SheerPower® 4GL
A Guide to the SheerPower Language

Touch Technologies, Inc.
10650 Scripps Ranch Blvd. Suite 100
San Diego, CA 92131

1-800-525-2527 or 1-858-566-3603
TouchTechSupport@Gmail.com

Note

® Windows 98, Windows ME, Windows 2000, Windows NT, Windows XP, and Windows Vista are registered trademarks of Microsoft.

® SheerPower is a registered trademark of Touch Technologies, Inc.

NOTICE

Touch Technologies, Inc. (TTI) has prepared this publication for use by TTI personnel, licensees, and customers. This information is protected by copyright. No part of this document may be photocopied, reproduced or translated to another language without prior written consent of Touch Technologies, Incorporated.

TTI believes the information described in this publication is accurate and reliable; much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material.

The information contained herein is subject to change without notice and should not be construed as a commitment by Touch Technologies, Inc.

Revised: October 29, 2009 for V5.0

Copyright ©2001-2009 Touch Technologies, Inc.

Preface

SheerPower 4GL is a next generation database language for Windows. It works on Windows 2000, Windows NT, Windows XP, Windows Vista, and Windows Server. It includes a Rapid Development Environment (SPDEV) and a fourth-generation Database Language (SP4GL) .

The SheerPower language:

- is concise and powerful
- promotes structured programming
- is easy to use--even for programming novices

SheerPower excels at both character and mathematical utility, combining them into a powerful, easy-to-use framework. SheerPower can be used to write programs of any size, from simple-input applications to vast database infrastructures and Web applications. SheerPower's layout may look somewhat similar to that of a structured BASIC program, but SheerPower is beyond BASIC!

SheerPower includes a transparent interface to the ARS (Advanced Record System) database engine as well as to other database engines. (ARS is bundled into SheerPower).

ARS is integrated into SheerPower featuring extremely high speed sequential and random access to database information. In addition, ARS is optimized for client/server applications through the use of high performance memory resident shared data.

The interface includes:

- simple structure statements for complex data manipulations
- numerous extensions which allow the technical programmer to develop advanced applications

The SheerPower environment is designed for developing complex applications rapidly. This interactive environment:

- allows immediate response and fast development
- allows you to interrupt program execution, change code and then continue execution
- has high-level debugging facilities for pinpointing program bugs quickly and accurately
- returns errors immediately--for rapid development--while code is optimized by the compiler

Development cycle time is the time it takes to edit, compile, link, run and debug code. SheerPower reduces this time by providing an interactive and fast response environment. SheerPower offers features that cut cycle time significantly--up to 80%.

You will find SheerPower an exciting and powerful programming tool.

SheerPower Reference Manuals

All SheerPower manuals are designed to provide information in a manner that is concise and easy to use. The manuals are:

- A Guide to the SheerPower 4GL Language
- SheerPower Coding Standards

Purpose of this Guide

The purpose of this Guide is to present the information you will need to develop programs with SheerPower. The intent is to provide you, the user, with simple and concise explanations of the SheerPower system, features, and language. This manual is designed to provide a basic and thorough explanation of each element. This manual can also be used as a reference guide by advanced users.

Intended Audience

This Guide is written for both experienced programmers and those self-learners who have had some exposure to computers and programming languages.

User Feedback and Technical Support

We want your feedback! **Tell us what YOU think about:**

- **SheerPower 4GL**
- the SheerPower 4GL **WEBSITE**
- the SheerPower 4GL **MANUAL**

We really want to know!

If you have any technical issues, please email them to us and we will respond!!

Send all feedback and technical support emails to:

TouchTechSupport@gmail.com

Thank you for using SheerPower 4GL!

Chapter 1
Getting Started in SheerPower

1.1 Getting Started

Download SheerPower 4GL

Download the latest version of SheerPower 4GL by clicking HERE .

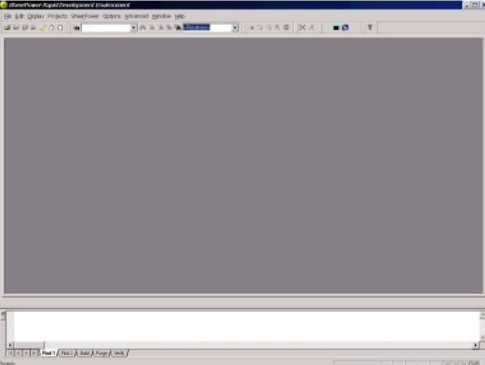
Click on the **FREE!! Download SheerPower 4GL NOW!** link.

Follow the instructions on the download webpage to download and install SheerPower 4GL.

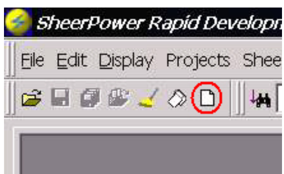
How to Start SheerPower Rapid Development Environment

To start SheerPower Rapid Development Environment (SPDEV), double click the **SheerPower** shortcut icon located on your desktop---a circle with a lightning bolt through it.

Below is an image of what SheerPower Rapid Development Environment looks like when it is running.



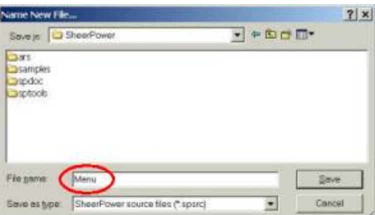
1.2 Creating a New Program in SheerPower

 To create a **new program** in SheerPower Rapid Development Environment, click once on the **New** icon in the toolbar---a white paper with one corner folded. This will create a new file for you to work in.

You can also click on **File** in the SPDEV toolbar and select **New** from the drop down menu of choices to create a new file.

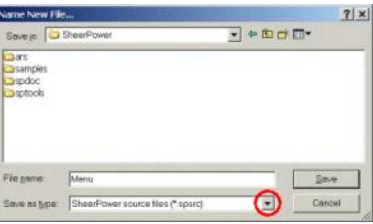
The **Name New File...** dialog box will appear and ask you to name your new file, and to specify the location you want to save it to.

Type the **name** of the new program file inside the **File name:** field. Name this new file **Menu**.



The default file type to save a file is **.SPSRC**. This is a SheerPower source program file. When you click on `[Save]` the file type extension will automatically be added on to the file name. Clicking on the down arrow will drop down a list of different file types to save your

files as.



Since .SPSRC is already selected as the file type, click on `[Save]`.

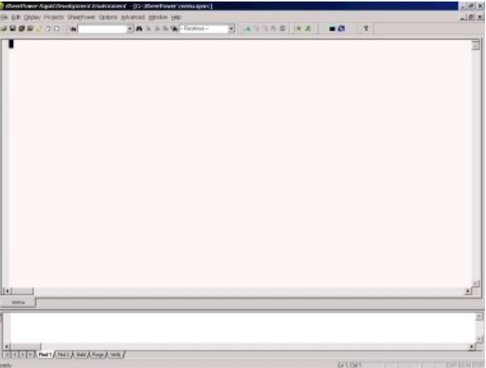
After you click on the `[Save]` button in the **"Name New File..."** dialog box, a new document window will open within SheerPower Rapid Development Environment.

A dialog box will also appear prompting you for your name, company name and program name.

When you fill in this information and click on `[OK]`, a **program template** will automatically be inserted into your new program file. See Appendix I, Developing Professional Applications with SheerPower for a detailed discussion on the special features built-in to SheerPower designed to save time and money when creating professional applications.

For the purpose of this basic tutorial, click on `[Cancel]`.

You can now begin to write your new SheerPower program inside the new document window.



1.2.1 Menu Program Example

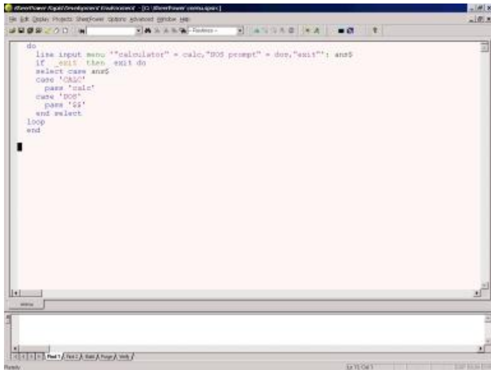
Copy and paste or type the following program into the new menu.spsrc file inside SPDEV.

Note

The **examples** show how commands or statements are used. Wherever possible, the example is a full program rather than a program fragment.

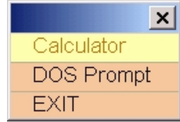
You are encouraged to type in (or copy/paste) the SheerPower examples into SPDEV throughout this manual and try running them.

```
do
  line input menu '"calculator" = calc,"DOS prompt" = dos,"exit"': ans$
  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
end
```



RUN this program by clicking on the **Run** icon in the SPDEV toolbar---the running man.

The following menu will appear:



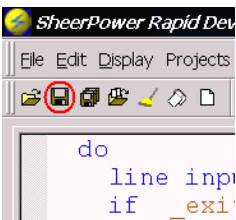
Click on the word **calculator** in the menu. The program will run your Windows calculator program.

Click on **DOS prompt** in the menu. The program will run the MS DOS Prompt (or Command Prompt) program.

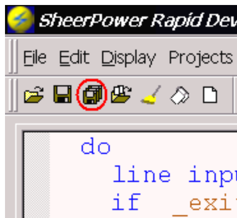
Click on **exit** or the `[X]` to close the menu. The console window will appear. Close it by typing in **exit** and pressing `[Enter]`, or by clicking on the `[X]` in the top right corner of the window.

To **SAVE** your program in SheerPower Rapid Development Environment, click once on the **Save** icon in the SPDEV toolbar---the floppy disk icon.

You can also click on **File** in the SPDEV toolbar and select **Save** from the drop down menu.



If you have multiple files open in SheerPower Rapid Development Environment that you wish to save, click on the **Save All** icon in the SPDEV toolbar---the icon with three floppy discs together.



Each time you **RUN** a program file from SPDEV, SheerPower automatically **SAVES** it.

SPDEV has a built-in **file backup** feature that allows you to keep up to the last 10 versions of any file edited in SPDEV. Each time you **save** your file in SPDEV, the old version is saved with a *version number* behind it:

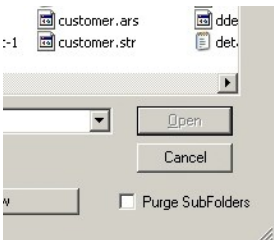
```
myprogram.spsrc   <-- current version
myprogram.spsrc-1 <-- last saved version
myprogram.spsrc-2 <-- 2nd last saved version
myprogram.spsrc-3 <-- 3rd last saved version
```

To choose the number of backup file versions for SPDEV to automatically save, click on **Options** in the SPDEV toolbar. Select **Change System Settings**, then enter in the **Number of Backup File Versions** to save (0-10).

To delete any unwanted SPDEV backup files, click on the **Purge** icon in the SPDEV toolbar. The **Purge Versions** dialog box will appear.

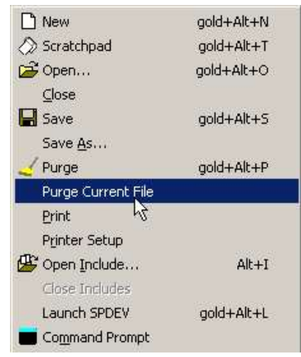
Select the folder or files that you wish to purge in the **Look in:** field. You can choose to purge either SheerPower (.spsrc) files or all file types in the drop down list at the bottom of the dialog box.

A **Purge Subfolders** option is also available inside the Purge Versions dialogbox. Just place a check inside the checkbox if you want to purge backup files that are inside the subfolders of the directory you have selected.



Once you have selected the folder or files to purge, and the file type, click on the **Purge Now** button. You will be prompted to confirm that you want to purge the files in the selected path. Click on the **OK** button. The **Purge Results** window will appear containing a list of all the backup files purged. If no backup files were in the folder nothing will be purged.

The option to purge only the backup versions of the current file you are working in is also available. Click on **File** in the toolbar options, then select **Purge Current File**.



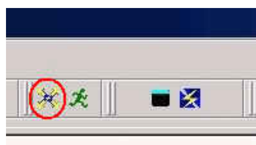
Note

Keep Your Valuable Source Code Hidden From Prying Eyes! *Optional* SheerPower 4GL GOLD Licenses are available.

Royalty FREE distribution---you can **always** distribute SheerPower-based applications to others---royalty free! And, you can do so **without** the *optional* SheerPower 4GL License, but you must provide them with the source code to your application.

Visit our website for more information.

In the SPDEV toolbar, click on the **DEPLOY** icon.



This will bring up the **Enter License Key** dialog box. You can copy and paste your SheerPower 4GL license key information in here. Click on `[OK]` to proceed with the deployment of the program.

Note

You will see the **Enter License Key** dialog box the FIRST TIME you do a DEPLOY when newly running SPDEV. If you restarted SPDEV 5 times in a day, you would see the "Enter License Key" dialog box appear 5 times (providing you deployed each time).

The **Deployment Properties** dialog box will appear. You can now enter a **Password** to protect this program file.

This is an **OPTIONAL PASSWORD** used to PROTECT your program. This way, only someone who **knows the password**can run the program.

So, you could post your .SPRUN file on the Internet---and anyone could download it. But, only those who have the PASSWORD could run it!!

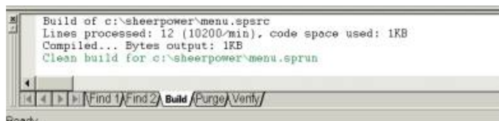
Place a check in the box beside **Don't show this dialog again** if you do not want to be prompted to enter a password every time you click on the deploy icon.

Note

If you want to deploy a program with a password, you have to specify a password each time you deploy.

Click on the `[OK]` button to deploy the program.

You will now see in the bottom frame of the SPDEV window that deploying the program **compiled** it. You will also see the number of lines of code, file size, and whether or not it was a *clean build* or if there were any *compile errors*.

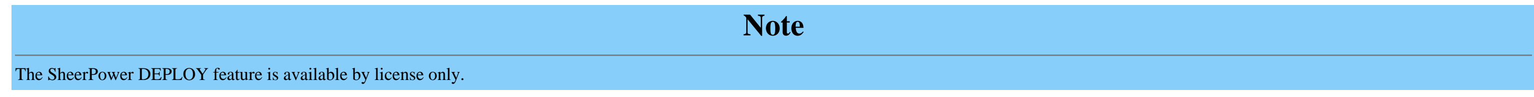


Look inside your SheerPower folder (c:\sheerpower is the default location). You will see that a file named **menu.sprun** has been created.

This **.SPRUN** file allows you to distribute applications without the recipients ever being able to see the source code.

Optional SheerPower 4GL Licenses Available

Royalty FREE distribution---you can **always** distribute SheerPower-based applications to others---royalty free! And, you can do so **without** the *optional* SheerPower 4GL Licenses, but you must provide them with the source code to your application.



Note

The SheerPower DEPLOY feature is available by license only.

Once you have written a program you can **DEPLOY** it by clicking on the **Deploy** icon in the toolbar.

Note

If you are editing a file in SheerPower Rapid Development Environment that is not a program file, the Deploy and Run icons will be disabled (greyed out).

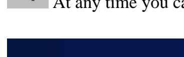
Deploying your main source code before you run your program will tell you immediately if there are any compile errors in your code. Any errors will appear inside the bottom window frame of SheerPower Rapid Development Environment. If there are no compile errors, SheerPower will tell you that you have a **clean build**. The name of your file, the number of lines written, the speed it was compiled at, and the size of the file will also appear inside the SheerPower Rapid Development Environment bottom window frame.

```
Build of C:\SheerPower\test.spsrc
Lines compiled: 58 (24800/min), code: 1KB
Clean build for C:\SheerPower\test.sprun
```

The **DEPLOY** feature also creates a **.SPRUN** file---such as **test.sprun**. SPRUN files are useful in that:

- All source code is hidden---preserving your intellectual property.
- A password can be attached to the program---only those that know the password can run the program.
- SPRUN files are textual---so they can easily be emailed or made available on a web site.

Deployed applications can include the **%COMPILE directive**. The %COMPILE directive allows you to enter in up to 100 lines of text inside the .SPRUN file. This text cannot be altered or the application will not run. See Section 3.9.1,%COMPILE for details.

 At any time you can **RUN** your main source program. To run a program in SheerPower Rapid Development Environment, click on the **Run** icon in the toolbar---the running man.

You can also **Run** a SheerPower program by double-clicking on its name using Windows Explorer. Both **SPSRC** and **SPRUN** program files can be run in this way. You can also **DRAG-and-DROP** either **SPSRC** or **SPRUN** program files onto the **DESKTOP**---then double-click to run the program.

How to Run a SheerPower Program on Another Computer

To run a SheerPower program on another computer:

- Download and install the SheerPower 4GL Virtual Machine to the other computer.
- Double-click on either the .SPSRC or .SPRUN file to run the program.

Temp Folders and Security in SheerPower 4GL

As a security feature, SheerPower programs cannot be run if they are inside of any temporary folder.

This prevents users from accidently running malicious code written in SheerPower that a hacker might place into an email or webpage.

In the program examples, **keywords** in SheerPower **statements and commands** are in **bold**, and information you (the user) supply is in regular text. The program examples are designed so that they can be typed, or copied and pasted into the SheerPower Rapid Development Environment and run. Below each example, the results are shown.

SheerPower Coding Standards

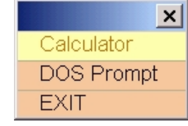
All examples shown in this manual conform to SheerPower coding standards. This includes all code being typed in **lower-case** and with proper line indentation. For more information on coding standards in SheerPower, please refer to the SheerPower Coding Standards manual and Appendix A, Coding Principles and Standards.

To assist you, SheerPower tries to automatically indent code according to the statement last typed in.

*The double quote (") can be used in place of the single quote (') as long as they are paired.

Try the following examples:

```
do
  line input menu '"calculator" = calc,"DOS prompt" = dos,"exit"': ans$
  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
end
```




```
// A simple quiz program

  woodpecker$ = 'sheerpower:samples\woodpecker.jpg'

  quiz_form$ = '<sheerpower persist><title>Quiz</title><form>' +
               '<center><h3>Skill Testing Question</center></h3>' +
               '<img src="' + woodpecker$ + '" border=3 align=middle>'
  quiz_form$ = quiz_form$ + '<font color=green> ' +
               '<b>What type of woodpecker' +
               ' is in this photograph?</b></font><p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Pileated Woodpecker"> ' +
               '<i>Pileated Woodpecker<p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Hairy Woodpecker"> Hairy Woodpecker<p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Redheaded Woodpecker"> ' +
               'Redheaded Woodpecker</i></b>'
  quiz_form$ = quiz_form$ + '<p><input type=submit name=submit value="Submit">' +
                            '<input type=submit name=exit>' +
                            '</form>'
  correct$ = 'Hairy Woodpecker'
  good$ = '<sheerpower width=500 height=300 color=green>' +
          '<form><h1>Congratulations!!  ' +
          correct$ + ' is the correct answer!!</h1>' +
          '<p><center><input type=submit></center></form>'

  do
    line input dialogbox quiz_form$: ans$
    if _exit then stop
    z0$ = element$(ans$, 1, chr$(26)) // get the first response
    value$ = element$(z0$, 2, '=')    // get the the value  (name=value)
    if value$ = correct$  then exit do
    message error: "Sorry, this is not a ";value$
  loop
```

```
  line input dialogbox good$: ans$
  end
```

```
// To present the top news story from CNN.COM
// Note:  From time to time CNN changes its format, so the
//        main$  and  end_main$  sometimes have to be changed.

  main$ = '<div class="CNN_homeBox">'
  end_main$ = '</div>'

  begin_form$ = '<form>' +
                '<h1><font color=green>' +
                'Top News from CNN' +
                '</font></h1>' +
                '<br><h2>'

  news_ch = _channel
  open #news_ch: name 'http://www.cnn.com'

  crlf$=chr$(13)+chr$(10)
  state$ = 'find_main'
  do
    line input #news_ch, eof eof?: text$
    if eof? then exit do
    select case state$
    case 'find_main'
      z0 = pos(text$, main$)
      if z0 = 0 then repeat do
      z0$ = mid(text$, z0+len(main$))
      z0  = pos(z0$, '>')
      if z0 = 0 then repeat do
      z0$ = mid(z0$, z0+1)
      dbox$ = begin_form$ +z0$
      state$ = 'gather_news'
    case 'gather_news'
      z0 = pos(text$, end_main$)
      if z0 > 0 then
        dbox$  = dbox$ + left(text$, z0-1)
        state$ = 'show_news'
        repeat do
      end if
      dbox$ = dbox$ + text$ + crlf$
      repeat do
    case 'show_news'
      // first add in the  http://www.cnn.com  to any anchors
      dbox$ = replace$(dbox$, '<a href="==<a href="http://www.cnn.com/',',', '==')
      // now finish off the input form with a single "submit" button
      dbox$ = dbox$ + '</h2><p><input type=submit></form>'
      line input dialogbox dbox$: ans$
      exit do
    end select
  loop
  end
```

Top News from CNN

President Bush said Monday his administration would use "all the tools at our disposal" to oust Iraqi President Saddam Hussein. "I do firmly believe that the world will be safer and more peaceful if there's a regime change in that government," Bush said at a news conference.
FULL STORY

```
// Evaluate an expression
// Type in the expression and press [enter]
//
// The expression can by very complex.  For example:
//        ((34-324)/14)*10.1234

  declare dynamic answer

  formtop$= '<sheerpower persist>' +
                '<title>Expression Evaluator</title>' +
                '<form><h2>Enter an expression</h2>' +
                'For example--> ((344574647-324978)/154574)*13350.1234' +
                '<input name=expr submit>' +
                '<p>'

  formend$ = '<p><input type=submit name=submit value="Submit">' +
    '<input type=submit name=exit value="Exit"></form>'

  form$ = formtop$ + formend$
  do
    input dialogbox form$: response$
    if _exit then exit do
    z0$ = element$(response$, 1, chr$(26))  // get the first field
    expr$ = element$(z0$, 2, '=')          // get the data for the first field
    if expr$ = '' then repeat do
    when exception in
      answer = eval(expr$)
    use
      answer = extext$
    end when
    if dtype(answer) <> 1 then answer = str$(answer)
    form$ = formtop$ +
                '<h2><font color=green>' + expr$ + '</font></h2>' +
                '<h3>' + answer + '</h3>' +
                formend$
  loop
  end
```

SheerPower Rapid Development Environment makes professional application development easy with **BUILT-IN TEMPLATES**, and **SPECIAL KEYSTROKES**.

Some of SheerPower's features are:

- Program Template
- Routine Header Template
- Auto-organize routines
- Auto-align "=" within routines and headers
- Easy commenting/uncommenting code
- Generate special DEBUG comment lines--easy to find!

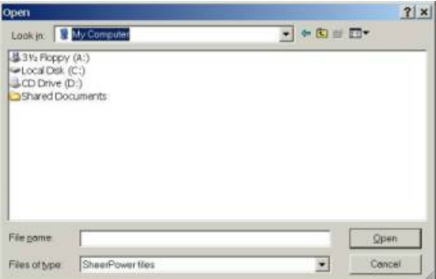
Plus many more special keystrokes that enable a programmer to quickly and efficiently write clean, readable and accurate code.

See Appendix I, Developing Professional Applications with SheerPower for a detailed discussion on the special features built-in to SheerPower designed to save time and money when creating professional applications.

See Appendix F, Keystrokes for SheerPower Rapid Development Environment for a complete reference guide to the special keystrokes in SheerPower Rapid Development Environment.

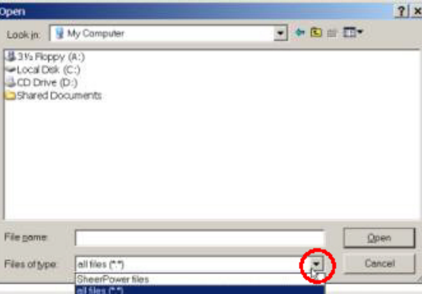
If you need to open an **existing file** stored on your computer to read or edit in SPDEV, click on the **Open** icon in the toolbar---an open file folder. You can then choose the directory and file you want to open from the **OPEN** dialog box that appears.

You can also open an existing file by clicking on **File** in the SPDEV toolbar, then selecting **Open** from the drop down menu. The **Open** dialog box will appear:

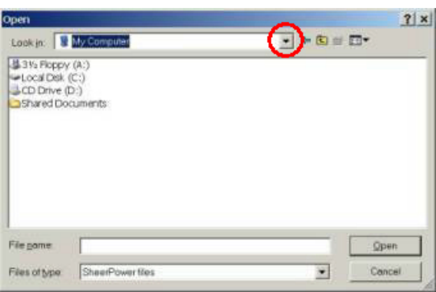


Whether you are creating a new file or opening an existing file to edit in SheerPower Rapid Development Environment, the default folder location is c:\sheerpower. The default file type being created/searched for is any of the SheerPower program files (*.spsrc, *.int, *.spinc and *.inc).

To open a different file type, click on the down arrow beside the *Files of type:* field (located at the bottom of the dialog box). Choose *All Files* to see all of files of the specified type within that folder.



To search for a file or to save your file to a different folder (other than c:\sheerpower), click on the down arrow beside the *Look in:* field (located at the top of the dialog box) and choose the folder you wish to search/save in.



SPDEV can open ANY kind of file that is not **binary data**.

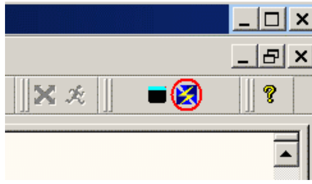
If you make an error in your program, such as misspelling the word **print**:

```
  input 'Please enter your name' : name$
  printt 'Hello, '; name$

Build of C:\SheerPower\tests.spsrc
file: C:\SheerPower\tests.spsrc
    (line 2, column 0):  Unrecognized statement
```

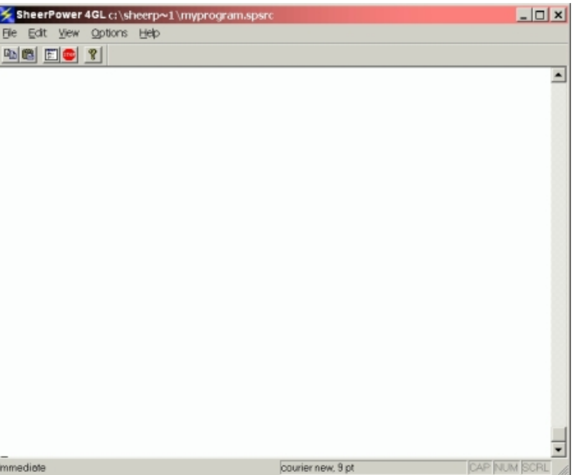
SheerPower will display an **error message** in the bottom frame of the window (the **results window**). You can double click on the line number given in the error message. SheerPower will highlight the line of code containing the error for you, and place the cursor at the beginning of the error. You can now retype the correction and rebuild your main source program.

When a main source program is run in SheerPower Rapid Development Environment, the **SP4GL Console Window** will open (also referred to as "console window").

To open the SP4GL Console Window without running a main source program, click on the **SP4GL Console Window** icon in the SPDEV toolbar---the square with a lightning bolt through it.



To close the SP4GL Console Window, type in **exit** and press the `[Enter]` key, or click the **X** in the top right corner of the window.



SP4GL Console Window

The console window must be closed *before* attempting to run a program again. The running man icon in the SheerPower Rapid Development Environment (SPDEV) toolbar will remain greyed out until the console window is closed.

More than one instance of the SP4GL Console Window can be opened by clicking on the SP4GL Console Window icon in the SPDEV toolbar.

In the console window the mouse can be used to select (highlight) text on the screen.

Below is a table containing the special keystrokes available when working in the SP4GL Console Window.

Table 1-1 SP4GL Console Window

Keystroke	Function Performed
ctrl/a	selects all text (both on and off screen)
ctrl/c	places selected text into the clipboard
ctrl/m	places contents of message history into the clipboard
ctrl/t	places contents of screen (including scroll buffers) into the clipboard

In SPDEV, you can get help by clicking on the **Help** icon in the toolbar---a question mark.

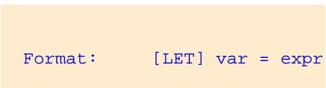
You can get help when in the console window by typing **help** and pressing the `[Enter]` key or by clicking on the **?** icon in the toolbar.

Selecting HELP in SPDEV or the console window will open up a new browser window that loads the online SheerPower documentation.

Utilize the **Contents** or the **Index** to help you find the required documentation.

User Conventions

In some places in this manual, the format of statements and commands is represented abstractly. For example:



```
Format:     [LET] var = expr
```

Certain conventions are used to represent values which you (the user) must supply. These conventions are always shown in *lowercase*. These are the user conventions:

Symbol	Description
array	Array element.

block of code	A SheerPower statement or a series of SheerPower statements.
chnl	An I/O channel associated with a file.
chnl_num	An I/O channel associated with a file.
cond_expr	Conditional expression.
col	Column. Used in the TAB or AT option to indicate the column to print at.
const	Constant value.
expr	Expression. May be a variable, constant, array element, structure reference, etc. or any combination of the previous, separated by operators.
field_expr	Field name expression.
func	Function.
handl	Exception handler. Pertaining to an exception handling routine.
handler	The block of code in an exception handler which is executed when an exception is generated.
int	Integer value.
int_expr	Integer expression.
int_var	Integer variable.
label	Alphanumeric statement label.
libr	Pertaining to libraries.
line_num	Program line number.
name	Name. Indicates the declaration of a name or the name of a program unit, such as a subprogram.
num	Numeric value. Num indicates that either a real or integer numeric value may be used.
num_expr	Numeric expression. Num_expr indicates that either a real or integer numeric value may be used.
num_var	Numeric variable.
param	An expression which is passed to a function, subprogram or other program unit. A parameter can be any valid SheerPower expression.
protected block	The block of code protected by an exception handling routine.
real	Real number. Indicates that only a real number may be used.
row	Row. Used in the AT option to specify the row to print at.
str	String. Str indicates that only a string value may be used.
str_expr	String expression.
str_var	String variable.
struc	A SheerPower structure.
struc_name	Structure name.
struc_expr	Structure expression.
sub	A SheerPower subprogram.
target	The target point of a branch statement. The target can be either a program line number, an alphanumeric label, or a ROUTINE statement.
uncond_expr	Unconditional expression. Uncond_expr indicates that only an unconditional expression can be used.
var	Variable. May be a simple variable, array element or a structure reference.

Other Conventions

Other conventions used in this manual are:

[ ]	**Optional portions** of a statement are enclosed in **brackets**: `INPUT [PROMPT str_expr] : var`
{ | }	**Braces combined with a vertical bar** indicate that one of the elements in the braces **must be chosen.** The elements are separated by the vertical bar: `EXIT {FOR | DO}`
[|]	**Brackets combined with a vertical bar** indicates that one of the options in the brackets **can be used**. The elements are separated by the vertical bar. Only one of the elements can be used: `DO [WHILE expr | UNTIL expr]`

...	An **ellipsis** indicates that the language element **can be repeated**. An ellipsis following an option enclosed in brackets indicates that you can repeat the format of the whole option. `INPUT [PROMPT str_expr] : var, var...`

dashes	**Three sets of dashes** indicate a **block of code**.
	
	`DO`
	`---`
	`--- block of code`
	`---`
	`LOOP`

. . .	**A vertical ellipsis** indicates the **continuation of a body of program code** which is not a defined block of code.

Part of this chapter and several other chapters in this Guide describe the commands and statements in the SheerPower language. The purpose of this section is to explain how the command and statement information is presented.

Statements that can only be used in connection with one another, such as **if then** or the **select case** statements, are described together. Each command and statement description includes the following information:

FORMAT:

```
input [options] var [, var...]
```

FORMAT:

```
extract structure struc_name: key field = expr1 [to expr2]
  ---
  ---  block of code
  ---
end extract
```

The **FORMAT** contains the command or statement format. This includes the required keywords and the optional elements, if there are any.

Keywords are words that must be typed exactly as shown in the format. Keywords inside the program examples are shown in **bold**; e.g., **extract structure, key, to, end extract**. The elements in regular text represent information that must be provided by the user in order to use the statement. For example, "struc_name" (the structure name) must be provided by the user in this statement. "field" and "expr1" information must also be provided.

Keywords and elements in brackets are optional. Multiple options are separated with commas.

SheerPower is space sensitive, so spaces must be included where they are shown in the format. Either the single quote or double quote can be used as long as they are paired.

EXAMPLE:

```
  input 'Enter full name', timeout 30, elapsed x: name$
  print name$
  print 'Elapsed time: '; x
  end


Enter full name? TTI Tester   <---- type in your name here, then press the [Enter]
key
TTI Tester
Elapsed time:  13
```

The **EXAMPLE** shows how the command or statement is used. Wherever possible, the example is a full program rather than a program fragment.

You are encouraged to type in (or copy/paste) the SheerPower examples into SPDEV throughout this manual and try running them.

PURPOSE:

In some cases, **PURPOSE** information is provided for usage clarity.

DESCRIPTION:

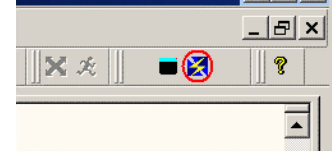
The **DESCRIPTION** tells about the command or statement, explains how it works, and other specific information.

Chapter 6, Built-in Functions of this Guide provides information on SheerPower's built-in functions, error and exception messages, and other general topics. You will want to get familiar with the **built-in functions**, as they will allow you to easily manipulate data.

Now that you have an idea of how to use this manual and work in SheerPower, the remainder of this Guide will provide the information you need to create and use SheerPower programs.

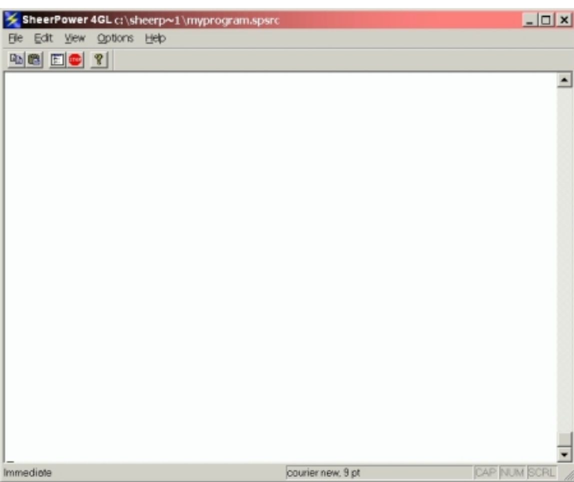
This chapter describes SheerPower debug facilities and how to use them. It also describes the commands that can be used in the SP4GL Console Window for debugging and experimentation purposes.

To open the SP4GL Console Window, click once on the **SP4GL Console Window** icon inside SheerPower Rapid Development Environment (SPDEV) in the top toolbar---the square with a lightning bolt through it.



Using the SP4GL Console Window

While writing a program in SPDEV, the **SP4GL CONSOLE WINDOW** (also referred to as the "console window") is utilized for debugging and experimentation purposes.



Note
When using the scrollbars on the console window, the maximum number of scrollback lines is 500.

SheerPower **COMMANDS** cause SheerPower to take some action immediately. SheerPower commands are not normally used *inside* programs. Here is a list of SheerPower commands that are generally used only in the console window:

- BUILD
- RUN
- LIST
- GO
- STEP
- SHOW FILES

SheerPower **STATEMENTS** are used *inside* program files (.SPSRC files inside SPDEV).

When a SheerPower statement is used inside the console window, it becomes a **COMMAND**. For example, the **PRINT statement** becomes a command when used in the console window. All SheerPower statements are available to be used as commands while debugging inside the console window.

The SP4GL Console Window is a limited editing environment. One or more commands can be performed at a time. Multiple commands must be separated by a backslash. If the line ends with a backslash, additional commands can be entered on the next line. The following example shows a single PRINT command inside the console window.

To perform the following example, open the SP4GL Console Window by clicking on the **SP4GL Console Window** icon in the SPDEV toolbar. Inside the console window, type:

print 'Hi there!'

and press the `[Enter]` key.

```
print 'Hi there!'  //<--- your entry

Hi there!  <--- result
```

PRINT command in the SP4GL Console Window

In the SP4GL Console Window, if you type in the name of a variable and press `[Enter]`, it will assume you want to print out the value of that variable. Therefore, the PRINT command is not necessary to type in.

Type the example below into the console window:

```
for i = 1 to 10\
print i\
next i
1  //<--result
2
3
4
5
6
7
8
9
10

OR:

for i = 1 to 10\print i\next i
1  //<--result
2
3
4
5
6
7
8
9
10
```

Here is another example you can try typing inside the console window. Note that the date$ function will return the **current date**. This will be different than the date you see returned in this example.

```
date$ //<--press [Enter]
20010710 //<--result
```

The SP4GL Console Window handles command recall. The user can recall the last 100 previously entered commands by using the UP and DOWN arrow keys.

To perform the following example, open the console window by clicking on the **SP4GL Console Window** icon in the SPDEV toolbar. Inside the console window, type:

```
print date$                   <--- type this line in and press [Enter]
20010716    //<--result
print date$(days(date$), 3)   <--- type this line in and press [Enter]
16-JUL-2001  //<--result
print date$(days(date$), 3)   <--- press the UP arrow key
print date$                   <--- press the UP arrow key, then press [Enter]
20010716  //<--result
```

When in the console window, the `[Tab]` key can be used to perform various operations that cut down the time it takes to debug program code. As you get familiar with SheerPower and the SheerPower language, you will find that the `[Tab]` key features can be very helpful.

In the SP4GL Console Window, enter part of a command and then press the `[Tab]` key. SheerPower will then supply the full name of the command. If the command is ambiguous, a selection menu of all valid choices will appear.

In the following example, SheerPower completes "**so**" to form the command "**sort**".

Open the SP4GL Console Window by clicking on the **SP4GL Console Window** icon in the SPDEV toolbar. At the flashing cursor in the console window, type in the word "**so**". Press the `[Tab]` key and you will see "**so**" completed to be the command "**sort**".

```
so[Tab]  <--- your entry, press [Tab]

sort  <--- SheerPower supplies
```

Next, type the word "**in**", then press the `[Tab]` key. In this example, "**in**" is ambiguous, so SheerPower provides a menu of choices:

```
in[Tab]                                   +--Choices---+
                                          |  include   |
                                          |  inf       |
                                          |  input     |
                                          +------------+
```

Command Completion in the SP4GL Console Window and SPDEV

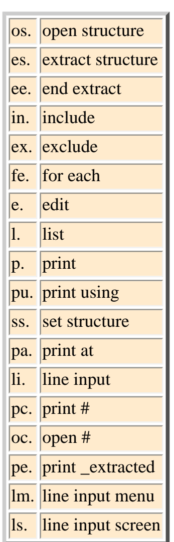
Command completion will also work in SPDEV. However, the SP4GL Console Window and SPDEV use different methods to perform command completion. Therefore, they will not always agree on which command or statement you are wanting to complete.

The `[Tab]` key can be used to correct misspelled commands in the console window, as well as misspelled statements in SPDEV. Just position the cursor on the misspelled word and press the `[Tab]` key. The word will be replaced with its correct spelling. In the example below, the misspelled command **SET STRCTRE** is changed *instantly* to the correct command spelling when the `[Tab]` key is pressed.

To perform the following example, open the SP4GL Console Window by clicking on the **SP4GL Console Window** icon in the SPDEV toolbar. Inside the console window, type:

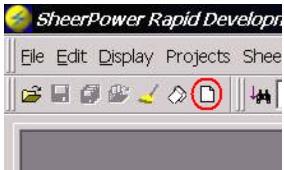
```
set strctre[Tab] //<-- press [Tab]
set structure
```

SheerPower uses abbreviations for a number of commands. The `[Tab]` key can be used with any of the specific abbreviations to expand the abbreviations. To use this feature, enter an abbreviation and then press the `[Tab]` key. The period '.' must be used after each abbreviation. The current abbreviations that can be used are:



os.	open structure
es.	extract structure
ee.	end extract
in.	include
ex.	exclude
fe.	for each
e.	edit
l.	list
p.	print
pu.	print using
ss.	set structure
pa.	print at
li.	line input
pc.	print #
oc.	open #
pe.	print _extracted
lm.	line input menu
ls.	line input screen

Open a new file inside SheerPower Rapid Development Environment by clicking once on the **New** icon in the toolbar---a white paper with one corner folded.



Name the new file "**sample_program**". Press the **Save** button inside the New File dialog box. The file type **.SPSRC** will automatically be added when the Save button is pressed.

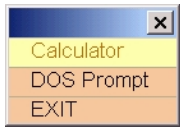
The **Get Program Info** dialogbox will appear, prompting you for your name, company name and program name. This will create a **Program Template** inside the new program file. For the purpose of this documentation, press the `[Cancel]` button. This will create an empty program file, with no program template.

See Section I.2, Program Template for more on creating a program template within a .SPSRC file. If you *do* create a program template in your new file, copy and paste or type the following program into the new file below the **Main Logic Area** above the word **stop**.

Otherwise, copy and paste or type the following program into the new program file:

```
do
  line input menu '"Calculator" = calc,"DOS Prompt" = dos,"EXIT"': ans$
  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
```

You can run this program by clicking once on the **Run** icon---the running man. The result of the program is shown below:



Choose 'EXIT' in the menu to complete running the program, then close the console window by typing '**exit**' or clicking on the '**X**' in the top right corner of the window.

This example will be used throughout the next few sections of this chapter.

Notice that after you completed running the sample program the following information appears in the **results window** (at the bottom of SPDEV) inside the **Build** tab:

```
Build of c:\sheerpower\samples\sample_program.spsrc
Lines processed: 10 (30000/min), code space used: 1KB
Clean build for c:\sheerpower\samples\sample_program.spsrc
```

If you had any compile errors in your source code, they will be listed here along with the location of the error, and the type of error. You can click directly on the line detailing the error and it will highlight the line of code containing the error directly in the program.

```
debug on
for i = 1 to 6
  prnt i  // typo error in the print statement
  if  i = 4  then halt
next i
end
```

When you run the program, the following information is displayed in the Build tab window at the bottom of SPDEV:

```
 Build of c:\sheerpower\samples\test.spsrc
file: c:\sheerpower\samples\test.spsrc
    (line 3, column 1):  Unrecognized statement
    Unrecognized statement at MAIN.0002
```

Left click directly on **(line 3, column 1): Unrecognized statement**. It will highlight that text in blue, and inside your program the line of code with the error will be highlighted in yellow. You can then correct the typo in the statement and rebuild the program.

In the console window, errors are returned either immediately after typing a syntactically incorrect line, or when the RUN command is given and your program is syntactically incorrect.

Errors happen when the program is compiling. Exceptions happen as the program is running.

In the case that the error happens outside of a routine (in the main logic area) the location of the error will always start with "**main**" followed by a period. Following the period is the source code line number.

I.E., **main.0006** means the error occurred in the 6th line from the top of the file.

I.E., **do_totals.0003** means the error occurred in 3 lines from the definition of the routine called **do_totals**.

For a list of the different SheerPower error and exception messages, see Appendix C, SheerPower's Error and Exception Messages.

In SPDEV the Alt + UP or DOWN arrow key can be used to move up or down a specific number of lines in your file. For more specialized programming keystrokes in SPDEV, see Appendix F, Keystrokes for SheerPower Rapid Development Environment.

FORMAT:

```
        [BUILD 'program_name']
```

EXAMPLE:

Important note on the following example

The following example assumes that you have a SheerPower program file called 'sample_program' in your SheerPower folder. This sample program was created in the previous section. See Section 2.2, Creating a Sample Program.

To run this example, open the console window by clicking once on the **SP4GL Console Window** icon in the SPDEV toolbar. Type in the example as shown below inside the SP4GL Console Window, then press `[Enter]`.

```
build 'sample_program'   //<--- type this line in and press [Enter]

Building c:\sheerpower\sample_program.spsrc ...
Lines compiled: 11 (33000/min), code: 1KB
```

PURPOSE:

Use the BUILD command to build a program in the SP4GL Console Window. Any build errors will be listed when you perform the BUILD command.

DESCRIPTION:

The BUILD command used in the console window defaults to the SheerPower folder. When building a program that is not stored inside the SheerPower folder, the path (location) of the program must be specified. See the following example:

```
build 'c:\windows\desktop\sample_program.spsrc'


Building c:\windows\desktop\sample_program.spsrc ...
Lines compiled: 11 (33000/min), code: 1KB
```

FORMAT:

```
        RUN ['file_spec']
```

EXAMPLE:

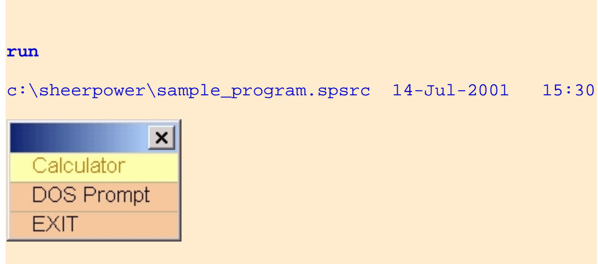
Important note on the following example

The following example assumes that you have a SheerPower program file called 'sample_program.spsrc' in your main SheerPower folder. This sample program was created in a previous section. See Section 2.2, Creating a Sample Program.

To perform this example, open **sample_program.spsrc** inside SheerPower Rapid Development Environment. Click once on the **Run** icon in the toolbar to run the program. A menu will appear on your screen with three choices. Choose '**Exit**', and the console window will appear.

Inside the console window, type 'RUN', and press `[Enter]`. The RUN command will cause the program to execute again, and the resulting menu created will appear.

```
run

c:\sheerpower\sample_program.spsrc  14-Jul-2001   15:30
```



PURPOSE:

The **RUN** command is used to execute a program after it has been written inside SPDEV. This way you can remain inside the SP4GL Console Window and run your program after you experiment with elements of your program.

The RUN command can be used:

- after previously running the program from SPDEV by pressing the RUN icon in the toolbar.
- after building the program within the console window using the BUILD command.

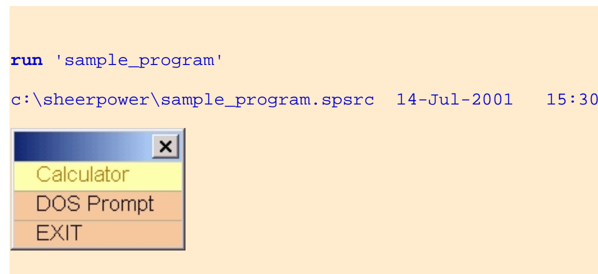
DESCRIPTION:

The RUN command executes programs. RUN with no file specification runs the current program.

When SheerPower executes the RUN command, SheerPower displays a header with the program name, current date and time. SheerPower then executes the program.

A file specification can be given with the RUN command. If a file specification is provided, SheerPower searches for the file, loads it, and then runs it. If no file type is given, SheerPower will use the default file type **.SPSRC**.

```
run 'sample_program'

c:\sheerpower\sample_program.spsrc  14-Jul-2001   15:30
```



Choose '**Exit**' from the menu to complete running the program, then close the SP4GL Console Window.

If you are running a program that is not stored inside the SheerPower folder, you must specify the path (location) of the program.

FORMAT:

```
        LIST [routine_name, routine_name, ...]
```

EXAMPLE:

To perform the following example, open the console window by clicking once on the **SP4GL Console Window** icon in SPDEV. Use the BUILD command to build the sample program (sample_program.spsrc). Then use the LIST command to display all the lines in the program.

```
    build 'sample_program.spsrc'
Building c:\sheerpower\sample_program.spsrc ...
Lines compiled: 11 (33000/min), code: 1KB
    list
c:\sheerpower\sample_program.spsrc  06-Jul-2001  16:29
do
  line input menu '"Calculator" = calc,"DOS Prompt" = dos,"EXIT"': ans$
  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
```

PURPOSE:

Use the **LIST** command to display all or part of your program source code.

DESCRIPTION:

LIST displays lines from the current program. The listing includes a header with the program name, current date and time. LIST by itself lists the entire program, including routine headers and comment lines.

Specific sections of a program can be listed by referencing the program's routines. You can also list combinations of routines. For example:

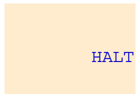
list do_input	lists the lines of code under the routine "do_input"
list date_routine	lists the lines in the "date_routine" routine
list do_input, date_routine	lists the lines from both routines

If a routine is specified which does not exist, nothing is listed.

You can use the LIST command:

- after previously running the program from SPDEV by pressing the RUN icon in the toolbar.
- after building the program within the console window using the BUILD command.

FORMAT:



```
        HALT
```

EXAMPLE:

Open SPDEV by clicking on the SPDEV shortcut icon on your desktop.

Open the sample program file **sample_program.spsrc** inside SPDEV by clicking on the **Open** icon inside the SPDEV toolbar.

Insert the **HALT** statement into the sample_program.spsrc source code as shown below.

Then run the program by clicking once on the **Run** icon in SPDEV.

```
do
  line input menu '"Calculator" = calc,"DOS Prompt" = dos,"EXIT"': ans$

  halt         //<--- insert HALT statement here

  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
```

Calculator
DOS Prompt
EXIT

Choose '**Calculator**' from the menu. The following will appear inside the SP4GL Console Window:

```
Halt at MAIN.0003
--
Call Stack Depth: 0
  MAIN.0003                            halt
--
Recently Called Routines
--
```

Keep the SP4GL Console Window open to continue with the next HALT statement example.

PURPOSE:

HALT is used to interrupt program execution, check values, and then continue execution.

The HALT statement must be inserted inside the source code before running the program. The HALT statement works the same way as the BREAK statement except that it **always** interrupts program execution. The BREAK statement will only interrupt program execution while DEBUG is ON. For a detailed explanation of the BREAK statement, please see Section 2.5.7, BREAK Statement.

DESCRIPTION:

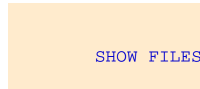
HALT interrupts program execution, but it does not close any files, nor does it write the active output buffer. Once halted, the user can then check values, enter debug commands or any SheerPower statements and expressions. Execution can be continued with the GO command.

You can continue to run the sample program in the console window as follows:

```
Halt at MAIN.0003
--
Call Stack Depth: 0
  MAIN.0003                            halt
--
Recently Called Routines
--
print ans$ <--- type this line in and press [Enter]
CALC
go<--- type in 'GO' and press [Enter]
```

The calculator program will appear when your program resumes execution. To exit the program, choose '**Exit**' in the menu, then close the console window.

FORMAT:



EXAMPLE:

To perform the following example, open the console window by clicking once on the **SP4GL Console Window** icon in SPDEV. Have the HALT statement inside the program. The **show files** command is generally used after a HALT statement halts a program's execution.

```
  line input menu '"Calculator" = calc,"DOS Prompt" = dos,"EXIT"': ans$
  halt
  if  _exit  then  exit do
  select case ans$
  case 'CALC'
    pass nowait: 'calc'
  case 'DOS'
    pass nowait: '$$'
  end select
loop
```

Use the BUILD command to build the sample program (sample_program.spsrc). Then use the RUN command to run the program.

```
    build 'sample_program.spsrc'
Building c:\sheerpower\sample_program.spsrc ...
Lines compiled: 11 (33000/min), code: 1KB
    run
```

PURPOSE:

Use the **SHOW FILES** command after a program **halts** to display a list of all open files and their status.

DESCRIPTION:

SHOW FILES displays a list of all open files and their status. If the files are ARS files (tables) then it some information will be displayed about their key structure, and recordsize, etc.

FORMAT:

EXAMPLE:

Copy/Paste or type the following example into a new file inside SPDEV. Name it 'test.spsrc'.

```
debug on
for i = 1 to 6
  print i
  if  i = 4  then halt
next i
end
```

Run the program by clicking once on the **Run** icon.

The following result will appear in the console window:

```
 1
 2
 3
 4
Halt at MAIN.0003
--
Call Stack Depth: 0
  MAIN.0003                           if  i = 4  then halt
--
Recently Called Routines
--
```

PURPOSE:

GO is used to continue a program after it has been interrupted.

DESCRIPTION:

GO resumes program execution after it has been interrupted. Once execution has stopped, you can enter immediate mode and debug commands, change code, etc. GO lets you resume execution even after changing code. If a HALT or BREAK statement was used, execution resumes at the first statement after the halt or break.

Type in the PRINT command as shown below inside the console window, and press `[Enter]`. The value will be printed out as requested. You can then type the **GO** command in the console window and press `[Enter]`. The program will then resume execution.

```
 1
 2
 3
 4
Halt at MAIN.0003
--
Call Stack Depth: 0
  MAIN.0003                           if  i = 4  then halt
--
Recently Called Routines
--
print sqr(i)//<---- type in this line and press [Enter]
2


go          //<---- type in 'go' and press [Enter]
5
6
```

SheerPower detects and announces exceptions and build errors. Sometimes errors occur which do not prevent execution, but do cause a program to execute incorrectly. SheerPower provides a high-level DEBUG system for detecting these more subtle errors.

DEBUG ON enables SheerPower's Debug System. DEBUG OFF disables the system.

The related function for the SheerPower Debug System is _DEBUG. See Section 6.8.1 for information on the _DEBUG system function.

Some DEBUG features automatically switch DEBUG ON or OFF when they are executed. Others require that DEBUG be enabled. (See DEBUG ON.)

Here is a list of SheerPower's DEBUG System features that require DEBUG to be enabled:

- TRACE ON and OFF
- STATS ON and OFF
- LIST STATS
- BREAK
- STEP

*Unlike most languages, SheerPower's debugging environment does not noticeably slow down program execution.

DEBUG ON/OFF

FORMAT:

```
DEBUG ON
```

The following example shows how the **DEBUG ON** statement is used inside an .SPSRC program to enable SheerPower's debug facility.

EXAMPLE:

```
debug on
print '1',
print '2',
break
print '3',
print '4',
print '5'
end


1                    2                   break at main.0003
```

PURPOSE:

DEBUG ON is used to enable SheerPower's debug system. SheerPower's debug system helps locate problems in the way a program runs.

DEBUG must be enabled in order to use its features. When DEBUG is enabled, all of SheerPower's debug features are available.

DESCRIPTION:

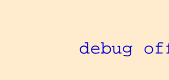
DEBUG ON can be issued in immediate mode or as a statement in a program.

If DEBUG ON is used as a statement in a program, SheerPower enables DEBUG when it encounters the statement. DEBUG remains enabled until a DEBUG OFF command or statement is executed or until a DEBUG feature is executed which disables it.

FORMAT:

```
DEBUG OFF
```

EXAMPLE:



```
debug off
```

PURPOSE:

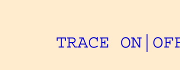
DEBUG OFF is used to disable SheerPower's DEBUG system. Set DEBUG OFF when you have finished correcting your program.

DESCRIPTION:

DEBUG OFF can be issued in immediate mode or as a statement in a program. If DEBUG OFF is used in a program, SheerPower disables DEBUG when it encounters the DEBUG OFF statement.

DEBUG will remain disabled until a DEBUG ON statement is executed, or until a DEBUG feature is executed which enables it.

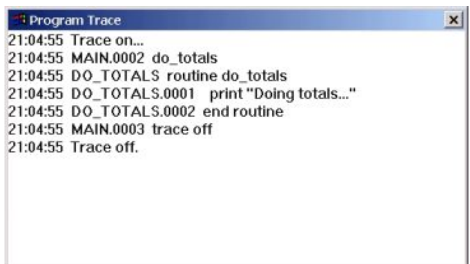
FORMAT:



```
TRACE ON|OFF
```

EXAMPLE:

```
debug on
trace on
do_totals
trace off
stop

routine do_totals
  print "Doing totals..."
end routine


Doing totals...
```



PURPOSE:

In complex applications, there is often a need to follow an application's program flow in order to figure out how the application works. This is also useful when debugging logic errors in an application.

DESCRIPTION:

TRACING is used to follow an application's logic flow. As each statement is executed, the trace window will display the label and line number and source line being executed.

To turn off tracing, just close the trace window. TRACE OFF will stop tracing, but leave the trace window open.

You can copy select or all of the text inside the trace window by highlighting the text with your mouse (CTRL/A will select all) and using CTRL/C to copy it.

In the console window toolbar, you can use the Trace icon to toggle the trace feature on or off.

The STATISTICS Features

STATISTICS records information on a program's execution. It records the time each program line takes to execute and the number of times the line is executed. The word "STATISTICS" can be abbreviated to "STATS" (STATS ON, STATS OFF, LIST STATS). This abbreviation will be used frequently in this Guide.

FORMAT:

EXAMPLE:

```
  debug on
  stats on
  dim name$(5)
  for i = 1 to 5
    input 'Please enter your name': name$(i)
    if  _exit  then exit for
    print 'Hello, '; name$(i); '!'
  next i
  end


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

PURPOSE:

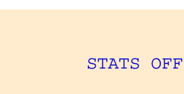
STATS ON is used to turn on the statistics feature, which stores the execution time and count for each program line. Use STATS to tell if statements are being executed the correct number of times and which parts of a program are taking the most time. STATS is especially useful for speeding up a program's execution time.

DESCRIPTION:

STATS ON enables SheerPower's statistics feature. SheerPower begins recording statistics when program execution begins. The statistics feature remains enabled until the STATS OFF statement is executed.

STATS ON can be executed in immediate mode or in a program. If STATS ON is executed in immediate mode, DEBUG is automatically switched on. If STATS ON is executed in a program, and DEBUG is off, SheerPower ignores the statement. When STATS ON is executed, any statistics previously recorded are lost.

FORMAT:



EXAMPLE:

```
stats off
```

PURPOSE:

STATS OFF is used to turn off the statistics feature.

DESCRIPTION:

STATS OFF turns off SheerPower's statistics feature. STATS OFF can be executed in immediate mode or in a program. If STATS OFF is executed in a program and DEBUG is off, SheerPower ignores the statement. STATS OFF leaves DEBUG on.

FORMAT:

```
LIST STATS [: routine_name, routine_name ,....]
```

EXAMPLE:

```
  debug on
  stats on
  dim name$(5)
  for i = 1 to 5
    input 'Please enter your name': name$(i)
    if  _exit  then exit for
    print 'Hello, '; name$(i); '!'
  next i
  end


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

Once you have run the program with the STATS ON, you can use the LIST STATS feature. Type in LIST STATS at the prompt, and the console window will display the file name, the date and time of day, and the statistics for each line in the program.

```
list stats
c:\sheerpower\list_stats.spsrc  04-SEP-2003 12:46
                          debug on
1        0.00      stats on
1        0.00      dim name$(5)
1        0.00      for i = 1 to 5
5        20.55       input 'Please enter your name': name$(i)
5        0.00        if  _exit  then exit for
4        0.00        print 'Hello, '; name$(i); '!'
4        0.00      next i
1        0.00      end


    stats off  <--- type in to turn off STATS
    debug off  <--- type in to turn off DEBUG
```

PURPOSE:

LIST STATS is used to display the statistics recorded by the STATISTICS feature.

DESCRIPTION:

LIST STATS lists each program line along with the number of times the line was executed and the execution time of each line.

The far left column lists the number of times each statement was executed. The next column gives the time each statement took to execute. The time is given in seconds and fractions of a second. (0.01 means the program line was executed in one-one hundredth of a second.) The last column lists the program itself. STATS must be ON for LIST STATS to be executed.

All the options available with the "LIST" statement are also available with LIST STATS. (See Section 2.4.3, LIST for more information.)

FORMAT:

EXAMPLE:

```
debug on
print '1',
print '2',
break
print '3',
print '4',
print '5'
end


1                    2                   break at main.0003
```

PURPOSE:

BREAK is used to stop program execution when DEBUG is ON. For instance, you might use BREAK to stop the program if a variable is assigned a wrong value.

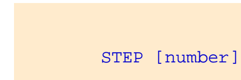
DESCRIPTION:

The BREAK statement can be used anywhere in a program. The BREAK statement will not take effect unless DEBUG is turned on. If DEBUG is off, SheerPower ignores any BREAK statements.

The HALT statement works the same way as the BREAK statement (see Section 2.4.4, HALT Statement) except that it **always** interrupts program execution.

When SheerPower executes a BREAK statement, it interrupts program execution and prints a BREAK message. The BREAK message tells what line the break occurred in. Program execution can be continued with the GO or STEP commands.

FORMAT:



EXAMPLE:

```
run
1                    2                    break at main.0003

step 2
3                    4
```

PURPOSE:

STEP is used to execute a specific number of program statements and then stop execution. That way, you can "step through" your program to find bugs.

DESCRIPTION:

STEP is used to step through a program---to execute a specified number of program statements. DEBUG must be ON for the STEP command to take effect. If DEBUG is not on and the STEP command is given, SheerPower ignores it. STEP must be given as a command. When the STEP command has been executed, SheerPower issues a BREAK and prints the break message. Issuing the STEP command without a number causes SheerPower to execute one program line.

Issuing the STEP command with a number causes SheerPower to execute the number of program lines specified. SheerPower begins executing program lines from the last line executed. It stops when the number of lines specified have been executed or when program execution ends.

You must start a program with RUN before you can use the STEP command.

In the console window the mouse can be used to select (highlight) text on the screen.

Below is a table containing the special keystrokes available when working in the console window.

Table 2-1 SP4GL Console Window Keystrokes

Keystroke	Function Performed
ctrl/a	selects all text (both on and off screen)
alt/b	causes program execution to HALT
ctrl/c	places selected text into the clipboard
ctrl/m	places contents of message history into the clipboard
ctrl/t	places contents of screen (including scroll buffers) into the clipboard

---

This chapter describes the basic elements that make up a SheerPower program. It also describes the types of data used with SheerPower programs and how the data is processed.

SheerPower programs have a default extension of **.SPSRC**. It is recommended that you use this extension for all of your SheerPower programs. SheerPower source programs are saved as **text files**. You can edit SheerPower source programs with any text editor.

A program is a series of instructions. These instructions describe how to manipulate data to produce a desired result. You determine what data the program will manipulate, how the program will manipulate it, and what the results of these manipulations will be.

When you create a program, you must use instructions that SheerPower understands. The SheerPower language provides these instructions. The language consists of statements. These statements are something like the words in this manual. You put them together in a meaningful order and SheerPower executes the program you write. Here is an example of a SheerPower program:

```
  input 'Please enter your name': name$
  print 'Hello, '; name$
  print 'Today is '; day$
  print name$; ', have a good '; day$
  end
```

The **INPUT** statement tells SheerPower to ask for a name. The **PRINT** statements tell SheerPower to print the information. **END** tells SheerPower it has reached the physical end of the program.

FORMAT:

```
PROGRAM prog_name
```

EXAMPLE:

```
program display_name
input 'Please enter your name': name$
print 'Hello, '; name$
end
```

PURPOSE:

The **PROGRAM** statement is used to name your program.

DESCRIPTION:

PROGRAM is used to name programs. *prog_name* is the program name. The program name must meet the following specifications for variable names:

- start with a letter
- can consist of letters, numbers, and/or underscore characters '_'
- can be up to, but not longer than, 39 characters

With SheerPower, when you execute this program, it will look like this:

```
Please enter your name? Tester  <---type your name here then press [Enter]
Hello, Tester
```

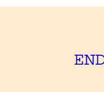
When you run the next program example, you will notice that the day is not asked for. **DAY$** is a reserved word that SheerPower uses for storing the day. There are several other reserved words that SheerPower uses. You can refer to Appendix B, Reserved Words to see a complete list of the reserved words.

```
input 'Please enter your name': name$
print 'Hello, '; name$
print 'Today is '; day$
print name$; ', have a good '; day$
end


Please enter your name? Julian
Hello, Julian
Today is Tuesday
Julian, have a good Tuesday
```

SheerPower programs are modular in structure. Every program can be divided into program units. The main unit is the main body of the program. This unit begins with the first program line and ends with the END statement.

```
first program line --   input 'Please enter your name': name$
                        print 'Hello, '; name$
                        print 'Today is '; day$
                        print name$; ', have a good '; day$
     end statement --   end
```

FORMAT:



```
END
```

EXAMPLE:

```
  input 'Please enter your name': name$
  print 'Hello, '; name$
  end

Please enter your name? John
Hello, John
```

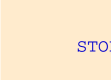
PURPOSE:

The **END** statement is used to mark the physical end of a program. It should be the last line of your program.

DESCRIPTION:

The **END** statement marks the end of a program. When SheerPower executes the END statement, it writes all active output buffers and closes all files in the current program.

FORMAT:



```
STOP
```

EXAMPLE:

```
  input 'Please enter your name': name$
  input 'How old are you': age
  if  age < 1  then
    print 'Not a valid age'
    stop
  end if
  print name$; ' is'; age
  end

Please enter your name? Ted
How old are you? .5
Not a valid age
```

PURPOSE:

STOP is used to terminate program execution where you do not want to mark the physical end of your program.

DESCRIPTION:

STOP behaves exactly as the END statement does. However, STOP does not mark the end of a program.

The STOP statement does not have to be the last physical line in a program. If there are subprograms, functions, etc., they can physically follow the STOP statement.

The STOP statement:

- closes all files
- closes all structures
- ends program execution

FORMAT:

```
        [PRIVATE] ROUTINE routine_name [: private varname, ...]
           ---
           --- block of code
           --- [REPEAT ROUTINE]
           --- [EXIT ROUTINE]
        END ROUTINE
```

EXAMPLE:

```
  get_username
  stop

  routine get_username
    input prompt 'Username: ': uname$
    if  _back or _exit  then exit routine
  end routine


Username: Tester
```

PURPOSE:

ROUTINES are block-structured subroutines. They provide a convenient way to name a block of code. By default, all variables in the main program are available inside the routine. To specify private variables that are not part of the main program use the **PRIVATE** option. To cause all variables inside of a routine to be treated as private, make the routine a **PRIVATE ROUTINE**.

Note

See Appendix M, SheerPower and Program Segmentation for more on routines and private routines in SheerPower.

It is a proven fact that short, simple "code chunks" are easiest to maintain -- and that 80% of software costs are in the maintenance of existing software. So, an excellent focus is to write short, simple "code chunks"---small, easy-to-maintain pieces of code.

SheerPower 4GL has powerful features that make it easy to write short, simple "code chunks":

- routines -- All variables in the main code are available from within the routine. This form of ROUTINE is designed to make it quick and easy to segment long pieces of source code into smaller, easy to maintain, "code chunks".
- private routines -- All variables are private to the routine. Main code variables are accessable by prefixing the variable with MAIN$---for example: "main$xyz" references the main code variable called "xyz". This form of PRIVATE ROUTINE is designed to make it easy to write reusable routines to be included into any number of applications.

Each variable in a program belongs to a "namespace". By default, they belong to a "namespace" called MAIN. So:

```
  abc = 123
  print abc
```

is the same as:

```
abc = 123
print main$abc
```

is the same as:

```
main$abc = 123
print abc
```

SheerPower 4GL supports both ROUTINES--- that use the MAIN "namespace" and PRIVATE ROUTINES---that have their own "namespace". For example:

```
routine show_display
  abc = 123
  print main$abc
end routine
```

In this ROUTINE, the variable "abc" belongs to the "namespace" of MAIN---sharing its variable names with the main program.

However in this PRIVATE ROUTINE:

```
private routine do_totals
  abc = 123
  print abc
end routine
```

the variable "abc" belongs to the "namespace" of "do_totals":

```
private routine do_totals
  abc = 123
  print do_totals$abc
end routine
```

Now, lets look at a more complex example:

```
abc = 123
do_totals
stop

private routine do_totals
  abc = 999
  print 'The DO_TOTALS version: '; abc
  print 'The MAIN version     : '; main$abc
end routine

end
```

Unlike the default ROUTINE feature in SheePower, that is used to segment program code into small, simple, easy-to-maintain chunks of code---The PRIVATE ROUTINE feature is used primarily to write reusable routines that will be used in a number of different applications. Here is a simple PRIVATE ROUTINE that is used to write messages to a message file:

```
private routine write_message with msg
  if msg_ch = 0 then open file msg_ch: name 'message.log', access output
  print #msg_ch: time$; ' '; msg
end routine
```

WRITE_MESSAGE has a single named parameter called "msg". The first time WRITE_MESSAGE is called, msg_ch will be zero, so a new message.log file is created and msg_ch receives the channel#. Then WRITE_MESSAGE writes out the current time and the message.

This PRIVATE ROUTINE can be called from any application, without worrying about variable name conflicts.

```
write_message with msg "this is a test"
```

The PRIVATE routine feature of SheerPower 4GL is designed to assist in writing routines that will be used in a number of different programs. The routines can be written without having to be concerned with accidental variable name conflicts---because all variable names in PRIVATE routines have their own, private, "namespace".

Routines can be edited as a complete unit.

To execute a ROUTINE, name the routine or use the **DISPATCH** statement. For more information on the dispatch statement see Section 10.9, DISPATCH.

To pass data in and out of routines, see Section 3.5, Passing Optional Parameters to Routines.

See Appendix M, SheerPower and Program Segmentation for more on routines in SheerPower.

DESCRIPTION:

ROUTINES are used to simplify programming. Routines simplify programs by breaking them up into small manageable pieces. Routines can be used to organize the individual thoughts and concepts in the program. The smaller the routines, the more successful you will be in writing code.

Each routine should be under 25 lines in length. If the number of lines exceeds 25, it is an indication that the routine is becoming too complex. Please see Appendix A, Coding Principles and Standards for more information.

Routine names must contain at least one underscore '_' (i.e., print_statistics).

```
request_info                // call request_info routine from main logic section
stop

routine request_info
  print 'Please enter in the required informaton'
  print
  get_username                // call get_username subroutine
end routine

routine get_username
  input prompt 'Username: ': uname$
  if  _back or _exit  then exit routine
end routine


Username: Tester    <---- Type in your name and press [Enter]
```

FORMAT:

```
EXIT ROUTINE
```

EXAMPLE:

```
  get_username
  end

  routine get_username
    input prompt 'Username: ': uname$
    if  _back or _exit  then exit routine
    if  uname$ = ''  then repeat routine
  end routine


Username: exit       <---- type in 'exit'
```

PURPOSE:

The **EXIT ROUTINE** statement enables you to exit from the current routine.

DESCRIPTION:

Use **EXIT ROUTINE** statement when you need to exit a routine early. For example,before you reach the end statement.

FORMAT:

```
        REPEAT ROUTINE
```

EXAMPLE:

```
  get_username
  end

  routine get_username
    input prompt 'Username: ': uname$
    if  _back or _exit  then exit routine
    if  uname$ = ''  then repeat routine
  end routine


Username:           <---- press the [Enter]
Username: Sunny     <---- type in your name or 'exit'
```

PURPOSE:

The **REPEAT ROUTINE** statement enables you to repeat execution of the current routine.

DESCRIPTION:

When SheerPower executes the **REPEAT ROUTINE** statement, control is passed to the first statement following the routine statement.

FORMAT:

```
ROUTINE routine_name [WITH input_param value [, input_param value, ...]]
  [, RETURNING output_param varname [, output_param varname,...]]
  [, PRIVATE varname,...]
```

EXAMPLE:

```
do_a_heading with option 45, title 'Big test',
                returning status s
print 'Status was: '; s
stop
routine do_a_heading with title, option, returning status
  print '** '; title; ' **... option:'; option
  status = -1
end routine
```

PURPOSE:

The purpose of the optional parameters is to pass data in and out of routines.

DESCRIPTION:

The parameter NAMES are used to pass data into and out of a routine. You can have up to 16 INPUT (with) and 16 OUTPUT (returning) parameters (32 total).

Currently the RETURNING parameters must all be non-array elements.

FORMAT:

```
    ROUTINE routine_name: PRIVATE var, var, var, ...
or
    ROUTINE routine_name: PRIVATE INTEGER var, STRING var, STRING var, ...
```

EXAMPLE:

```
do_totals
end

routine do_totals: private mytotal, desc$
  mytotal = 15
  desc$ = 'Test Totals'
  print desc$; mytotal
end routine


Test Totals 15
```

DESCRIPTION

SheerPower allows you to use **private variables** in a routine. Private variables are variables identified with a specific routine. This option allows you to use the same variable names more than once because, internally, SheerPower prefixes the variables with the routine name and a "$" character.

In the above example, the private variables "mytotal" and "desc$" are internally known to SheerPower as:

do_totals$mytotal

do_totals$desc$

From *inside* the routine, you can reference the variables by their private names. For example: mytotal, desc$

From *outside* the routine (or while debugging the code), you can reference the private variables by their internal known names. For example: do_totals$mytotal, do_totals$desc$

Note

See Appendix M, SheerPower and Program Segmentation for more on routines and private routines in SheerPower.

FORMAT:

```
ROUTINE routine_name [WITH input_param value [, input_param value, ...]]
  [, RETURNING output_param varname [, output_param varname,...]]
  [, PRIVATE varname,...]
```

EXAMPLE:

```
  do_a_heading with option 45, title 'Big test',
                returning status s
  print 'Status was: '; s
  stop
  routine do_a_heading with title, option, returning status, private tlen
    tlen = len(title)
    print '** '; title; ' **... option:'; option; ', len: '; tlen
    status = -1
  end routine


** Big test **... option: 45 , len: 8
Status was: -1
```

If you try to access a parameter incorrectly an error message will result:

```
  do_totals with tttitle "my title"
  stop

  routine do_totals with title
    print title
  end routine
```

The above example would generate TWO errors:

```
Possible coding error - variables used, but not assigned:
    DO_TOTALS$TITLE
(no data value got into the TITLE parameter)

 Inconsistant ROUTINE parameter passing:
    Routine DO_TOTALS, Parameter TTTITLE
(This isn't the right spelling of the TITLE parameter)
```

Each program unit is made up of **program lines**. A program line consists of one or more statements.

When SheerPower executes a program, it starts at the first line and executes all the statements in that program line. Then it goes to the next line and executes all the statements in that line and so on.

Multiple statements can be placed on a single line, delimited by the \ (backslash). CHR$(10) also signals the same as a \. A \ terminating the line implies an "&".

```
  for z1 = 4. to 7. step .5 \ print z1 \ next z1
  end


run
4
4.5
5
5.5
6
6.5
7
```

Note

Putting multiple statements on a single line is not recommended. It makes changing code difficult and the logic flow is not easy to follow.

You can continue a statement on the next physical line by using an **ampersand** (&). This allows you to continue a statement that is too long to fit on one line.

To continue a statement, place the ampersand at the end of the continued line. You can also place an optional ampersand at the beginning of the next line. For example:

```
  input 'Please enter your name': name$
  print 'Hello, '; name$
  print 'Today is '; day$
  print name$; &
      ', have a good '; &      // required ampersand
      & day$                   // optional ampersand
  end


Please enter your name? Julian
Hello, Julian
Today is Tuesday
Julian, have a good Tuesday
```

Note

In the above example, the comment text follows the line continuation character (&). Whenever you comment a continued line, the (//) double forward slash *must* come AFTER the ampersand.

You can continue statements between words; however, you cannot break up a word.

An ampersand (&) may be used to identify a statement that is continued on the next physical line. The ampersand is acceptable, but is no longer required in some cases. *Implied continuation* has been implemented for:

- comma (,) list items
- all operators having two arguments; i.e. trailing "+", "AND", "OR", "*", "/", etc., implies that the rest of the code is on the following line

Note

Comma-separated list continuation does *not* work in a PRINT statement. PRINT statements still require the trailing "&" for continuation.

```
  open structure cust: name 'sptools:customer',     // no ampersand
    access outin,                                  // no ampersand
    lock
  close structure cust
  end
```

```
  print 'Today is the first day of the ' +    // no ampersand
      'rest of your life.'
  a$ = 'walk'
  b$ = 'walk'
  c$ = 'walk'
  if a$ = b$ and                              // no ampersand
    a$ = c$  then
    print 'All are the same.'
  end if
  end


Today is the first day of the rest of your life.
All are the same.
```

Inserting **COMMENTS** in programs will allow the code to be easily understood in the future. Comments are not executable statements. They are simply included in source code for informational purposes. They are seen when a program is listed or printed out. However, SheerPower will ignore them when it executes a program.

There are two types of comments allowed in SheerPower: **exclamation point (!)** and **double-forward slash (//)**. The exclamation point commenting is used for creating the program and routine headers. The double-forward slash commenting is used for inserting comments *within* routines.

```
  dim name$(10)                               // setup array
  rem main logic
  for i = 1 to 10                             // Begin the loop
    input 'Please enter your name': name$(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    <---- type in 'exit'
```

FORMAT:

```
        // comment_text
```

EXAMPLE:

```
  input 'Please enter your name': name$   // ask for a name
  print 'Hello, '; name$                  // say hello
  end


Please enter your name? Mike
Hello, Mike
```

PURPOSE:

All programs should be commented. **Comments** teach future programmers, as well as yourself, how a program works. The proper use of comments can dramatically enhance the ability to maintain a program.

Comments are used to put headers in routines, and to comment code within routines. Always document code thoroughly in each routine header. Ideally, commenting should rarely be done inside the actual routine. If the routine is written simply and kept short, then no comments are needed within the code. Keep the size and scope of the routine limited and obvious for future reference.

DESCRIPTION:

Comments can be used to clarify parts of your program as shown in the example above.

When the program is listed or printed, the commented line is displayed as it was written in the source code.

When SheerPower executes the above program, it executes the INPUT statement, ignores the "//" and the comment text following it, and continues execution at the PRINT statement. The "//" does not have to be placed at the beginning of a physical line. It can be used anywhere on a program line.

In the SheerPower Rapid Development Environment, the GOLD/p keystroke will automatically create the program header template. The GOLD/r keystroke will automatically create the routine header template. The 'GOLD' keys in SPDEV are the `[Esc]` (Escape) key or the `[NumLock]` (Numbers Lock) key. For more special keystrokes in SPDEV, refer to Appendix F, Keystrokes for SheerPower Rapid Development Environment.

Below is an example of a **Routine Header**:

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! f i n d _ p e a k _ c a m s _ a n d _ v i e w e r s
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!
! Brief description:
!   This routine finds the peak number of cams logged in and peak
!   number of live viewers watching each day.
!
! Expected on entry:
!   total_live_viewers = number of live viewers
!   live_cameras       = number of live cams
!
! Locals used:
!
!
!
! Results on exit:
!   peak_viewers       = peak number of live viewers
!   peak_cameras       = peak number of live cameras
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The **double-forward slash** can be used after an ampersand to document continued lines. When a line is continued with an ampersand, any comments must **follow** the ampersand. For example:

```
  input a$
  if  a$ = ''  then  print a$; &      // here is the trailing
      ' is OK.'                       // comment text
  end
```

Code that is inserted into a program for the sole purpose of debugging the program should be marked with a **debug comment**. SheerPower can automatically create a debug comment line for you with a special keystroke.

The **GOLD/C** keystroke is mapped to create a debug comment line. In SPDEV, a **GOLD** key is a special key that allows you to utilize other keys for different purposes. The GOLD key is either the `[Esc]` (Escape) key or the `[NumLock]` (Numbers Lock) key. See Appendix F, Keystrokes for SheerPower Rapid Development Environment for more special SheerPower Rapid Development Environment keystrokes.

To perform a GOLD/C keystroke, place the cursor where you want the debug comment line created. Press once on the `[Esc]` key or the `[NumLock]` key. Let go and look in the bottom right hand corner of the SPDEV window. SheerPower tells you if your GOLD key is activated by highlighting a small square in the bottom frame in black, with gold letters that say 'Gol'. Now you can press the `[C]` key, and a small dialog box will appear asking you for your initials. Leaving your initials will allow future programmers (as well as yourself) to know who inserted this line of debug code. Enter **your initials**, and click on '**OK**'.

```
!++ debug sw June 08, 2001
```

The '**!**++' at the beginning of every line of debug comment makes it very easy to perform a search and find all lines containing debug code.

The following sections describe the directives that are available for use in your programs. These directives are invoked when the compiler builds a program and/or when a program is compiled.

FORMAT:

```
        %COMPILE 'quoted_text'
```

EXAMPLE:

```
  print 'This is a test program.'
  %compile 'Text that you want to be seen inside the SPRUN file...'
  %compile 'Up to 100 lines of text can be seen here!'
  end
```

DESCRIPTION:

Using the DEPLOY feature you can make *portable runnable* applications (**.SPRUN** files). These files are plain text, and can be password protected.

SPRUN files allow you to distribute SheerPower programs without your source code being seen.

An **SPRUN** file can be easily copy/pasted, emailed or embedded into a website. This allows for simple distribution of an application.

The %compile text is truncated at 72 characters. There can be up to 100 %compile text lines in a program.

Once the text is inside the SPRUN file, if it is altered in any way the SPRUN file will not run! This is a good place to put copyright notices and license agreements, etc.

FORMAT:

```
        %MESSAGE 'quoted_text'
```

EXAMPLE:

```
  %message 'Including HELP module'
  %include 'sptools:help'
  end
```

DESCRIPTION:

This compiler directive displays a message, *quoted_text*, on the message line. The message is displayed when a program is built into a workspace or compiled.

FORMAT:

```
        %MESSAGE ERROR: 'quoted_text'
```

EXAMPLE:

```
%message error: 'Using experimental HELP module'
%include 'sptools:help'
end
```

DESCRIPTION:

This compiler directive makes an alert sound and displays the message when a program is compiled.

FORMAT:

```
        %INCLUDE 'file_spec'
```

EXAMPLE:

```
  %include 'sptools:example'
```

PURPOSE:

%INCLUDE allows you to put common subroutines into a separate file to be shared among applications.

DESCRIPTION:

%INCLUDE includes a source code into the current SheerPower program. The default extension for the included file is **.SPINC**.

FORMAT:

```
        %include conditional: 'file_spec'
```

DESCRIPTION:

If the file to be included does not exist, no error is generated. This allows programs to conditionally include modules.

%INCLUDE CONDITIONAL includes a source code file into the current SheerPower program if the file to be included is found. The default extension for the included file is **.SPINC**.

Programs manipulate data. The data can be numeric or textual, but the data must be presented according to the rules of the SheerPower language.

SheerPower accepts three basic types of data:

1. integer numbers
2. real numbers
3. string (textual) data

The following sections describe each of the three types of data.

An **integer** number is a whole number with no fractional part. The following are examples of integers:

```
        +5%       894%       -369004%
```

Integers can be positive or negative. Integer constants can end with a trailing percent sign (%). About integers:

- Names for integer variables, arrays or functions must be designated by a trailing percent sign (%) or declared with a DECLARE statement.
- A negative integer constant is designated by a leading minus sign.
- A positive integer constant can be designated by a leading plus sign or by the absence of a sign.
- Integers have a size of 32 bits.
- SheerPower allows integer numbers anywhere in the range of -2147483648 to +2147483647.

A **real** number can have a fractional part. The following are real constants:

```
        5.4             894.0              -369004
```

Real numbers can be positive or negative. Any number that is not designated as an integer or a string is treated as a real number.

A real number without a decimal point is called an **ambiguous constant**. About real numbers:

- A variable, array or function whose name has no trailing % or $ is treated as a real number unless it is declared.
- A negative real constant has a leading minus sign.
- A positive real constant can have a leading plus sign or no sign.
- Ambiguous constants are treated as real numbers with a fractional part of 0 (5 = 5.0).
- Real numbers can have 18 digits to the LEFT of the decimal point, and 16 digits to the right. The largest number is: 999999999999999999.9999999999999999
- SheerPower REAL NUMBERS use a patent-pending "exact math" internal representation that avoids the "penny rounding" problems found in traditional programming languages.

Here is an example of SheerPower exact math:

```
  x = 0
  for i = 1 to 10000
    x = x + .01
  next i
  if  x = 100.00  then print 'It is right.'


It is right.
```

If you try this example in another programming language (Visual Basic or C++ for example), you will *not* get the correct answer. Here is another example:

```
  print 123456789.012345 * 87654321.123456789


10821521028958940.344459595060205
```

String data consists of text. Text can consist of any characters. All of the following are valid string constants:

```
'Hello, Fred.'

'Account number (##-####)'

'65'
```

About string data:

- String constants must be enclosed in single or double matching quotes. String variables must be designated by a trailing dollar sign ($) or declared with a DECLARE statement.
- The maximum size of a SheerPower 4GL string is 64 MB, or 67,108,864 bytes.
- String variables are dynamic in length.

Boolean variables represent either a TRUE or FALSE condition. These variables can be expressed with a trailing "?". For example:

```
  done? = FALSE
  do
    input 'Ready': ans$
    if  _exit  then done? = true
    if  _back  then done? = true
  loop until done?
  end


Ready? exit
```

Expressions can be:

- constants
- simple variables
- array elements, which are a type of variable
- structure references, which are a feature of the SheerPower data structure system
- substrings
- compound expressions, which are any combination of the above operands separated by operators

A **constant** is a value that does not change during a program execution. Constants can be any of the three data types: integer numeric, real numeric or string.

Integer numeric constants are written with digits, with no decimal fraction. An integer constant can end with a percent sign. Examples:

```
        235             412%
```

Real numeric constants are written with a sign, digits, and a decimal point. For example, 5.38 is a real numeric constant.

String constants must be enclosed in quotes. The quotes are called **string delimiters**. They tell SheerPower where the string begins and ends. The quotes are not considered part of the string. Everything within the quotes *is* considered part of the string. Note that the following strings are different:

```
  print 'Account number (##-####)'
  print '    Account number     (##-####)'
  end


Account number (##-####)
    Account number     (##-####)
```

An empty string is called a **null string**. A null string is indicated by a pair of quotes with nothing between them ("").

String delimiters can be single quotes (') or double quotes ("). However, the quotes must match and be paired.

Quotes can be included as part of a string by using them within the string delimiters. For example:

```
"Message'Time to go home!'"
```

The string part of this is:

```
 Message 'Time to go home!'
```

The delimiter quotes can be used within a string by repeating them twice. For instance:

```
"Message ""Time to go home!"""
```

The string part of this is:

```
 Message "Time to go home!"
```

Variables are a way of storing values that can change in the program during execution. A variable names a storage location. The *value* of the variable is stored in this location. Here are two types of variables:

- simple variables such as: AVERAGE and NAME$
- array elements such as: AVERAGE(3) and NAME$(2,12)

Two other constructs are used in ways similar to variables---you can use them in expressions and assign values to them:

- substrings such as: NAME$[7:12]
- structure references such as: SALES(AVERAGE) and CLIENT(PHONE)

Variables are represented by a name consisting of a letter or series of letters, numbers and underscore characters. Variables:

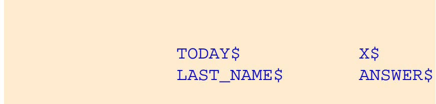
- must start with a letter
- can be followed by a proper data type designation (% for integers, $ for strings and ? for BOOLEANS)
- can be declared explicitly

Some examples of valid variables are:

```
        TODAY$            X%
        Z                 INDEX
```

- Variables can be **integer numeric, real numeric** or **string**.
- Integer variables can contain only integer values.
- If a real numeric value is assigned to an integer variable, SheerPower rounds the real number and uses the resulting integer.
- String variable names can have a trailing dollar sign ($).
- If numeric data is assigned to a string variable, an exception occurs.

The following are examples of string variables:



```
        TODAY$            X$
        LAST_NAME$        ANSWER$
```

- A variable that is not followed by a $, % or ? is treated as a real numeric variable unless it is declared.
- If string data is assigned to a real numeric variable, an error will result.
- If integer data is assigned to a real numeric variable, SheerPower treats it as a real number with a fractional part of 0.

The following are examples of real numeric variables:

```
INDEX                Z
COUNTER              AMOUNT
```

SheerPower uses the last assigned value when a variable is referenced. See Section 5.7, Private Variables in Routines, for more information on using variables.

An **array** is a variable that consists of a group of elements. Each element represents a storage location. A different value can be assigned to each element. Here is an example of an array:

```
QUANTITY(1)
QUANTITY(2)
QUANTITY(3)
QUANTITY(4)
QUANTITY(5)
```

To indicate which element to access, use numbers after the array name, called *subscripts*. The subscripts are enclosed in parentheses. Example: amount(3,2)

Subscripts must be given as integers. If a real numeric subscript is given, it will be rounded and the integer portion used. Arrays can contain real numeric, integer or string values. String and integer arrays are designated by placing the appropriate symbols after the array name and before the subscript. For example:

```
a$(5,10)        a%(5,10)
```

About arrays:

- Arrays must be defined with the DIM statement and can be redefined with the REDIM statement.
- Arrays can have up to 32 dimensions.
- An array can have as many elements as memory allows.
- Integer and string arrays must be designated by the appropriate symbol (% for integers, $ for strings) or else they must be declared.
- Subscripts must be integers. Real subscripts are rounded.
- Multiple subscripts must be separated by commas.
- The default low bound for an array is 1 or the number indicated with the OPTION BASE statement (See Section 5.8.4).
- You can indicate the low bound explicitly with a TO clause:
  DIM COSTS(5 TO 10).

Substrings are a way to refer to a part of a string. The format of a substring is:

```
str_var [begin : end]
```

str_var is the name of a string variable. *begin* is the position in the string where your substring begins. *end* is the position at which the substring ends. For example, here is a string called ALPHABET$:

```
  alphabet$ = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  print alphabet$[9:14]
  end


IJKLMN
```

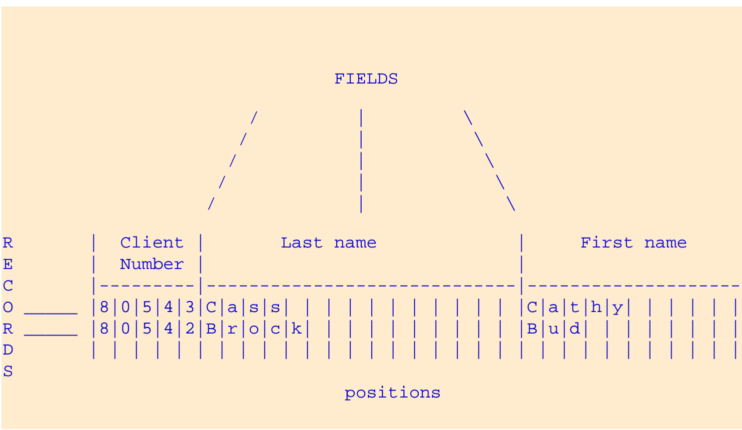
The substring ALPHABET$[9:14] tells SheerPower to begin at the ninth character and count to the 14th character. Everything between and including these two positions makes up the substring. Therefore, the substring is "IJKLMN".

begin and *end* are integers. If real numbers are given for these positions, SheerPower rounds them and uses the remaining integers. A substring can be manipulated like any other expression, and data can be stored in the substring. Substrings can be used also to change the value of a string. For example:

```
let a$ = 'Your tests are in.'
print a$
let a$[6:10] = 'results'
print a$
end


Your tests are in.
Your results are in.
```

Another type of variable is a **structure reference**. SheerPower includes a transparent interface to several record management systems, including the Windows file management system. One of the major features of SheerPower is its ability to perform database operations as a part of the language. SheerPower's data structure statements allow manipulation of stored data from within a user's programs. (See Chapter 15, Data Structure Statements for information on the SheerPower data structure statements.)

SheerPower stores data in **structures**. Structures look something like this:



Each structure is made up of **records** and **fields**. In the CLIENT structure above, we have a record for each customer.

Each record consists of fields. For example, our customer records might contain a field for the customer's ID number, last name, first name, address, phone number, company name, etc.. Each of these pieces of data is stored in its own field - the name field, address field, phone number field, etc.. These fields appear as columns in the example shown above.

Structures and fields

For information on creating structures and defining fields, see Chapter 16, Database Setup.

The field data in structures can be referenced by using structure references. To reference a field, indicate the structure name and the expression of the field whose contents you want to access:

```
struc_name(field_expr)
```

struc_name is the name associated with the structure. *field_expr* is the name of a field in the structure. When a field is referenced, SheerPower searches the current record for this field and reads its contents. Some examples of structure references are:

```
CLIENT(PHONE)          CATALOG(PART_NUM)
```

The *field_expr* can be either a string or numeric expression.

A string constant can be used to specify the field name. If the field name is given as a string constant, it need not be enclosed in quotes. SheerPower will use the string constant as the field name:

```
                  PRINT CL(LAST)
                            /
         the field is specified by its field name
```

If the field is specified as an expression, the expression will need to be preceded by a pound sign (#). The pound sign tells SheerPower that the following characters are an expression, not the field name. If the pound sign is not included, SheerPower will interpret the characters as a field name. Here are two examples:

```
                 PRINT CL(#FIELDNAME$)
                             /
        the field is specified by the variable FIELDNAME$

                 PRINT CL(#FIELDNUM)
                          /
        the field is specified by the variable FIELDNUM
```

See Section 15.8.1.1, FIELD Expressions for an example of a program that uses field expressions.

Fields with **multiple occurrences** (single dimension array) are supported.

When defining a field with multiple occurrences, the length of the field must be the length of a single occurrence.

Each occurrence of a field is accessed by including the occurrence number in the field expression. For example, to access the second occurrance of the field "address":

```
 print cust(address#2)
```

Compound expressions can be used in programs. Compound expressions consist of operators and operands. There are three types of compound expressions:

- numeric expressions
- string expressions
- conditional expressions

Numeric expressions consist of numeric (integer or real) variables, constants or expressions separated by arithmetic operators. The arithmetic operators are +, -, *, /, and ^.

```
                  Constants    Variables

      +         Add

                  4%+2%        Z + TWO16

      -         Subtract

                  4%-2%        Z - TWO16

      /         Divide

                  4%/2%        Z / TWO16

      *         Multiply

                  4%*2%        Z * TWO16

      ^         Raise to a power

                  4%^2%        Z ^ TWO16
```

Any number of these operators can be combined in an expression.

```
        4 + Z ^ TWO16                Z * TWO16 / 2
```

Generally, two arithmetic operators cannot be used next to each other. However, a + or - sign can be used to indicate a positive or negative number. For example:

```
                total * -2   =   total * (-2)
                total / +2   =   total / (+2)
```

If all the values in an arithmetic expression are of the same data type, the result of the expression will be of that data type. For example, if an expression consists only of integer numbers, it will yield an integer result. If an expression consists only of real numbers, the result will be a real number. If an expression consists of integers and real numbers, the result will be a real number. If the target of a real calculation is an integer (a% = 1.5 + 2.8), the result is rounded before it is assigned to the target.

String expressions are strings concatenated (joined). String expressions can be joined by a plus sign (+) or by an ampersand (&). SheerPower evaluates this type of string expression by concatenating the strings. For example:

```
  z$ = 'MO' + 'TH' & 'ER'
  print z$
  end


MOTHER
```

In the above example, SheerPower joins the strings separated by a plus sign and an ampersand, and assigns their value to z$. String constants, variables, functions, etc. can be included in your expressions. For example:

```
  let last$ = ' is it.'
  print 'This' + last$
  end


This is it.
```

Conditional expressions are expressions which yield a **TRUE** (1) or **FALSE** (0) value. Conditional expressions are created by using either relational or logical operators. When SheerPower evaluates a conditional expression, it returns a value of either TRUE or FALSE. If the expression is TRUE, SheerPower returns the integer 1. If the expression is FALSE, SheerPower returns the integer 0.

Relational operators are similar to those used in algebra. The relational operators are:

```
=      equals                          X=Y          X is equal to Y

<      less than                       X<Y          X is less than Y

>      greater than                    X>Y          X is greater than Y

<=     less than or equal to           X<=Y         X is less than or equal
                                                      to Y

>=     greater than or equal to        X>=Y         X is greater than or
                                                      equal to Y

<>     not equal to                    X<>Y         X is not equal to Y
```

X and Y can be any unconditional or conditional expression.

When relational operations are performed on strings, SheerPower determines which string occurs first in the ASCII collating sequence and returns **TRUE** or **FALSE**. For instance, when you perform relational operations on two strings, SheerPower checks the ASCII values for each character in each string. SheerPower compares the strings character by character using these ASCII values, and determines where there is a difference in the values.

When SheerPower finds a character that differs, it compares the two and determines which one has a smaller ASCII code number. SheerPower then returns a TRUE or FALSE value depending on the relational expression. For example:

```
  a$ = 'TEXT'
  b$ = 'TEST'
  MESSAGE$ = 'Strings are equal'
  if  a$ < b$  then message$ = a$ + ' is less than ' + b$
  if  b$ < a$  then message$ = b$ + ' is less than ' + a$
  print message$
  end


TEST is less than TEXT
```

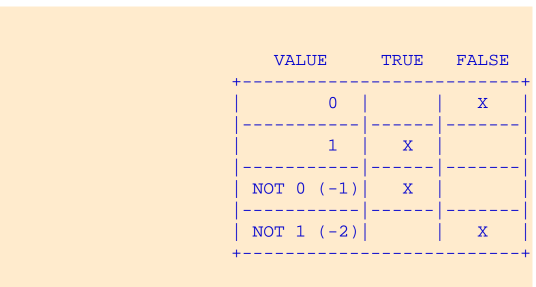
SheerPower compares the two strings. They are identical up to the third character. The ASCII value of S is 53. The ASCII value of X is 58. Therefore SheerPower prints "TEST is less than TEXT".

The logical operators are:

```
NOT     NOT X           TRUE  if X is false and
                        FALSE if X is true.

AND     X AND Y         TRUE  if X and Y are true.

OR      X OR Y          TRUE  if X or Y is true.

XOR     X XOR Y         TRUE  if X is true, or if Y is true but
                          FALSE if both X and Y are true.

EQV     X EQV Y         TRUE  if X and Y are true, or
                        TRUE  if X and Y are false,
                          but FALSE otherwise.

IMP     X IMP Y         TRUE if X is true and Y is false.
```

X and Y can be any expressions. Logical operators are usually used on integers or expressions which yield an integer result such as conditional expressions. Logical operators will always yield an integer result. If a logical operator is used on a real number, the real number is rounded and the resulting integer is used.

Logical expressions always return an integer value. If the integer value is a 1, the expression is TRUE. If the integer value is a 0, the expression is FALSE. (NOT 0 is equal to -1 and is TRUE. NOT 1 is equal to -2 and is FALSE.)



VALUE	TRUE	FALSE
0		X
1	X	
NOT 0 (-1)	X	
NOT 1 (-2)		X

- If a logical operator is used on a real expression, the expression is rounded and the resulting integer value is operated upon.
- Integers are represented as a signed 32-bit quantity.

Logical operators can be used to do **bit manipulation**. Computers represent values in a binary code, using ones and zeros. SheerPower integer values are represented as a 32-bit binary longword. A bit which is set to 1 is considered on. A bit which is set to 0 is off. The value of the word is equal to the value of all the bits which are on, added together. For example:

```
0 0 0 1 0 1 1 1  =  16 + 4 + 2 + 1 = 23
```

The last bit has a value of 1. The second to the last bit has a value of 2. The third bit has a value of 4, the fourth a value of 8, the fifth bit has a value of 16, and so on. Each bit has a value double that of the previous one:

```
         0    0    0    0    0    0    0    0
---------------------------------------------
       128   64   32   16   8    4    2    1
```

Bits can be manipulated and tested using logical operators. The logical operators work on bits. They compare each position in each word according to the particular rules of the logical operator. For instance, here is the **AND** operator used on two values:

```
  let a% = 23%        // 00010111
  let b% = 37%        // 00100101
  let c% = (a% and b%)
  print c%
  end


5
```

When SheerPower executes this program, it compares the two values. It sets a bit in the result to 1 (on) only if both the bits at a given position are on (1). The value of the resultant word is 5:

```
A%    0 0 0 1 0 1 1 1 = 23
B%    0 0 1 0 0 1 0 1 = 37
      ---------------
C%    0 0 0 0 0 1 0 1 =  5
```

When SheerPower evaluates an expression, it evaluates it in a specific order. SheerPower evaluates expressions from **left to right**.

```
1+Z+4                  equals          (1+Z)+4
1+Z-4                  equals          (1+Z)-4

3*4/QUANTITY           equals          (3*4)/QUANTITY
12/QUANTITY*3          equals          (12/QUANTITY)*3
```

The following priorities take precedence over the left to right evaluation rule:

1. SheerPower always evaluates expressions in parentheses first. Parentheses, ( ), can be used to change the order of any of the following operations. If parentheses are nested, SheerPower evaluates them from the inside out. For example:

   ```
   Z%-(X% / (Y% + AMOUNT))
   ```

   SheerPower evaluates the expression Y% + AMOUNT first. Next, it divides the X% by AMOUNT to determine that result. Finally, it subtracts the entire sum from Z%.
2. SheerPower performs functions second.
3. SheerPower performs exponentiation.
4. SheerPower performs multiplication and division.
5. SheerPower performs addition and subtraction.
6. SheerPower performs relational operations from left to right. (The relational operators are: =, <, >, <=, >= and <>.) The only exception is the assignment of the result. The result is always assigned last.
7. SheerPower performs logical operations in the following order:

   NOT
   AND

OR
XOR
IMP
EQV

Parentheses are used around every set of operations. This makes it easy to pick out the beginning and end of an operation, and will make absolutely clear what is intended without depending on the order of precedence of operators. If someone reading the code can know with absolute certainty how it was intended to be executed via the use of parentheses, then they will not have to wonder if a particular bug is due to the language used creating the code, other than what was intended. See Appendix A, Coding Principles and Standards for more on Coding Principles and Standards in SheerPower.

```
if  a > b * c or d + 4 = f  then x = x + 1           <--- hard to read!

if  ((a>(b * c)) or ((d + 4) = f))  then  x = x + 1  <--- clarity with parenthesis
```

Variables specify storage locations. Numeric and string variables are assigned values with the LET statement. String variables can also be assigned values with the LSET, RSET and CSET statements.

The LET statement assigns the value of an expression to a variable. The expression is evaluated and its value is stored in the location specified by the variable. The data types of the expression and the variable must match. Thus, you must assign a string variable string values and you must assign numeric variables numeric values. A string variable must end with a dollar sign "$" unless you declare it. An integer variable must end with a percent sign (%) unless you declare it.

The DECLARE statement allows you to dispense with data type designations. DECLARE declares a variable as either string, integer or real. Once the variable has been declared, you do not need to attach a $ or % to it. Functions, arrays, etc., can also be declared with the DECLARE statement.

FORMAT:

```
        DECLARE [STRING | INTEGER | REAL | BOOLEAN | OBJECT] var, var...
```

EXAMPLE:

```
  declare string name, sex
  declare integer age
  declare real amount
  declare object anything
  declare boolean is_male
  input 'Enter your name': name
  input 'Enter your age': age
  input 'Enter your sex': sex
  input 'Enter an amount': amount
  if  sex = 'male'  then is_male = true  else  is_male = false
  print
  print name; ' is a'; age; 'year old '; sex
  print name; ' entered the amount'; amount
  if  is_male  then print name; ' is male'
  anything = 4 + 5
  print 'This object (or dynamic) variable contains a numeric value: '; anything
  anything = 'kitty'
  print 'Now it contains a string value: '; anything
  end


Enter your name? Sammy
Enter your age? 28
Enter your sex? male
Enter an amount? 25.38

Sammy is a 28 year old male
Sammy entered the amount 25.38
Sammy is male
This object (or dynamic) variable contains:  9
```

```
Now it contains a string value: kitty
```

PURPOSE:

DECLARE is used to specify the data types of variables, functions, etc. Once the data type of a variable or function has been declared, it is not necessary to designate them with a trailing $ or %.

DESCRIPTION:

DECLARE declares the data type of a variable or function. The **STRING** option indicates that the following are string variables or functions. **INTEGER** declares integer numeric. **REAL** indicates real numeric. The **BOOLEAN** option indicates that the following are Boolean variables. Only one of the four data type options can be used in each DECLARE statement. Any number of variables can be declared with a DECLARE statement.

DECLARE OBJECT declares one or more variables to be of type OBJECT. A variable of type OBJECT receives the data type of the data that is put into it.

Note

OBJECT and **DYNAMIC** are synonyms in this statement. DECLARE DYNAMIC is the same as DECLARE OBJECT.

To find out the current data type of an object variable, the DTYPE function can be used (see Section 6.9.2).

Multiple data types can be declared by using the following format:

```
        DECLARE datatype var, var, datatype var, var, datatype var, var, ...
```

For example:

```
  declare string name, sex, integer age, year, real amount
  input 'Enter your name': name
  input 'Enter your age': age
  input 'Enter your sex': sex
  input 'Enter the year': year
  input 'Enter an amount': amount
  print
  print name; ' is a'; age; 'year old '; sex; ' in'; year
  print name; ' entered the amount'; amount
  end


Enter your name? Terry
Enter your age? 25
Enter your sex? female
Enter the year? 2000
Enter an amount? 582.69

Terry is a 25 year old female in 2000
Terry entered the amount 582.69
```

FORMAT:

```
        DECLARE STRUCTURE struc_name1 [, struc_name2 ...]
```

EXAMPLE:

```
declare structure str
open structure cl: name 'sptools:client'
ask structure cl: id cl_id$
set structure str: id cl_id$
extract structure str
end extract
for each str
  print str(#1); ' '; str(#2)
next str
end


20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

DECLARE STRUCTURE declares one or more symbols to be of type **STRUCTURE**. Once a symbol has been declared to be of type STRUCTURE, it can be used in statements such as SET STRUCTURE...ID to write generalized routines where you do not know at compile time which structure you are going to use.

Usage example: this statement could be used in the situation where you have a transaction structure and a transaction history structure and, optionally, want a report on one or the other. You could use one report program and the DECLARE STRUCTURE statement to declare which structure to use when the user makes the report selection.

FORMAT:

```
OPTION REQUIRE DECLARE
```

EXAMPLE:

```
option require declare
declare string name, comment
input 'Please enter your name': name
line input 'Enter a comment in quotes': comment
print name; ' says, '; comment
end


Please enter your name? George
Enter a comment in quotes? 'Have a nice day!'
George says, 'Have a nice day!'
```

DESCRIPTION:

OPTION REQUIRE DECLARE causes SheerPower to require all variables in the program to be declared. If the OPTION REQUIRE DECLARE statement is used and a variable is left undeclared, SheerPower will return an error when program execution is attempted. The OPTION REQUIRE DECLARE statement should occur before any DECLARE statements and before any variables are assigned.

FORMAT:

```
OPTION BASE  [0 | 1]
```

DESCRIPTION:

OPTION BASE sets the lowest subscript or base for arrays. The base can be either zero or one. If you use OPTION BASE 0, the lowest element of an array has the subscript zero (0). If you use OPTION BASE 1, the lowest element is subscript one (1).

See Section 5.8.4 for an example and detailed information on this statement.

FORMAT:

```
[LET] var = expr
```

EXAMPLE:

```
  input 'Last name': last$
  input 'First name': first$
  let name$ = first$ & ' ' & last$
  print name$
  end


Last name? Taylor
First name? Rick
Rick Taylor
```

PURPOSE:

The **LET** statement is used to store information into a variable or data structure.

DESCRIPTION:

var is the variable being assigned a value. *expr* is an expression. The expression is evaluated and its result is assigned to the variable. The expression can be any SheerPower expression (see Chapter 4.) The variable and the expression data types must match. For instance, if *var* is a string variable, *expr* must be a string expression.

NOTE: The keyword LET is optional. For example:

```
LET name$ = first$ & ' ' & last$
```

can be stated as:

```
name$ = first$ & ' ' & last$
```

When SheerPower executes the **LET** statement, it first evaluates the expression on the right side of the equal sign. It then assigns this value to the variable on the left side of the equal sign. The variable represents a location in memory. The value of the expression is stored in this location. Each time a new value is assigned, the old value is lost and the new value is stored in its memory location.

Assigning numeric values:

- Assigning a numeric value to a string variable results in an error.

- Assigning a real value to an integer variable causes the real number to be rounded to the nearest integer and assigned.

```
  input 'Amount': amount
  let rounded% = amount
  print 'Real numeric amount:'; amount
  print 'Integer amount (after rounding):'; rounded%
  end


Amount? 1.54
Real numeric amount: 1.54
Integer amount (after rounding): 2
```

Incrementing Numeric Variables

FORMAT:

```
        num_var++
```

It is common place to increment a numeric variable:

```
  total = total + 1

  lines = lines + 1
```

SheerPower 4GL provides a shorthand syntax for incrementing numeric variables:

```
  total++

  lines++
```

- Assigning a string value to a numeric variable results in an error.
- String variables are dynamic in length, and can be assigned values of up to 12 million characters in length.

LSET, RSET and **CSET** assign string values to variables. LSET left-justifies, RSET right-justifies, and CSET center-justifies the new value. These statements can be used to assign only *string* values.

LSET, RSET and CSET justify the new value *within* the length of the previous value. For example, in the following program, HEADING$ has a length of twenty characters and consists of twenty dots:

```
  heading$ = repeat$('.', 20)          // Twenty dots
  print '('; heading$; ')'
  end

(....................)
```

In the following example, the RSET statement is used to assign the new value 'Page 12' to HEADING$. SheerPower uses the current length of HEADING$, 20 characters, and replaces it with the new value, 'Page 12'. SheerPower *right-justifies* this value by padding it with 13 leading spaces. Thus, HEADING$ still has a length of twenty characters.

```
heading$ = repeat$('.', 20)          //  Twenty dots
print '('; heading$; ')'
rset heading$ = 'Page 12'
print '('; heading$; ')'
end


(....................)
(             Page 12)
```

FORMAT:

```
LSET str_var = str_expr
```

EXAMPLE:

```
heading$ = repeat$('.', 20)          // Twenty dots
print '('; heading$; ')'
lset heading$ = 'Page 12'
print '('; heading$; ')'
end


(....................)
(Page 12             )
```

PURPOSE:

LSET is used to left-justify string data.

DESCRIPTION:

When SheerPower executes an **LSET** statement, it evaluates the string expression on the right side of the equal sign. SheerPower assigns the new value to the string variable and left-justifies this value within the length of the old value. If the new value has leading or trailing spaces, these spaces are carried over and the string is justified with those spaces. LSET can be used only with strings.

FORMAT:

```
RSET str_var = str_expr
```

EXAMPLE:

```
  heading$ = repeat$('.', 20)          // Twenty dots
  print '('; heading$ ;')'
  rset heading$ = 'Page 12'
  print '(' ; heading$ ; ')'
  end

(....................)
(             Page 12)
```

PURPOSE:

RSET is used to right-justify string data.

DESCRIPTION:

When SheerPower executes the **RSET** statement, it evaluates the string expression on the right side of the equal sign. SheerPower assigns the new value to the string variable and right-justifies this value within the length of the old value. If the new value has leading or trailing spaces, these spaces are carried over and the string is justified with those spaces. RSET can be used only with strings.

FORMAT:

```
        CSET str_var = str_expr
```

EXAMPLE:

```
  heading$ = repeat$('.', 20)          // Twenty dots
  print '('; heading$; ')'
  cset heading$ = 'Page 12'
  print '('; heading$; ')'
  end

(....................)
(       Page 12      )
```

PURPOSE:

CSET is used to center string data.

DESCRIPTION:

When SheerPower executes the **CSET** statement, it evaluates the expression on the right side of the equal sign. Next, SheerPower assigns the new value to the string variable and centers it within the length of the old value. If the string value has leading or trailing spaces, the spaces are carried over and the string is centered with those spaces. CSET can be used only with strings.

FORMAT:

```
        LSET | RSET | CSET FILL str_expr: str_var = expr
```

EXAMPLE:

```
  heading$ = repeat$('.', 20)          // Twenty dots
  print '('; heading$; ')'
  cset fill '*': heading$ = 'Page 12'
  print '('; heading$; ')'
  end

(....................)
(*******Page 12******)
```

DESCRIPTION:

The value of *expr* is left-justified, right-justified or centered inside the *str_var*. The remaining part of the string is filled with the pattern specified by *str_expr*. If *str_expr* is the null string, no filling occurs---the remaining part of the string is left as is.

FORMAT:

```
        DATA [num_const | str_const] [,[num_const | str_const]...]
                      .
                      .
                      .
        READ [num_var | str_var] [,[num_var | str_var]...]
```

EXAMPLE:

```
  dim months$(6)
  data January, February, March
  data April, May, June
  for i = 1 to 6
    read months$(i)
    print months$(i)
  next i
  end

January
February
March
April
May
June
```

PURPOSE:

DATA and **READ** statements are used to assign data to variables in cases where the data will not change with successive runs of the program.

DESCRIPTION:

The **DATA** and **READ** statements assign data to variables. DATA specifies a list of data to assign. The data must be given as constants and can be string, numeric or integer types. Multiple data items must be separated by commas.

The READ statement specifies a list of variables to assign data to. The variables can be string, numeric or integer variables. They can be substrings, array elements, etc..

When SheerPower executes the first READ statement, it goes to the first DATA statement and assigns the items in the DATA list to the variables in the READ list. The first variable in the READ list is assigned the first value in the DATA list. The second variable in the READ list is assigned the second value in the DATA list, and so on.

```
DATA constant, constant, constant, constant...
.
.       |         |         |         |
.
READ variable, variable, variable, variable...
```

If the data item contains a comma, the data item should be enclosed with single or double quotes. For example:

```
dim amounts$(3)
data '$25,000', '$250,000', '$2,500,000'
read amounts$(1), amounts$(2), amounts$(3)
print amounts$(1), amounts$(2), amounts$(3)
end


$25,000             $250,000            $2,500,000
```

The variable types and data types must match or an exception will result. For example, if the third item in the DATA list is a string constant, and the third variable in the READ list is a numeric variable, an exception will result.

When the second READ statement is executed, SheerPower starts reading from the first unread data item in the DATA list. For example:

```
dim months$(4)
data January, February, March, April, May, June
read months$(1), months$(2)
read months$(3), months$(4)
print months$(1), months$(2), months$(3), months$(4)
end


January     February      March      April
```

In the example above, when the first READ statement is executed, SheerPower reads the months January and February. When the second READ statement is executed, SheerPower will continue at the first unread month---March---and read it into months$(3).

If you attempt to read more data than exists; that is, if your READ list has more items than your DATA list, an exception will result. You can avoid this by using the **RESTORE** statement to restore the DATA list and read from the beginning again.

The READ and DATA statements must occur in the same program unit. For example, you cannot not have your DATA statements in the main program unit and your matching READ statements in a subprogram.

See Section 5.6.2 for information on using RESTORE.

FORMAT:

```
RESTORE
```

EXAMPLE:

```
  dim months$(3)
  dim more_months$(3)
  data January, February, March
  for i = 1 to 3
    read months$(i)
    print months$(i)
  next i
  restore
  print
  for i = 1 to 3
    read more_months$(i)
    print more_months$(i)
  next i
  end


January
February
March

January
February
March
```

PURPOSE:

RESTORE is used to access the same set of data (from a DATA statement) for a number of READ statements.

DESCRIPTION:

RESTORE restores the DATA statements in a program unit so they can be used again. When the RESTORE statement is executed, all the DATA statements which have been read are restored. The next READ statement causes SheerPower to go back to the first DATA statement and begin assigning the items in its list.

In the example program, the months will be read and assigned to the array MONTHS$. When the RESTORE is executed, the DATA statements will be restored. When the READ statement is executed, the months will be read into the new array MORE_MONTHS$.

FORMAT:

```
    ROUTINE routine_name: PRIVATE var, var, var, ...
or
    ROUTINE routine_name: PRIVATE INTEGER var, STRING var, STRING var, ...
```

EXAMPLE:

```
  do_totals
  end

  routine do_totals: private mytotal, desc$
    mytotal = 15
    desc$ = 'Test Totals'
    print desc$; mytotal
  end routine


Test Totals 15
```

DESCRIPTION

SheerPower allows you to use **private variables** in a routine. Private variables are variables identified with a specific routine. This option allows you to use the same variable names more than once because, internally, SheerPower prefixes the variables with the routine name and a "$" character.

In the above example, the private variables "mytotal" and "desc$" are internally known to SheerPower as:

do_totals$mytotal

do_totals$desc$

From *inside* the routine, you can reference the variables by their private names. For example: mytotal, desc$

From *outside* the routine (or while debugging the code), you can reference the private variables by their internal known names. For example: do_totals$mytotal, do_totals$desc$

Note

See Appendix M, SheerPower and Program Segmentation for more on routines and private routines in SheerPower.

Arrays are a type of variable. They are used to store and manipulate tables of variable information. An array must be defined before it is used in a program. Array variables are described in Section 4.5.1, Arrays.

Arrays are dimensioned with a **DIM** statement. The **REDIM** statement can be used to redimension an array; that is, to change the dimensions of an array which has been defined with the DIM statement. The **OPTION BASE** statement changes the default low bound. By default, the low bound is 1.

About arrays:

- The highest bound allowed is 2147483648.
- The lowest bound allowed is -2147483647.
- Each array can have up to 32 dimensions.
- See Section 4.5.1, Arrays for more information on the storage allocation of arrays.
- If an array element outside of the specified bounds is referenced, an exception is generated.

FORMAT:

```
        DIM [INTEGER | REAL | STRING | BOOLEAN]
                  array_name ([int_expr TO] int_expr [, ...])
```

EXAMPLE:

```
  dim name$(4)
  for i = 1 to 4
    input 'Enter a name': name$(i)
  next i
  print
  for i = 1 to 4
    print i; ' '; name$(i)
  next i
  end


Enter a name? Jim
Enter a name? Jane
Enter a name? Bob
Enter a name? Betty

1  Jim
2  Jane
3  Bob
4  Betty
```

PURPOSE:

DIM is used to dimension arrays. Arrays are used to store tables of variable information. An array must be dimensioned before it can be used.

DESCRIPTION:

The simplest version of a **DIM** statement is:

```
        DIM array_name(int_expr)
```

array_name is the name of the array being defined. The array name must meet the rules for variable names. *int_expr* is the high bound for the array---the highest element allowed in a dimension. The low bound is the lowest element allowed in a dimension. The low bound defaults to 1. For example:

```
        DIM NAME$(4)
```

This statement defines a one-dimensional array with four elements:

```
NAME$(1)
NAME$(2)
NAME$(3)
NAME$(4)
```

Multiple Dimensions

An array can have up to 32 dimensions. A high bound must be specified for each dimension.

```
        DIM array_name(int_expr [, int_expr, ...])
```

For example:

```
          dim name$(4,2)
```

This statement defines the following two-dimensional array:

```
NAME$(1,1)   NAME$(1,2)
NAME$(2,1)   NAME$(2,2)
NAME$(3,1)   NAME$(3,2)
NAME$(4,1)   NAME$(4,2)
```

Low Bounds

The **low bound** is the lowest element a dimension can have. Low bounds can be specified for each dimension of an array. If no low bound is specified, the default is 1. To specify a low bound, use the following format:

```
        DIM array_name (int_ expr TO int_expr)
```

The number preceding TO is the low bound. For example:

```
dim name$(4,18 to 20)
```

This statement creates an array whose first dimension contains elements 1-4 and whose second dimension contains elements 18-20:

```
NAME$(1,18)   NAME$(1,19)   NAME$(1,20)
NAME$(2,18)   NAME$(2,19)   NAME$(2,20)
NAME$(3,18)   NAME$(3,19)   NAME$(3,20)
NAME$(4,18)   NAME$(4,19)   NAME$(4,20)
```

FORMAT:

```
        REDIM array_name (int_expr, int_expr...) ...

             OR

        REDIM array_name [( [int_expr TO] int_expr,
                      [int_expr TO] int_expr... )] ...
```

EXAMPLE:

```
  dim name$(2)
  input 'How many names': num
  redim name$(num)
  for i = 1 to num
    input 'Enter a name': name$(i)
  next i
  do
    print
    for i = 1 to num
    if  name$(i) = ''  then
      print i; ' '; 'empty slot'
    else
      print i; ' '; name$(i)
    end if
    next i
    print
    input 'How many names': num
    if  _back or _exit  then exit do
    redim name$(num)
  loop
  end


How many names? 3
Enter a name? Tim
Enter a name? Sammy
Enter a name? Fred

1  Tim
2  Sammy
3  Fred

How many names? 4

1  Tim
2  Sammy
3  Fred
```

```
4  empty slot

How many names? exit
```

PURPOSE:

The **REDIM** statement is used to change the size of an array.

DESCRIPTION:

REDIM redimensions arrays. REDIM can be used only on arrays that have already been dimensioned with the DIM statement. The REDIM statement has the same rules, options and limits as the DIM statement.

Arrays can be dynamically expanded as needed. If you REDIM a single dimension array or the first dimension of a multi-dimensioned array to a larger size, the old values are kept. If you REDIM any array to a smaller size or REDIM two or more dimensions in a multi-dimensioned array to a larger size, the old values are lost.

If your application depends on REDIM initializing all array values, change your code as follows:

```
        Old Code:       REDIM X(100)

        New Code:       REDIM X(1)
                        REDIM X(100)
```

The REDIM X(1) forces all array values to be initialized by the second REDIM statement.

FORMAT:

```
        OPTION BASE [0 | 1]
```

EXAMPLE:

```
  option base 0
  dim name$(4)
  for i = 0 to 4
    input 'Enter a name': name$(i)
    print i; ' Hello, '; name$(i)
  next i
  end


Enter a name? June
 0  Hello, June
Enter a name? Tony
 1  Hello, Tony
Enter a name? Sandy
 2  Hello, Sandy
Enter a name? Carl
 3  Hello, Carl
Enter a name? Liz
 4  Hello, Liz
```

PURPOSE:

OPTION BASE is used to set the default low bound for arrays to suit your needs. You have the option of starting the array with element O or element 1.

DESCRIPTION:

When no low bound is specified for a dimension, the default is 1. The **OPTION BASE** statement lets you specify a default low bound of 0 or 1. When any following DIM or REDIM statements are executed, SheerPower defaults the low bound to 0 or 1 as specified.

SheerPower has numerous built-in functions. This chapter describes the system and other built-in functions.

The following are common math functions that SheerPower performs:

CEIL(x) returns the ceiling of x. The ceiling of x is equal to the smallest integer that is not less than x.

```
  print ceil(1.543)


2
```

The DIV0 function divides *num_expr1* by *num_expr2*. If *num_expr2* (divisor) is 0, 0 is returned.

```
  print div0(0.8, 0.000004)
  print div0(0.8, 0.0)
  print div0(6, 3)
  print div0(6, 0)
  end


200000
0
2
0
```

Given a number, the FP function returns the fractional part of the number. See Section 6.1.6, IP(num_expr).

```
  print fp(238.304)


 .304
```

INT returns the whole portion of a real number as a real number.

```
  print int(148.8432)


148
```

INTEGER changes any numeric expression into an integer value and assigns the integer value to the variable specified.

```
  z = integer(4 + (993 * 35))
  print z
  end


34759
```

IP truncates the value of a real number at the decimal point and returns the integer portion. See Section 6.1.3, FP(num_expr).

```
  print ip(1234.56)


1234
```

MAX(x,y) returns the larger of the two values x and y. See also "MIN function".

```
  print max(5, 9)


9
```

MIN(x,y) returns the lesser of the values x and y. See also "MAX function".

```
  x = 43
  y = 19
  print min(x, y)


19
```

MOD gives the remainder of one number divided by another.

```
  print mod(36, 13)


10
```

REAL changes any numeric expression into a real or floating-point value and assigns the real value to the variable specified.

```
  input 'Your age': age%
  let decimal_age = real(age%)
  print 'Your number is'; decimal_age
  end


Your age? 31
Your number is 31
```

REMAINDER(x,y) returns the remainder when X is divided by Y. It differs subtly from MOD. MOD(-4,3) = 2 while REMAINDER(-4,3) = -1.

```
  print remainder(-4,3)


-1
```

or

RND(num_expr)

RND returns a random number greater than or equal to zero and less than one. If a numeric expression (*num_expr*) is given, RND returns a whole number between one and the numeric expression.

```
 print rnd


.9409720199182629
```

ROUND rounds a *num_expr* to the specified number of decimal places (*int_expr*). The default int_expr is 0.

```
 print round(21.83492, 2)


21.83
```

This function truncates a real number to a given number of decimal places.

```
 print truncate(123.45678, 2)
 print truncate(123.45678, 4)
 end


123.45
123.4567
```

The following are transcendental functions that SheerPower performs:

ABS returns the absolute value of a specified numeric expression.

```
 print abs(-5)


5
```

The arccosine of X (ACOS(x)) returns the angle whose COS is x. The angle is returned in radians. ACOS has to be between -1 and +1 (inclusive).

```
 print acos(.75)


.722734247813
```

Given X and Y coordinates, the ANGLE function returns the angle from 0,0 in radians.

```
 print angle(4,9)


1.152571997216
```

The arcsine of X (ASIN(x)) returns the angle whose SIN is x. The angle is returned in radians. ASIN has to be between -1 and +1 (inclusive).

```
print asin(.3)

.304692654015
```

Arctangent (ATN) returns the angle, in radians, of a specified tangent.

```
print atn(33)

1.540502566876
```

COS returns the cosine of an angle the user specifies in radians.

```
print cos(64)

.39185723043
```

COSH returns the hyperbolic cosine of a passed real number.

```
print cosh(31)

14524424832623.712890625
```

Cotangent (COT(X)) is equivalent to 1/TAN(X).

```
print cot(31)

-2.2640027937804799
```

CSC(x) is the cosecant of X. It is shorthand for 1/SIN(x).

```
print csc(187)

-1.0028370028157145
```

Given an angle in radians, the DEG function returns the number of degrees.

```
print deg(14)

802.140913183152502
```

EXP function returns the value of the mathematical constant, "e", raised to a specified power.

```
  print exp(5)

148.413159102577
```

LOG returns the natural logarithm of a specified number.

```
  print log(100)

4.605170186
```

LOG2 returns a number's base 2 logarithm.

```
  print log2(100)

6.643856189775
```

LOG10 returns a number's common logarithm.

```
  print log10(100)

2
```

Returns the value 3.1415926535897932.

```
  print pi

3.1415926535897932
```

Given a measurement in degrees, the RAD function returns the number of radians.

```
  print rad(85)

1.4835298641951801
```

SEC returns a secant of a given angle (1/COS(num_expr)). *num_expr* is a passed angle.

```
  print sec(5)

3.5253200858189003
```

SGN returns the sign of a number. It returns a +1 if the expression is positive, a -1 if the expression is negative, and 0 if the expression is zero.

```
  print sgn(-238)
  print sgn(238)
  print sgn(0)

-1
 1
 0
```

SIN returns the sine of an angle specified in radians.

```
  print sin(23)

-.846220404
```

SINH(X) returns the hyperbolic sine X.

```
  print sinh(23)

4872401723.124451637268
```

SQR returns the square root of a number.

```
  print sqr(64)

8
```

TAN returns the tangent of an angle that is specified in radians.

```
  print tan(0.2)

 .202710035509
```

TANH returns the hyperbolic tangent of the numeric expression given.

```
  print tanh(0.5)

 .46211715726
```

The following are date and time functions that SheerPower performs:

DATE returns today's date in YYDDD format. The 'DDD' is the number of days that have gone by so far this year.

```
print date

3117
```

The DATE$ function returns the date in image format. *int_expr1* is a given Julian day number, the default is today's date. *int_expr2* indicates the desired output format for the date. The Julian day is the number of days since January 1, 1600.

Table 6-1 DATE$ function - integer values

Value (int_expr2)	Output Date Format
0	YYYYMMDD format
1	MMDDYYYY format
2	DDMMYYYY format
3	dd-Mon-yyyy format
4	Month dd, yyyy format

```
print date$                 -  gives 20010424

print date$(days(date$),1)  -  gives 04242001

print date$(days(date$),2)  -  gives 24042001

print date$(days(date$),3)  -  gives 24-Apr-2001

print date$(days(date$),4)  -  gives April 20, 2001
```

Given a date in CCYYMMDD or YYMMDD format, the DAYS function returns the number of days since January 1, 1600 (this date is day 1). This number is called the Julian day.

```
print days('20000122')
print days('990122')
end

146119
145754
```

int_num indicates the desired input format for the date. The default input format is zero. If the century is not included, it assumes 1900 as the century.

Table 6-2 DAYS function - integer values

Value (int_num)	Input Date Format
0	CCYYMMDD or YYMMDD
1	MMDDCCYY or MMDDYY
2	DDMMCCYY or DDMMYY
3	DD-Mon-CCYY or DD-Mon-YY
4	Month DD, CCYY

```
  print days('20000103',0)
  print days('01032000',1)
  print days('03012000',2)
  print days('03-Jan-2000',3)
  print days('January 3, 2000',4)
  end


146100
146100
146100
146100
146100
```

Given an integer expression specifying the number of days since January 1, 1600, DAY$ returns the day of the week. If no integer expression is given, DAY$ returns the day of the week for today's date. The day is returned as a string expression (Friday, Saturday, etc.).

```
  print day$


Saturday
```

Given the number of seconds since the SheerPower base date, the FULLTIME$ function returns the date and time in one of the formats given below.

float_expr is the number of seconds since the SheerPower base date. The default is the current date and time. January 1, 1600 00:00:00 is considered the second 0.

Table 6-3 FULLTIME$ function - integer values

Value (int_var)	Output Data Format
0	CCYYMDD HHMMSS
1	MMDDCCYY HHMMSS
2	DDMMCCYY HHMMSS
3	DD-Mon-CCYY HH:MM:SS
4	Month DD, CCYY HH:MM:SS

```
  print fulltime$
  sec = seconds('20000121 115042')
  print fulltime$(sec, 0)
  print fulltime$(sec, 1)
  print fulltime$(sec, 2)
  print fulltime$(sec, 3)
  print fulltime$(sec, 4)
  end


20000208 232653
20000121 115042
01212000 115042
21012000 115042
21-Jan-2000 11:50:42
January 21, 2000 11:50:42
```

Given a full-time string in CCYYMMDD HHMMSS, YYMMDD HHMMSS, HHMMSS or HHMM format, the SECONDS function returns the number of seconds since the SheerPower base date (January 1, 1600 00:00:00).

The number of seconds is returned as a floating point number.

```
  z  = seconds('20000122 103050')
  z1 = seconds('990122 103050')
  z2 = seconds('103050')
  z3 = seconds('1030')
  print 'Seconds cymdhms ='; z
  print 'Seconds  ymdhms ='; z1
  print 'Seconds     hms ='; z2
  print 'Seconds     hm  ='; z3
  end


Seconds cymdhms = 12624633050
seconds  ymdhms = 12593097050
seconds     hms = 37850
seconds     hm  = 37800
```

The value returned by the TIME function depends on the value of *int_expr*.

If *int_expr* = 0, TIME returns the number of seconds since midnight.

If *int_expr* = 1, TIME returns the CPU time of the process in tenths of a second.

If *int_expr* = 2, TIME returns connect time of the current process in minutes.

```
  print time(0)
  print time(1)
  print time(2)
  end


67004
1
0
```

TIME(5) returns the number of seconds since SheerPower was invoked. This function can be used to time events to the nearest 100th/sec.

```
  print time(5)


 .03
```

or

TIME$(num_expr)

If *num_expr* is NOT specified, TIME$ returns the current time of day in HH:MM:SS format.

num_expr is the number of seconds since midnight. The result is returned in HH:MM format.

```
  print time$(1800)
  print time$(54178)
  print time$
  end


00:30
15:02
11:33:27
```

Many applications allow the end-user to enter a six-digit date. For a six-digit date, SheerPower 4GL needs to know if the YEAR is in the 19th century or the 20th century. For example:

```
  161231
```

Is this 1916, December 31st or is this 2016, December 31st?

By default, SheerPower assumes that if a six-digit date is given, and the YEAR is less than 20, then this is the 20th century. In the example above,:

```
  161231 --> December 31, 2016
```

The default **PIVOT DATE** is year 20.

The default pivot date can be changed by creating a logical:

```
  SheerPower_Y2K_PIVOT
```

This can be done with the following small program:

```
  set system, logical 'SheerPower_Y2K_PIVOT': value '17'
```

You can add the following to the c:\sheerpower\sp4gl_YOURNAME.ini file:

```
[logicals]
SheerPower_Y2K_PIVOT=17
```

Then the value will automatically be setup for all SheerPower applications. The logical is checked for only ONCE at sp4gl.exe STARTUP time.

The following are string manipulation functions that SheerPower performs:

The ASCII function returns the decimal ASCII value of a string's first character. It is returned as an integer. The Section 6.4.4 is the opposite of the ASCII function.

```
  print ascii('A')


65
```

CHANGE$ changes specified characters in *str_expr1*. *str_expr1* is the source string. *str_expr2* contains the target characters, and *str_expr3* specifies the substitution characters. CHANGE$ returns the changed string.

CHANGE$ searches for the target characters within the source string and replaces these characters with the substitution characters. The substitution characters are mapped onto the target characters.

```
  let a$ = 'bdbdbdbd'
  let b$ = 'b'
  let c$ = 'c'
  let changed$ = change$(a$, b$, c$)
  print a$
  print changed$
  end


bdbdbdbd
cdcdcdcd
```

CHARSET$ returns the character set specified. The optional string expression can be used to specify the character set to return. The available character sets are:

Table 6-4 Available character sets for CHARSET$

	UCASE	all upper-case letters (A-Z)
	LCASE	all lower-case letters (a-z)
	CONTROL	all control characters (ASCII 0-31)
	ASCII	the ASCII character set, in order (0-255)

ASCII is the default character set for CHARSET$.

```
  line input 'Enter your text': text$
  // change upper-case to lower-case
  ct$ = change$(text$, &
       charset$('ucase'), &
       charset$('lcase'))
  print 'Lower-case version is:'; ct$
  end


Enter your text? TESTER
Lower-case version is: tester
```

CHR$ returns a string with the specified ASCII value (*int_expr1*) repeated the specified number of times (*int_expr2*). If no count is specified, a default count of one is used.

```
  x = 65
  print chr$(x)  // prints A -- the 65th ASCII character
  end


A
```

Given an integer (*int_expr1*) and an optional length (*int_expr2*), which defaults to four, the CONVERT$ function returns a string mapping of the integer.

If the optional data type (*int_expr3*) is 17, the returned string will be a packed floating (PF).

The following data types are supported:

Table 6-5 CONVERT$ function - supported data types

Data Type	Conversion Result
1	Integer (2 or 4 byte)

7	COBOL comp-3 (C3 packed decimal)
17	Packed floating (PF)

```
  a$ = convert$(16961)
  print a$
  end


AB
```

Given a string containing a mapped integer, the CONVERT function returns the integer value.

```
  a$ = 'AB'
  b = convert(a$)
  print b
  end


16961
```

The CONVERT and CONVERT$ functions can be used in situations such as building segmented keys consisting of multiple data types.

CPAD$ returns a new string, padded on the left and on the right with pad characters. *text_str* is the string to be centered, *size* is the size of the new string. The default pad character is a space.

```
  print cpad$('123', 9, '0')
  end


000123000
```

EDIT$ performs one or more editing operations on the supplied string argument, depending on the value of the integer expression. The integer expression is one of the integers below, or a sum of integers below for the desired edit functions:

Table 6-6 EDIT$ function - operation values

Value	Edit Operation
1	Trim parity bits.
2	Discard all spaces and tabs.
4	Discard characters: CR, LF, FF, ESC, RUBOUT and NULL.
8	Discard leading spaces and tabs.
16	Reduce spaces and tabs to One space.
32	Convert lower case to upper case.
64	Convert "[" to "(" and "]" to ")".
128	Discard trailing spaces and tabs.
256	Do not alter characters inside quotes.

```
  print edit$('hi there, how are you today?' , 32)


HI THERE, HOW ARE YOU TODAY?
```

The ELEMENTS function returns the number of elements in a string expression that contains a list of elements. *str_expr1* is the string containing the list of elements. *str_expr2* is the separator. A comma is the default separator.

- Leading and trailing spaces are removed from the text.

- Quoted data is treated as one element.

A given element is considered **quoted** only if the first non-blank character of the element is a single or double quote mark.

```
  alphabet$ = 'a;b;c;d;e;f;g;h;i;j;k;l;m;n;o;p;q;r;s;t;u;v;w;x;y;z'
  print elements(alphabet$, ';')
  end


26
```

ELEMENT$ returns the element from *str_expr1* which is specified by the *num_expr*. *str_expr1* contains a set of elements with separators between them. The default separator is a comma:

```
  let a$ = element$('ADD,DEL,EXIT',2)
  print a$
  end


DEL
```

A separator other than the comma can be specified with *str_expr2*.

```
  let sentence$ = 'This is a test.'
  let a$ = element$(sentence$,2,' ')
  print a$
  end


is
```

More than one separator in a row returns a null for the corresponding element.

Note on the following example:

The following example shows how having two commas [the default separator] in a row will cause the result to be *nothing* [null] for that particular element.

```
  let sentence$ = 'This,, is, a, test'
  print element$(sentence$, 2)
  end
```

The ENCODE$ function returns a string containing a number converted to the base specified. *num_expr* is the value to convert. *num_int* is the base to convert. For instance, '2' indicates binary, etc. See also Section 6.9.1, DECODE(str_expr, int_expr)

```
  do
    input 'Enter a number to convert to hex': decnum
    if  decnum = 0 or _exit  then exit do
    print 'Hex for'; decnum;' is '; encode$(decnum,16)
  loop
  end


Enter a number to convert to hex? 34.56
Hex for 34.56 is 22
Enter a number to convert to hex? 255
Hex for 255 is FF
```

Given an expression and a format, FORMAT$ returns the result of the expression in the format indicated.

The '@' format character causes the character not to be translated by the formatter. The '<' and '>' are treated like an '@' character. You can justify a character string, but avoid zero suppression and zero insertion.

The FORMAT$ function takes an expression of any data type for the data to format (the first argument), including string expressions.

```
  z$ = format$('1234567', '###~-####')
  print 'Phone number: '; z$
  end


Phone number: 123-4567
```

The FORMAT$ function returns all asterisks "*" in the case of overflow.

```
  z$ = format$(12.23,'#.##')
  print z$
  end


****
```

FORMAT$() returns the same string data as given by the following:

```
        PRINT USING str_expr: expr
```

The FORMAT$ function supports the DATE format and date arguments. Given a date in YYMMDD or CCYYMMDD format, FORMAT$ returns the date in the date format requested.

```
        FORMAT$(z$, '{DATE [argument]}?')
```

The **?** can be replaced with a mask. If no date argument is provided, the default is MDCY.

```
  z1$ = format$('990122', '{date mdcy}?')
  z2$ = format$('990122', '{date mdcy}##/##/####')
  z3$ = format$('20000122', '{date mdcy}?')
  z4$ = format$('20000122', '{date mdcy}##/##/####')
  print z1$, z2$
  print z3$, z4$
  end


01221999            01/22/1999
01222000            01/22/2000
```

Table 6-7 FORMAT$ function - date arguments

DATE Argument	YYMMDD Input	Result	CCYYMMDD Input	Result
none	990207	02072000	20000207	02072000
YMD	990207	990207	20000207	990207
CYMD	990207	20000207	20000207	20000207
MDY	990207	022199	20000207	022199
MDCY	990207	02072000	20000207	02072000
DMY	990207	070299	20000207	070299
DMCY	990207	07022000	20000207	07022000
DMONY	990207	07-Feb-99	20000207	07-Feb-99
DMONCY	990207	07-Feb-2000	20000207	07-Feb-2000
MONTHDY	990207	February 7, 99	20000207	February 7, 99
MONTHDCY	990207	February 7, 2000	20000207	February 7, 2000

The FORMAT$ function supports character rotation. The ROTATE option rotates the last nn characters of a string to the first position in the string.

```
      FORMAT$(z$, '{ROTATE n}?')
```

The **?** can be replaced with a mask.

The GETSYMBOL$ function is used to return script variables and symbols. These can be the results of an HTML form submission, CGI environment variables, SheerPower symbols, DNS symbols, operating system symbols, or any variable you have defined as a symbol.

The GETSYMBOL$ function has an optional parameter that allows you to not trim the leading and trailing spaces of a symbol returned. The default parameter is set to **TRUE**. "True" is implied and means that the leading and trailing spaces will be trimmed. The **FALSE** parameter can be used if you do not want the leading and trailing spaces trimmed from the symbol retrieved.

```
  set system, symbol 'test': value '  hi there'
  print '<';getsymbol$('sp:test');'>'
  print '<';getsymbol$('sp:test', false);'>'
  print '<';getsymbol$('sp:test', true);'>'

<hi there>
<  hi there>
<hi there>
```

Below are a few more examples of GETSYMBOL$:

```
  //  print the contents of the symbol "city" from a HTML form submit.
  print getsymbol$('city')
```

```
  // print the contents of the environment symbol REMOTE_ADDR
  print getsymbol$('env:REMOTE_ADDR')
```

```
  // print the contents of the operating system symbol PATH
  print getsymbol$('os:PATH')
  // and list the contents of the TEMP directory
  print getsymbol$('os:TEMP')


C:\PROGRA~1\Java\JRE16~2.0_0\bin;C:\PROGRA~1\Java\JRE16~2.0_0\bin;C:\WINDOWS\sys
tem32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\PROGRA~1\ABSOLU~1;C:\Program Files\
QuickTime\QTSystem\;C:\Program Files\IDM Computer Solutions\UltraEdit\;C:\Progra
m Files\IDM Computer Solutions\UltraCompare;.
C:\DOCUME~1\User\LOCALS~1\Temp
```

Below is a list of **symbol prefixes** supported:

Table 6-8 Supported Symbol Prefixes

Symbol Prefix	Description
env:	CGI environment variables (see Section 19.3.8, Summary of CGI Environment Variables)
os:	Operating system symbols. If the symbol begins with a \ then it is a registry symbol.
sp:	SheerPower symbols (see Section 11.14.15, SET SYSTEM, SYMBOL: VALUE)
dns:	DNS symbols (see Section 11.14.14, Performing DNS Lookups with ASK SYSTEM, SYMBOL)

The HASH$ function changes the plain text in *str_expr1* into a hashed eight-byte string value. It can be used to develop one-way hashed passwords. The optional text in *str_expr2* and optional *int_expr* can be used to further make the hashed value unique.

```
  password$ = hash$('TRUTH')
  input 'Password': pwd$
  if  hash$(pwd$) = 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.
```

LCASE returns a string expression with all letters in lower case. See Section 6.4.41, UCASE$(str_expr).

```
  print lcase$('IT HAS BEEN A WONDERFUL DAY!')


it has been a wonderful day!
```

LEFT$ returns the leftmost **nn** characters from a string. *int_expr* is the last character position to be included in the resulting string.

```
  print left$('Hello there!', 3 )

Hel
```

LEN returns the length of a string. It returns an integer.

```
  print len('These are the built-in functions of SheerPower.')

47
```

LPAD$ pads a string on the left with pad characters. The default pad character is a space.

```
  print lpad$('123', 6, '0')

000123
```

LTRIM$ removes all leading spaces (those on the left side of the string).

```
  print ltrim$('   This function gets rid of leading spaces to the left of a
string.')
  end

This function gets rid of leading spaces to the left of a string.
```

MAXLEN returns the maximum number of characters that a string variable can contain. Since all string variables are variable length with a maximum of 16711425, this function always returns 16711425.

```
  print maxlen('Hi')

16711425
```

MEM() returns a zero-terminated string where the first byte starts at the given memory address.

If the memory address passed to the MEM() function is not readable, MEM() returns a zero-length string.

The format for MEM() is:

```
  string_result = mem(int_memory_address)
```

Where STRING_RESULT is the zero-terminated string pointed to by the given memory address.

```
  library 'msvcrt.dll'
  call 'ctime' (0%)         // returns a pointer to a zero-terminated string
  mytime$ = mem(_integer)   // get the string with the MEM() fuction
  print '-- '; mytime$
  end


-- Wed Dec 31 16:00:00 1969
```

MID$ or MID returns a substring from the middle characters of a specified string, leaving the string unchanged. *int_expr1* is the starting position of the substring, and *int_expr2* is the length of the substring. MID$(str_expr,int_expr1) will return the rest of the string.

```
  a$      = 'beginmiddleend'
  middle$ = mid$(a$, 6, 6)
  end$    = mid$(a$, 6)
  print middle$, end$
  end

middle     middleend
```

Given the ASCII name of a character, the ORD function returns the location of that character in the ASCII character table. It returns an integer number.

```
  print ord('H')

72
```

Given the location of a character in the ASCII character table, the ORDNAME$ function returns the name of that character.

```
  print ordname$(69)

E
```

PARSE$ splits a string into tokens and returns each token separated by a space. Letters are UPPERCASE except within quotes. Tail comments are ignored. Embedded "$" characters are allowed. Names/words can start with digits. The maximum line length is 1024 characters.

```
  a$ = 'company$ = 123abc$ + "and sons" !rnn'
  print parse$(a$)
  end

COMPANY$ = 123ABC$ + "and sons"
```

and

PIECE$ returns an element from *str_expr1* specified by *num_expr*. *str_expr1* contains a list of elements. The separator can be indicated by *str_expr2*. The default separator is a carriage-return line-feed pair.

These two functions are similar to the ELEMENTS() and ELEMENT$() functions except that:

- no leading or trailing spaces are removed from the text
- quoted data is not skipped over
- the default separator is the two-character sequence. CHR$(13)+CHR$(10), the CR/LF sequence used by WRAP$()

```
message 'Enter in a long line of text to be rewrapped. Then click DONE'
line input area 5, 10, 8, 60: text$
print at 10, 1: 'Rewrapped text'
wt$ = wrap$(text$, 1, 30)
print 'Number of lines: '; pieces(wt$)
print wt$
print
print 'First line was: '; piece$(wt$, 1)
end


        +-------------------------------------------+
        |This line of text is long enough to be     |
        |rewrapped into more than one line.         |
        |                                           |
        |                                           |
        +-------------------------------------------+
               Done    Back   Exit   Help


 Enter in a long line of text to be rewrapped. Then click DONE.


         This line of text is long enough to be rewrapped into
         more than one line.


Rewrapped text
Number of lines:  3
This text is long
enough to be rewrapped into
more than one line.

First line was: This text is long
```

PRETTY$ converts text so that the text displays on any terminal. Named control characters show up with their names. Other control characters show up as {X} where "X" is the letter to press or as {XX} where "XX" is the hexadecimal value of the character.

```
a$ = 'Hello' + chr$(5) + chr$(161) + chr$(7)
print pretty$(a$)
end


Hello{^E}{A1}{bel}
```

The QUOTE$ function encloses a string expression in double quotes. If the string expression is already enclosed in double quotes, QUOTE$ leaves it alone. If the string expression is already wrapped in single quotes, QUOTE$ replaces them with double quotes. Elements double-quoted within the string expression are given another pair of double quotes (see following example). Elements single-quoted within the string expression are ignored.

```
do
  clear
  print at 1,1:
  message 'Enter a line of text to be quoted'
  print 'Text:'
  input '', length 30: line$
  if  _back or _exit  then exit do
  if  line$ = ''  then repeat do
  print
  print 'Quoted text using the QUOTE$ function...'
  print quote$(line$)
  delay
loop
end
```

```
Text:
? The little boy cried "wolf!"

Quoted text using the QUOTE$ function...
"The little boy cried ""wolf!"""
```

REPEAT$ creates a string composed of the specified string repeated the specified number of times.

```
  print repeat$('Hi!', 9)

Hi!Hi!Hi!Hi!Hi!Hi!Hi!Hi!Hi!
```

REPLACE$ searches for a list of patterns in the *str_expr1* and replaces it with the output string from *str_expr2*. REPLACE$ returns the replaced string expression.

str_expr1 is a list of patterns to search for.

str_expr2 is the replacement list.

str_sep1 is the optional separator for replacement items. The default is a comma.

str_sep2 is the optional separator between the input and output text in items. Default is =.

```
  text$ = '01-Mar-1989'
  print replace$(text$, 'r=y 8=9' , ' ')
  end

01-May-1999
```

RIGHT$ returns the rightmost characters from a string. *int_exp* is the character position of the last character to be included in the substring COUNTING FROM THE RIGHT.

```
  ans$ = right$('Daniel', 2)
  print 'rightmost characters = '; ans$
  end

rightmost characters = el
```

RPAD$ pads a string on the right with pad characters. The default pad character is a space.

```
  print rpad$('123', 6, '0')
  end

123000
```

RTRIM$ returns a string without any trailing spaces (those on the right side).

```
  let a$ = '    HELLO    '
  print '*'; a$; '*'
  let stripped$ = rtrim$(a$)
  print '*'; stripped$; '*'


*    HELLO    *
*    HELLO*
```

The SEG$ function uses a first and last character position to extract the substring.

```
  print seg$('abcdefghijklmnop', 3, 8)


cdefgh
```

This function sorts the elements from a *str_expr1* in ASCII value order; returns a list of the sorted elements.

str_expr1 contains the list of elements to be sorted.

str_expr2 is an optional separator. Default is a comma.

```
  a$ = 'code area is'
  a_sort$ = sort$(a$, ' ')
  print a_sort$
  end


area code is
```

SPACE$ returns the number of spaces indicated by num_expr.

```
  indent  = 10
  indent$ = space$(10)
  print indent$; 'This text is indented'; indent; 'spaces.'
  end


          This text is indented 10 spaces.
```

STR$ changes a number to a numeric string. The string that is created does not have any extra leading or trailing spaces.

```
  age = 22
  me$ = "I am " + str$(age) + " years old."
  print me$
  end


I am 22 years old.
```

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to the right to a specified column.

```
  print tab(20); 'Hello there!'

                   Hello there!
```

TRIM$ returns the string specified stripped of any leading or trailing spaces and tabs.

```
  let a$ = '    HELLO    '
  print '*'; a$; '*'
  let stripped$ = trim$(a$)
  print '*'; stripped$; '*'

*    HELLO    *
*HELLO*
```

UCASE returns a string expression with all letters in upper case. See also Section 6.4.15, LCASE$(str_expr).

```
  print ucase$('are you enjoying this manual so far?')

ARE YOU ENJOYING THIS MANUAL SO FAR?
```

The UNQUOTE$ function removes one set of quotes from a quoted string expression. If the string expression is not quoted, UNQUOTE$ leaves the string alone. UNQUOTE$ does not affect internally quoted elements.

```
  do
    print at 1,1:
    message 'Enter a line of text to be unquoted'
    print 'Text:'
    input '', length 50: line$
    if  _back or _exit  then exit do
    if  line$ = ''  then repeat do
    print
    print 'Quotes removed using the UNQUOTE$ function...'
    print unquote$(line$)
    delay
  loop
  end

Text:
? "I will not take these 'things' for granted."

Quotes removed using the UNQUOTE$ function...
I will not take these 'things' for granted.
```

The URLENCODE$ function takes a string and converts it to a new string that can be used as data in a browser URL. Spaces in the string are converted to + signs, and in the example below, the ampersand & is converted to **%26**. To decode a converted string, see Section 6.4.44, URLDECODE$(str_expr).

```
  mydata$ = '?phrase=' + urlencode$('Dogs & cats')
  print mydata$
  end


?phrase=Dogs+%26+cats
```

The URLDECODE$ function takes the encoded string data in a browser URL and decodes it into plain text. In the example below, the + signs are converted to spaces, and the %26 is converted to an ampersand (&). To encode string data for use in a browser URL see Section 6.4.43, URLENCODE$(str_expr).

```
  print urldecode$('?phrase=Dogs+%26+cats ')
  end


?phrase=Dogs & cats
```

VAL returns the floating-point value of a numeric string.

```
  text$ = "My age is 20"
  z0$   = element$(text$, 4, ' ')
  age   = val(z0$)
  print 'In 10 years I will be'; age + 10
  end


In 10 years I will be 30
```

WRAP$ returns a word-wrapped text string, given left and right margins. Each line of the string is separated with a CR/LF.

Where *string_expr* = text string to wrap, *int_expr1* = left margin, *int_expr2* = right margin.

```
  input 'Type in a sentence' ; text$
  text$ = wrap$(text$, 5, 15)
  print text$
  end


Type in a sentence? This is an example of the wrap$ function.
     This is an
     example of
     the wrap$
     function.
```

The XLATE$ function translates one string to another by referencing a table you supply. For example, the XLATE$ function can translate from EBCDIC to ASCII. The first *str_expr* is the string to be translated. The second *str_expr* is the translation table.

```
  a$ = charset$
  a$[66:66] = 'x'  // change the "A" to a lowercase "x"
  print xlate$('DAN', a$)
  end


DxN
```

The following are string searching and comparing functions that SheerPower performs:

The COMPARE function compares two strings and returns a numeric value ranging from 0 (no match) to 100 (an exact match).

```
  options$ = 'LEFT,RIGHT,UP,DOWN'
  best = 0
  best$ = ''
  input 'Enter an option': usr_opt$
  for idx = 1 to elements(options$)
    opt$  = element$(options$, idx)
    score = compare(opt$, usr_opt$)
    if  score > best  then
      best  = score
      best$ = opt$
    end if
  next idx
  select case best
  case 0
    print 'Unknown option: '; usr_opt$
  case 100
    print 'Option okay, it was: '; usr_opt$
  case else
    print 'Misspelled option: '; usr_opt$
    print using 'Did you mean ? (## percent)': best$, best
  end select
  end


Enter an option? dwn
Misspelled option: dwn
Did you mean DOWN (92 percent)
```

The ITEM function returns the number of the first item that matches the whole or partial item name given. A negative number is returned if more than one item matches.

Quoted data is treated as one element; the quotes are not removed.

```
  z = item('ADD,EXIT,EXTRACT,MODIFY', 'MOD')
  print z
  end


4
```

```
  z = item('ADD,EXIT,EXTRACT,MODIFY', 'EX')
  print z
  end


-2
```

str_expr1 contains a list of elements separated by commas. *str_expr2* contains a string. MATCH compares *str_expr2* with each of the elements in *str_expr1* and gives the number of the element that matches.

The TRUE parameter is used if the match is to be *case-exact*. The default is case-insensitive.

```
  a$ = 'BLUE'
  b$ = 'Blue'
  c$ = a$
  do
    x = match('Red,White,Blue', c$, true)
    print c$;
    if  x > 0  then
      print ' is a match.'
    else
      print ' is not a match.'
      c$ = b$
      repeat do
    end if
  end do
  end


BLUE is not a match.
Blue is a match.
```

This function matches any characters in text (*str_expr1*) with the pattern (*str_expr2*). *str_expr1* is the text to search and *str_expr2* is the pattern being searched for. It returns the location of the first character in the text that contains the pattern. If the characters cannot be found, it returns zero.

PATTERN Options and Examples

Pattern options can be mixed and matched with unlimited complexity.

?

The "?" matches any character in the text.

```
  if  pattern('The quick brown fox', &
     'f?x') > 0  then
     print 'Found'
  end if
  end


Found
```

*

The "*" matches one or more of a character that precedes the single asterisk (*).
It matches zero or more of a character that precedes the double asterisk (**).

```
  if  pattern('aaa   01/26/99', 'a* *01/26/99') > 0  then
    print 'The date is found'
  end if
  end


The date is found
```

The {} is used to define a group of characters. The characters in the enclosed group can be ranges such as {a-z} or individual characters such as {AQX}.

```
  text$ = 'A1N5V7N0'
  rule_str$ = '{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}{A-Z}{0-9}'
  if  pattern(text$, rule_str$) > 0  then
    print "Driver's license is correct."
  end if
  end


Driver's license is correct
```

{^}

The {^} looks for characters that are NOT {^A-Z}. The result below shows the difference between using '?' and {^}.

```
  print pattern('Mary J. Smith','{^Mar}')
  print pattern('Mary J. Smith','J. {^S}')
  print pattern('Mary J. Smith','J. S?')
  end


4
0
6
```

~

The '~' (quote) character looks for a pattern of text that IS an * (stands for itself).

```
  text$ = '$4,670.00'
  if  pattern(text$, '$~4,670.00') > 0  then
    print 'Your text is correct.'
  end if
  end


Your text is correct.
```

{|nnn,nnn,nnn|}

This feature lets you designate binary data for detecting patterns that are not printable text characters. The characters in the enclosed group can be individual byte values such as {|13|} or groups such as {|0,13,200|}. Since {A-Z} is the range of A through Z, the proper syntax for this same range using byte values would be {{|65|}-{|90|}}.

```
  text$ = 'Help yourself to some ' + chr$(13) + 'food'
  if  pattern(text$, '{|13|}') > 0  then
    print 'Carriage return found.'
  end if
  end


Carriage return found.
```

{<cc|ccc|c>}

The {<cc|ccc|c>} looks for text in *str_expr1* that matches the enclosed groups.

```
  text$ = 'The area code is 619'
  if  pattern(text$, 'is {<619|714|916>}') > 0  then
    print 'Your area code is on the list.'
  end if
  end


Your area code is on the list.
```

{(word_text)}

(word_text) looks for a match on a word. The word text can contain an embedded "$".

```
  a$ = 'There are dogs and cats in the house.'
  b$ = 'Everyone would like to make big$bucks!'
  if  pattern(a$, '{(cats)}') > 0  then
    print 'The word was found.'
  end if
  if  pattern(b$, '{(big$bucks)}') > 0  then
    print 'The $ word was found.'
  end if
  end


The word was found.
The $ word was found.
```

{|directive|}

Directives can be used with the PATTERN function. Multiple directives can be used. Directives are processed from left to right. The valid directives are:

nocase	can be uppercase, lowercase or a combination of the two
case	unless NOCASE specified, default is CASE (i.e. case-exact)
bol	beginning of a line
eol	end of a line

```
  a$ = 'ElEpHaNtS have trunks'
  b$ = 'water goes splash'
  if  pattern(a$, '{|bol|}{|nocase|}elephants') > 0  then
    print 'elephants was found.'
  end if
  if  pattern(b$, 'splash{|eol|}') > 0  then
    print 'splash was found.'
  end if
  end


elephants was found.
splash was found.
```

POS searches for a substring within a string. It returns the substring's starting character position. *str-exp1* is the string to be searched, *str-exp2* is the substring to locate, and *int-exp* is the OPTIONAL character position at which to begin the search. (The default is the start of the string.) If the substring is not found, zero is returned.

```
  text$ = "Hello and goodbye"
  andpos = pos(text$, 'and')
  if  andpos > 0  then print 'AND found at position'; andpos
  end


AND found at position 7
```

This function scans *str_expr1* for the characters in *str_expr2* and returns the position at which *str_expr2* begins. *int_expr* specifies a character position at which the search is to begin.

The characters in *str_expr2* need not appear contiguously in *str_expr1*.

```
  let a$ = 'Cameron Whitehorn'
  let b$ = 'amr Wtor'
  let position = scan(a$, b$)
  print position
  end


2
```

SKIP returns the position of the character following the last skipped character.

str_expr1 is the text string to be searched.

str_expr2 contains the list of characters which are to be skipped. If only one argument is given, SKIP will skip over spaces, tabs and nulls.

int_expr contains the search start position. This parameter is optional.

```
  a$ = '31415 hello'
  z  = skip(a$, '1234567890 ')
  print mid(a$, z)
  end


hello
```

The following are end user interface system functions that SheerPower performs:

_BACK returns a TRUE or FALSE value. TRUE if the `[esc]`or the UP ARROW was pressed at the last prompt.

```
  message 'Press the Escape key or the Up Arrow key'
  input 'Please enter your age' : age$
  if  _back  then
    print '_back is set to true'
  end if
  end


Please enter your age? [Esc]  <---- press the Escape key
_back is set to true
```

_EXIT returns a TRUE or FALSE value. TRUE if EXIT was entered at the last prompt.

```
  do
    input 'Please enter your name' : name$
    if  _exit  then
    print '_exit is set to true'
    exit do
    end if
  loop
  end


Please enter your name? [Ctrl/Z]      <------ hold down the Ctrl key, then press the
Z key
_exit is set to true
```

_HELP returns a TRUE or FALSE value. TRUE if HELP or a question mark (?) was entered at the last prompt.

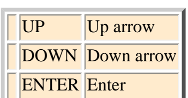
_HELP should be checked before _BACK and/or _EXIT because, in some cases, all three are set on. For example, if "EXIT" is the default and HELP is entered, both _HELP and _EXIT are set on.

```
  input 'Do you need help' : location$
  if  _help  then
    print '_help is set to true'
  end if
  end


Do you need help? help
_help is set to true
```

_REPLY returns the user's reply to the last prompt. The reply is returned as a string.

```
  last$ = 'Enter new text'
  do
    line input area 5,10,15,50, default last$: text$
    if  _exit  then exit do
    last$ = 'You said ' + _reply
  loop
  end
```

The _TERMINATOR function returns the name of the key that terminated the last INPUT statement. The values returned are:



UP	Up arrow
DOWN	Down arrow
ENTER	Enter

F1, F2, F3, F4, F5, F7, F8, F9, F11, F12

```
  do
    line input 'name': yourname$
    if  _exit  then exit do
    print 'Terminator was: '; _terminator
  loop
  end


name? [F3]  <----- press the F3 key
Terminator was: F3
name? exit
```

VALID is used to validate user responses.

text_str is the text to be validated.

rule_str is the list of validation rules.

Multiple validation rules are separated by a semicolon. If given characters are NOT between quotes, they are to be uppercase.

VALID returns an error if there is an invalid validation rule.

```
'Illegal validation rule' (-4021)
```

VALID returns TRUE or FALSE according to the following validation rules:

- **ALLOW text1, text2, text3 [to text4]**
  ALLOW compares the *text_str* with each element on the *rule_str* list (defaults to CHARACTER) and returns TRUE if found.
  For numeric comparison, "NUMBER" or "INTEGER" *must* be used, otherwise, VALID() will do a string-compare validation...left justified.

```
  text$    = 'ann'
  vrule$   = 'allow ann, dan, tom'
  number$  = '10'
  vrules$  = 'number; allow 1 to 6; maxlength 2'
  number2$ = '12'
  vrules2$ = 'number; allow 1 to 24, 99'

  if  valid(text$, vrule$)  then print 'true'
  if  not valid(number$,  vrules$) &
        then print 'false'
  if  valid(number2$, vrules2$)  then print 'true'
  end


true
false
true
```

- **DISALLOW text1, text2, text3 [to text4]**
  DISALLOW returns TRUE if *text_str* cannot be found on the *rule_str* list.

```
  number$ = '10'
  vrules$ = 'disallow 01, 03, 05; minlength 2'
  if  valid(number$, vrules$) &
    then print 'true'
  end


true
```

- **MINLENGTH nn**
  Minimum of nn characters long.

```
  text$  = 'Hello there'
  vrule$ = 'minlength 5'
  if  valid(text$, vrule$) &
    then print 'true'
  end


true
```

- **MAXLENGTH nn**
  Maximum of nn characters long.

```
  text$  = 'Hello'
  vrule$ = 'maxlength 5'
  if  valid(text$, vrule$) &
    then print 'true'
  end


true
```

- **LENGTH nn**
  Exactly nn characters long.

```
  text$  = 'abcdefghijklmnopqrstuvwxyz'
  vrule$ = 'length 26'
  if  valid(text$, vrule$) &
    then print 'true'
  end


true
```

- **CHARACTERS "ccc"**
  Returns TRUE if the given characters (letters or numbers) are on the *rule_str* list.

```
  text$  = 'abc123'
  vrule$ = 'characters "abc123"'
  if  valid(text$, vrule$) &
    then print 'true'
  end


true
```

- **NOCHARACTERS "ccc"**
  Returns TRUE if the given characters are not on the *rule_str* list.

```
  text$  = 'abc123'
  vrule$ = 'nocharacters "def456"'
  if  valid(text$, vrule$) &
    then print 'true'
  end


true
```

- **LETTERS**
  Returns TRUE if given *text_str* consists only of A - Z, a - z or space.

```
  text$   = 'It is a sunny day today'
  vrules$ = 'letters'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **LCASE**
  Returns TRUE if given *text_str* consists of lowercase letters.

```
  text$   = 'hi there'
  vrules$ = 'lcase'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **UCASE**
  Returns TRUE if given *text_str* consists of uppercase characters.

```
  text$   = 'HI THERE'
  vrules$ = 'ucase'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **DIGITS**
  Returns TRUE if *text_str* consists of the numbers 0 - 9.

```
  text$   = '983745'
  vrules$ = 'digits'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **DECIMALS nn**
  Specifies the maximum number of decimal places.

```
  text$   = '9837.45'
  vrules$ = 'decimals 2'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **NUMBER**
  Indicates that *text_str* is numeric (validates that number).

```
  text$   = '100'
  vrules$ = 'number'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **INTEGER**
  Returns TRUE if given integer number is up to 32 bits long (i.e. 2147483647).

```
  text$   = '2147483647'
  vrules$ = 'integer'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **INTEGER WORD**
  Returns TRUE if given integer number is up to 16 bits long (i.e. 32767).

```
  text$   = '32767'
  vrules$ = 'integer word'
  if  valid(text$, vrules$) &
    then print 'true'
  end


true
```

- **DATE**
  Validates a given date in: YYMMDD or CCYYMMDD format.

	DATE	YMD	YYMMDD format
	DATE	DMY	DDMMYY format
	DATE	MDY	MMDDYY format
	DATE	MDCY	MMDDCCYY format

Important Note

For all dates, use 'LENGTH nn' to control whether a 6- or 8-character date is required.

- **DATE DMONY**
  Checks if *text_str* is a legal date in DD-Mon-YY format and returns TRUE (1) or FALSE (0).

```
  text$  = '01-Jan-99'
  vrule$ = 'date dmony'
  print valid(text$, vrule$)
  end


1
```

- **DATE DMONCY**
  Checks if *text_str* is a legal date in DD-Mon-CCYY format and returns TRUE (1) or FALSE (0).

```
  text$  = '01-Jan-2000'
  vrule$ = 'date dmoncy'
  print valid(text$, vrule$)
  end


1
```

- **FULLTIME**
  Checks if *text_str* is a legal date and time in CCYYMMDD HHMMSS or YYMMDD HHMMSS format and returns TRUE (1) or FALSE (2).

```
  text$  = '20000122 010101'
  text2$ = '990122 010101'
  vrule$ = 'fulltime'
  print valid(text$, vrule$)
  print valid(text2$, vrule$)
  end


1
1
```

- **PATTERN**
  Matches *text_str* with PATTERN's *rule_str*. For details see Section 6.5.4, PATTERN(str_expr1, str_expr2).
- **REQUIRED**
  Returns TRUE if *text_str* is not null or space(s).

```
  text$   = 'a b c d'
  text2$  = ' '
  vrule$  = 'required'
  print valid(text$, vrule$)
  print valid(text2$, vrule$)
  end


1
0
```

- **YES/NO**
  Returns TRUE if *text_str* consists only of the following: YES, NO, Y or N.

```
  text$   = 'yes'
  text2$  = 'n'
  text3$  = 'a'
  vrule$  = 'yes/no'
  print valid(text$, vrule$)
  print valid(text2$, vrule$)
  print valid(text3$, vrule$)
  end


1
1
0
```

- **VRULES**
  Checks if a *text_str* is a legal set of validation rules, and returns TRUE (1) or FALSE (0).

```
  print valid('integer', 'vrules')
  end


1
```

- **PRINTMASK**
  PRINTMASK checks the format of *text_str* and returns TRUE (1) if it is legal or FALSE (0) otherwise.

```
  text_str$ = '##.##'
  vrule$    = 'printmask'
  print valid(text_str$, vrule$)
  end


1
```

- **EXPRESSION**
  EXPRESSION validates a *text_str* and returns TRUE (1) if it is a legal SheerPower expression or FALSE (0) otherwise.

```
  text_str$  = 'total = a% + 30'
  text_str2$ = '##~-###'
  vrule$     = 'expression'
  print valid(text_str$, vrule$)
  print valid(text_str2$, vrule$)
  end


1
0
```

- **CODE**
  CODE checks for VALID SheerPower code syntax.

```
  a$ = "print 'hello, ' name$"
  if  not valid(a$, 'code')  then print 'false'
  end


false
```

- **MENU**
  MENU validates a *text_str* and returns TRUE (1) if it is a legal menu description for an INPUT MENU statement, or FALSE (0) otherwise.

```
  text_str$ = '%multi,a,b,c'
  vrule$    = 'menu'
  print valid(text_str$, vrule$)
  end


1
```

- **ROUTINE**
  ROUTINE checks to see if the given routine name exists or not. TRUE (1) is returned if the routine name exists and FALSE (0) is returned if it does not exist.

```
  text$  = 'do_totals'
  text2$ = 'test_nos'
  vrule$ = 'routine'

  print valid(text$, vrule$)
  print valid(text2$, vrule$)
  end

  routine do_totals: private mytotal, desc$
    mytotal = 15
    desc$ = 'Test Totals'
    print desc$; mytotal
  end routine


1
0
```

- **FILTER**
  Filters the *text_str* before the validation. The following filters are allowed:
  - **LTRIM$** - Removes leading spaces.
  - **RTRIM$** - Removes trailing spaces.
  - **TRIM$** - Removes both leading and trailing spaces.
  - **UCASE$** - Converts to uppercase characters.
  - **LCASE$** - Converts to lowercase characters.
  - **REPLACE** - If *text_str* contains a few 'old_t1' characters, all of them will be replaced by 'new_t1'.

    ```
            old_t1=new_t1, old_t2=new_t2, ....
    ```

  - **REMOVE** - Removes any characters from the *text_str*.
  - **CHANGE** - Changes specified characters. For details see the CHANGE$ function.

```
  text$    = 'abcd10'
  vrules$  = 'filter remove "10"; letters'
  text2$   = 'ab1cd1'
  vrules2$ = "filter replace '1'='e';letters"
  text3$   = ' 1234  '
  vrules3$ = "filter trim; number"

  if  valid(text$, vrules$)  then print 'true'
  if  valid(text2$, vrules2$)  then print 'true'
  if  valid(text3$, vrules3$)  then print 'true'
  end

true
true
true
```

- **RESTORE** - Restores data to pre-filtered state for further validations. Can be used for multiple validation rules.

```
  text$  = "123"
  vrule$ = "filter replace '1'='a';restore; number"

  if  valid(text$, vrule$)  then print 'true'
  end

true
```

The following are file and structure access functions that SheerPower performs:

_CHANNEL returns the next available channel number.

Note on the following example:

In the following example, the text *'This is a test.'* will be printed out to a new file called 'test.txt' created in your SheerPower folder. To view the results of this example, open 'test.txt' in SPDEV *after* running it.

```
  out_ch = _channel
  open #out_ch: name 'sheerpower:test.txt', access output
  print #out_ch: 'This is a test.'
  close #out_ch
  end
```

_EXTRACTED tells how many records were extracted in the last extract.

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
    include cl(state) = 'CA'
  end extract
  print 'Number of Californians:'; _extracted
  end


Number of Californians: 7
```

FILEINFO$ parses a file specification and returns either a full file specification or specific file specification fields.

str_expr1 is the file specification to be parsed. If no file specification is given, the device and directory you are currently running from are returned.

str_expr2 is a list of field names, separated by commas, which are to be returned. The field names are:

CONTENTS	file contents
DEVICE	drive name
DIRECTORY	directory name
NAME	file name
TYPE	type or extension name
LOCATION	device and directory names
BACKUP_DATE	last file backup date
CREATION_DATE	date file was created
EXPIRATION_DATE	file expiration date
REVISION_DATE	date file was last modified
REVISION	the number of times a given file has been revised (given that the underlying OS supports this)
SIZE	the size of the file in bytes
ALL or ""	full file specification

str_expr3 is the default file specification. This parameter is optional.

FILEINFO$ can be used in various formats.

```
  print fileinfo$('x.y', 'ALL')
  print fileinfo$('', 'ALL')
  end


c:\sheerpower\x.y
c:\sheerpower
```

```
  x$ = 'sheerpower:samples\client'
  print fileinfo$(x$, 'ALL', '.ars')
  end


c:\sheerpower\samples\client.ars
```

```
  print fileinfo$('sheerpower:\samples\client', 'all', '.ars')
  print fileinfo$('sheerpower:\samples\client', 'location')
  print fileinfo$('sheerpower:\samples\client', 'location, name')
  print fileinfo$('sheerpower:\samples\client.ars')
  end


c:\sheerpower\samples\client.ars
c:\sheerpower\samples\
c:\sheerpower\samples\client
c:\sheerpower\samples\client.ars
```

The **CONTENTS** option of FILEINFO$ returns the entire contents of the file 'some_file.xxx' into all_of_file$ (see example below). If the file cannot be found, then returns a null string. A zero-length file will also return a null string.

FORMAT:

```
  all_of_file$ = fileinfo$('some_file.xxx', 'contents')
```

```
 all_of_file$ = fileinfo$('sheerpower:sheerpower.ini', 'contents')
 print all_of_file$
 end

! the specified file contents will display in the console window when the program
! is run:


[license]
LicenseKey=F0CE-2E43-7583-3130-3030-3030-3131-0003-873F-000A
Username=non-commercial
EmailAddress=non-commercial
```

```
sourcefile$ = 'source.xxx'
destfile$   = 'destination.xxx'
contents$   = fileinfo$(sourcefile$, 'contents')
open file dest_ch: name destfile$, access output, unformatted
print #dest_ch: contents$
close #x
end
```

Given a file name to find, FINDFILE$ returns the complete file specification of the first file found that matches the name given. If no file is found, the function returns a null string.

FINDFILE$ calls can be nested if the inner call has only one argument (i.e., the file specification, but no index number).

str_expr	The name of the file to search for. This can be just part of the full file specification.
int_expr	Which file specification to return if multiple files are found. This parameter is optional. The default is to return the first file found.
result	The complete file specification of the file found.

```
  print findfile$('sheerpower:\samples\*.spsrc')


c:\sheerpower\samples\cdplayer.spsrc
```

```
  do
    line input 'File specification': spec$
    if  _exit  then exit do
    for i = 1 to 9999
      file$ = findfile$(spec$, i)
      if  file$ = ''  then exit for
      print file$
    next i
  loop
  end


File specification? sheerpower:samples\client.*  <---- type this in
c:\sheerpower\samples\client.ars
c:\sheerpower\samples\client.def
c:\sheerpower\samples\client.fdl
c:\sheerpower\samples\client.str
File specification? exit
```

The following are debugging and exception handling functions that SheerPower performs:

_DEBUG returns a TRUE or FALSE value. TRUE if DEBUG is on. FALSE if DEBUG is off.

```
  debug on
  if  _debug  then
    print 'I am debugging'
  end if
  debug off
  if  _debug  then
    print 'I am still debugging.'
    else print 'I am no longer debugging.'
  end if
  end


I am debugging.
```

```
I am no longer debugging.
```

_STATUS returns the value given to SheerPower by the operating system for the last operating system request. It also returns the operating exception number for the last exception that occurred. This function is useful when debugging system level errors. The _STATUS function is often used with the SYSTEXT$ function.

```
  when exception in
    open #1: name 'c:\stuff\otherstuff\myfile.txt'
    close #1
  use
    print 'The error was: '; systext$(_status)
  end when
  end


The error was: The system cannot find the path specified.
```

Upon the completion of an INPUT MENU statement, the concept _STRING contains the menu path taken by the user when selecting the menu item. (i.e., "#2;#3" means the 3rd item of the 2nd submenu.)

```
  line1$       = '%width 12, %menubar, %autovbar ON,'
  line2$       = 'file = {new, get_file, save, save_as},'
  line3$       = 'edit = {cut, copy, paste},'
  line4$       = 'paragraph = {font, alignment, spacing, tabs, headers, footers},'
  line5$       = 'options = {ruler = {on, off}, side_bar = {on, off},'
  line6$       = 'view = {enlarged, normal, small}},exit'
  test_menu$   = line1$ + line2$ + line3$ + line4$ + line5$ + line6$
  the_default$ = ''

  do
    input menu test_menu$, default the_default$: ans$
    if  _exit  then exit do
    message 'Menu path was', _string
    the_default$ = _string
  loop
  end


+-------------------------------------------------------------------------------+
|  FILE      |   EDIT      |   PARAGRAPH   |   OPTIONS   |   EXIT               |
+---------------------------------------------+---OPTIONS---+-------------------+
                                             |  RULER    [>|
                                             |  SIDE_BAR +----VIEW-----+
                                             |  VIEW    |  ENLARGED   |
                                             +-----------|  NORMAL    |
                                                         |  SMALL      |
                                                         +-------------+


The menu path was:  #4;#3;#2
```

EXLABEL$ returns the routine name and line number executing when the last exception occurred, e.g., DO_INPUT.4

```
  try_it
  stop
  routine try_it
    when exception in
      open #1: name 'xx.yy'
    use
      print 'Open error at '; exlabel$
      print 'Error was: '; extext$(extype)
    end when
  end routine


Open error at TRY_IT.0002
Error was: File not found
```

EXTEXT$ returns explanatory text associated with a specified exception number. See also the EXLABEL$ function example above.

```
  print extext$(2001)


Subscript out of bounds
```

EXTYPE returns the number of the last exception that occurred. It is returned as an integer.

```
  try_it
  stop
  routine try_it
    when exception in
      open #1: name 'xx.yy'
    use
      print 'Open error at '; exlabel$
      print 'Error was: '; extype
    end when
  end routine


Open error at TRY_IT.0002
Error was: 7110
```

or

SYSTEXT$[(int_expr)]

SYSTEXT$ returns the text associated with the operating system status specified. If no *int_expr* is supplied, SheerPower returns the text for the last system status. SYSTEXT$ is often used with _STATUS system function.

```
  print systext$


The operation completed successfully.
```

The following are miscellaneous functions that SheerPower performs:

With the DECODE function, given the string representation of a number and the base that the value is in (*int_expr*), SheerPower returns the value in base 10. The number is returned as a real number. See also Section 6.4.11, ENCODE$(num_expr, num_int).

```
  do
    line input 'Enter a HEX value', default 'ff': hex$
    if  _exit  then exit do
    print 'Decimal value is'; decode(hex$,16)
  loop
  end

Enter a HEX value? ff
Decimal value is 255
Enter a HEX value? exit
```

The DTYPE function returns as an integer value, the data type of an expression or object variable: 1 = string, 2 = integer, 3 = real. See [Section 5.1, DECLARE] for more on declaring variable data types.

```
  declare object x
  x = 45.6
  print dtype(x)
  end


3
```

This function evaluates the expression described in *str_expr* and returns the result. If the variable being assigned the result is dynamic, the function puts the result in whatever data type the expression result was in. If the variable is NOT dynamic and the result is the wrong data type, a "Data type mismatch" error is returned.

```
  line input 'Enter an expression': a$
  print 'The result is '; eval(a$)
  end


Enter an expression?  5 + 4
The result is  9
```

FALSE returns the constant 0. It is returned as an integer. See also "TRUE function".

```
  input 'Enter your age': age
  of_age? = false
  if  age >= 18  then of_age? = true
  print 'Given your age, you are ';
  if  of_age?  then print 'plenty old!' else print 'too young!'
  end


Enter your age? 22
Given your age, you are plenty old!

run
Enter your age? 7
Given your age, you are too young!
```

Given an array and a dimension number, this function returns the low bound for that dimension. The default dimension is 1.

```
  dim temperature(-40 to 100)
  print 'Lowest temperature we can handle: '; lbound(temperature,1)
  print 'Highest temperature we handle    : '; ubound(temperature,1)
  end


Lowest temperature we can handle:  -40
Highest temperature we handle    :   100
```

MAXNUM returns the largest number available in this implementation of SheerPower.

```
  print maxnum


9223372046854775807
```

SIZE returns the number of elements in one dimension of an array.

array-name	Array to examine
int-expr	Dimension to get the size of. The default dimension is 1.

```
  dim calendar(366)
  print 'Size of this calendar: '; size(calendar)


Size of this calendar: 366
```

TRUE returns the constant 1. It is returned as an integer. See also "FALSE" function.

```
  input 'Enter your age': age
  of_age? = false
  if  age >= 18  then of_age? = true
  print 'Given your age, you are ';
  if  of_age?  then print 'plenty old!' else print 'too young!'
  end


Enter your age? 38
Given your age, you are plenty old!

rnh
Enter your age? 5
Given your age, you are too young!
```

Given an array and a dimension number, UBOUND returns the upper bound for that dimension. It returns an integer value. The default dimension is 1.

```
  dim temperature(-40 to 100)
  print 'Lowest temperature we can handle: '; lbound(temperature,1)
  print 'Highest temperature we handle    : '; ubound(temperature,1)
  end


Lowest temperature we can handle:  -40
Highest temperature we handle    :   100
```

The **PRINT** statement prints or displays text on the screen. The printed text can be formatted using a mask or directive and/or highlighted using video options. This section describes the various ways that text can be displayed on the screen.

FORMAT:

```
        PRINT [[AT row, col] [,ERASE] [,WIDE] [,BLINK] [,REVERSE]
        [,BOLD] [,USING "print_mask"]:] expr [{, | ;} expr...] [, | ;]
```

EXAMPLE:

```
  input name$
  print 'Hello, '; name$
  print bold: 'Here is a number: 1.93'
  end


? Rick
Hello, Rick
Here is a number: 1.93
```

DESCRIPTION:

The simplest version of the **PRINT** statement is:

```
        PRINT expr
```

expr is an expression to print. *expr* can be any SheerPower expression. SheerPower prints the value of the expression at the current cursor position and then generates a new line. A PRINT statement without an expression simply generates a new line.

```
  print 'Line 1'
  print
  print 'Line 3'
  end


Line 1

Line 3
```

One PRINT statement can print several items. Multiple items must be separated with a *comma* or a *semicolon*. The separator determines where the **next** expression will be printed.

Two additional features can be used to position the cursor:

- The *TAB* function positions the cursor at the column specified.
- The *AT* option positions the cursor at the coordinates specified. It can only be used once---at the beginning of the PRINT statement.

Semicolons

Separating print items with a **semicolon** causes the items to immediately follow one another. When the items are printed, no spaces appear between the expressions.

```
  alpha$ = 'ABCDEFGHIJKLM'
  bet$ = 'NOPQRSTUVWXYZ'
  print alpha$; bet$
  end


ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Commas and Print Zones

Print in columns by using **print zones.** Each print zone has a default width of twenty characters. To change the width, see Section 11.16.2, SET ZONEWIDTH.

```
        |-------------------|-------------------|-------------------|
        1                  20                  40                  60
```

Separating items with a comma causes the item **following** the comma to be printed in a new print zone. The terminal width determines the number of zones in each line. (See Section 11.9.1, ASK MARGIN statement to determine the terminal width.)

```
  input name_1$, name_2$
  print name_1$, 'MARY', name_2$
  end


? FRED, JOHN                       <------ type in FRED, JOHN
FRED                MARY                JOHN
```

If an item is longer than the zone width, SheerPower continues it into the next print zone. SheerPower uses as many print zones as necessary to print an item.

```
? FRED, DILLENSCHNEIDER & SONS
FRED                MARY                DILLENSCHNEIDER & SONS
```

SheerPower writes data sequentially. If an item is too long (over 132 characters) to write in one record, SheerPower continues it in the next record.

```
  open  #1: name 'test.txt', access output
  set   #1: margin 80
  print #1: repeat$('+-', 70)
  close #1
  open  #1: name 'test.txt', access input
  line input #1: record_1$, record_2$
  print 'Record 1: '; record_1$
  print 'Record 2: '; record_2$
  close #1
  end
```

```
Record 1:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
Record 2: +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The **TAB** function moves the print position to a specified column.

FORMAT:

```
        TAB(column)
```

Column is the column position to print. **TAB** positions the cursor at the column specified. TAB always moves forward (to the right). If the cursor is at or past the column specified, SheerPower jumps to the next line. The TAB function is a print item. Therefore, it must be separated from other print items with a comma or a semicolon.

```
  input name_1$, name_2$
  print tab(5); name_1$; tab(20); 'MARY'; tab(35); name_2$
  end


? FRED, JOHN
    FRED           MARY           JOHN
```

The **AT** option moves the cursor to a specified row and column on the screen.

FORMAT:

```
        AT row, column
```

Row is the row to print at. *Column* is the column to print at. Row and column can be any integer numeric constants.

```
  input name_1$, name_2$
  clear
  print at 3,10: name_1$; tab(20); 'Mary'; tab(35); name_2$
  end


? Fred, John               <------- type in 'Fred, John'


       FRED      MARY           JOHN
```

The **ERASE** option erases from the end of the printed text to the end of the line.

```
  print at 10, 1: 'Mary had a little lamb'
  delay 2
  print at 10, 1: 'Jeff'
  delay 2
  print erase, at 10, 1: 'Caroline'
  delay 2
  end


 Mary had a little lamb
.
.
.
Jeff had a little lamb
.
.
.
Caroline
```

The following rules govern the printing of numbers:

- Numbers are always followed by a space, even if the separator is a semicolon; positive numbers and zero are also preceded by a space.

```
  let x = 7
  let y = 10
  print x; y
  end


7  10
```

- Negative numbers are preceded by a minus sign.

```
  let x = -7
  let y = -10
  print x; y
  end


-7 -10
```

- If a number can be represented as an integer of 12 or fewer digits, it is printed as such.

```
  let x = 700000000001
  print x
  end


700000000001
```

SheerPower prints:

- 10 digits of precision for integer numbers
- 12 digits of precision for real numbers

If a separator (comma, semicolon, TAB function or AT option) is the last item in the PRINT statement, SheerPower advances the cursor to the position specified and **does not** generate a new line.

```
print tab(5); 7,
print 10;
print 20
end


    7              10  20
```

The **video attribute options** highlight text on the screen. Separate the options from the print list with a colon. The options are:

BLINK	causes the expressions in the print list to blink in low and high intensity
BOLD	causes the expressions in the print list to appear in bold (high intensity)
REVERSE	causes the print list to appear in reverse video

The video options can be combined. For example, the following program will print "Hello" boldfaced and in reverse video. It will then blink the word "Goodbye".

```
print bold, reverse: 'Hello'
print blink: 'Goodbye'
end
```

The **USING** option is used to format text. The print mask indicates the format for the data in the print list.

FORMAT:

```
        USING print_mask
```

The *print_mask* consists of **fields** or **directives** and text. The text can precede or follow a field or directive. The print mask tells how to print the expressions in the print list.

```
let a$  = "#.## ##.##" //<---- two fields
print using "amount = #.##": 1.9
print using a$: 1.93, -1.93
end


amount = 1.90
1.93 -1.93
```

The USING Print List

Expressions in the print list with the **USING** option are separated by commas. A trailing semicolon is allowed. The expressions are printed according to the format. However, a trailing semicolon causes SheerPower to leave the cursor at the end of the print line.

```
print using "###.## ###.##": 22.88, 45;
print " and others."
end


22.88  45.00 and others.
```

Fields are made up of format characters. The format characters tell SheerPower how to print the expressions in the print list.

character

The # is used to indicate a character position---a place in the format where a character can occur. For example:

```
  print using '#### ##': 'Test', 'Hi'
  end


Test Hi
```

In the above example, there are two fields. When the first string is printed, the word "Test" occupies all four character positions. When the second expression is printed (Hi), only two character positions are used.

If the string expression being printed is smaller than the field, the expression will be printed **centered** within the field.

```
  print using '#### ####': 'Test', 'Hi'
  print '123456789'
  end


Test  Hi
123456789
```

If the string expression is longer than the field, SheerPower generates an exception.

character

The # can also be used to specify digits. Each # represents one numeric digit position.

```
   print using "##": 19
   end


19
```

If more positions than the numeric expression contains are indicated, the expression will be right-justified and padded with spaces.

```
  print '1st 2nd 3rd'
  print using "### ### ###": 193, 19, 1
  end


1st 2nd 3rd
193  19   1
```

SheerPower prints a minus sign in front of negative numbers. SheerPower does not print a sign in front of positive numbers.

```
  print '1st 2nd 3rd'
  print using "### ### ###": 193, 19, -1
  end


1st 2nd 3rd
193  19  -1
```

If more positions to the left of the decimal point than the expression contains are indicated, the expression will be printed with leading spaces.

```
  print using "###.##": 1.9
  end


1.90
```

If more positions to the right of the decimal point than the expression contains are indicated, the expression will be printed with trailing zeros.

```
  print '--1-- --2--'
  print using "##.## ##.##": 1.3, 1.25
  end


--1-- --2--
 1.30  1.25
```

< character

The less than sign left-justifies text within a field. The less than sign must appear at the beginning of a field. The less than sign counts as a character position. In this example, justification occurs only in the second field.

```
  print using "#### <###": 'Test', 'Hi'
  print '123456789'
  end


Test Hi
123456789
```

In the above example, there are two fields. When the first string is printed, the word "Test" occupies all four character positions. The less than sign (<) causes SheerPower to left-justify the second expression.

> character

The greater than sign is used to right-justify text within a field. The greater than sign must appear at the beginning of a field. The greater than sign counts as a character position.

```
  print using "#### >###": 'Test', 'Hi'
  print '123456789'
  end


Test   Hi
123456789
```

In the above example, there are two fields. The greater than sign (>) causes SheerPower to right-justify the second expression.

@ character

The @ indicates one character position with no translation.

```
  print using '####': 0001
  print using '@@@@': 0001
  end


   1
0001
```

. character

You can include a decimal point in a number by putting a period or decimal point in the format.

```
  print using "###.##": 19.3
  end


19.30
```

, character

Include commas in your numbers by putting commas in the format.

```
  a$ = "##,###.##"
  print using a$: 28290.06
  print using a$: 8290.06
  print using a$: 290.06
  end


28,290.06
 8,290.06
   290.06
```

Commas cannot be used in exponential format.

% character

The **%** character pads on the left with zeros.

```
  print '-1- -2- -3-'
  print using "%%% %%% %%%": 193, 19, 1
  end


-1- -2- -3-
193 019 001
```

* character

The ***** character pads on the left with asterisks. This symbol can be used to set up check amounts.

```
  print using '***,***.**': 19.42
  end


*****19.42
```

If the expression is smaller than the format, SheerPower will right justify the expression and pad it with **asterisks**.

```
  print '-1- -2- -3-'
  print using "*** *** ***": 193, 19, 1
  end


-1- -2- -3-
193 *19 **1
```

+ character

A plus sign causes SheerPower to print a leading plus or minus sign. SheerPower will print a plus sign in front of positive numbers and a minus sign in front of negative numbers.

The "+" sign adds a character position to the format. The character position is used for the sign of the number.

```
  print ' -1-  -2-  -3-'
  print using "+### +### +###": 193, 19, -1
  end


 -1-  -2-  -3-
+193  +19   -1
```

- character

The - character prints a leading or trailing minus sign for negative numbers, and a leading space for positive numbers. The "-" adds a character position to the format. The character position is used to print the minus sign or space.

```
  print ' -1-  -2-  -3-'
  print using "-### -### -###": 193, 19, -1
  end


-1-  -2-  -3-
193   19   -1
```

~ character

The ~ (tilde) character marks the character following it as literal data.

```
  print using '###~-###~-####': '5556667777'
  end


555-666-7777
```

$ character

The $ character prints a floating dollar sign. The dollar sign appears before the number. $ causes SheerPower to print '$-' for negative numbers and '$' for positive numbers. The minus sign appears immediately **after** the dollar sign and **before** the number.

```
  print "1st col 2nd col"
  print using "$###.## $###.##": 11.93, -1.93
  end


1st col 2nd col
 $11.93  $-1.93
```

$+ characters

$+ characters print a floating dollar sign. The dollar sign appears before the numeric expression. $+ causes SheerPower to print a minus sign before negative numbers, and a plus sign before positive numbers. The sign appears after the dollar sign and before the number.

```
  print "1st  col 2nd  col"
  print using "$+###.## $+###.##": 11.93, -1.93
  end


1st  col 2nd  col
 $+11.93   $-1.93
```

-$ characters

-$ characters print a floating dollar sign. The dollar sign appears immediately before the numeric expression. -$ causes SheerPower to print a minus sign before negative numbers and a **space** before positive numbers. The minus sign or space appears immediately before the dollar sign.

```
  print "1st  col 2nd  col"
  print using "-$###.## -$###.##": 11.93, -1.93
  end


1st  col 2nd  col
  $11.93   -$1.93
```

+$ characters

+$ causes SheerPower to print a floating dollar sign. The dollar sign appears immediately before the number. +$ causes SheerPower to print a plus sign before positive numbers and a minus sign before negative numbers. The plus or minus sign appears immediately before the dollar sign.

```
  print "1st  col 2nd  col"
  print using "+$###.## +$###.##": 11.93, -1.93
  end


1st  col 2nd  col
 +$11.93   -$1.93
```

Notice that +$ adds two character positions to the format. One position contains the dollar sign, the other contains the plus or minus sign.

$- characters

$- characters prints a floating dollar sign. The dollar sign appears before the number. $- causes SheerPower to print a minus sign before negative numbers and a **space** before positive numbers. The minus sign or space appears after the dollar sign and before the number.

```
  print "1st  col 2nd  col"
  print using "$-###.## $-###.##": 11.93, -1.93
  end


1st  col 2nd  col
 $ 11.93   $-1.93
```

If your expression is too large to fit in a field, SheerPower gives an exception.

The **directives** used with the USING option of the PRINT statement tell SheerPower what to do with the text.

FORMAT:

```
        PRINT USING 'directive' : str_expr
```

The **UCASE** directive converts the str_expr to uppercase characters.

```
  print using '{ucase}?' : 'march'
  end


MARCH
```

The **LCASE** directive converts the str_expr to lowercase characters.

```
  print using '{lcase}?' : 'MARCH'
  end


march
```

The **HYPHEN** directive causes SheerPower to suppress the hyphen character if it is the last non-blank character after the format is applied.

```
  print using '<#####~-####' : '92123'
  print using '{hyphen}<#####~-####' : '92123'
  end


92123 -
92123
```

Given a str_expr that contains a date in the format YYMMDD or CCYYMMDD, the **DATE** directive converts the str_expr to a default or specified, optionally-masked date format.

These date arguments can be used: YMD, CYMD, MDY, MDCY, DMY, DMCY, DMONY, DMONCY, MONTHDY, MONTHDCY. If no argument is provided, the default is MDCY. (See Section 6.4.12, FORMAT$(expr, str_expr) for examples of date argument usage.)

To format the resulting data, include a **?** in the print mask.

```
print using '{date}?': '990122'
print using '{date dmy}?': '990122'
print using '{date dmcy}?': '990122'
print
print using '{date mdy}?': '20000115'
print using '{date mdy}##/##/##': '20000115'
print using '{date mdcy}##/##/####': '20000115'
end


01221999
220199
22011999

011500
01/15/00
01/15/2000
```

The **ROTATE** directive rotates the last n characters in a str_expr to the first position in the str_expr. Optionally, the resulting str_expr can be masked by replacing the **?** with a print mask.

```
print using '{rotate 3}?': '5552527800'
print using '{rotate 3}###~ ###~-####': '5552527800'
print
print using '{rotate 5}?': 'TuneTommy'
print using '{rotate 5}#####~ ####': 'TuneTommy'
end


8005552527
800 555-2527

TommyTune
Tommy Tune
```

Given a str_expr containing a 4-digit time in HHMM or HH:MM format or a 6-digit time in HHMMSS or HH:MM:SS format, the **TIME** directive converts the str_expr to HH:MM AM/PM or HH:MM:SS AM/PM.

```
print using '{time}?': '1022'
print using '{time}?': '19:45'
print
print using '{time}?': '102255'
print using '{time}?': '19:45:36'
end


10:22 AM
07:45 PM

10:22:55 AM
07:45:36 PM
```

Given a str_expr containing a 5-, 6- or 9-digit Zip code, the **ZIPCODE** directive converts the str_expr to an appropriate Zip code format.

```
  print '5 character zipcode : ';
  print using '{zipcode}?': '92126'
  print '6 character zipcode : ';
  print using '{zipcode}?': 'K8A3P9'
  print '9 character zipcode : ';
  print using '{zipcode}?': '931327845'
  end


5 character zipcode : 92126
6 character zipcode : K8A 3P9
9 character zipcode : 93132-7845
```

The **MESSAGE** statement prints a message at line 23 (the default line) on the screen.

FORMAT:

```
        MESSAGE [ERROR: | DELAY:] expr [; | , expr]
```

EXAMPLE:

```
  print at 1,1:
  do
    message 'Enter EXIT to exit'
    input 'Please enter your name': name$
    if  _exit  then
      message 'The End'
      exit do
    else
      print name$
      repeat do
    end if
  end do
  end


Please enter your name?

               Enter EXIT to exit       (first message)

Please enter your name? Tester
Tester
Please enter your name? exit

                    The End             (second message)
```

DESCRIPTION:

SheerPower displays messages at the bottom of the screen. Below the message line there is a scrollable MESSAGE HISTORY window. Error messages are displayed in red. The **MESSAGE** statement can be used to display your own messages and errors on this line.

The MESSAGE statement can print several items. Each item can be any SheerPower numeric or string expression. Multiple items must be separated with a comma or a semicolon. The separator determines where the *next* expression will be printed.

Semicolons

Separating message items with a semicolon causes the items to immediately follow one another. When the items are printed, no spaces appear between the items.

Commas

Separating items with a comma puts a space between each item.

SheerPower would display this message:

```
MESSAGE 'number is', 123; 456; 789
```

as:

```
number is 123456789
```

ERROR Option

SheerPower displays a message for at least three seconds before clearing the message. When the **ERROR** option is used the following things occur:

- a yellow box displaying the error message pops up on the screen
- an error tone is heard
- typeahead is purged (Typeahead is the feature that accepts characters typed ahead of the computer's request for input.)

FORMAT:

```
MESSAGE ERROR: expr [; | , expr]
```

EXAMPLE:

```
  print at 1,1:
  input 'Enter your age': age$
  message error: 'Is this really your age?'
  end


Enter your age? 99

            Is this really your age?
```

DELAY Option

The **DELAY** option of the MESSAGE statement causes SheerPower to set an automatic delay, giving the user time to view the message before clearing the message. Starting with a minimum delay of approximately three seconds, SheerPower increases the delay a little for lengthier messages.

FORMAT:

```
MESSAGE DELAY: expr [; | , expr]
```

EXAMPLE:

```
z$ = 'This is a very, very, very, very, very, very long message'
message delay: z$
message delay: 'Short message'
end


 This is a very, very, very, very, very, very long message

                      Short message
```

FORMAT:

```
        DELAY [num_expr]
```

EXAMPLE:

```
print 'Waiting a bit'
delay 4.5
print 'Done'
end


Waiting a bit
Done
```

PURPOSE:

Use **DELAY** when you need to cause a timed delay before continuing program execution; for instance, to give the user time to read a message before clearing the screen.

DESCRIPTION:

DELAY causes SheerPower to pause for the specified number of seconds before continuing program execution. The numeric expression (*num_expr*) can be a whole number or a fraction. For example:

```
        delay 3.5
```

The resolution of DELAY is +/- 10th of a second.

If *num_expr* is omitted, SheerPower prints this message at the bottom of the screen:

```
                    Press the ENTER key to continue
```

and waits for the user to respond.

If, at the "Press ENTER..." prompt, a user enters:

[Ctrl/Z]	_EXIT is set to TRUE (1)
[esc] or UP-arrow	_BACK is set to TRUE (1)
[Help]	_HELP is set to TRUE (1)

A DELAY that waits for the `[Enter]` key can also be completed by a MOUSE click.

The **CLEAR** statement can be used to clear the SheerPower 4GL screen (everything within the SheerPower 4GL screen) or to clear a specific area of the screen. CLEAR can be used to clear any rectangular area within the screen. This statement clears the screen before executing code or printing information on the screen.

FORMAT:

```
CLEAR [AREA [BOX] [, attr_list:] row_1, col_1, row_2, col_2]
```

EXAMPLE:

```
  clear
  input 'Please enter your name': name$
  print 'Hello, '; name$
  end


Please enter your name? Tester
Hello, Tester
```

DESCRIPTION:

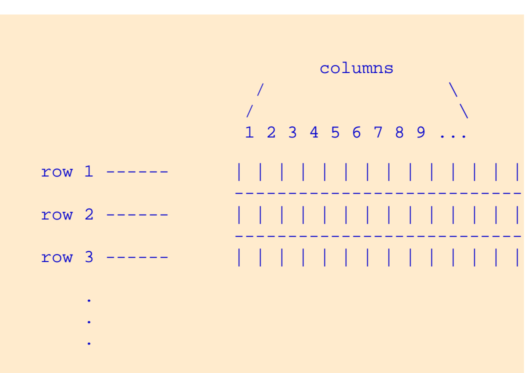
CLEAR, by itself, clears all text from the screen. It removes any message text that is displayed within the screen.

AREA Option

The **AREA** option clears a specific section of the screen. The cleared area is rectangular in shape.

row specifies a vertical position on the screen. Rows are numbered sequentially from the top of the screen to the bottom. The default setting for number of rows is 30.

col specifies a column--a horizontal position on the screen. Columns are numbered sequentially from the first character position on the left of the screen to the last character position on the right of the screen. The default setting of columns is 80.



Two coordinates must be specified. These coordinates mark opposite corners of a rectangular area. SheerPower clears the rectangular area specified by these two coordinates. For instance, the statement CLEAR AREA 2,3,8,20 would clear a rectangular area:

```
   1st coordinate (2,3)   +----------------+
                          |                |
                          |                |
                          |                |
                          |                |
                          |                |
                          +----------------+   (8,20) 2nd coordinate
```

The first coordinate (2,3) indicates the upper left corner. The second coordinate (8,20) indicates the lower right corner.

FORMAT:

```
        CLEAR AREA [, attr_list:] row_1, col_1, row_2, col_2
```

EXAMPLE:

```
clear area reverse: 5, 10, 11, 60
print at 7, 20: 'Cleared area is in Reverse video'
end


                  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
                 |                                                 |
                    Cleared area is in Reverse video
                 |                                                 |

                 |                                                 |
                  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
```

CLEAR AREA allows the following attributes to be used when clearing an area: **BOLD**, **BLINK**, **REVERSE**. Multiple attributes used in one statement are separated by commas.

BOX Option

The **BOX** option creates an empty box with a frame.

The **BOLD**, **BLINK**, and **REVERSE** attributes can also be used with the BOX option. Separate attributes in one statement with commas.

FORMAT:

```
        CLEAR AREA BOX [, attr_list:] row_1, col_1, row_2, col_2
```

EXAMPLE:

```
clear area box, bold: 5, 10, 11, 60
print at 7, 20: 'Inside the box'
print at 12, 1: 'Outside the box'
end


              +--------------------------------------------------+
              |                                                  |
              |         Inside the box                           |
              |                                                  |
              |                                                  |
              |                                                  |
              +--------------------------------------------------+
      Outside the box
```

This chapter describes the various ways that data can be entered at the pc and stored into variables.

FORMAT:

```
        [line] input var, var...

        [key] [line] input [  ['Prompt_text']
          [, prompt str_expr] [, erase]
          [, at row, column] [, length num_expr] [, default str_expr]
          [, VALID str_expr] [, timeout time_limit] [, elapsed num_var]
          [, area num_expr, num_expr, num_expr, num_expr] [, attributes attr_list]
          [, screen '[text] <format>...']
          [, dialogbox str_exp,]
          [, menu str_expr: str_var] :] var [,var. . .]
```

EXAMPLE:

```
  input 'Please enter your first name': first$
  input 'Now, enter your last name': last$
  line input 'Where do you live': city$
  print
  print 'Hello '; first$; ' '; last$
  print 'From '; city$
  end


Please enter your first name? Sunny
Now, enter your last name? Day
Where do you live? San Diego, California

Hello Sunny Day
From San Diego, California
```

PURPOSE:

The **INPUT** statement is used to ask questions from the user and store the answers for use in a program.

DESCRIPTION:

The **INPUT** statement reads data typed by the user and assigns it to variables. *var* is a variable that the data is being assigned to. When SheerPower executes an INPUT statement, it prints any prompt given and waits for the user's response. The user's response is then

assigned to the variable(s) specified.

For information on INPUT from a text file, see Chapter 14, File Handling.

The user enters data in response to the INPUT statement. The input data must be the same data type as the variable, or SheerPower generates an exception. If, in response to the INPUT statement, the user presses the `[Enter]` key and:

- the variable is numeric, SheerPower assigns it a 0
- the variable is a string, SheerPower assigns it a null string ("")

There are three types of INPUT statements:

- **INPUT**
  The INPUT statement reads one or more comma-separated data items.
- **LINE INPUT**
  The LINE INPUT statement reads an entire line of data.
- **KEY INPUT**
  The KEY INPUT statement reads a single keystroke.

There are four input styles:

1. Simple input

```
  input 'Please enter your name': name$
  print 'Hello '; name$
  end


Please enter your name? Toby
Hello Toby
```

2. Formatted data entry screens (see Section 8.17, SCREEN Option)

```
  input 'Please enter your name': name$
  input screen 'Soc. sec.: <DIGITS: ###-##-####>': ssn
  print name$, 'Social security number:'; ssn
  end


Please enter your name? Fred
Soc. sec.:  ___-__-____        (before input)
Soc. sec.:  324-11-4533        (after input)
Fred                Social security number: 324114533
```

3. Free format multi-line text input (see Section 8.15, AREA Option)

```
  line input area 5, 10, 8, 60: text$
  print at 10, 1: 'Rewrapped text'
  wt$ = wrap$(text$, 1, 30)
  print wt$
  end


This is an example of wrapped text.  The text is
wrapping.__________________________________________
___________________________________________________
___________________________________________________

Rewrapped text
This is an example of wrapped
text.  The text is wrapping.
```

4. Pop-up menus (see Section 8.18, MENU Option)

```
sample_menu$ = '%title "Options",' &
    + 'Add, Change, Delete, Inquire'
line input menu sample_menu$: selection$
print 'Menu option was '; selection$
end


        +--Options---+
        |  ADD       |
        |  CHANGE    |
        |  DELETE    |
        |  INQUIRE   |
        +------------+

      Menu option was CHANGE
```

The **INPUT** statement has the following options, which are described in detail in following sections of this chapter:

PROMPT	displays the prompt text
AT row, col	positions the cursor on the desired row and column
LENGTH nn	limits the number of characters that a user can type
DEFAULT	lets you provide defaults for INPUT statements
VALID	validates user responses
TIMEOUT	limits the time the user has to respond to the INPUT prompt
AREA	does free format multi-line text input from an area on the screen
SCREEN	creates formatted data entry screens
MENU	displays and receives input from "pop-up" menus
ELAPSED	keeps track of the time it takes the user to respond to an INPUT prompt
ATTRIBUTES	displays in BOLD, BLINK, REVERSE
ERASE	clears the input line prior to accepting input and after the input has been completed
DIALOGBOX	presents the end user with simple to complex input forms. See Chapter 9, Input Dialogbox - Creating GUI Forms with SheerPower

At an input prompt, there are several commands and keystrokes that the user can enter instead of an answer to the prompt. These are:

EXIT or `[Ctrl/Z]`

Type the word **EXIT** or press `[Ctrl/Z]` to exit from a prompt or procedure. If one of these options is entered, SheerPower sets the internal variable **_EXIT** to TRUE (1).

`[esc]` or UP-arrow

If **`[esc]`** or the **Up-arrow** key is entered, the internal variable **_BACK** will be set to TRUE (1).

HELP or `[Help]`

Get help for an input item by typing the word **HELP** or by pressing the `[Help]` button in the toolbar. If one of these options is input, SheerPower sets the internal variable **_HELP** to TRUE (1).

_HELP should be checked before _BACK and/or _EXIT because, in some cases, all three are set on. For example, if "EXIT" is the default and the [Help] key is pressed, both _HELP and _EXIT are set on.

The internal variables _BACK, _EXIT and _HELP can be examined within a program and appropriate action can be taken.

The following keys allow the user to edit input:

Left-arrow	move left

Right-arrow	move right
Delete key	delete character(s) to the left of the cursor
TAB	move to next input field or terminate input
[CTRL/A]	toggle between insert and overstrike mode
[CTRL/E]	move to end of line
[CTRL/H]	move to beginning of line
[CTRL/J]	delete the word to the left of the cursor.
[CTRL/U]	delete all the characters on a line.
[Enter]	terminate the input; [Enter] can be pressed anywhere in the line, not only at the end of the line.

If the INPUT is to a string variable, the user can enter an unquoted or a quoted string. If the user enters an unquoted string, SheerPower ignores any leading spaces, trailing spaces, or tabs. An unquoted string cannot contain commas.

If the user enters a quoted string, SheerPower removes the quotes, and includes any leading or trailing spaces and tabs.

```
  input 'Enter your name': name1$
  input 'Enter your name in quotes': name2$
  input 'Enter your name in quotes with spaces': name3$
  input 'Enter last name, comma, space, first name in quotes': name4$
  print
  print name1$
  print name2$
  print name3$
  print name4$
  end


Enter your name? Tony
Enter your name in quotes? 'Tony'
Enter your name in quotes with spaces? '  Tony  '
Enter last name, comma, space, first name in quotes? 'Smith, Tony'

Tony
Tony
  Tony
Smith, Tony
```

A single INPUT statement can be used to input several variables. The input items and variables must be separated with commas.

```
  input 'Enter 3 names separated by commas': name1$, name2$, name3$
  print name1$, name2$, name3$
  end


Enter 3 names separated by commas? Tom, Sue, George

Tom                 Sue                 George
```

If an INPUT statement contains a list of variables, SheerPower asks for input until all of the variables have a value. If the user enters less data or more data than there are variables, SheerPower generates an exception. If an exception occurs, SheerPower restarts from the beginning.

Users can enter the data for a variable list in one of the following two ways:

1. The user can enter each piece of data on a separate line by typing a comma as the last character on each continuing line. The comma tells SheerPower that there is more data to come.

```
Enter 3 names separated by commas? Tom,
? Sue,
? George

Tom                 Sue                 George
```

2. Users can enter the data on one line by separating the data with commas as shown in the first example above.

Whichever method is used, SheerPower will continue accepting input data until all the variables have values.

Validating Data

It is best to not input directly into a structure(field)---but to instead always go through an intermediate variable that is a numeric or string variable:

Instead of:

```
line input 'Last name': client(last_name)
```

Use:

```
line input 'Last name': last_name$
client(last_name) = last_name$
```

This validates the data prior to storing it into the structure. For example:

```
do
  input 'Last name': last_name$
  if _exit then exit do
  if len(last_name$) < 2 then
    message error: 'Too short of a name'
    repeat do
  end if
  client(last_name) = last_name$
end do
```

A number of variables can be input with one **LINE INPUT** statement. Simply list the variables separated by line terminators.

```
  line input 'Enter a comment, press Enter, enter a comment': &
       comment_1$, comment_2$
  print
  print comment_1$
  print comment_2$
  end


Enter a comment, press Enter, enter a comment? This is the first comment
? This is the second comment

This is the first comment
This is the second comment
```

SheerPower asks for input until all of the variables listed have received a value.

Unlike the INPUT statement, commas cannot be used to separate answers. Each variable is prompted for separately. If a comma is in a user's response, the comma is just taken as part of the text.

By default, SheerPower prints a question mark and a space and then waits for the user's response. To display prompt text before the question mark, enclose the prompt text in quotes and follow it with a colon. The colon separates the prompt text from the variable(s). The prompt text must follow the keyword INPUT and must be separated from the variable list by a colon.

When SheerPower executes the INPUT statement, it prints the prompt text ("Your name" in the example below) followed by a question mark and a space.

```
  input 'Your name': name$
  print name$
  end


Your name? Fred
Fred
```

The **PROMPT** option displays the specified prompt text *without* the question mark and space.

FORMAT:

```
        PROMPT str_expr
```

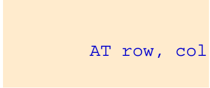
str_expr is a string expression which contains the prompt text. *str_expr* can be any string expression. The prompt text is separated from the variable list with a colon (:).

EXAMPLE:

```
  input prompt 'Please enter your name: ': name$
  print 'Hello '; name$
  end


Please enter your name: Jackie
Hello Jackie
```

The **AT** option positions the cursor on the specified row and column. This is the position where the INPUT statement starts the prompt, not where the user enters data.

FORMAT:



```
        AT row, col
```

row is the row to print at. *col* is the column to print at. *row* and *col* can be any integer numeric constants.

EXAMPLE:

```
  print at 1, 1:
  input at 3, 10, prompt 'Please enter your name: ': name$
  print 'Hello '; name$
  end


      Please enter your name: Jack    <-----this line gets printed at row 3, column
10
Hello Jack
```

The **ATTRIBUTES** option allows input with attributes. The available attributes are:

BOLD
BLINK
REVERSE

Multiple attributes used in one INPUT statement are separated by commas.

FORMAT:

```
        ATTRIBUTES attr_list
```

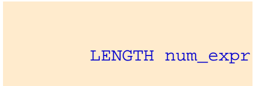
attr_list contains a list of input attributes.

EXAMPLE:

```
  name_attr$ = 'bold, blink'
  line input 'Enter your name', attributes name_attr$: name$
  print 'Hello '; name$
  end


Enter your name? Susan
Hello Susan
```

The **LENGTH** option limits the number of characters that a user can enter. It causes SheerPower to display underscore characters following the prompt. The number of underscore characters is the length.

FORMAT:



```
        LENGTH num_expr
```

num_expr is the number of characters the user can enter.

EXAMPLE:

```
input 'Enter your name', length 15: name$
input 'Enter a city', length 20: city$
print name$, city$
end


Enter your name? Betty__________
Enter a city? 'San Diego'_________
Betty                San Diego
```

DEFAULT lets you provide defaults for INPUT statements. SheerPower automatically formats the default appropriately. The user can press `[Enter]` to accept the default you provide.

FORMAT:

```
        DEFAULT str_expr
```

str_expr is a string expression that will be used as the default. When SheerPower executes an INPUT statement with a default, it prints the default after the prompt.

EXAMPLE:

```
input 'Enter the state code', default 'CA': state$
print 'The state was: '; state$
end


Enter the state code? CA
The state was: CA
```

If the user does not want the default, they can simply type over the default text.

When performing an **INPUT MENU**, the **DEFAULT** option can be used to specify a default menu path. The default takes the format of:

```
        #item1;#item2;...
```

#item1 is the number of the item on the first menu, *#item2* is the number of the item on the second menu, and so on.

Upon the completion of a INPUT MENU statement, the concept **_STRING** contains the menu path taken by the user when selecting the menu item; i.e., **#2;#3** means the 3rd item of the 2nd submenu. (For information on menus, see Section 8.18, MENU Option.)

EXAMPLE:

```
line1$ = '%width 12, %menubar,'
line2$ = 'file = {new, get_file, save, save_as},'
line3$ = 'edit = {cut, copy, paste},'
line4$ = 'paragraph = {font, alignment, spacing, tabs, headers, footers},'
line5$ = 'options = {ruler = {on, off}, side_bar = {on, off},'
line6$ = 'view = {enlarged, normal, small}},exit'
test_menu$ = line1$ + line2$ + line3$ + line4$ + line5$ + line6$
the_default$ = ''
do
  input menu test_menu$, default the_default$: ans$
  if  _exit  then  exit do
```

```
    message 'Menu path was', _string
    the_default$ = _string
  loop
  end


+--------------------------------------------------------------------------------+
|  FILE      |   EDIT      |   PARAGRAPH   |   OPTIONS    |    EXIT              |
+---------------------------------------------+---OPTIONS---+-------------------+
                                             |  RULER    [>|
                                             |  SIDE_BAR +----VIEW-----+
                                             |  VIEW      |  ENLARGED   |
                                             +-----------|  NORMAL     |
                                                         |  SMALL      |
                                                         +-------------+
```

The menu path was: **#4;#3;#2**

The **ERASE** option clears the input line prior to accepting input. After input has been completed, the input line is cleared again.

```
  print at 1,1:
  input 'Please enter a name', at 3,1: name_1$
  input 'Enter a 2nd name', at 4,1, erase: name_2$
  print '1st name: '; name_1$
  print '2nd name: '; name_2$
  end


Please enter a name? James

1st name: James
2nd name: Tony
```

The **VALID** option validates user responses according to specified validation rules.

FORMAT:

```
        VALID str_expr
```

str_expr is the list of validation rules.

Refer to Section 6.6.6, VALID(text_str, rule_str) for information on all the validation rules.

EXAMPLE:

```
  input 'Enter name', length 20: name$
  input 'Enter age', length 5, valid 'integer': age$
  print name$, age$
  end


Enter name? Aaron_______________
Enter age? 32___
Aaron                32
```

The **TIMEOUT** option limits the time the user has to respond to the INPUT prompt. A time limit must be specified with the TIMEOUT option. If the user does not complete the INPUT statement within the specified time, an exception ("timeout on input at xx") is generated.

FORMAT:

```
        TIMEOUT num_expr
```

num_expr is a numeric expression that represents the number of seconds allowed for the response.

EXAMPLE:

```
  input 'Name', timeout 4.5: name$
  end


Name? Timeout on input at 10
```

TIMEOUT 30 gives the user approximately 30 seconds to enter a name. Fractions of a second can be indicated by including decimal digits. TIMEOUT 4.5 gives the user approximately 4.5 seconds to enter a name.

The **AREA** option does input from an area on the screen.

FORMAT:

```
        LINE INPUT AREA top, left, bottom, right: str_expr
```

Top	Top row of the area on the screen
Left	Left column of the area
Bottom	Bottom row of the area
Right	Right column of the area

EXAMPLE:

```
  line input area 5, 10, 8, 60: text$
  print at 10, 1: 'Rewrapped text'
  wt$ = wrap$(text$, 1, 30)
  print wt$
  end


        This is an example of wrapped text.  The text is
        wrapping.__________________________________________
        ___________________________________________________
        ___________________________________________________

Rewrapped text
This is an example of wrapped
text.  The text is wrapping.
```

A BELL is outputted if user-entered text goes outside the area.

During a LINE INPUT AREA, the following features are available:

- automatic word-wrap while text is being entered
- insert and delete capability
- use of LEFT, RIGHT, UP and DOWN arrow keys

While you are entering text in an area, you can use the following commands:

Text	To enter text, just type it
Arrow keys	To move around, press UP, DOWN, LEFT, RIGHT
`[CTRL/H]`	Moves cursor to beginning of line
`[CTRL/E]`	Moves cursor to end of line
`[esc]`	Exit (abort) the input
`[Help]`	Exits from the input and sets the variable _HELP to true

The "escape" `[esc]` keystroke has meaning if it is in the first position of the area (top left corner).

When a LINE INPUT AREA is completed, SheerPower removes any trailing line feeds and spaces.

Field size limit

Because of line breaks, etc. it is possible to reach the field size limit before reaching the fill-in limit. A BELL is outputted when you have entered TOTAL characters that reach the field size (including line separators, etc.).

INPUT DIALOGBOX provides the user with all the power of HTML forms without using a web browser. For a complete discussion of INPUT DIALOGBOX please see Chapter 9.

FORMAT:

```
INPUT DIALOGBOX str_exp: str_var
```

str_expr specifies the dialogbox format.

The **SCREEN** option is used to create formatted data entry screens. The SCREEN option lets the user enter data into a fill-in area.

FORMAT:

```
SCREEN str_expr
```

str_expr specifies the screen format.

EXAMPLE:

```
  input 'Your name, please': name$
  input screen 'Soc. sec.: <DIGITS:###-##-####>': SSN
  print name$, 'Social security number:'; SSN
  end


Your name, please? John
Soc. sec.: ___-__-____
```

When the program executes, SheerPower displays the prompt "Soc. sec.:" and a fill-in area with underscore characters where the user must fill in a number. The fill-in area is highlighted. As the user enters the social security number, it appears in the fill-in area and the underscores disappear. When the `[Enter]` key is pressed, the number is assigned to the variable SSN.

```
Your name, please? John
Soc. sec.: 555-48-6662
John                Social security number: 555486662
```

The SCREEN option can be used with any data type. A string expression follows the keyword SCREEN. The expression contains any text to print and the format for the fill-in areas.

A number of commands can be used within the format to control how the fill-in area works. These fall into two categories:

- format options
- format characters

Format options come first. They are followed by a colon and then by format characters.

#

The # is used to specify **digits**). Wherever a # sign appears, only these types of characters are allowed as input. For example, the # might be used for a phone number:

```
input screen '<:(###) ###-####>': phone
print phone
end
```

When SheerPower executes this program, it displays the following fill-in area:

```
(___) ___-____


(555) 554-7879
5555547879
```

##

The @ is used to specify any printable character, including letters and numbers.

```
input screen 'License #: <@@@ @@@@>': license$
print 'License: '; license$
end

License #: INF 5783          <-------- type in a license number
License: INF5783
```

.

A **decimal point** is used to align the fractional portion of a floating point number. If the screen format is in the form <###.####>, when the user presses the `[ . ]` key the digits will be right-justified to the decimal point. When the field is complete, the digits to the right of the decimal will be left-zero filled.

```
print 'Input .59, please'
input screen 'Amount: <###.####>': amt$
print 'Amount = '; amt$
end


Amount:    .         <------ type in .59 here
Amount = .5900
```

^

The ^ is used to specify an UPPERCASE letter. If the next character inputted is a lowercase letter, SheerPower will change it to uppercase.

```
input screen 'First name: <^@@@@@@@@@@@@@>': first$
input screen 'Middle initial: <^>.': middle$
input screen 'Last name: <^@@@@@@@@@@@@@@@>': last$
print first$; ' '; middle$; '. '; last$
end


First name: John___________           <-------- type in john
Middle initial: B.                    <-------- type in b.
Last name: Smithy___________          <-------- type in smithy
John B. Smithy
```

```
input screen 'Comparison: <### ~> ###>': a$
print a$
end
```

If format options are used, they must be placed *before* the format characters and followed by a colon. The following format options are available:

UCASE

UCASE uppercases all letters that are typed. If a letter is typed in lowercase, it is changed to uppercase.

```
print 'Type in some text!'
input screen '<ucase:@@@@@@@@@@@@@@@@@@@@>': text$
print 'Here is your text: '; text$
end


Type in some text!
JOHN B. SMITHY______                 <-------- type in 'john b. smithy
Here is your text: JOHN B. SMITHY
```

LCASE

LCASE lowercases all letters that are typed. If a letter is typed in uppercase, it is changed to lowercase.

```
  print 'Type in some text!'
  input screen '<lcase: @@@@@@@@@@@@@@@@@@@@>': text$
  print 'Here is your text: '; text$
  end


Type in some text!
john b. smithy                  <-------- type in JOHN B. SMITHY
Here is your text: john b. smithy
```

NOECHO

With the *NOECHO/bold* screen format option, typed characters are not echoed (printed) on the screen.

```
  input screen '<noecho:Password? @@@@@>' : pw$
  print pw$
  end


Password? _____
Elene
```

DIGITS

The **DIGITS** screen format option allows only digits to be entered in the field. The format character # also allows the minus sign and the decimal point.

```
  input screen 'Phone: <digits:###-####>': phone
  print phone
  end


Phone: 555-6798
 5556798
```

AJ

AJ causes SheerPower to jump automatically to the next field when the current field has been filled in. The field must be completely filled in before SheerPower will jump to the next field.

```
  input screen 'Phone: <AJ, digits:###-####>': phone
  input screen 'Soc. sec.: <digits:###-##-####>': ssn
  print phone, ssn
  end


Phone: 555-3839
Soc. sec.: 899-75-3432
 5553839            899753432
```

REQ

REQ specifies that input is required and something must be entered. The computer will beep and prompt until valid input is entered.

AT row, column

The **AT** option displays the field on the screen at the row and column position specified. *row* specifies the row to print at. *column* specifies the column to print at.

Defaults can be supplied for each field in the screen. The format of the defaults is a list of values separated with line feeds. Defaults are set up in the following example.

```
  city$ = 'St Paul'
  state$ = 'Minnesota'
  df$ = city$ + chr$(10) + state$
  clear
  scr$ = '<at 6, 10: City @@@@@@@@@@@@@@@@@@@@@@@@@@>' &
      +'<at 7, 10: State  ^^>'
  input screen scr$, default df$: city$, state$
  end


City     St. Paul
State    MI
```

VALID

The **VALID** option allows full validation of each field on the screen. (See the VALID function in Chapter 6 for a list of validation rules.) The format of the VALID option is:

```
        VALID 'rule_str'
```

rule_str is the list of validation rules. Multiple validation rules are separated by semicolons.

```
  input screen 'code = <valid "integer":###-##-##>': ans$
  print ans$
  end


code = 121-23-47
1212347
```

ELAPSED

The **ELAPSED** option keeps track of the time it takes the user to respond to an INPUT prompt. SheerPower assigns the elapsed time to the variable specified. The ELAPSED option starts marking time when SheerPower finishes displaying the prompt. Time stops when all the variables listed in the INPUT statement have values. The format is:

```
        ELAPSED num_var
```

num_var is the numeric variable the elapsed time is assigned to. SheerPower returns the elapsed time in seconds. If a real numeric variable is specified, SheerPower returns the time to a hundredth of a second (1/100).

```
  input 'Name', elapsed x: name$
  print name$; ', your elapsed time is'; x
  end


Name? John
John, your elapsed time is 1.39
```

If an integer variable is specified, SheerPower rounds the time to the nearest second.

```
input 'Name', elapsed x%: name$
print name$; ', your elapsed time is'; x%
end


Name? Julie
Julie, your elapsed time is 1
```

BOLD, BLINK, REVERSE

The **BOLD**, **BLINK**, **REVERSE** options allow each input string to be displayed with its own attributes. If these format options are used together with the ATTRIBUTES option, the ATTRIBUTES option will be suppressed.

```
input screen '<at 5,10, bold,blink:@@@@@@>' + &
             '<at 6,10, reverse:@@@@@@>': str1$, str2$
end
```

When a single input screen statement is used to create a fill-in screen, certain keys on the keyboard have special functions. The following keys can be used to complete the screen:

TAB

The **TAB key** `[Tab]` is used to jump to the next field in the input screen statement. Once a field is filled in, the `[Tab]` key must be pressed to jump to the next field.

[Esc]

The **Escape key** `[Esc]` can be used to return to a previous field. When the `[Esc]` key is pressed, the cursor will jump back to the beginning of the previous field and the entry can be changed.

Enter

The `[Enter]` key is used to complete a data entry screen. If more than one format is included in a screen option, when you have finished filling in all the fields, the `[Enter]` key must be pressed to complete data entry. You must be at the last field when the `[Enter]` key is pressed. Until the `[Enter]` key is pressed, you are free to change your response to any field.

```
a$ = '<at 6, 10: Full name @@@@@@@@@@@@@@@@@@@@@@@@>' + &
     '<at 8, 10: Address @@@@@@@@@@@@@@@@@@@@@@@@@@>' + &
     '<at 10, 10: City @@@@@@@@@@@@>' + &
     '<at 10, 38, aj, req, letters: State ^^>' + &
     '<at 10, 50, req, digits: Zip #####>'
clear
input screen a$: name$, addr1$, city$, state$, zip$
delay
clear
print at 6, 10: name$
print at 8, 10: addr1$
print at 10, 10: city$; ', '; state$; '   '; zip$
end


Full name Sunny Day_______________

Address 2356 Main St._____________

City San Diego___           State CA    Zip 92131


               Press the Enter key to continue

Sunny Day

2356 Main St.
```

```
San Diego, CA   92131
```

The **MENU** option is used to create pop-up menus.

FORMAT:

```
        [LINE] INPUT MENU str_expr: str_var
```

str_expr is a string expression that describes the menu. It consists of multiple, comma-separated elements. The elements can be either items or directives.

EXAMPLE:

```
  title$ = '%title "structure",%multi,'
  box_loc$ = '%at 10,15,'
  client$ = '"TTI_Client"={id,name={Cass,Brock,Errant},phone},'
  address$ = 'address={%bar,street,city,state,country},'
  misc$ = 'misc={%replace,mail,record}'
  menu$ = title$ & box_loc$ & client$ & address$ & misc$
  line input menu menu$: ans$
  end


               +---STRUCTURE---+
               |  TTI_CLIENT  +---ADDRESS--+
               |  ADDRESS     |------------|
               |  MISC...     |  STREET    |
               +--------------|  CITY      |
                              |  STATE     |
                              |  COUNTRY   |
                              +------------+
```

Menu **items** consist of either a description or a description and = sign followed by either a data value or a submenu indicator. If no = sign is given, the data value is the same as the description. If the description contains spaces or special characters, it must be enclosed within quotes. Up to 5000 menu items can be handled.

Menu **descriptions** are displayed on the menu for the user to choose.

The **data value** is the value returned to the application when the menu item is chosen.

A **submenu indicator** consists of a "{" followed by menu elements followed by "}".

The following directives can be used with the [LINE] INPUT MENU statement:

%AT row, column

The menu or submenu box can be placed at a given row and column. **CENTER** can be used in place of row and/or column to center a menu or submenu.

```
        %at row, center
        %at center, column
        %at center, center
```

%ATTACHED

The **%ATTACHED** menu directive causes the resulting menu window to be "attached" to the SP4GL Console Window (when there is some console output). If the console window is minimized or restored, the attached menu window will minimize or be restored with it.

FORMAT:

```
%attached
```

```
print 'Illustrating %attached menu directive...'
print
print 'Minimize the SheerPower 4GL Console Window...'
print 'and the "attached" menu will minimize along with it!'
menu$ ='%attached, OPEN, SHOW, PRINT, %heading "GUIDE_OPTIONS", NOSYSTEM, "MENU ON|OFF"'
line input menu menu$ : ans$
end
```

%BAR

Separates DESCRIPTIVE items with "----". The separated line consists of the characters specified in a "text_str". If no text is given, the separated line defaults to a dashed line.

FORMAT:

```
%BAR ['text_str']
```

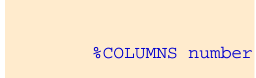
EXAMPLE:

```
menu$ = '"DATA ENTRY", "REPORTS", %bar"*", MAIL, EXIT'
line input menu menu$ : ans$
end


      +---------------+
      |  DATA ENTRY   |
      |  REPORTS      |
      |  **********   |
      |  MAIL         |
      |  EXIT         |
      +---------------+
```

%COLUMNS

The **%COLUMNS** directive sets the number of columns for the menu.

FORMAT:



```
%COLUMNS number
```

EXAMPLE:

```
menu$ = '%columns 3,"DATA ENTRY","REPORTS",MAIL,HELP,EXIT'
line input menu menu$ : ans$
end


      +--------------------------------------+
      |  DATA ENTRY   |   HELP   |   EXIT    |
      |  REPORTS      |   MAIL   |           |
      +--------------------------------------+
```

%HEADING

The **%HEADING** directive displays a blank line and a "text_str" between menu items. If no "text_str" is given, the line defaults to a dashed line.

FORMAT:

```
%HEADING ['text_str']
```

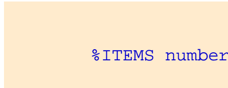
EXAMPLE:

```
menu$ ='OPEN, SHOW, PRINT, %heading "GUIDE_OPTIONS", NOSYSTEM, "MENU ON|OFF"'
line input menu menu$ : ans$
end


      +-------------------+
      |  OPEN             |
      |  SHOW             |
      |  PRINT            |
      |                   |
      |  GUIDE_OPTIONS    |
      |  NOSYSTEM         |
      |  MENU ON|OFF      |
      +-------------------+
```

%ITEMS

The **%ITEMS** directive creates a multi-column menu. "Number" represents the number of items per column. **%ITEMS** creates as many columns as is necessary. Horizontally scroll the columns as needed.

FORMAT:



```
%ITEMS number
```

EXAMPLE:

```
menu$='ENGINES={%items 3, DBMS, ARS, ADABASE, RDB, DBASE}'
input menu menu$: ans$
end


      +----------+-----------------------+
      |  ENGINES |  DBMS       |   RDB      |
      +----------|  ARS        |   DBASE    |
                 |  ADABASE    |            |
                 +-----------------------+
```

%LOCKSTEP

The **%LOCKSTEP** directive controls column scrolling when there are multiple columns in one menu. If it is turned OFF, the columns scroll independently. Otherwise, the columns scroll together.

FORMAT:

```
        %LOCKSTEP [ON|OFF]
```

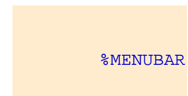
EXAMPLE:

```
menu$ = '%lockstep off, %SIZE 4, 1, 2, 3, 4, 5, 6, %split, 7, 8, 9, 10, 11, 12'
line input menu menu$ : ans$
end



      +---------------+
      |  ... |   7    |
      |  4   |   8    |
      |  5   |   9    |
      |  6   |   ...  |
      +---------------+
```

%MENUBAR

The **%MENUBAR** directive creates menu bars (menus with choices listed horizontally) with pull-down submenus.

FORMAT:



```
        %MENUBAR
```

EXAMPLE:

```
item1$='%menubar, OPEN, SELECT={INCLUDE,EXCLUDE}, SORT, PRINT'
line input menu item1$ : ans$
end


      +-----------------------------------------------------------------------------+
      |  OPEN      SELECT      SORT      PRINT                                      |
      +----------+---SELECT---+-------------------------------------------------+
                 |  INCLUDE   |
                 |  EXCLUDE   |
                 +------------+
```

%MESSAGE

The **%MESSAGE** directive displays a message when the menu or submenu is displayed.

FORMAT:

```
        %MESSAGE 'message'
```

EXAMPLE:

```
menu$ = 'open, show, print, %message "Select a menu option"'
line input menu menu$: ans$
end


      +----------+
      |  OPEN    |
      |  SHOW    |
      |  PRINT   |
      +----------+



                            Select a menu option
```

%MULTI

The **%MULTI** directive allows multiple items to be selected from a menu or submenu.

%REPLACE

The **%REPLACE** directive can be located in any submenu. The calling menu is not kept on the screen. Pressing '\' will return to the calling menu.

%SIZE number

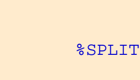
The **%SIZE number** directive determines the number of DESCRIPTIVE items that are located in the menu_box.

There is no limit to the number of the items that can be used. If all of the items do not fit within the menu box, the items are vertically scrolled using the UP, DOWN, LEFT and RIGHT ARROW keys. "Str_var" contains the name of the selected item. If the %MULTI directive is used and the `[Select]` key is pressed, "Str_var" contains the names of the items. Each item is separated by a line feed character.

%SPLIT

The **%SPLIT** directive instructs SheerPower to start a new column at that specified point in the menu.

FORMAT:



```
%SPLIT
```

EXAMPLE:

```
item1$='title, chapter, page={1201, 1202, 1203, 1204, %split, 1305, 1306, 1307}'
line input menu item1$ : ans$
end


      +------------+
      |  TITLE     |
      |  CHAPTER +-------PAGE---------+
      |  PAGE    |  1201   |   1305   |
      +----------|  1202   |   1306   |
                 |  1203   |   1307   |
                 |  1204   |          |
                 +--------------------+
```

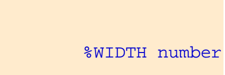
%TITLE ".........."

The **%TITLE ".........."** directive assigns a title to a menu or submenu.

%WIDTH

The **%WIDTH** directive controls the minimum width of the current column.

FORMAT:



```
%WIDTH number
```

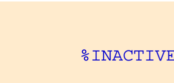
EXAMPLE:

```
menu$ = '%width 20,%items 2, a, b, c,d, f, g, h, i, j'
input menu menu$ : ans$
end


        +-----------------------------------------------------------------------------
-+
        |<<  C                  |   F                  |   H
>>|
        |    D                  |   G                  |   I
|
        +-----------------------------------------------------------------------------
-+
```

%INACTIVE

Normally menus are ON TOP of all windows and ACTIVE. The **%INACTIVE** directive makes the menus be like "normal" windows where other windows can be on top of them.

FORMAT:



```
%INACTIVE
```

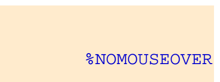
EXAMPLE:

```
title$ = '%inactive, %title "structure",%multi,'
box_loc$ = '%at 10,15,'
client$ = '"TTI_Client"={id,name={Cass,Brock,Errant},phone},'
address$ = 'address={%bar,street,city,state,country},'
misc$ = 'misc={%replace,mail,record}'
menu$ = title$ & box_loc$ & client$ & address$ & misc$
line input menu menu$: ans$
end


             +---STRUCTURE---+
             |  TTI_CLIENT  +---ADDRESS--+
             |  ADDRESS     |------------|
             |  MISC...     |  STREET    |
             +--------------|  CITY      |
                            |  STATE     |
                            |  COUNTRY   |
                            +------------+
```

%NOMOUSEOVER

Usually as you move your mouse over a menu item, the menu item under the mouse becomes active. %NOMOUSEOVER turns off the mouse-over feature.

FORMAT:



```
%NOMOUSEOVER
```

EXAMPLE:

```
menu$ ='%nomouseover, OPEN, SHOW, PRINT, %heading "GUIDE_OPTIONS",' +
   'NOSYSTEM, "MENU ON|OFF"'
line input menu menu$ : ans$
end



      +------------------+
      |  OPEN            |
      |  SHOW            |
      |  PRINT           |
      |                  |
      |  GUIDE_OPTIONS   |
      |  NOSYSTEM        |
```

```
    |  MENU ON|OFF     |
    +------------------+
```

The following is the user interface in a [LINE] INPUT MENU:

- UP and DOWN arrow keys move the menu cursor from one item to another.
- `[esc]` or 'LEFT arrow' key returns to the calling menu.
- `[Ctrl/Z]` sets _EXIT to TRUE.
- 'SPACE', 'ENTER', 'RIGHT arrow' and clicking the pointing device selects the item if the item is a submenu.
- 'SPACE' should be used with '%MULTI' to select or deselect an item; clicking the pointing device can also be used.
- The 'DEL' key moves the menu cursor back to the first item in the menu.

FORMAT:

```
        KEY INPUT [[#channel] [, PROMPT str_expr]
             [, TIMEOUT time_limit]
             [, ELAPSED num_var]
             :] var, [var,...]
```

EXAMPLE:

```
  print 'See how quick you are.'
  key input prompt 'Press a key, quick!', &
      elapsed x: press$
  print
  print 'You took'; x; 'seconds to Press '; press$; '.'
  end


See how quick you are.
Press a key, quick!
You took 1.99 seconds to Press h.
```

PURPOSE:

KEY INPUT is used to input a keystroke from the user and stores the value of the key in the string variable specified.

DESCRIPTION:

Some applications require the use of special keys or keystroke level validation. The **KEY INPUT** statement is used for these applications.

KEY INPUT does not echo the key pressed, nor does it generate a line feed.

All the options available with the preceding "INPUT" statement are also available with KEY INPUT. KEY INPUT returns the following:

_back
_exit
_help
_terminator

_TERMINATOR returns control (Ctrl) characters in the format "CTRL/X". For example, if the user presses `[Ctrl/G]`, _TERMINATOR returns "CTRL/G".

```
key input 'Press a terminator' : z$
print
print 'You pressed '; _terminator
end


Press a terminator?            (User presses [CTRL/G])
You pressed CTRL/G
```

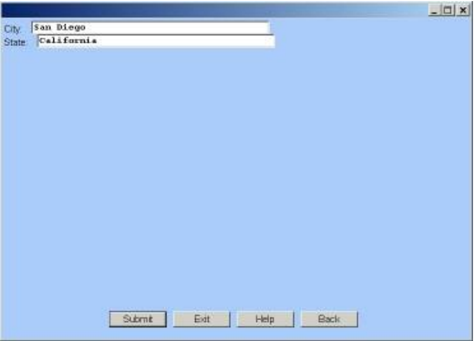
FORMAT:

```
INPUT DIALOGBOX str_exp[, DEFAULT str_exp]: str_var
```

EXAMPLE:

```
form$ = '<form>'
form$ = form$ + 'City: <input type=text name=city><br>'
form$ = form$ + 'State: <input type=text name=state>'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```

When the program executes, SheerPower displays the "City" field and a fill-in area where the user can fill in a city, followed by the "State" field and a fill-in area where the user can fill in a state. At the bottom of the form, the default form buttons are displayed: SUBMIT, EXIT, HELP and BACK.



PURPOSE:

INPUT DIALOGBOX presents the end user with simple to complex input forms.

DESCRIPTION:

INPUT DIALOGBOX is a very powerful feature of SheerPower. INPUT DIALOGBOX provides the user with the power of HTML forms without using a web browser.

When the user submits the form, the results entered are returned in the following format:

```
fieldname=result
```

The **fieldname** is what the *name* of each input field is given inside the form code:

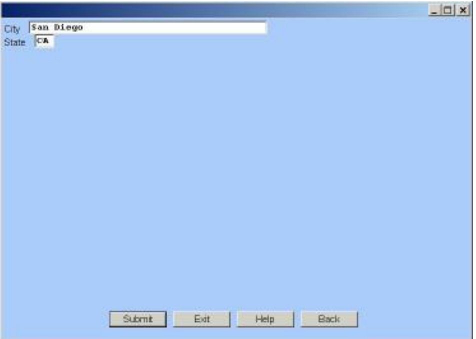
```
<input type=text name=firstname>
```

If there are multiple fieldnames that contain user-entered results, the data is stored into a list separated by a **chr$(26)**:

```
fieldname=result chr$(26) fieldname=result chr$(26) fieldname=result
```

Below is an example of how to parse the results of a submitted form:

```
myform$ = '<form>City <input type=text name=city  size=40><br>' +
                'State <input type=text name=state size=2></form>'

input dialogbox myform$: ans$

for item = 1 to pieces(ans$, chr$(26))
  z0$ = piece$(ans$, item, chr$(26))
  varname$ = element$(z0$, 1, '=')
  value$   = element$(z0$, 2, '=')
  select case varname$
  case 'city'
    print 'City was  : '; value$
  case 'state'
    print 'State was : '; value$
  case else
    print 'Unknown: '; varname$
  end select
next item
end
```



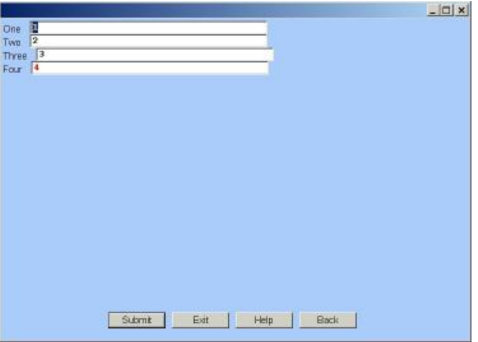
Type in a city and state, then click on `[SUBMIT]`. The results will be printed out to the SP4GL window as seen below:

There are 3 directives available for use with the **Input Dialogbox statement**. These directives are used as part of the **default** option.

- %error
- %focus
- %readonly

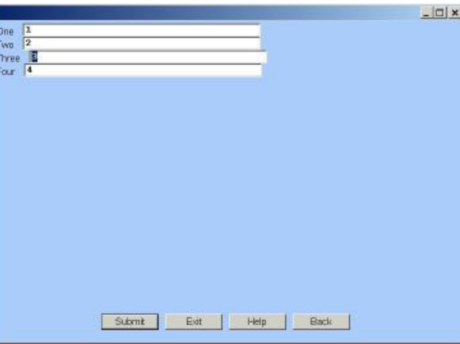
The **%ERROR** directive places the focus on the erroneous field(s) and makes the field text red. Multiple %error directives can be used for multiple errors.

```
test$ = '<form>'
test$ = test$ + 'One <input name=one value="1"><br>'
test$ = test$ + 'Two <input name=two value="2"><br>'
test$ = test$ + 'Three <input name=three value="3"><br>'
test$ = test$ + 'Four <input name=four value="4">'
test$ = test$ + '</form>'
def$ = "%error=one"+chr$(26)+"%error=four"
input dialogbox test$, default def$: answer$
end
```



The **%FOCUS** directive allows the focus to be placed on a selected field. Multiple %focus directives can be used if there are multiple fields to move focus to.

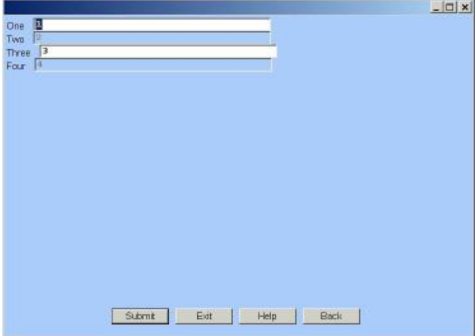
```
test$ = '<form>'
test$ = test$ + 'One <input name=one value="1"><br>'
test$ = test$ + 'Two <input name=two value="2"><br>'
test$ = test$ + 'Three <input name=three value="3"><br>'
test$ = test$ + 'Four <input name=four value="4">'
test$ = test$ + '</form>'
def$ = "%focus=three"
input dialogbox test$, default def$: answer$
end
```



The **%READONLY** directive allows for fields to display data that cannot be changed by the end-user.

To specify multiple fields as "read-only" the chr$(26) separator must be included.

```
test$ = '<form>'
test$ = test$ + 'One <input name=one value="1"><br>'
test$ = test$ + 'Two <input name=two value="2"><br>'
test$ = test$ + 'Three <input name=three value="3"><br>'
test$ = test$ + 'Four <input name=four value="4">'
test$ = test$ + '</form>'
def$ = "%readonly=two"+chr$(26)+"%readonly=four"
input dialogbox test$, default def$: answer$
end
```

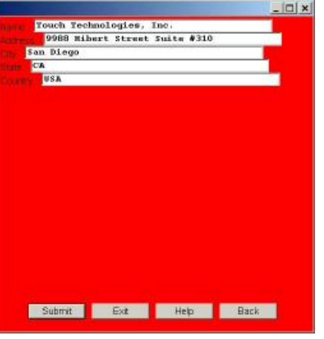


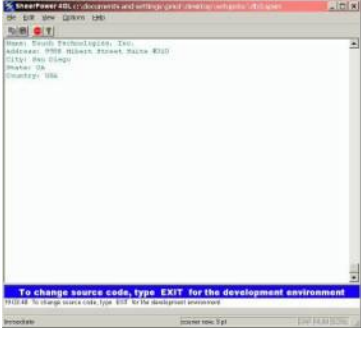
INPUT DIALOGBOX supports a subset of standard HTML tags along with dialogbox enhancements. For a complete listing of supported HTML tags refer to Appendix G, Input Dialogbox--supported HTML tags.

The **SHEERPOWER TAG** is a powerful feature of INPUT DIALOGBOX. The SheerPower tag allows you to change the size, color and title of a form as well as other attributes. In addition, the SheerPower tag includes a **TYPE** attribute which specifies the type of dialog box to be presented. The different types are: HTML FORM (default), OPEN, SAVEAS, and SELECT.

When using the HTML form type the SheerPower tag must be the **first** tag used before the FORM tag.

```
form$ = '<sheerpower color=red height=500 width=500>'
form$ = form$ + '<form>'
form$ = form$ + 'Name <input type=text name=name><br>'
form$ = form$ + 'Address <input type=text name=address><br>'
form$ = form$ + 'City <input type=text name=city><br>'
form$ = form$ + 'State <input type=text name=state><br>'
form$ = form$ + 'Country <input type=text name=country><br>'
form$ = form$ + '</form>'
input dialogbox form$: results$

for item = 1 to pieces(results$, chr$(26))
  z0$ = piece$(results$, item, chr$(26))
  varname$ = element$(z0$, 1, '=')
  value$   = element$(z0$, 2, '=')
  select case varname$
  case 'name'
    print 'Name: '; value$
  case 'address'
    print 'Address: '; value$
  case 'city'
    print 'City: '; value$
  case 'state'
    print 'State: '; value$
  case 'country'
    print 'Country: '; value$
  case else
    print '??: '; varname$
  end select
next item
end
```





SheerPower Tag Attributes

Table 9-1 SHEERPOWER Tag Attributes

Attribute	Function
color	specify the background color of the form
background	specify a .JPG image to be used for the form background
title	specify the title of the form
height	specify the height of the form (in pixels)
width	specify the width of the form (in pixels)
src	specifies location of a URL or local file to grab HTML form tag code from (URL must contain only form HTML code)
persist	keeps background canvas, text and images in place when form autosubmits (no blinking screen)
autosubmit	automatically submits form after specified seconds of inactivity
type (HTML form/open/saveas/select)	opens the Open, Saveas or Select dialogbox
attached	causes the resulting dialogbox window to be "attached" to the SP4GL Console Window (when there is some console output). If the console window is minimized or restored, the attached dialogbox window will minimize or be restored with it.

The SheerPower tag allows you to manipulate the color, height and width of an HTML form. Percentages can also be specified for height and width instead of pixels. For example:

```
<sheerpower title="This is MY form!" color="green" height="50%" width="75%">
```

You can also use the title attribute to insert a title in the form.

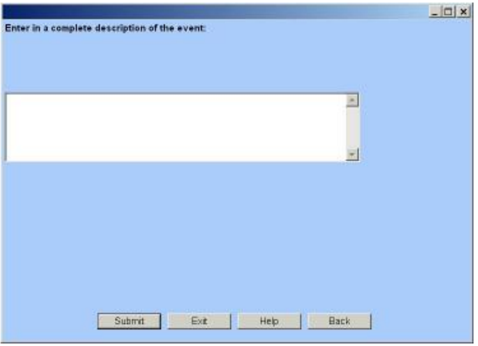
```
  form$ = '<sheerpower title="This is MY form!" color="green" height="400"
width="500">'
  form$ = form$ + '<form>'
  form$ = form$ + 'Name <input type=text name=name><br>'
  form$ = form$ + 'Address <input type=text name=address ><br>'
  form$ = form$ + 'City <input type=text name=city><br>'
  form$ = form$ + 'State <input type=text name=state><br>'
  form$ = form$ + 'Country <input type=text name=country><br>'
  form$ = form$ + '</form>'
  input dialogbox form$: results$
  end
```

This example illustrates the BACKGROUND attribute to the <sheerpower> tag where any .JPG image file can be used as the dialogbox form background.

```
  form$ = '<form>'
  form$ = form$ + '<sheerpower background="sheerpower:samples\woodpecker.jpg">'
  form$ = form$ + '</form>'
  input dialogbox form$: ans$
  end
```

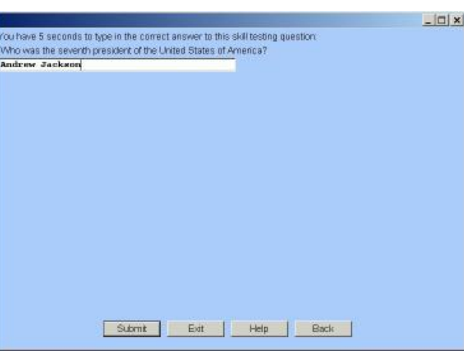
The next example illustrates the SRC attribute. Using the SRC attribute inside the <sheerpower> tag allows you to take the existing HTML code between <form></form> tags from a file located on your local machine and use it inside your program. Note the syntax used to locate the file when using the SRC attribute.

```
form$ = '<sheerpower src="file://c:\sheerpower\samples\src_form.html">'
input dialogbox form$: response$
end
```



The **AUTOSUBMIT** attribute lets you automatically submit a form after a specified number of seconds of inactivity. The following example illustrates the use of the AUTOSUBMIT attribute to the SheerPower tag.

```
form$ = '<sheerpower autosubmit="5">'
form$ = form$ + '<form>'
form$ = form$ + 'You have 5 seconds to type in the correct answer '
form$ = form$ + 'to this skill testing question:<p> Who was the seventh '
form$ = form$ + 'president of the United States of America? '
form$ = form$ + '<input type=text name=answer>'
form$ = form$ + '</form>'
input dialogbox form$: skill$
print skill$
end
```



The **AUTOSCALE** attribute is used to dynamically adjust the fontsizes for the end-user so that the dialogbox looks like the one that the programmer designed.

```
<sheerpower autoscale=true>
```

The **ATTACHED** attribute causes the resulting dialogbox window to be "attached" to the SP4GL Console Window (when there is some console output). If the console window is minimized or restored, the attached dialogbox window will minimize or be restored with it.

```
<sheerpower attached>
```

The **TYPE** attribute enables you bring up the OPEN, SAVEAS or SELECT dialog box. The dialogbox is used to get the list of files from the user, but does not perform any processing on the files.

The DEFAULT PATH can be any location specified as shown in the examples below.

You can choose which file types to display in the dialog boxes by using a filter:

```
input dialogbox '<sheerpower type=xxx filter="filter_string">': f$
```

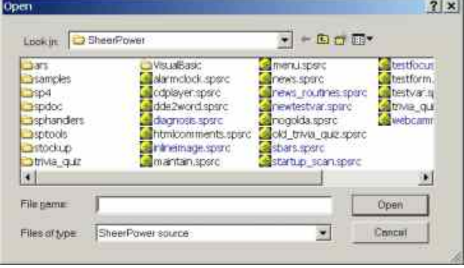
The filter_string will look like:

```
filtername = spec,spec,spec; filtername=spec2,spec2
```

When writing a program that opens a dialog box for the user to select a file you must also check to see if the user selects the `[Cancel]` button. The way to check if a user clicked on CANCEL or closed the dialogbox window is:

```
if _exit then print 'they closed the window or clicked on cancel'
```

```
path$ = "c:\sheerpower"
filter$ = "SheerPower source=*.spsrc,*.spinc"
input dialogbox '<sheerpower type=open filter=' +
                quote$(filter$) +'>', default path$: f$
```



To specify a root-level for the BROWSING of files as well as a default file name, use the following syntax:

```
input dialogbox '<sheerpower type=open>', default xxx$: f$
```

Where xxx$ can be a specific file path with or without the file name and extension.

```
pathfile$ = "c:\sheerpower\samples\news.spsrc"
input dialogbox '<sheerpower type=open>', default pathfile$: f$
end
```

```
input dialogbox '<sheerpower type=saveas>': f$
end
```

```
select_file$ = "c:\sheerpower\"
input dialogbox '<sheerpower type=select>', default select_file$: f$
end
```

<form>. . . </form>

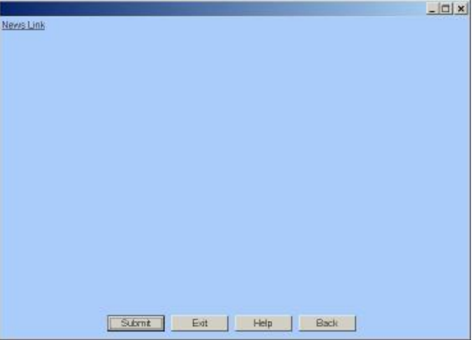
The **FORM** tag defines an input form.

```
form$ = '<form>'
form$ = form$ + 'Name <input type=text name=name ><br>'
form$ = form$ + 'Address <input type=text name=address ><br>'
form$ = form$ + 'City <input type=text name=city><br>'
form$ = form$ + 'State <input type=text name=state><br>'
form$ = form$ + 'Country <input type=text name=country><br>'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```

<a>...

The **ANCHOR** tags can be used to insert a clickable **link** inside a form. Clicking on the text link will open a web browser to the URL indicated inside the anchor tags.

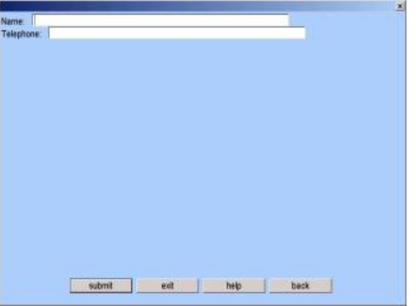
```
f$ = '<form><a href="http://www.cnn.com">News Link</a></form>'
input dialogbox f$: a$
end
```



<input>

The **INPUT** tag is used to specify a simple input element inside a form. There is no terminating INPUT tag.

```
form$ = '<form>'
form$ = form$ + 'Name: <input type=text name=name><br>'
form$ = form$ + 'Telephone: <input type=text name=telephone><br>'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```



INPUT Tag Attributes

Table 9-2 INPUT Tag Attributes

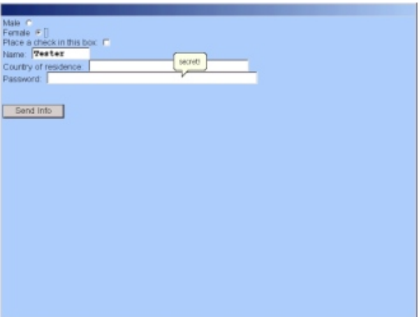
Attribute	Function
type	see list below for type attributes
name	symbolic name for this input field
value	can be used to specify the default contents of the field; also specifies the value of a checkbox or radio button when it is checked
checked	specifies that this checkbox or radio button is checked by default; this is only appropriate for checkboxes and radio buttons
size	the physical size of the input field in characters

maxlength	the maximum number of characters that are accepted as input
message	message text displayed when hovering over input field

The **TYPE** attributes are:

- **text** - text entry field
- **password** - text entry field; entered characters are represented as asterisks
- **checkbox** - a single toggle button; on or off
- **radio** - a single toggle button; on or off
- **submit** - a pushbutton that causes the completion of the form
- **reset** - a pushbutton that causes the various input elements in the form to be reset to their default values
- **hidden** - allows authors to include form data without having it rendered to the user.

```
  test$ = '<form>'
  test$ = test$ + 'Male<input type=radio name=gender><br>'
  test$ = test$ + 'Female<input type=radio name=gender checked><br>'
  test$ = test$ + 'Place a check in this box:<input type=checkbox name=check><br>'
  test$ = test$ + 'Name:<input type=text name=name size=10 value="Tester"><br>'
  test$ = test$ + 'Country of residence:<input type=text name=country message="type
here" size=30><br>'
  test$ = test$ + 'Password:<input type=password name=password message="secret!"
maxlength=5><br>'
  test$ = test$ + '<input type=hidden name=test_form_only value=complete><p>'
  test$ = test$ + '<input type=submit name=submit value="Send Info">'
  test$ = test$ + '</form>'
  input dialogbox test$: answer$
  end
```



The default SUBMIT buttons are:

- Submit
- Exit
- Help
- Back

To create your own custom submit buttons insert the following HTML code inside your form code:

```
<input type=submit name=submit value="Send Info">
```

The same can be done to custom create your own Exit, Help and Back buttons.

You can place the submit buttons whereever you want on the form.

```
  form$ = '<form>'
  form$ = form$ + '<h2>Test Question</h2><p>'
  form$ = form$ + '1. Solve the following equation: <b>1=sin(3x) - cos(6x)</b>'
  form$ = form$ + '<p>Type in your final answer inside the space provided below.<p>'
  form$ = form$ + '<input type=text name=solution value="type your solution
here"><p>'
  form$ = form$ + '<input type=submit name=submit value="Send my solution!">'
  form$ = form$ + '<input type=submit name=exit value="Uh-Uh! Get me outta here!">'
  form$ = form$ + '<input type=submit name=help value="Help...me....">'
  form$ = form$ + '<input type=submit name=back value="Back up!">'
  form$ = form$ + '</form>'
  input dialogbox form$: ans$
  end
```

Images can be used for submit buttons in INPUT DIALOGBOX. The format is:

```
  <input type=submit name=submit src="url_of_image.jpg">
```

```
  img_location$ = 'sheerpower:samples'
  form$ = '<form>'
  form$ = form$ + '<sheerpower color="white">'
  form$ = form$ + '<h2>Test Question</h2><p>'
  form$ = form$ + '1. Solve the following equation: <b>1=sin(3x) - cos(6x)</b>'
  form$ = form$ + '<p>Type in your final answer inside the space provided below.<p>'
  form$ = form$ + '<input type=text name=solution value="type your solution here">'
  form$ = form$ + '<input type=submit name=submit src="'+ img_location$ +
'\help_submit.jpg">'
  form$ = form$ + '</form>'
  input dialogbox form$: ans$
  end
```

<select>...</select>

The **SELECT** tag creates a drop down menu inside a form. Inside SELECT, only a sequence of OPTION tags is allowed. Each sequence can be followed by an arbitrary amount of plain text.

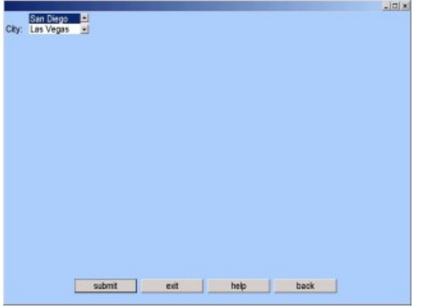
```
form_menu$ = '<form>'
form_menu$ = form_menu$ + 'City: <select name=city>'
form_menu$ = form_menu$ + '<option value="San Diego">San Diego, CA'
form_menu$ = form_menu$ + '<option value="Las Vegas">Las Vegas, NV'
form_menu$ = form_menu$ + '<option value="Minneapolis">Minneapolis, MN'
form_menu$ = form_menu$ + '</select>'
form_menu$ = form_menu$ + '</form>'
input dialogbox form_menu$: choice$
end
```

SELECT Tag Attributes

Table 9-3 SELECT Tag Attributes

Attribute	Function
name	the symbolic name for this SELECT element (must be present)
size	the value of SIZE then determines how many items will be visible
multiple	if present (no value), specifies that the SELECT should allow multiple selections (n of many behavior)

```
form_menu$ = '<form>'
form_menu$ = form_menu$ + 'City: <select multiple size=2 name=city>'
form_menu$ = form_menu$ + '<option value="San Diego">San Diego'
form_menu$ = form_menu$ + '<option value="Las Vegas">Las Vegas'
form_menu$ = form_menu$ + '<option value="Minneapolis">Minneapolis'
form_menu$ = form_menu$ + '<option value="Pheonix">Pheonix'
form_menu$ = form_menu$ + '<option value="New York">New York'
form_menu$ = form_menu$ + '<option value="New Jersey">New Jersey'
form_menu$ = form_menu$ + '</select>'
form_menu$ = form_menu$ + '</form>'
input dialogbox form_menu$: choice$
end
```

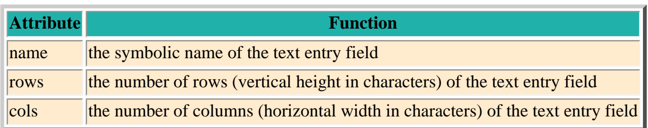


<textarea>...</textarea>

The **TEXTAREA** tag is used to place a multiline text entry field with optional default contents in a fill-out form.

TEXTAREA Tag Attributes

Table 9-4 TEXTAREA Tag Attributes



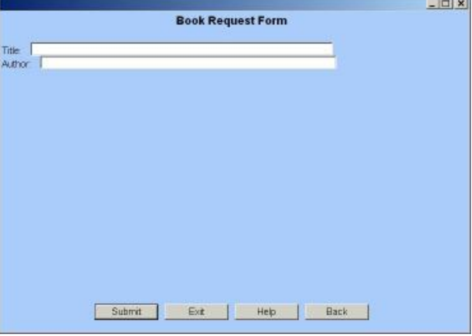
Attribute	Function
name	the symbolic name of the text entry field
rows	the number of rows (vertical height in characters) of the text entry field
cols	the number of columns (horizontal width in characters) of the text entry field

```
form_box$ = '<form>'
form_box$ = form_box$ + '<textarea name=comments rows=10 cols=30>'
form_box$ = form_box$ + 'Please type your comments in here.'
form_box$ = form_box$ + '</textarea>'
form_box$ = form_box$ + '</form>'
input dialogbox form_box$: choice$
end
```

<center>...</center>

The **CENTER** tag defines text that should be centered.

```
test$ = '<form>'
test$ = test$ + '<center><b><h3>Book Request Form</center></h3></b><p>'
test$ = test$ + 'Title: <input type=text name=title size=46><br>'
test$ = test$ + 'Author: <input type=text name=author size=45>'
form$ = test$ + '</form>'
input dialogbox test$: answer$
end
```



<p>...</p>

The **PARAGRAPH** tag starts a new paragraph by separating one paragraph from another with white space. The </P> tag is optional if the tag is only to insert space between two paragraphs, but vital when attributes (for example, ALIGN="center") are to apply to the whole paragraph. The ALIGN attribute can be one of LEFT, RIGHT, or CENTER.

```
  text$ = '<form>'
  text$ = text$ + '<b>Paragraph Example</b>'
  text$ = text$ + '<p>The following paragraphs will '
  text$ = text$ + 'illustrate the left, right and center attributes to the '
  text$ = text$ + 'paragraph tag.'
  text$ = text$ + '<p align=left>'
  text$ = text$ + 'This Agreement is governed by, and construed in accordance with, '
  text$ = text$ + 'the laws of the State of California, without regard to your actual
state or '
  text$ = text$ + 'country of residence. Any modification of any provision of this
Agreement will'
  text$ = text$ + ' be effective only if in writing and signed by TTI.</p>'
  text$ = text$ + '<p align=right>'
  text$ = text$ + 'If, for any reason, a court of competent jurisdiction '
```

```
  text$ = text$ + 'finds any provision or portion of this Agreement to be 
unenforceable, the remainder '
  text$ = text$ + 'of this Agreement will continue in full force and effect.</p>'
  text$ = text$ + '<p align=center>'
  text$ = text$ + 'This Agreement constitutes the entire agreement between '
  text$ = text$ + 'you and TTI, with respect to the subject matter at hand. This 
Agreement supersedes '
  text$ = text$ + 'and replaces all other understandings or agreements, written or 
oral, regarding such '
  text$ = text$ + 'subject matter.</p>'
  input dialogbox text$: answer$
  end
```

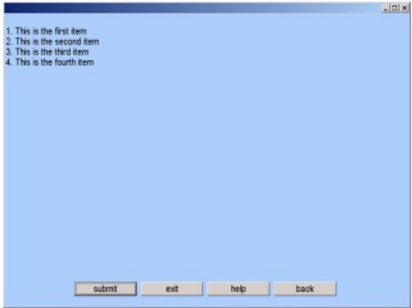

The **LINE BREAK** tag breaks the current line of text. There is no </br> tag.

```
  test$ = '<form>'
  test$ = test$ + '1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>10'
  test$ = test$ + '</form>'
  input dialogbox test$: answer$
  end
```

...

The **ORDERED LIST** tag introduces an ordered (numbered) list, which is made up of List Item (LI) tags.

```
nlist$ = '<form>'
nlist$ = nlist$ + '<ol>'
nlist$ = nlist$ + '<li>This is the first item'
nlist$ = nlist$ + '<li>This is the second item'
nlist$ = nlist$ + '<li>This is the third item'
nlist$ = nlist$ + '<li>This is the fourth item'
nlist$ = nlist$ + '</ol>'
nlist$ = nlist$ + '</form>'
input dialogbox nlist$: example$
end
```



...

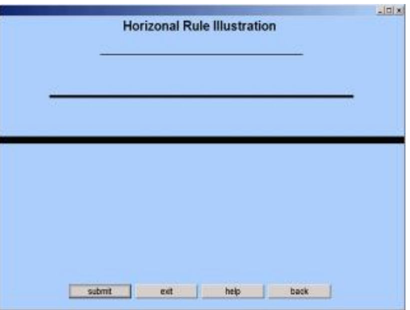
The **UNORDERED LIST** tag introduces an unordered (bulleted) list, which is made up of List Item (LI) tags.

```
blist$ = '<form>'
blist$ = blist$ + '<ul>'
blist$ = blist$ + '<li>This is the first item'
blist$ = blist$ + '<li>This is the second item'
blist$ = blist$ + '<li>This is the third item'
blist$ = blist$ + '<li>This is the fourth item'
blist$ = blist$ + '</ul>'
blist$ = blist$ + '</form>'
input dialogbox blist$: example$
end
```

<hr>

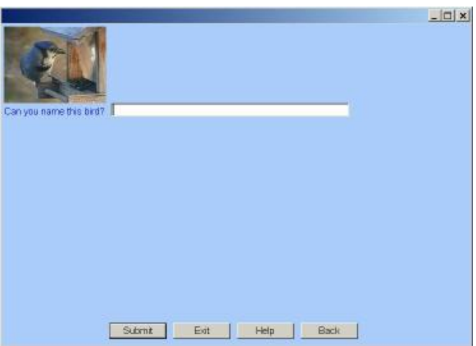
The **HORIZONTAL RULE** tag causes a horizontal line to be drawn across the screen. There is no </hr> tag.

```
form$ = '<form>'
form$ = form$ + '<center><h2>Horizontal Rule Illustration</h2></center>'
form$ = form$ + '<p><hr width=50%><p>'
form$ = form$ + '<p><hr width=75% size=5><p>'
form$ = form$ + '<p><hr size=10><p>'
form$ = form$ + '</form>'
input dialogbox form$: ans$
end
```



The **INLINE IMAGE** tag displays an image referred to by a URL. It must contain at least an SRC attribute.

```
image$ = '<form>'
image$ = image$ + '<img src="c:\sheerpower\samples\bluejay.jpg"><p>'
image$ = image$ + '<font color=blue>Can you name this bird?</font>'
image$ = image$ + '<input type=text name=birdname>'
image$ = image$ + '</form>'
input dialogbox image$: source$
end
```



INLINE IMAGE TAG ATTRIBUTES

Table 9-5 INLINE IMAGE Tag Attributes

Attribute	Function
src="URL"	**URL** identifies the image source
width="number"	**number** specifies the width of the image in pixels

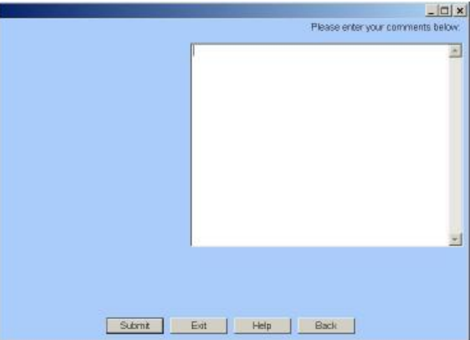
height="number"	**number** specifies the height of the image in pixels
border="number"	**number** is the border thickness in pixels
align="alignment"	**alignment** left or right for horizontal alignment; top, texttop, middle, center, bottom and baseline for vertical alignment

```
// A simple quiz program

  woodpecker$ = 'sheerpower:samples\woodpecker.jpg'

  quiz_form$ = '<sheerpower persist><title>Quiz</title><form>' +
               '<center><h3>Skill Testing Question</center></h3>' +
               '<img src="' + woodpecker$ + '" border=3 align=middle>'
  quiz_form$ = quiz_form$ + '<font color=green> ' +
               '<b>What type of woodpecker' +
               ' is in this photograph?</b></font><p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Pileated Woodpecker"> ' +
               '<i>Pileated Woodpecker<p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Hairy Woodpecker"> Hairy Woodpecker<p>'
  quiz_form$ = quiz_form$ + '<input type=radio name=birdname ' +
               'value="Redheaded Woodpecker"> ' +
               'Redheaded Woodpecker</i></b>'
  quiz_form$ = quiz_form$ + '<p><input type=submit name=submit value="Submit">' +
                            '<input type=submit name=exit>' +
                            '</form>'
  correct$ = 'Hairy Woodpecker'
  good$ = '<sheerpower width=400 height=170 color=green>' +
          '<form><h1>Congratulations!!  ' +
          correct$ + ' is the correct answer!!</h1>' +
          '<p><input type=submit></form>'

  do
    input dialogbox quiz_form$: ans$
    if _exit then stop
    value$ = element$(ans$, 2, '=')
    if value$ = correct$  then exit do
    message error: "Sorry, this is not a ";value$
  loop
  input dialogbox good$: ans$
  end
```

<div>...</div>

The **DIVISON** tag is used to divide a document up into different sections, such as chapters, sections, abstract, and appendix. The CLASS attribute specifies what section this is. The ALIGN attribute can be one of LEFT, RIGHT, or CENTER.

```
form$ = '<div align=right><form>'
form$ = form$ + 'Please enter your comments below: <p>'
form$ = form$ + '<textarea name=comment cols=30 rows=15></textarea>'
form$ = form$ + '</form></div>'
input dialogbox form$: comment$
end
```

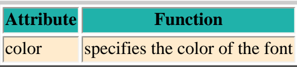


...

The **FONT** tag defines text with a smaller or larger font than usual. The normal font size corresponds to 3; smaller values of number will produce a smaller font, and larger values of number will produce a larger font.

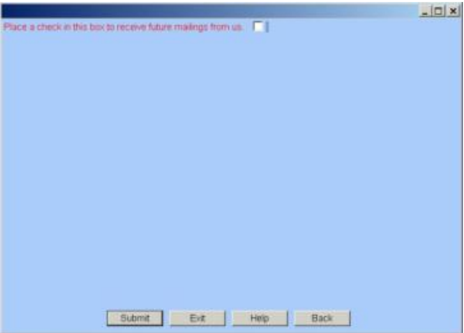
FONT Tag Attributes

Table 9-6 FONT Tag Attributes



Attribute	Function
color	specifies the color of the font

```
form$ = '<form>'
form$ = form$ + '<font color=red>'
form$ = form$ + 'Place a check in this box to receive future mailings from us. '
form$ = form$ + '<input type=checkbox name=yesmail></font>'
form$ = form$ + '</form>'
input dialogbox form$: reply$
end
```



Note

The **COLOR ATTRIBUTE** to the font tag allows you to utilize **hexadecimal** color values the same as HTML.

SheerPower supports 6 levels of **HEADINGS**, H1 through H6. H1 is the largest heading size. H6 is the smallest heading size.

<h1>...</h1>

The **HEADING 1** tag defines a level 1 heading (the largest heading size).

```
form$ = '<form>'
form$ = form$ + '<h1>Level 1 Heading Tag</h1><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```

<h2>...</h2>

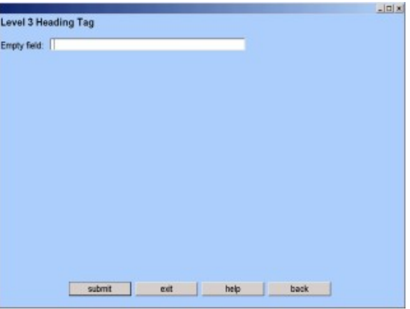
The **HEADING 2** tag defines a level 2 heading.

```
form$ = '<form>'
form$ = form$ + '<h2>Level 2 Heading Tag</h2><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```

<h3>...</h3>

The **HEADING 3** tag defines a level 3 heading.

```
form$ = '<form>'
form$ = form$ + '<h3>Level 3 Heading Tag</h3><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```



<h4>...</h4>

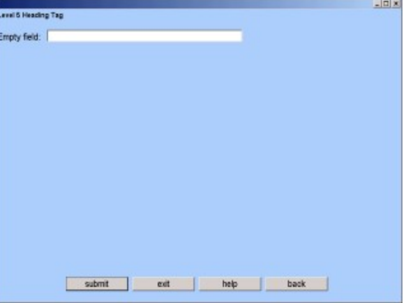
The **HEADING 4** tag defines a level 4 heading.

```
form$ = '<form>'
form$ = form$ + '<h4>Level 4 Heading Tag</h4><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```

<h5>...</h5>

The **HEADING 5** tag defines a level 5 heading.

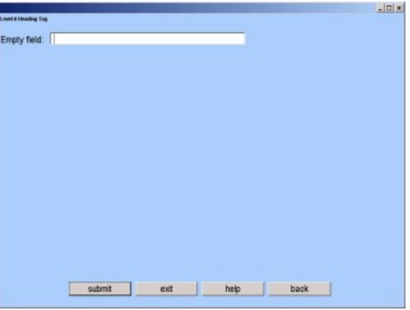
```
form$ = '<form>'
form$ = form$ + '<h5>Level 5 Heading Tag</h5><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```



<h6>...</h6>

The **HEADING 6** tag defines a level 6 heading (the smallest size heading).

```
form$ = '<form>'
form$ = form$ + '<h6>Level 6 Heading Tag</h6><br>'
form$ = form$ + 'Empty field: <input type=text name=field size=30>'
form$ = form$ + '</form>'
input dialogbox form$: entry$
end
```



...

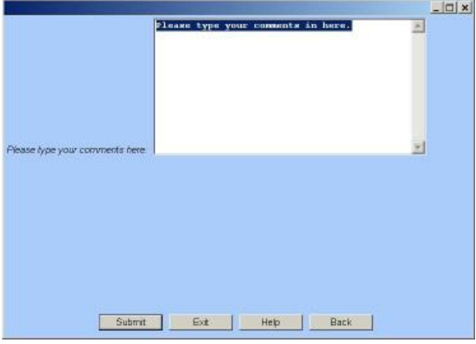
The **BOLD** tag defines text that should be shown in boldface.

```
form$ = '<form>'
form$ = form$ + 'This example illustrates the <b>BOLD</b> tag<br>'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```

<i>...</i>

The **ITALIC** tag defines text that should be shown in italics.

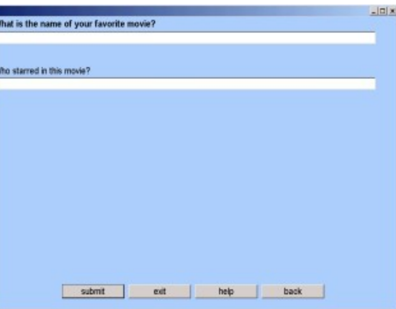
```
form$ = '<form>'
form$ = form$ + '<i>Please type your comments here:</i>'
form$ = form$ + '<textarea name=comments rows=10 cols=30>'
form$ = form$ + 'Please type your comments in here.'
form$ = form$ + '</textarea>'
form$ = form$ + '</form>'
input dialogbox form$: comments$
end
```



...

The **EMPHASIZED** tag defines text that should be emphasized.

```
form$ = '<form>'
form$ = form$ + '<em>What is the name of your favorite movie?</em>'
form$ = form$ + '<input type=text name=movie size=60><br><p>'
form$ = form$ + 'Who starred in this movie?'
form$ = form$ + '<input type=text name=star size=60>'
form$ = form$ + '</form>'
input dialogbox form$: ans$
end
```



<pre>...</pre>

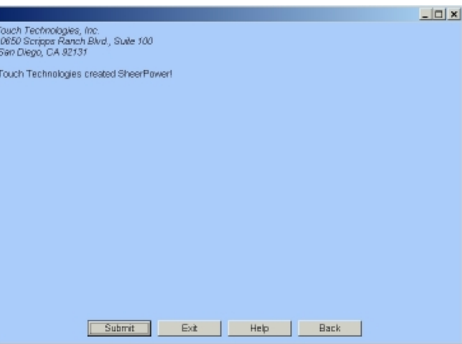
The **PREFORMATTED TEXT** tag defines text that should be shown in a fixed width font with whitespace specified by the form author. Multiple spaces will be displayed as multiple spaces.

```
poem$ = '<form>'
poem$ = poem$ + '<pre>'
poem$ = poem$ + '          I will confess<br>'
poem$ = poem$ + '        With cheerfulness,<br>'
poem$ = poem$ + '   Love is a thing so likes me,<br>'
poem$ = poem$ + '        That, let her lay<br>'
poem$ = poem$ + '          On me all day,<br>'
poem$ = poem$ + "I'll kiss the hand that strikes me.<p>"
poem$ = poem$ + " - excerpt from Robert Herrick's A Hymn to Love"
poem$ = poem$ + '</pre>'
poem$ = poem$ + '</form>'
input dialogbox poem$: test$
end
```

<address>...</address>

The **ADDRESS** tag defines text that gives an address or other contact information. The text is displayed in italics.

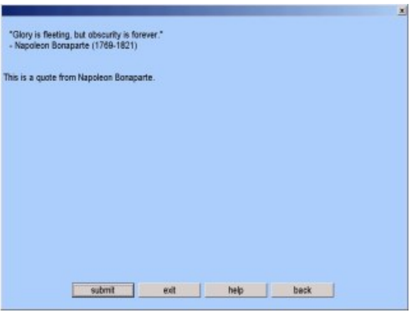
```
form$ = '<form>'
form$ = form$ + '<address>Touch Technologies, Inc.<br>'
form$ = form$ + '10650 Scripps Ranch Blvd., Suite 100<br> '
form$ = form$ + 'San Diego, CA 92131</address>'
form$ = form$ + '<p> Touch Technologies created SheerPower!'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```



<blockquote>...</blockquote>

The **BLOCKQUOTE** tag defines text that is quoted from elsewhere. The text is displayed in an indented block surrounded by blank lines.

```
form$ = '<form>'
form$ = form$ + '<blockquote>"Glory is fleeting, but obscurity is forever."<br>'
form$ = form$ + '- Napoleon Bonaparte (1769-1821)</blockquote>'
form$ = form$ + '<p>This is a quote from Napoleon Bonaparte.'
form$ = form$ + '</form>'
input dialogbox form$: response$
end
```



<table>...</table>

The **TABLE** tag creates a table of columns and rows.

```
info_form$ = '<form>'
info_form$ = info_form$ + '<table>'
info_form$ = info_form$ + '<tr><td>Please enter<br> your age:'
info_form$ = info_form$ + '<td><input type=text name=age size=6>'
info_form$ = info_form$ + '<td>Please enter<br> your height:'
info_form$ = info_form$ + '<td><input type=text name=height size=6>'
info_form$ = info_form$ + '<td>Please enter<br> your weight:'
info_form$ = info_form$ + '<td><input type=text name=weight size=6>'
info_form$ = info_form$ + '</tr>'
info_form$ = info_form$ + '</table>'
info_form$ = info_form$ + '</form>'
input dialogbox info_form$: data$
end
```

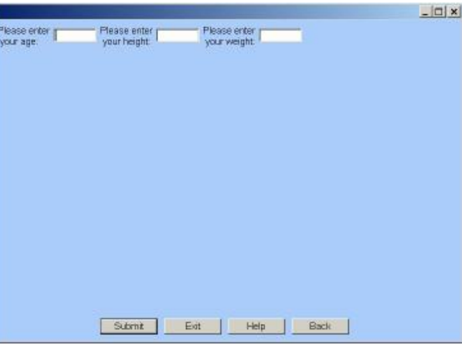


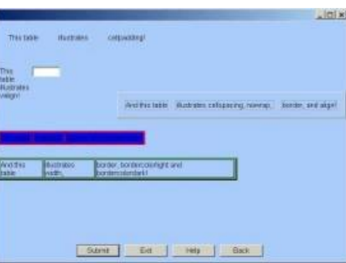
TABLE TAG ATTRIBUTES

Table 9-7 TABLE Tag Attributes

Attribute	Function
align	left, right, center
width	sets how wide the table can be
border	causes the table to be drawn with a border
border= *number*	draws the table with a border *number* pixels thick
cellpadding= *number*	separates the cell borders and the text with a padding of *number* pixels
cellspacing= *number*	separates cells with a gutter of *number* pixels
bgcolor= *colorname*	sets the background colour for the entire table
bordercolor= *colorname*	sets the border colour for the entire table
bordercolorlight= *colorname*	sets the border highlight colour for the entire table
bordercolordark= *colorname*	sets the border shadow colour for the entire table
valign	sets the vertical alignment for the entire table. "valign" is TOP or BOTTOM
nowrap	prevents word wrap within table entries

```
  form$ = '<form>'
  form$ = form$ + '<table cellpadding=20>'
  form$ = form$ + '<tr><td>This table'
  form$ = form$ + '<td>illustrates'
  form$ = form$ + '<td>cellpadding!'
  form$ = form$ + '</tr></table><p>'
  form$ = form$ + '<table valign=top><tr><td>'
  form$ = form$ + 'This<br>table<br>illustrates<br>valign!'
  form$ = form$ + '<td><input type=text size=5 name=blank>'
  form$ = form$ + '</tr></table><p>'
  form$ = form$ + '<table border=2 cellspacing=15 align=right nowrap>'
  form$ = form$ + '<tr><td>And this table'
  form$ = form$ + '<td>illustrates cellspacing, nowrap, '
  form$ = form$ + '<td>border, and align!'
  form$ = form$ + '</tr></table><p>'
  form$ = form$ + '<table border=2 bgcolor=blue bordercolor=red>'
  form$ = form$ + '<tr><td>This table'
  form$ = form$ + '<td>illustrates'
  form$ = form$ + '<td>bgcolor and bordercolor!'
  form$ = form$ + '</tr></table><p>'
  form$ = form$ + '<table border=2 bordercolorlight=green bordercolordark=black 
width=500>'
  form$ = form$ + '<tr><td>And this table'
  form$ = form$ + '<td>illustrates width,'
  form$ = form$ + '<td>border, bordercolorlight and bordercolordark!'
  form$ = form$ + '</tr></table>'
  form$ = form$ + '</form>'
  input dialogbox form$: data$
```

```
end
```

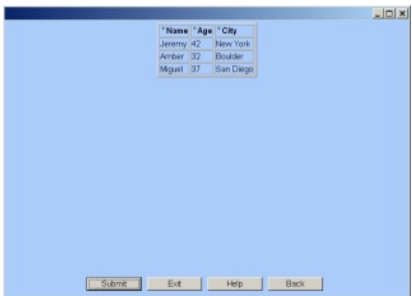


<th>...</th>

Valid only in a **table row**, the **TABLE HEADER** tag defines a header cell. The header is usually in bold text.

An optional **sort** attribute causes the table column defined by the table header to be sortable by the end user. When the table is presented to the end user, there will be a clickable diamond beside the table header. Clicking once on the diamond toggles the column between ascending and descending order.

```
form$ = '<form>'
form$ = form$ + '<table align=center border=2>'
form$ = form$ + '<tr><th sort>Name</th>'
form$ = form$ + '<th sort>Age</th>'
form$ = form$ + '<th sort>City</th>'
form$ = form$ + '<tr><td>Jeremy'
form$ = form$ + '<td>42'
form$ = form$ + '<td>New York</tr>'
form$ = form$ + '<tr><td>Amber'
form$ = form$ + '<td>32'
form$ = form$ + '<td>Boulder</tr>'
form$ = form$ + '<tr><td>Miguel'
form$ = form$ + '<td>37'
form$ = form$ + '<td>San Diego</tr>'
form$ = form$ + '</table>'
form$ = form$ + '</form>'
input dialogbox form$: data$
end
```



<tr>...</tr>

Valid only in a **table**, the **TABLE ROW** tag defines a row of cells that are defined with <td> tags.

```
form$ = '<form>'
form$ = form$ + '<table border=2>'
form$ = form$ + '<tr><td>This is'
form$ = form$ + '<td>one row of'
form$ = form$ + '<td>cells in a table!'
form$ = form$ + '</tr>'
form$ = form$ + '</table>'
form$ = form$ + '</form>'
input dialogbox form$: data$
end
```

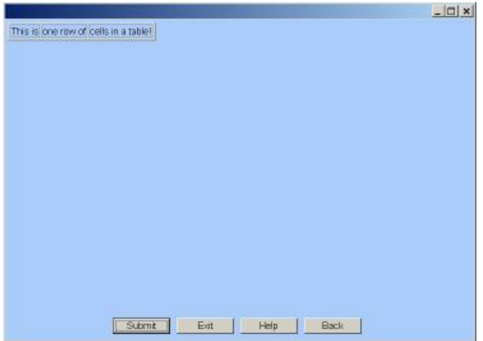


TABLE ROW TAG ATTRIBUTES

Table 9-8 TABLE ROW Tag Attributes

Attribute	Function
align	alignment of the text within the table cell (left, right, center)
valign	alignment of the text within the table cell (top, middle, bottom)
bgcolor= *colorname*	sets the background colour for the table row
bordercolor= *colorname*	sets the border colour for the table row
bordercolorlight= *colorname*	sets the border highlight colour for the table row
bordercolordark= *colorname*	sets the border shadow colour for the table row

<td>

Valid only in a **table row** tag, the **TABLE DATA** tag defines a table cell.

```
form$ = '<form>'
form$ = form$ + '<table border=2>'
form$ = form$ + '<tr><td>Table data...'
form$ = form$ + '<td>More table data....'
form$ = form$ + '<td>And more table data!'
form$ = form$ + '</tr>'
form$ = form$ + '</table>'
form$ = form$ + '</form>'
input dialogbox form$: data$
end
```

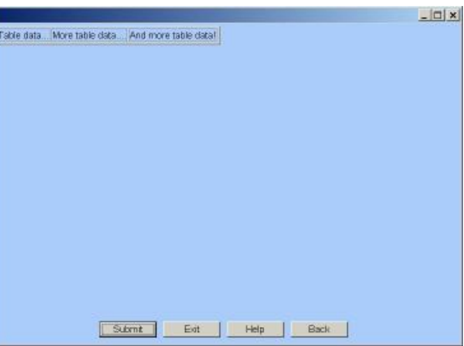


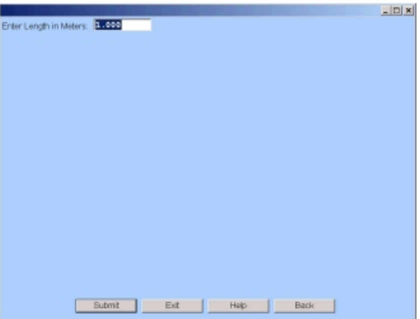
TABLE DATA TAG ATTRIBUTES

Table 9-9 TABLE DATA Tag Attributes

Attribute	Function
colspan= *number*	the number of columns this cell occupies
rowspan= *number*	the number of rows this cell occupies
nowrap	prevents word wrap within the cell
align	alignment of the text within the table cell (left, right, center)
valign	alignment of the text within the table cell (top, middle, bottom)
bgcolor= *colorname*	sets the background colour for the table cell
bordercolor= *colorname*	sets the border colour for the table cell
bordercolorlight= *colorname*	sets the border highlight colour for the table cell
bordercolordark= *colorname*	sets the border shadow colour for the table cell

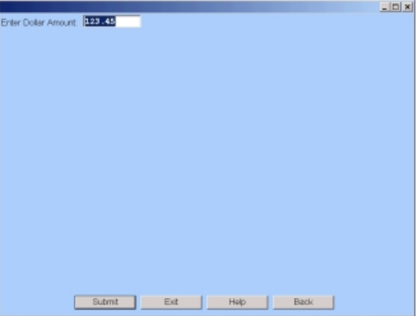
To insert string variable string data into a form:

```
a$ = '1.000'
form$ = form$ + '<form>Enter Length in Meters:' +
'<input name=one size=10 type=text value=' + quote$(a$) + '></form>'
 input dialogbox form$: ans$
```



To insert numeric variable data into a form:

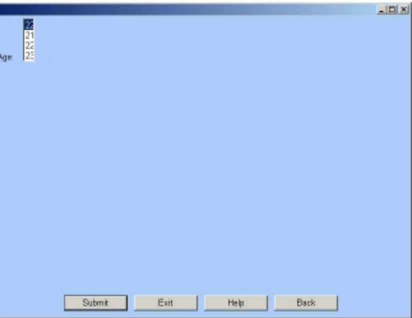
```
cash_amt = 123.45
form$ = form$ + '<form>Enter Dollar Amount:' +
'<input name=one size=10 type=text value=' + quote$(str$(cash_amt)) + '></form>'
input dialogbox form$: ans$
```



Values chosen from a dialogbox form drop-down menu can be stored into variables and used to be displayed later.

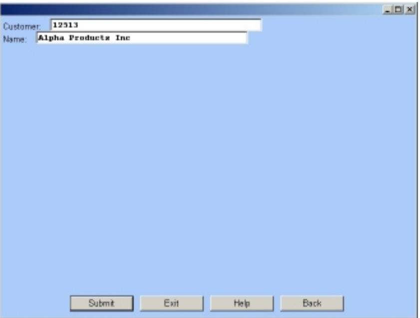
The way to do this is to regenerate the **DIALOGBOX** string with the default where it needs to be.

```
form_menu$ = '<form>'
form_menu$ = form_menu$ + 'Age: <select name=age>'
form_menu$ = form_menu$ + '<option value="_age_">_age_'
form_menu$ = form_menu$ + '<option value="21">21'
form_menu$ = form_menu$ + '<option value="22">22'
form_menu$ = form_menu$ + '<option value="23">23'
form_menu$ = form_menu$ + '</select>'
form_menu$ = form_menu$ + '</form>'

last_age$ = "23"
do
  default_age$ = last_age$
  default_form$ = replace$(form_menu$,'_age_='+default_age$)
  input dialogbox default_form$: choice$
  if _exit then exit do
  for item = 1 to pieces(choice$, chr$(26))
    z0$ = piece$(choice$, item, chr$(26))
    varname$ = element$(z0$, 1, '=')
    value$   = element$(z0$, 2, '=')
    select case varname$
    case 'age'
      last_age$ = value$
    case else
    end select
  next item
loop
end
```



Below is an example of creating a dialogbox form where data records from a data structure are accessed and able to be updated.

```
// Simple customer query
open structure cust: name 'sheerpower:\samples\customer', access outin
cust$ = '12513'
set structure cust, field custnbr: key cust$  // do the search
cname$   = cust(name)

do

  dbox$ = '<form>' +
          'Customer: <input type=text name=cnbr value=' + quote$(cust$) + '>' +
          '<br>' +
          'Name: <input type=text name=cname value=' + quote$(cname$) + '>' +
          '<br>' +
          '</form>'
  input dialogbox dbox$: formdata$
  if _exit or _back then exit do
  if _help then repeat do

  z0$ = element$(formdata$, 1, chr$(26))
  cust$ = element$(z0$, 2, '=')

  z0$ = element$(formdata$, 2, chr$(26))
  cname$ = element$(z0$, 2, '=')

  set structure cust, field custnbr: key cust$  // do the search
  if _extracted = 0 then
    message error: 'Cannot find '; cust$
    repeat do
  end if
  if cname$ = cust(name) then repeat do // nothing to do
  cust(name) = cname$  // update the name
loop
end
```



References used to gather information on HTML tags

http://www.htmlhelp.com

http://archive.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimerAll.html

http://www.webspawner.com/cc/html/alpha.html

http://www.willcam.com/cmat/html/crossref.html

http://www.htmlgoodies.com

A **LOOP** is a section of code that can be repeated. There are two types of loops.

- FOR loops execute a block of code a specific number of times
- DO loops execute a block of code either infinitely until a specified condition is met, or as long as a specified condition remains true

A **FOR loop** is used to repeat a block of code a specific number of times. FOR loops also inform the user how many times the loop was executed. A FOR loop might be used to input 10 similar data items or as a counter; for example, to count from 1 to 1000, or to make calculations from each of the 10 data items entered.

DO loops are used to execute a block of code until a specific condition is met. For instance, a DO loop might be useful to enter numbers until a 0 is entered. Additionally, DO loops are used to do calculations until two numbers match, or to continue a process until either the user chooses to stop or until a final result is reached.

Loops are *constructs*--they are created by using several statements which can only be used in conjunction with one another (FOR/NEXT, DO/LOOP). The statements which make up the constructs are described together.

FORMAT:

```
        FOR index_var = num_expr1 TO num_expr2 [STEP num_expr3]
                ---
                ---  block of code
                ---
        NEXT index_var
```

EXAMPLE:

```
  dim name$(4)
  for j = 1 to 4
    input 'Enter name': name$(j)
    print j; ' '; name$(j)
  next j
  print 'Finished'
  print 'Final value:'; j
  end


Enter name? Jack
 1  Jack
Enter name? Tom
 2  Tom
Enter name? Sue
 3  Sue
Enter name? Toby
 4  Toby
Finished
Final value: 5
```

PURPOSE:

The **FOR** loop executes a block of code a specific number of times. This construct can be used to repeat a section of code a certain number of times.

DESCRIPTION:

In the above example, the INPUT and PRINT statements make up the body of the loop. This block of code is executed each time the loop is repeated. (For clarity, the body of the loop is indented two spaces.) The **FOR** statement marks the beginning of the loop and determines how many times the loop is repeated. The **NEXT** statement marks the end of the loop.

```
        index variable
               |
               V
            for J = 1 to 4   <-- limit expression
                    ^
                    |
            initial expression
```

General Information

- *index_var* can be an integer or a real numeric variable. **It cannot be an array element**.
- Each FOR statement must have a corresponding NEXT statement or an exception will be generated.
- The index variable can be changed from within the loop. However, this can affect the number of times the loop is repeated.

The **index variable** keeps track of how many times the loop has been executed. The **initial expression** is the number SheerPower begins counting at. The **limit expression** is the number SheerPower stops counting at. In the example, SheerPower counts from 1 to 4, so the loop executes four times.

When SheerPower runs the example program, it executes the loop four times. The first time the FOR statement is executed, the variable J is set to 1. SheerPower executes the body of the loop. Since J = 1, SheerPower inputs NAME$(1), Jack, and prints NAME$(J).

```
        J = 1        Enter name? Jack
                      1  Jack
```

When SheerPower reaches the NEXT J, it adds one to J and jumps back to the beginning of the loop. J is now 2. SheerPower checks to see if J is greater than 4. Since J isn't greater than 4, SheerPower repeats the loop. When SheerPower executes the loop for the last time, it jumps back to the beginning of the loop and checks to see if J is greater than 4. Since J **is** greater than 4, SheerPower jumps to the statement following the NEXT J (PRINT 'Finished') and continues normal program execution.

```
J = 1        Enter name? Jack
              1  Jack
J = 2        Enter name? Tom
              2  Tom
J = 3        Enter name? Sue
              3  Sue
J = 4        Enter name? Toby
4  Toby
Finished
Final value: 5
```

By default, when a FOR loop is executed, SheerPower increments the index variable by one. The increment can be changed with the STEP option. The format of the FOR statement with the STEP option is:

```
        FOR index_var = num_expr1 TO num_expr2 STEP num_expr3
            ---
            ---  block of code
            ---
        NEXT index_var
```

num_expr3 is a numeric expression specifying the increment. Each time the FOR statement is executed, SheerPower adds the increment to the index variable. SheerPower stops executing the loop when the index variable is greater than the limit.

```
  dim name$(4)
  for j = 1 to 4 step 3
    input 'Enter name': name$(j)
    print j; ' '; name$(j)
  next j
  print 'Finished'
  print 'Final value:'; j
  end


Enter name? Fred
 1  Fred
Enter name? John
 4  John
Finished
Final value: 7
```

FOR loops can be nested. A nested loop is a loop which begins and ends inside of another loop. **Loops cannot overlap**. The inner loop must begin and end completely within the outer loop.

```
  dim name$(4)
  for j = 1 to 4      //<--- start of outer loop
    input name$(j)
    for k = 1 to j
      print name$(k); '  ';            //<--- inner loop
    next k
    print
  next j                               //<--- end of outer loop
  print 'Finished'
  end


? FRED                  <--- type in FRED
FRED
? JOHN                  <--- type in JOHN
FRED  JOHN
? MARY                  <--- type in MARY
FRED  JOHN  MARY
? KATE                  <--- type in KATE
FRED  JOHN  MARY  KATE
Finished
```

FORMAT:

```
        EXIT FOR
```

EXAMPLE:

```
  for i = 1 to 5
    input 'Your name, please': name$
    if  _exit  then exit for
    print 'Hello, '; name$
  next i
  print 'Finished'
  end


Your name, please? James
Hello, James
```

```
Your name, please? Marian
Hello, Marian
Your name, please? exit
```

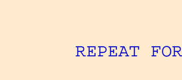
PURPOSE:

EXIT FOR is used to exit from a FOR loop.

DESCRIPTION:

When SheerPower executes an **EXIT FOR** statement, it jumps to the first statement following the matching NEXT statement. EXIT FOR can be used only within FOR loops. If EXIT FOR is used within a nested loop, SheerPower exits the innermost loop.

FORMAT:



```
REPEAT FOR
```

EXAMPLE:

```
  for i = 1 to 3
    print i
    input 'Your name, please': name$
    if  name$ = ''  then repeat for
    print 'Hello, '; name$
  next i
  end


 1
Your name, please? George
Hello, George
 2
Your name, please?
 2
Your name, please? Sam
Hello, Sam
 3
Your name, please? Tom
Hello, Tom
```

PURPOSE:

REPEAT FOR is used to increment the index variable.

DESCRIPTION:

REPEAT FOR repeats all or part of the body of a loop. REPEAT FOR can be used only in FOR loops. When SheerPower executes REPEAT FOR, it jumps to the first statement following the FOR statement.

If REPEAT FOR is used within a nested loop, SheerPower repeats the innermost loop.

```
for i = 1 to 10
  for j = 1 to 5
       print j                               //SheerPower will
    input 'Your name, please': name$         //repeat this
    if  name$ = ''  then repeat for          //inner loop
    print 'Hello, '; name$
  next j
  print 'We now have'; i; 'set(s) of names.'
next i
end
```

FORMAT:

```
        ITERATE FOR
```

EXAMPLE:

```
  for i = 1 to 3
    print i
    input 'Your name, please' : name$
    if  name$ = 'Skip'  then iterate for
    print 'Hello, '; name$
  next i
  end


 1
Your name, please? Toby
Hello, Toby
 2
Your name, please? Skip
 3
Your name, please? Sam
Hello, Sam
```

PURPOSE:

ITERATE FOR is used to skip code processing.

DESCRIPTION:

When SheerPower executes **ITERATE FOR**, it jumps to the NEXT statement. Any statements between the ITERATE FOR and the NEXT statement will be skipped.

If ITERATE FOR is used in a nested loop, SheerPower iterates the innermost loop.

```
  // count to ten, but skip a few numbers
  for idx = 1 to 10
    if idx = 2 or idx = 6 then iterate for
    print 'On '; idx
  next idx
  end


On  1
On  3
```

```
On  4
On  5
On  7
On  8
On  9
On  10
```

FORMAT:

```
        DO [WHILE expr | UNTIL expr]
                ---
                ---  block of code
                ---
        LOOP [WHILE expr | UNTIL expr]
```

EXAMPLE:

```
  a = 3
  do until a = 0
    input 'Your name, please': name$
    print 'Hello, '; name$
    a = a - 1
  loop
  end


Your name, please? Sam
Hello, Sam
Your name, please? Sue
Hello, Sue
Your name, please? Bart
Hello, Bart
```

PURPOSE:

A **DO LOOP** is used to execute a block of code repeatedly until a specified condition is met.

DESCRIPTION:

The simplest type of **DO LOOP** is an infinite DO LOOP:

```
  do
    input 'Your name, please' : name$
    print 'Hello, '; name$
  loop
  end
```

In the above example, the INPUT and PRINT statements make up the body of the loop. This block of code is executed each time the loop is repeated. **DO** begins the loop. **LOOP** marks the end of the loop. When SheerPower reaches the LOOP statement, it jumps back to DO and executes the loop again.

The `[Alt/B]` command or clicking on the STOP icon in the toolbar can be used to break out of an infinite DO loop.

DO loops can be nested. **Loops cannot overlap**. The inner loop must be completely within the DO and LOOP statements of the outer loop.

```
      start of
      outer loop  ---     do
                            a = 5
                            do until a = 0
                         /      input 'Your name' : name$
             inner loop         print 'Hello, '; name$
                          \     a = a - 1
                            loop
         end of             print 'Done with a loop'
         outer loop ---   loop
                          end
```

DO loops can be made conditional with WHILE and UNTIL options. WHILE and UNTIL set up a condition. The loop is executed if the condition is met.

FORMAT:

```
        WHILE cond_expr
```

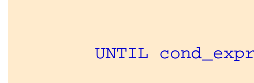
EXAMPLE:

```
  a = 3
  do
    input 'Your name, please': name$
    print 'Hello, '; name$
    a = a - 1
  loop while a > 0
  print 'Finished'
  end


Your name, please? FRED
Hello, FRED
Your name, please? JOHN
Hello, JOHN
Your name, please? KATE
Hello, KATE
Finished
```

DESCRIPTION:

cond_expr is a conditional expression. When SheerPower executes the **WHILE** option, it evaluates the conditional expression as either **TRUE** (1) or **FALSE** (0). If the expression is TRUE, the condition is met and SheerPower executes the loop. SheerPower continues executing the loop until the expression becomes FALSE. When the expression becomes FALSE, the condition is not met. SheerPower stops executing the loop and jumps to the statement following LOOP.

FORMAT:



```
        UNTIL cond_expr
```

EXAMPLE:

```
  a = 3
  do until a = 0
    input 'Your name, please': name$
    print 'Hello, '; name$
    a =  a - 1
  loop
  print 'Finished'
  end


Your name, please? FRED
Hello, FRED
Your name, please? JOHN
Hello, JOHN
Your name, please? KATE
Hello, KATE
Finished
```

DESCRIPTION:

cond_expr is a conditional expression. When SheerPower executes the **UNTIL** option, it evaluates the conditional expression as either **TRUE** (1) or **FALSE** (0). If the expression is FALSE, SheerPower executes the loop. SheerPower continues executing the loop until the expression becomes TRUE. When the expression becomes TRUE, SheerPower stops executing the loop and jumps to the statement following LOOP.

Placement of WHILE and UNTIL

WHILE and UNTIL can be attached to the DO and/or to the LOOP statements. Whenever SheerPower encounters a WHILE or UNTIL clause, it checks whether to execute the loop. The placement of the WHILE and UNTIL clauses affects the execution of the loop.

If a WHILE or UNTIL is attached to the DO statement, SheerPower first checks to see whether the condition is TRUE or FALSE before it executes the loop (again). In the case of a WHILE statement, if the condition is still met (i.e., is TRUE), SheerPower executes the loop. If the condition is not met (i.e., is FALSE or is no longer TRUE), SheerPower does not execute the loop.

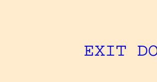
In the case of an UNTIL statement, if the condition has not been met (or is still FALSE), SheerPower executes the loop once more. If the condition has been met (i.e., is TRUE), SheerPower does not execute the loop again.

Creating Two Conditions

WHILE and UNTIL options can be placed at both ends of the loop. SheerPower evaluates each expression in turn. When it finds that one of the conditions has or has not been met (depending upon whether it is a WHILE or UNTIL clause), SheerPower stops executing the loop. For example, when the following program runs, SheerPower executes the loop until A equals 5 or the user enters EXIT.

```
  dim name$(4)
  a = 1
  do until a = 5
    input 'Your name, please' : name$(a)
    a = a + 1
  loop while not _exit
  print 'Finished'
  end
```

FORMAT:



EXAMPLE:

```
do
  input 'Your name, please' : name$
  if _exit  then  exit do
  print 'Hello, '; name$
loop
print 'Finished'
end


      Your name, please? Fred
Hello, Fred
Your name, please? exit       <---- type in 'exit'
Finished
```

PURPOSE:

EXIT DO is used to exit from a DO loop.

DESCRIPTION:

When SheerPower executes an **EXIT DO** statement, it jumps to the first statement following the LOOP or END DO statement. If EXIT DO is used within a nested loop, SheerPower exits the innermost loop.

DO...END DO is a single iteration loop. The code between DO and END DO is processed only once unless conditional code specifies exiting or repeating the DO.

FORMAT:

```
        REPEAT DO
```

EXAMPLE:

```
do
  input 'Your name, please': name$
  if  _exit  then exit do
  if  name$ = ''  then repeat do
  print 'Hello, '; name$
loop
end

Your name, please? Fred
Hello, Fred
Your name, please?
Your name, please? exit   <---- type in 'exit'
```

PURPOSE:

REPEAT DO is used to repeat part of a DO loop.

DESCRIPTION:

REPEAT DO repeats all or part of the body of a loop. When SheerPower executes REPEAT DO, it jumps to the first statement following the DO statement.

If REPEAT DO is used within a nested loop, SheerPower repeats the innermost loop.

```
do
  i = i + 1
  do                         //<------SheerPower will repeat this inner loop
    input 'Your name, please': name$
    if  _exit  then exit do
    if  name$ = ''  then repeat do
    print 'Hello, '; name$
  loop                       //<------
  print 'We now have'; i; 'set(s) of names.'
loop
end
```

FORMAT:

```
        ITERATE DO
```

EXAMPLE:

```
do
  input 'Your name, please': name$
  if  _exit  then exit do
  if  name$ = 'SKIP'  then iterate do
  print 'Hello, '; name$
loop
end


Your name, please? FRED
Hello, Fred
Your name, please? SKIP
Your name, please? exit
```

PURPOSE:

ITERATE DO is used to repeat a loop, skipping part of the loop.

DESCRIPTION:

ITERATE DO repeats a loop. When SheerPower executes ITERATE DO, it jumps to the LOOP or END DO statement. Any statements between the ITERATE DO and the end of the DO block statement will be skipped.

If ITERATE DO is used in a nested loop, SheerPower iterates the innermost loop.

```
do
  let i = i + 1
  do                             //<----- SheerPower will iterate
    input 'Your name, please' : name$
    if  name$ = 'SKIP'  then iterate do
    if  _exit  then exit do
    print 'Hello, '; name$
  loop                           //<---- this inner loop
  print 'We now have'; i; 'set(s) of names.'
loop
end
```

FORMAT:

```
EXECUTE str_expr
```

EXAMPLE:

```
input 'Enter a video attribute': video$
z$ = 'print ' + video$ + &
    ': "This will be printed using ' + video$ + '"'
execute z$
end

Enter a video attribute? bold
This will be printed using bold
```

EXAMPLE:

```
nbr_fields = 5
dim check$(nbr_fields)
check$(1) = &
  'do \' &
  + '  if  len(ans$) <> 9  then \' &
  + '    message error : "SSN must be 9 digits" \' &
  + '    exit do \' &
  + '  end if \' &
  + '  if  not valid(ans$, "number")  then  \' &
  + '    message error : "SSN must be numeric" \' &
  + '    exit do \' &
  + '  end if \' &
  + '  print "SSN is valid" \' &
  + 'end do'
field_nbr = 1
input 'SSN' : ans$
execute check$(field_nbr)
end

SSN? 123456789
SSN is valid
```

DESCRIPTION:

EXECUTE allows new code to be incorporated into the program at runtime. It is used mostly for generalized procedures, utilities, and tools.

A string is built which contains the SheerPower statements to execute. Multiple SheerPower statements are separated by either a line feed character [chr$(10)] or a "\" character.

When an EXECUTE statement is encountered, SheerPower compiles the code contained in the string and then runs that code. All program variables are available to the executed code, and any variables established by the executed code are available to the rest of the program.

An executed string is compiled only once. When a string has been compiled, the code contained within that string is processed as efficiently as the main program code.

Note: There can not be any COMMENTS in an execute string because the first "!" or "//" causes the rest of the string to be assumed as part of the comment. The "\" or CHR$(10) is an "end-of-statement" indicator (not "end-of-line").

The EXECUTE statement makes the coding of powerful generalized routines very easy.

Conditionals are constructs which specific blocks of code to be executed depending on one or more conditions. For instance, suppose you are doing a tax program. You need to use the EZ form if certain conditions are met, the short form if others are met,

and the long form otherwise. You can use a conditional to determine which form to use.

There are two types of conditionals: IF and SELECT. The IF construct is useful to check one or more conditions and execute a different block of code depending on the result. For instance, say that you need to print one statement if the user is male and under 20, another if the user is male and between 20 and 40, and still another if the user is male and over 40. The IF construct works well in this kind of situation.

The SELECT CASE construct is useful when comparing one main expression with several values and executing a different block of code for each possible match. For instance, suppose that in the tax program we mentioned before, you need to execute a different block of code depending on the user's tax bracket. SELECT CASE would let you compare a main expression---BRACKET---with all the possible tax brackets (10000-15000, 15000-25000, etc.).

FORMAT:

```
IF cond_expr THEN statement [ELSE statement]

        or

IF cond_expr1 THEN
    ---
    ---  block of code
    ---
[ELSEIF cond_expr2 THEN
    ---
    ---  block of code
    ---   ...]
[ELSE
    ---
    ---  block of code
    ---      ]
END IF
```

EXAMPLE:

```
find_age_and_sex

routine find_age_and_sex
  input prompt 'Enter your age: ': age
  input prompt 'Enter your sex: ': sex$
  if  ucase$(sex$[1:1]) = 'M'  then exit routine
  if  age < 20  then
    print 'Please go to line A.'
  elseif  age > 19  and age < 40  then
    print 'Please go to line B.'
  else
    print 'Please go to line C.'
  end if
end routine


Enter your age: 25
Enter your sex: female

Please go to line B.
```

PURPOSE:

The **IF** construct is used to execute a statement or block of code only under specific conditions.

DESCRIPTION:

The simplest form of the **IF** construct is a one-line statement:

```
IF cond_expr THEN statement
```

cond_expr is a conditional expression. SheerPower evaluates this expression as either **TRUE** (1) or **FALSE** (0). If the condition is TRUE, SheerPower executes the statement following the **THEN**. If the condition is FALSE, SheerPower skips the statement following the THEN and goes to the next line.

In the example program, when SheerPower executes the first IF statement, it evaluates the conditional expression, SEX$[1:1] = 'M'. IF the user is 'Male' the condition is TRUE and SheerPower executes the statement following the THEN and exits the routine.

IF can be used to execute a block of code. The IF block construct looks like this:

```
IF cond_expr THEN
      ---
      ---  block of code
      ---
END IF
```

If the conditional expression is TRUE, SheerPower executes the block of code beginning on the next line. **END IF** marks the end of this block of code. If the expression is FALSE, SheerPower skips to the statement following the END IF.

The **ELSE** option executes a statement if the conditional expression is FALSE. The format of the IF statement with the ELSE option is:

```
IF cond_expr THEN statement ELSE statement
```

When SheerPower executes the IF statement, it evaluates the conditional expression. If the expression is TRUE, the statement following the THEN is executed. If the expression is FALSE, the ELSE statement is executed. (Please refer to previous example.)

```
Enter your age: 19
Enter your sex: Female

Please go to line A.
```

In the above program, when SheerPower executes the first IF statement, it evaluates the expression, SEX$[1:1] = 'M'. Since the user is Female, the expression is FALSE, so SheerPower skips the THEN clause and jumps to the ELSE clause. SheerPower executes the code between the ELSE clause and the END IF.

The ELSE option can be used to execute a block of code if the conditional expression is FALSE. The IF construct with the ELSE option looks like this:

```
IF cond_expr THEN
      ---
      ---  block of code
      ---
ELSE
      ---
      ---  block of code
      ---
END IF
```

If the conditional expression is TRUE, SheerPower executes the block of code between the IF and the ELSE statements. If the expression is FALSE, SheerPower executes the block of code between the ELSE and the END IF.

```
find_age_and_sex

routine find_age_and_sex
  input prompt 'Enter your age: ': age
  input prompt 'Enter your sex: ': sex$
  if  ucase$(sex$[1:1]) = 'M'  then exit routine
  if  age < 40  then
    print 'Please go to line A.'
  else
    print 'Please go to line B.'
  end if
end routine


Enter your age: 45
Enter your sex: female

Please go to line B.
```

In the above program, when SheerPower executes the second IF statement, it checks to see if "AGE < 40". Since AGE is not less than 40, the condition is FALSE. SheerPower skips the code following the THEN and jumps to the ELSE clause. SheerPower executes the code following the ELSE clause.

The **ELSEIF** option sets up a new condition to check. The format of the IF construct with the ELSEIF option is:

```
        IF cond_expr1 THEN
              ---
              ---  block of code
              ---
        ELSEIF cond_expr2 THEN
              ---
              ---  block of code
              ---
        ELSE
              ---
              ---  block of code
              ---
        END IF
```

ELSEIF establishes a new condition. SheerPower evaluates this condition. If the condition is TRUE (1), SheerPower executes the code following the ELSEIF. If the condition is FALSE (0), SheerPower jumps to the next clause in the IF construct.

```
find_age_and_sex

routine find_age_and_sex
  input prompt 'Enter your age: ': age
  input prompt 'Enter your sex: ': sex$
  if  ucase$(sex$[1:1]) = 'M'  then exit routine
  if  age < 20  then
    print 'Please go to line A.'
  elseif  age > 19 and age < 40  then
    print 'Please go to line B.'
  else
    print 'Please go to line C.'
  end if
end routine


Enter your age: 25
Enter your sex: female

Please go to line B.
```

In the above program when SheerPower executes the second IF statement, it checks to see if "AGE < 20". Since AGE is not less than 20, the first condition is FALSE. SheerPower skips the code following the THEN and checks the condition set up by the

ELSEIF. Since "AGE > 19" and "AGE < 40", the second condition is TRUE and SheerPower executes the code following the ELSEIF and prints 'Please go to line B.', then exits the conditional.

FORMAT:

```
        SELECT CASE main_expr
        CASE expr1[, expr2,...]
            ---
            ---  block of code
            ---
        [CASE expr3[, expr4,...]
            ---
            ---  block of code
            ---            ...]
        [CASE IS {numeric operator | boolean operator} expr5
            ---
            ---  block of code
            ---            ...]
        [CASE ELSE
            ---
            ---  block of code
            ---            ...]
        END SELECT
```

EXAMPLE:

```
  do
    input 'Your income per year': income
    if  _back or _exit  then exit do
    select case income
    case 0
      print 'No income?'
    case is < 0
      print 'A negative income?  You are in debt!'
    case is > 0
      print 'A positive income.'
    end select
  loop
  end


Your income per year? 0
No income?
Your income per year? -15000
A negative income? You are in debt!
Your income per year? 30000
A positive income.
Your income per year? exit
```

PURPOSE:

Sometimes it is necessary to compare one main expression with several values and execute a different block of code for each possible match. **SELECT CASE** is used to check a set of conditions and execute code depending on the results.

DESCRIPTION:

The **SELECT CASE** statement begins the construct and gives the main expression (*main_expr*). In the example, the main expression is INCOME. The **CASE** statements are compared with the main expression. The first CASE expression (*expr1*) is 0. Following this CASE is a block of code. If INCOME = 0 the block of code following CASE 0 is executed.

- The main expression and all CASE expressions must be the same data type; for example, if the main expression is a string expression, all the CASE expressions must be strings also. If the data types do not match, an exception is generated.
- If the main expression is an integer and a real numeric CASE expression is given, the CASE expression is rounded and the integer portion is compared.
- There must be at least one CASE expression.
- If none of the CASE expressions match (and there is no CASE ELSE), an exception is generated.

Each **CASE** statement can include several expressions separated by commas. SheerPower compares each of the expressions in a CASE statement. If any of them match, the block of code following the CASE is executed.

```
  do
    input 'Procedure (add, del, exit)': pro$
    if  _exit  then exit do
    pro$ = ucase$(pro$)
    select case pro$
    case 'ADD'
      print 'Adding...'
    case 'DEL', 'delete'
      print 'Deleting...'
    end select
  loop
  end


Procedure (add, del, exit)? add
Adding...
Procedure (add, del, exit)? del
Deleting...
Procedure (add, del, exit)? exit
```

The following example illustrates how to check for a range of values:

```
  a = 5
  select case a
    case 1 : print 'one'
    case 2 to 6 : print 'range'
    case else : print 'else'
  end select
  b$ = 'c'
  select case b$
    case 'a' : print 'a'
    case 'b' to 'e' : print 'range'
    case else : print 'else'
  end select
  end


range
range
```

The **CASE ELSE** option executes a block of code only if none of the CASE expressions match. SELECT CASE with the CASE ELSE option looks like this:

```
        SELECT CASE main expr
        [CASE expr1, expr2...
            ---
            ---  block of code
            ---             ...]
        [CASE IS {numeric operator| boolean operator} expr3
            ---
            ---  block of code
            ---             ...]
        CASE ELSE
            ---
            ---  block of code
            ---
        END SELECT
```

CASE ELSE must follow the last CASE statement. If none of the CASE expressions match, SheerPower executes the block of code following the CASE ELSE statement.

```
  do
    input 'Procedure (add, del, exit)' : pro$
    if  _exit  then exit do
    pro$ = ucase$(pro$)
    select case pro$
    case 'ADD'
      print 'Adding...'
    case 'DEL', 'DELETE'
      print 'Deleting...'
    case else
      message error: 'Procedure must be: add, del or exit'
      repeat do
    end select
  loop
  end


Procedure (add, del, exit)? add
Adding...
Procedure (add, del, exit)? del
Deleting...
Procedure (add, del, exit)? funny

      Procedure must be add, del, or exit

Procedure (add, del, exit)? EXIT
```

CASE IS is used to form a conditional expression to be checked against the main expression. The format of the CASE IS option is:

```
        CASE IS {relational operator} expr
```

When the CASE IS statement executes, SheerPower compares *expr* to the *main_expr* using the relational operator.

```
  do
    input 'Your income per year': income
    if  _back or _exit  then exit do
    select case income
    case 0
      print 'No income?'
    case is < 0
      print 'A negative income?  You are in debt!'
    case is > 0
      print 'A positive income.'
    end select
  loop
  end


Your income per year? -15000
A negative income?  You are in debt!
Your income per year? 0
No income?
Your income per year? 25000
A positive income.
Your income per year? exit
```

FORMAT:

```
CHAIN 'file_spec'
```

EXAMPLE:

```
  line input 'Your name (last, first)': name$
  open #1: name 'storage.txt', access output
  print #1: name$
  close #1
  input 'Add to CLIENT structure (Y/N)': reply$
  if  reply$ = 'Y'  then chain 'ADD'
  end


Your name (last, first)? Woods, Jack
Add to CLIENT structure (Y/N)? N
```

DESCRIPTION:

CHAIN exits the current program and runs the program specified.

file_spec is the specification for the program being chained to. The file specification can be any string expression. SheerPower searches for the file specified, then exits the current program and runs the program named. Control **does not** return to the current program when the chained program is finished. If SheerPower cannot find the file, or if the file is not an executable program, an exception is generated.

When SheerPower executes the CHAIN statement, it:

- writes all active output buffers, closes all files in the current program and releases all storage
- exits the current program and executes the program named
- does not pass any variables, functions, etc., from the previous program

FORMAT:

```
PASS [NOWAIT | NORETURN | WINDOW | TIMEOUT] [:] STRING_EXPR
```

EXAMPLE:

Important note on the following example:

The following example will run the calculator program in your computer.

```
  input 'What program would you like to run': prog$
  pass prog$
  end


What program would you like to run? calc           <--------- type in 'calc'
```

PURPOSE:

PASS is used to perform system commands without leaving the SheerPower environment or exiting a program. In Windows, SheerPower *passes* the command to the operating system.

SheerPower supports using the PASS command from a *captive* account. This allows you to use SheerPower 4GL for captive menu situations.

DESCRIPTION:

PASS passes the specified string expression to the operating system command interpreter. Generally, it passes the string to the operating system. The operating system will respond to the string as it would if you entered it at the DOS prompt. When the system finishes, control returns to the SheerPower program. By default, the Command Prompt window is completely suppressed when PASS is used.

When the **NOWAIT** option is used with PASS, the operating system executes the passed command and immediately returns to SheerPower without waiting for the passed command to finish.

EXAMPLE:

```
print 'Start the calculator'
pass 'calc'
print 'We are back from using the calculator'
delay
print 'Now we start the calculator, but return instantly.'
pass nowait: 'calc'
print 'We are back already -- even though the calculator is still active.'
delay
end
```

The **PASS NORETURN** statement passes a command to the operating system but does not return to SheerPower.

EXAMPLE:

```
print 'B E F O R E'
delay 2
pass noreturn: 'calc' //<--- start up Windows calculator
end


B E F O R E
```

By default, the **PASS** statement suppresses the COMMAND PROMPT window. This then allows one to run commands in the background without the Command Prompt window displaying or flashing on the screen at all.

To display the Command Prompt window when the PASS statement is executed, use the **PASS WINDOW:** statement.

When the SP4GL Console Window is displayed

When a program is run from SPDEV, the SP4GL Console Window will always open allowing you to debug the program. To test the PASS WINDOW: statement, run the program directly from the SP4GL Console Window or save the program file and run it by double-clicking on the program file. In this way you will see the Command Prompt window when the PASS WINDOW: statement is used.

When the debug console window is closed, any pending pass command is also terminated

The following sample program will create a file called "x.txt" in your SheerPower directory that contains a listing of the directory contents.

EXAMPLE:

```
 pass window: 'dir>x.txt'
 end

// file contents
 Volume in drive C is OS
 Volume Serial Number is 9999-4444

 Directory of C:\SheerPower

09/06/2008  06:45 PM    <DIR>          .
09/06/2008  06:45 PM    <DIR>          ..
07/24/2007  10:39 PM    <DIR>          ars
07/12/2008  10:38 PM    <DIR>          samples
07/12/2008  10:40 PM               127 sheerpower.ini
09/03/2008  11:43 PM         2,183,227 sp4gl.exe
09/06/2008  06:45 PM               498 sp4gl_system_info.txt
09/03/2008  11:41 PM         1,916,993 SPDev.exe
09/05/2008  09:06 PM               889 spdev_nonsp_user.ini
09/06/2008  06:38 PM             5,710 spdev_profile_user.ini
09/05/2008  06:32 PM             9,404 spdev_sp4gl_user.ini
09/06/2008  05:51 PM               472 spdev_system_info.txt
07/25/2007  10:35 AM             4,517 spdev_tools_user.ini
07/24/2007  10:39 PM    <DIR>          spdoc
09/01/2008  03:31 PM    <DIR>          sphandlers
08/31/2008  10:43 PM    <DIR>          sptools
09/05/2008  06:33 PM             1,151 Uninstall.dat
09/05/2008  06:33 PM           200,704 Uninstall.exe
09/06/2008  06:46 PM                 0 x.txt
              12 File(s)      4,323,692 bytes
               7 Dir(s)  34,208,325,632 bytes free
```

PASS WINDOW will work with the NOWAIT, NORETURN and TIMEOUT options. The sample program below will create a file called "y.txt" in your SheerPower directory that contains a list of the contents of the directory.

```
 pass nowait, window: 'dir>y.txt'
 end

// file contents
 Volume in drive C is OS
 Volume Serial Number is 9999-4444

 Directory of C:\SheerPower

09/06/2008  06:45 PM    <DIR>          .
09/06/2008  06:45 PM    <DIR>          ..
07/24/2007  10:39 PM    <DIR>          ars
07/12/2008  10:38 PM    <DIR>          samples
07/12/2008  10:40 PM               127 sheerpower.ini
09/03/2008  11:43 PM         2,183,227 sp4gl.exe
09/06/2008  06:45 PM               498 sp4gl_system_info.txt
09/03/2008  11:41 PM         1,916,993 SPDev.exe
09/05/2008  09:06 PM               889 spdev_nonsp_user.ini
09/06/2008  06:38 PM             5,710 spdev_profile_user.ini
09/05/2008  06:32 PM             9,404 spdev_sp4gl_user.ini
09/06/2008  05:51 PM               472 spdev_system_info.txt
07/25/2007  10:35 AM             4,517 spdev_tools_user.ini
07/24/2007  10:39 PM    <DIR>          spdoc
09/01/2008  03:31 PM    <DIR>          sphandlers
08/31/2008  10:43 PM    <DIR>          sptools
09/05/2008  06:33 PM             1,151 Uninstall.dat
09/05/2008  06:33 PM           200,704 Uninstall.exe
09/06/2008  06:46 PM                 0 y.txt
              12 File(s)      4,323,692 bytes
               7 Dir(s)  34,208,325,632 bytes free
```

PASS TIMEOUT allows you to specify the maximum amount of time for a command to complete (in seconds). If the command has not already completed before the time specified, it will be completed at that time, and regular processing will continue.

The TIMEOUT option has no effect if used with NOWAIT and NORETURN. It does work with the WINDOW option.

The following sample program will open the Windows Calculator program, and close it automatically after 10 seconds has passed.

EXAMPLE:

```
pass timeout 10: 'calc'
end
```

FORMAT:

```
pass print: string_expr
```

EXAMPLE:

```
//  Create your output text file:
outfile$ = 'myfile.txt'
open file out_ch: name outfile$, access output
for i=1 to 100
  print #out_ch: i, sqr(i)
next i
close #out_ch

// Now print it out to the default printer
pass print: outfile$
end
```

PURPOSE:

The **PASS** statement can be used to print output from SheerPower.

DESCRIPTION:

PASS PRINT will locate the program associated with the filetype being used, then ask that program to print the file to whatever printer is currently selected for that application.

FORMAT:

```
pass url: str_exp
```

EXAMPLE:

```
//  Create your output html file
outfile$ = 'myfile.html'
open file out_ch: name outfile$, access output
print #out_ch: '<html><body>'
print #out_ch: '<table border=3 bgcolor=lightblue>'
for i=1 to 100
  print #out_ch: '<tr>'
  print #out_ch: '<td>'; i; '<td>'; sqr(i)
  print #out_ch: '</tr>'
next i
print #out_ch: '</table>'
print #out_ch: '</body></html>'
close #out_ch

// Now envoke the browser to open the file.
pass url: outfile$
end
```

PURPOSE:

PASS URL opens any URL. The above example illustrates PASS URL opening an .HTML file.

DESCRIPTION:

The PASS URL statement can be used to open any webpage, either local or remote. A new browser window is opened by PASS URL. If the URL references a filetype handled by an external program (such as an .AVI movie file), then the appropriate external program will be run (such as the Windows Media Player).

By using the PASS URL statement, web-based applications and multi-media features can be integrated into SheerPower 4GL applications

FORMAT:

```
        DISPATCH str_expr
             .
             .
             .
        target
             ---
             ---  block of code
             ---
        END ROUTINE
```

EXAMPLE:

```
  input 'Routine name', default 'add_info': routine$
  dispatch routine$
  stop

routine add_info
  print 'Adding information...'
end

routine change_info
  print 'Changing information...'
end


Routine name? add_info
Adding information...
```

PURPOSE:

DISPATCH executes a routine that the program determines at runtime.

DESCRIPTION:

DISPATCH looks at the contents of the string expression (*str_expr*), searches for a routine with that name and goes to the routine.

str_expr is the name of the subroutine to execute.

SET and **ASK** statements find and change characteristics within a SheerPower program. SET sets various characteristics, and ASK returns the value of various characteristics. SET and ASK have several different options.

SET and ASK can be used on a channel of a device. SET and ASK are used to do special printing to the screen. ASK is used to find the screen's current print zone width and right margin setting. If they are not correct, SET is used to change them and then print your material to the screen.

For more information:

For information on SET #chnl and ASK #chnl statements, refer to Chapter 14, File Handling.

For information on SET STRUCTURE and ASK STRUCTURE statements, refer to Chapter 15, Data Structure Statements.

FORMAT:

```
        SET AUTOEXIT num_expr
```

EXAMPLE:

```
  set autoexit 1
  do
    input 'Who': a$
    if  _exit  or  _back  then  exit do
  loop
  print 'Finished'
  end


Who? Greg
Who? Sammy
Who?              (when user fails to respond within 1 minute.)
Finished
```

PURPOSE:

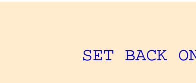
SET AUTOEXIT slowly backs a user out of a program if the computer is left idle.

DESCRIPTION:

SET AUTOEXIT causes an idle terminal waiting at an input prompt to set _EXIT to TRUE and complete the input. *num_expr* is the length of time in minutes. If *num_expr* is assigned a value of 0, SheerPower turns off the feature.

If the terminal is left idle for *num_expr* minutes at the input prompt, EXIT will be forced as the response, the _EXIT flag will be set to on and the program will execute the code indicated for _EXIT, if any.

FORMAT:



```
SET BACK ON
```

> **Note**
>
> The `[Esc]` key or the `[up arrow]` key will set _BACK to TRUE.

EXAMPLE:

```
  line input 'Name', length 30: reply$
  print _back
  set back on
  print _back
  end


Name? TESTER________________________
0
1
```

DESCRIPTION:

SET BACK ON sets the internal variable _BACK to TRUE (1).

FORMAT:

```
SET BACK OFF
```

EXAMPLE:

```
  line input 'Name', length 30: reply$
  print _back
  set back off
  print _back
  end


Name? [Esc]_____________________________  <--- press the Escape key or Up arrow key
1
0
```

DESCRIPTION:

SET BACK OFF sets the internal variable _BACK to FALSE (0).

FORMAT:

```
SET ERROR ON
```

EXAMPLE:

```
do
  input 'Enter the age': age
  if  age < 1  then
    print 'Too young:'; age
    set error on
  else
    set error off
  end if
loop while _error
end


Enter the age? .5
Too young: .5
Enter the age? 38
```

PURPOSE:

SET ERROR ON is used to set the **_ERROR** flag on.

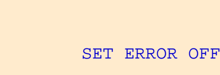
DESCRIPTION:

_ERROR is a general-purpose error flag. It is used to indicate that an error has occurred, and to test later whether an error has occurred.

The following statements SET the _ERROR flag:

- SET ERROR ON
- the MESSAGE ERROR: statement
- the execution of the END WHEN statement

FORMAT:



```
SET ERROR OFF
```

EXAMPLE:

```
do
  input 'Enter the age': age
  if  age < 1  then
    print 'Too young:'; age
    set error on
  else
    set error off
  end if
loop while _error
end


Enter the age? .5
Too young: .5
```

```
Enter the age? 38
```

PURPOSE:

SET ERROR OFF is used to clear the **_ERROR** flag.

DESCRIPTION:

_ERROR is a general purpose error flag. You can use it to indicate that an error has occurred, and to test later whether an error has occurred.

The following statements CLEAR the _ERROR flag:

- SET ERROR OFF
- the DISPATCH statement
- the WHEN EXCEPTION IN statement

FORMAT:

```
        ASK ERRORS num_var
```

EXAMPLE:

```
  do
    input 'Enter the age': age
    if  age < 1  then
      message error: age; ' Too Young'
      repeat do
    else
      exit do
    end if
  loop
  ask errors num_errors
  print 'Errors:'; num_errors
  end


Enter the age? 0                  0 Too Young
Enter the age? .5                 .5 Too Young
Enter the age? 21
Errors: 2
```

DESCRIPTION:

ASK ERRORS asks for the number of user errors. The MESSAGE ERROR: statement increments this internal counter.

FORMAT:

```
        SET EXIT ON
```

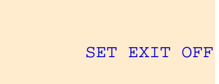
EXAMPLE:

```
  line input 'Name', length 30: reply$
  print _exit
  set exit on
  print _exit
  end

Name? ELAINE________________________
0
1
```

DESCRIPTION:

SET EXIT ON sets the internal variable _EXIT to TRUE (1).

FORMAT:



```
        SET EXIT OFF
```

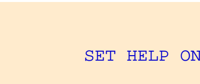
EXAMPLE:

```
  line input 'Name', length 30: reply$
  print _exit
  set exit off
  print _exit
  end

Name? EXIT__________________________
1
0
```

DESCRIPTION:

SET EXIT OFF sets the internal variable _EXIT to FALSE (0).

FORMAT:



```
        SET HELP ON
```

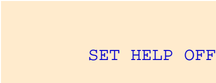
EXAMPLE:

```
line input 'Name', length 30: reply$
print _help
set help on
print _help
end

Name? MIKE___________________________
0
1
```

DESCRIPTION:

SET HELP ON sets the internal variable _HELP to TRUE (1).

FORMAT:



```
SET HELP OFF
```

EXAMPLE:

```
line input 'Name', length 30: reply$
print _help
set help off
print _help
end

Name? HELP___________________________
1
0
```

DESCRIPTION:

SET HELP OFF sets the internal variable _HELP to FALSE (0).

FORMAT:

```
SET ICON 'str_exp'
```

EXAMPLE:

```
// dynamically change the taskbar icon

  set icon "c:\sheerpower\samples\smiley.ico"
  print "Check out the taskbar icon... it's a smiley!"
  delay
  set icon "c:\sheerpower\samples\frowny.ico"
  print "and now... it's changed to a frown!"
  end
```

DESCRIPTION:

SET ICON changes the Icon displayed on the taskbar for each running SheerPower application. A unique icon can be specified for every SheerPower application. The icon can be dynamically changed during program execution.

FORMAT:

```
        ASK KEYSTROKES num_var
```

EXAMPLE:

```
  input 'Please enter your name': name$
  print 'Hello '; name$
  ask keystrokes strokes
  print 'Keystrokes:'; strokes
  end


Please enter your name? Maryanne
Hello Maryanne
Keystrokes: 8
```

DESCRIPTION:

ASK KEYSTROKES asks for the number of user-entered keystrokes.

FORMAT:

```
        ASK MARGIN num_var
```

DESCRIPTION:

ASK MARGIN finds the right margin of the device specified and assigns its value to the numeric variable *num_var*.

FORMAT:

```
        SET MARGIN num_expr
```

EXAMPLE:

```
  print repeat$('.' ,200)
  print
  ask margin old_marg
  input 'What do you want the margin set to': new_marg
  set margin new_marg
  print repeat$('.' ,200)
  set margin old_marg
  end


..................................................
..................................................
..................................................
..................................................

What do you want the margin set to? 20
....................
....................
....................
....................
```

DESCRIPTON:

SET MARGIN sets the right margin on the device specified to the number indicated. *num_expr* specifies the column to set the margin to. The margin must be greater than zonewidth.

FORMAT:

```
        ASK PAGESIZE num_var
```

EXAMPLE:

```
  ask pagesize no_lines
  print 'There are'; no_lines; 'lines or rows on this screen'
  end

There are 24 lines or rows on this screen
```

DESCRIPTION:

ASK PAGESIZE returns the number of rows or lines of screen output.

FORMAT:

```
        ASK RESPONSES num_var
```

EXAMPLE:

```
  input 'Please enter your name': name$
  input 'What day is this': what_day$
  print 'Hello '; name$
  print 'Have a good '; what_day$
  ask responses answers
  print
  print 'Responses:'; answers
  end


Please enter your name? Ginger
What day is this? Wednesday
Hello Ginger
Have a good Wednesday

Responses: 2
```

DESCRIPTION:

ASK RESPONSES asks for the number of completed input responses.

FORMAT:

```
        SET SCROLL num_expr1, num_expr2
```

EXAMPLE:

```
  print at 21, 1: 'This text will not scroll.'
  set scroll 5, 20
  print at 20, 1:;
  delay 1
  print 'This'
  delay 1
  print 'text'
  delay 1
  print 'will'
  delay 1
  print 'scroll.'
  delay 1
  set scroll  1,24
  end



This
text
will
scroll

This text will not scroll.
```

DESCRIPTION:

SET SCROLL statement sets up a scrolling region from line *num_expr1* to line *num_expr2*.

FORMAT:

```
RANDOMIZE
```

EXAMPLE:

```
  randomize
  x = rnd
  print x
  end

run
.244013674718

run
.524856061388
```

PURPOSE:

RANDOMIZE gives the RND function a new starting point. This ensures a different random number sequence each time a program executes.

DESCRIPTION:

SheerPower uses a pseudo-random number sequence to generate random numbers. It uses a SEED value to start the sequence. The SEED changes each time that the RND function is used. SheerPower used the system clock to set the initial SEED value.

RANDOMIZE tells SheerPower to pick a random SEED value. This ensures that a different series of random numbers is returned each time the program executes. (See Section 6.1.12, RND for information on the RND function. See Section 11.13, ASK | SET SEED for the ASK/SET SEED statement.)

On using RANDOMIZE within a program

RANDOMIZE should only be done ONCE per the run of the program. It just uses the system clock to start the random number seed. RANDOMIZE should not be used inside of a LOOP.

FORMAT:

```
ASK SEED num_var
SET SEED num_expr
```

EXAMPLE:

```
  randomize
  ask seed seed_num
  for i = 1 to 3
    print rnd(1000)
  next i
  print 'Reset the random sequence'
  set seed seed_num
  for i = 1 to 3
    print rnd(1000)
  next i
  end
```

```
608
88
506
Reset the random sequence
608
88
506
```

PURPOSE:

ASK SEED sets or resets the pseudo-random number sequence.

DESCRIPTION:

ASK SEED returns the current starting point of a pseudo-random sequence and stores the number in *num_var*.

SET SEED sets the starting point of a pseudo-random sequence with the number in *num_expr*.

There are a number of ASK SYSTEM and SET SYSTEM statements. These are described in the following sections. The ASK/SET statements ask about and set various system operation features.

FORMAT:

```
        ASK SYSTEM: COMMENT str_var
```

EXAMPLE:

```
  set system: comment 'Invoice Entry'
  ask system: comment c$
  print c$
  end

Invoice Entry
```

DESCRIPTION:

The **ASK SYSTEM: COMMENT** statement asks for the SheerPower operating system comment for the process.

FORMAT:

```
        SET SYSTEM: COMMENT str_expr
```

EXAMPLE:

```
set system: comment 'Invoice Entry'
ask system: comment c$
print c$
end

Invoice Entry
```

FORMAT:

```
ASK SYSTEM: DIRECTORY str_var
```

EXAMPLE:

```
ask system: directory z$
print 'Current directory is: '; z$
end

Current directory is: c:/sheerpower
```

DESCRIPTION:

ASK SYSTEM: DIRECTORY asks the operating system for the current default device and directory.

FORMAT:

```
SET SYSTEM: DIRECTORY str_var
```

Important note on the following example:

The following example assumes that you have a folder named "examples" inside of your SheerPower folder [c:\SheerPower\examples].

EXAMPLE:

```
ask system: directory z0$
print 'Current directory     '; z0$
set system: directory 'c:\sheerpower\examples'
ask system: directory z1$
print 'Directory set to      '; z1$
delay 2
set system: directory z0$
print 'Directory set back to '; z0$
end

Current directory      c:\sheerpower
Directory set to       c:\sheerpower\examples
Directory set back to  c:\sheerpower
```

DESCRIPTION:

SET SYSTEM: DIRECTORY sets the default device and directory.

FORMAT:

```
        ASK SYSTEM, LOGICAL str_expr: VALUE str_var
```

EXAMPLE:

```
  ask system, logical "SheerPower": value scr$
  print '"SheerPower" points to: '; scr$
  end


"SheerPower" points to: c:\SHEERPOWER\
```

DESCRIPTION:

ASK SYSTEM, LOGICAL asks the operating system to translate the logical name in *str_expr* and place the result into the variable specified by *str_var*.

FORMAT:

```
        SET SYSTEM, LOGICAL str_expr1: VALUE str_expr2
```

EXAMPLE:

```
  set system, logical 'SheerPower': value 'c:\sheerpower\examples\tester'
  ask system, logical 'SheerPower': value z$
  print 'Logical set to '; z$
  end


Logical set to c:\sheerpower\examples\tester
```

DESCRIPTION:

SET SYSTEM, LOGICAL: VALUE statement sets the operating system logical name in *str_expr1* to the value in *str_expr2*.

You can also set logical values by editing the **SP4GL.INI** file, creating a **[logicals]** section, then defining the logical. For example:

```
  [logicals]
  mylogical=c:\somewhere
```

FORMAT:

```
ASK SYSTEM: MODE str_var
```

EXAMPLE:

```
  ask system: mode process_mode$
  print 'Process Mode: '; process_mode$
  end


Process mode: BATCH
```

DESCRIPTION:

ASK SYSTEM: MODE statement returns the mode of the process which is one of the following:

INTERACTIVE
BATCH
NETWORK
OTHER

FORMAT:

```
ASK SYSTEM: PARAMETER str_var
```

> **Note**
>
> ASK SYSTEM: PARAMETER works only in Windows 2000 and Windows NT.

EXAMPLE:

```
  ask system: parameter param$
  print 'Parameter was: '; param$
  delay
  end


c:\sheerpower> test.spsrc hello  <---- type in 'test.spsrc hello' at the Command
(DOS)
                                       prompt to run this program.

Parameter was: hello             <---- this will appear in the console window
```

PURPOSE:

ASK SYSTEM: PARAMETER returns any parameter from the command line given after the program name and places it in *str_var*.

DESCRIPTION:

ASK SYSTEM: PARAMETER lets you obtain the command line that invoked SheerPower. The statement gives you the part of the command line after the program name.

FORMAT:

```
ASK SYSTEM: PROCESS str_var
```

EXAMPLE:

```
ask system: process process$
print 'Process is: '; process$
end


Process is: SheerPower 4GL
```

DESCRIPTION:

ASK SYSTEM: PROCESS *str_var* asks the operating system for the current process name.

FORMAT:

```
SET SYSTEM: PROCESS str_expr
```

EXAMPLE:

```
ask system: process process$
curr_process$ = process$
print 'Current process is: '; curr_process$
new_process$ = 'do_test'
set system: process new_process$
ask system: process process$
print 'New process is: '; process$
set system: process curr_process$
ask system: process process$
print 'Old process restored: '; process$
end


Current process is: SheerPower 4GL
New process is: DO_TEST
Old process restored: SheerPower 4GL
```

DESCRIPTION:

SET SYSTEM: PROCESS *str_expr* changes the operating system process name to *str_expr*.

FORMAT:

```
ASK SYSTEM: PROGRAM str_var
```

EXAMPLE:

```
// make a sample program file called
// mysample.spsrc in the SheerPower Samples folder
// and paste in the following code:

ask system: program x$
print 'This program is '; x$
end


This program is c:\sheerpower\samples\mysample.spsrc
```

DESCRIPTION:

The **ASK SYSTEM: PROGRAM** statement returns the full file specification of the running program. This is helpful for programs that check their own revision dates and for .SPRUN files that want to scan for compiled in licenses.

FORMAT:

```
ASK SYSTEM: RIGHTS str_var
```

Important note on the following example:

Win9x does not provide any specific rights for users. ASK SYSTEM: RIGHTS will work only with Windows 2000 and Windows NT.

EXAMPLE:

```
ask system: rights process_rights$
print 'Your process rights are: '; process_rights$
end


Your process rights are: FAST_ACCESS,TEST_ACCESS
```

DESCRIPTION:

ASK SYSTEM: RIGHTS asks the operating system to return a list of the rights explicitly granted to the calling process.

FORMAT:

```
ASK SYSTEM, SYMBOL str_expr: VALUE str_var
```

EXAMPLE:

```
set system, symbol 'SheerPower': value 'SheerPower 4GL'
ask system, symbol 'SheerPower': value symbol$
print 'Value of symbol SheerPower is: '; symbol$
end


Value of symbol SheerPower is: SheerPower 4GL
```

DESCRIPTION:

ASK SYSTEM, SYMBOL: VALUE statement asks the operating system to translate the symbol name in *str_expr* and place the result into the variable specified by *str_var*.

FORMAT:

```
ASK SYSTEM, SYMBOL 'DNS:xxx': VALUE str_var

(where 'xxxx' is a domain name or IP address to lookup)
```

EXAMPLE:

```
ask system, symbol 'dns:mail.ttinet.com': value x$
print x$


38.112.130.3
```

```
ask system, symbol 'dns:mymail.ttinet.com': value x$
print x$  // it returns a blank since mymail.ttinet.com does not exist
```

DESCRIPTION:

**ASK SYSTEM, SYMBOL 'DNS:xxx': VALUE a$** returns into a$ the IP address associated with xxx. If the DNS lookup fails, a$ will be a null string.

FORMAT:

```
SET SYSTEM, SYMBOL str_expr1: VALUE str_expr2
```

EXAMPLE:

```
set system, symbol 'mysym': value 'hello'
ask system, symbol 'mysym': value z$
print 'Symbol set to '; z$
end


Symbol set to hello
```

For an example on how to read and write to the Windows registry using SET SYSTEM, SYMBOL: VALUE and ASK SYSTEM, SYMBOL: VALUE, see Section 12.1, Read/Write to the Windows Registry
.

DESCRIPTION:

SET SYSTEM, SYMBOL: VALUE statement sets the operating system symbol name in *str_expr1* to the value in *str_expr2*.

FORMAT:

```
        ASK SYSTEM, SYMBOL 'OS:xxx': value str$
        //returns into str$ the Operating system symbol value of 'xxx'
```

EXAMPLE:

```
ask system, symbol 'os:path': value mypath$
print 'The path value is: '; mypath$
end


The path value is: c:\windows;c:\windows\system32;C:\SheerPower\
```

DESCRIPTION:

The **os:** prefix to a symbol says that we are referencing the OPERATING SYSTEM SYMBOL. In the case of WINDOWS, this is the COMMAND window symbol.

FORMAT:

```
        ASK SYSTEM: USER str_var
```

EXAMPLE:

```
ask system : user uname$
print 'User is: '; uname$
end


User is: Default
```

PURPOSE:

ASK SYSTEM: USER statement returns the operating system name or ID for the user.

There are various ASK WINDOW and SET WINDOW statements. These are described in the following sections. The ASK/SET WINDOW statements ask about and reset different screen features.

FORMAT:

```
        ASK WINDOW AREA row, col, row, col: DATA str_var
```

EXAMPLE:

```
  print at 10, 4: 'Mary had a';
  print at 11, 4: 'little lamb';
  ask window area 10, 4, 11, 15: data x$
  print
  print x$
  end


   Mary had a
   little lamb
Mary had a
          little lamb
```

DESCRIPTION:

ASK WINDOW AREA statement reads the text displayed on the screen within the area defined by the given upperleft/lowerright coordinates into a string variable, *str_var*. The coordinates are specified by upper-left row, upper-left column, lower-right row, lower-right column. The statement returns a <LF> delimited string. No screen attributes are stored.

FORMAT:

```
        SET WINDOW AREA row, col, row, col: DATA str_expr
```

EXAMPLE:

```
  x$ = 'Mary had a' + chr$(10) + 'little lamb'
  set window area 6, 5, 7, 15: data x$
  end


    Mary had a
    little lamb
```

DESCRIPTION:

SET WINDOW AREA statement sets the screen within the area defined by the given upperleft/lowerright coordinates to the specified string. This is the mirror image of ASK WINDOW AREA row, col, row, col: DATA str_var.

FORMAT:

```
ASK WINDOW: COLUMN num_var
```

EXAMPLE:

```
print at 5,10:;
ask window: column cur_col
print 'Cursor is at column'; cur_col
end

Cursor is at column 10
```

DESCRIPTION:

ASK WINDOW: COLUMN statement returns the current column of the cursor's position.

FORMAT:

```
SET WINDOW: COLUMN num_expr
```

EXAMPLE:

```
print at 5,10:;
set window: column 4
print 'Hi!'
end

   Hi!
```

DESCRIPTION:

SET WINDOW: COLUMN statement positions the cursor at the *num_expr* column within the current row.

FORMAT:

```
ASK WINDOW: CURRENT str_var
SET WINDOW: CURRENT str_var
```

EXAMPLE:

```
print at 1,20, blink: 'Sample screen'
do
  line input 'Name', at 5,1, length 30: name$
  if  _back or  _exit then exit do
  if  _help  then
    ask window: current old_w$
    clear area box: 1, 5, 10, 50
    print at 3, 10, reverse: 'This is some help'
    delay
    set window: current old_w$
    repeat do
  end if
end do
end


                   Sample screen


             Name? help___________________            <------- type in 'help'



          +---------------------------------------------+
          |                                             |
          |    This is some help                        |
          |                                             |
      Name|                                             |
          |                                             |
          |                                             |
          +---------------------------------------------+


                      Press the ENTER key to continue
```

DESCRIPTION:

ASK WINDOW: CURRENT and **SET WINDOW: CURRENT** saves the image of the current screen and later restore it easily. This is useful for help messages and menus, where you must temporarily change the screen and then restore it back to what it was.

FORMAT:

```
        ASK WINDOW: DATA str_var
```

EXAMPLE:

```
print at 10, 4: 'Mary had a';
print at 11, 4: 'little lamb';
ask window: data x$
print
print x$
end


  Mary had a
  little lamb
.
.
.
Mary had a
```

```
        little lamb
```

DESCRIPTION:

ASK WINDOW: DATA statement reads the text displayed on the whole screen into a string variable. The statement returns a <LF> delimited string. No screen attributes are stored.

FORMAT:

```
        SET WINDOW: DATA str_expr
```

EXAMPLE:

```
  clear
  print at 1,1: ;
  x$ = 'Mary had a' + chr$(10) + 'little lamb'
  set window: data x$
  print at 10,1: 'done'
  end


Mary had a
little lamb







done
```

DESCRIPTION:

SET WINDOW: DATA statement sets the whole screen to the specified string. This is the mirror image of ASK WINDOW: DATA str_var.

FORMAT:

```
        ASK WINDOW: KEYMAP str_var
        SET WINDOW: KEYMAP str_expr
```

EXAMPLE:

```
  print 'Save the current keymap, reset keymap to default value.'
  ask window: keymap old_keymap$
  set window: keymap ''

  print 'Changing dollar sign key to *'
  set window keystroke '$': value '*'
  line input 'Press the dollar sign key, then ENTER': e$

  print 'Restore saved keymap'
  set window: keymap old_keymap$
  line input 'Press the DOWN key': down$
  line input 'Press the dollar sign key, then ENTER' : e$
```

```
  end


Save the current keymap, reset keymap to default value.
Changing DOWN key to be the EXIT key
Press the DOWN key? EXIT
Changing dollar sign key to *
Press the dollar sign key, then ENTER? *
Restore saved keymap
Press the DOWN key?
Press the dollar sign key, then ENTER? $
```

PURPOSE:

ASK WINDOW: KEYMAP and **SET WINDOW: KEYMAP** allow a generalized routine to save the current keymap, change the meaning of keys, and then restore the original keymap when done.

DESCRIPTION:

ASK WINDOW: KEYMAP and **SET WINDOW: KEYMAP** are used to save the image of the keymap and later restore it. This is helpful for applications the meaning of the keys must be temporarily changed using the SET WINDOW KEYSTROKE statement. The keymap can be restored to its default setting with SET WINDOW: KEYMAP.

FORMAT:

```
        SET WINDOW KEYSTROKE str_expr1: VALUE str_expr2
```

EXAMPLE:

```
  print 'Saving the current keymap.'
  ask window: keymap old_keymap$
  set window: keymap ''

  print 'Changing dollar sign key to *'
  set window keystroke '$': value '*'
  line input 'Press the dollar sign key, then ENTER': e$

  print 'Restoring saved keymap.'
  set window: keymap old_keymap$
  line input 'Press the DOWN key': down$
  line input 'Press the dollar sign key, then ENTER' : e$
  end


Saving the current keymap.
Changing DOWN key to be the EXIT key
Press the DOWN key? EXIT
Changing dollar sign key to *
Press the dollar sign key, then ENTER? *
Restoring saved keymap.
Press the DOWN key?
Press the dollar sign key, then ENTER? $
```

PURPOSE:

SET WINDOW KEYSTROKE changes the meaning of a keystroke within a SheerPower program. This allows complete redefinition of the keyboard for a given application.

DESCRIPTION:

str_expr1 describes the name of a key to be changed. It can be a single keystroke name or a comma-separated list of names. Keystroke names can be a single letter, or the name of the letter (such as `[Tab]`, or

[Ctrl/Z]).

str_expr2 defines the new meaning of the keystroke. A keystroke meaning consists of one or two components: the keystroke value and the keystroke concept. For example, the [Ctrl/Z] key usually has a value of CHR$(26), and the concept of EXIT. The keystroke value and/or the keystroke concept can be changed. If changing both a value and a concept, separate the two with a comma. You can restore the original meaning of the key by using " ".

The following keystroke concepts are supported:

Table 11-1 Supported Keystroke Concepts

Concept name	Description
_EXIT	an EXIT key
_BACK	a BACK key
_HELP	a HELP key
_IGNORE	ignore this keystroke
_INVALID	beep when pressed
_TERMINATOR	keystroke is a line terminator

FORMAT:

```
        ASK WINDOW: ROW num_var
```

EXAMPLE:

```
  print at 5,10:;
  ask window: row cur_row
  print 'Cursor is at row'; cur_row
  end


Cursor is at row 5
```

DESCRIPTION:

ASK WINDOW: ROW statement returns the current row of the cursor's position.

FORMAT:

```
        SET WINDOW: ROW num_expr
```

EXAMPLE:

```
  set window: row 3
  print 'Hi!'
  end


Hi!
```

DESCRIPTION:

SET WINDOW: ROW statement positions the cursor at the *num_expr* row within the current column.

FORMAT:

```
        ASK WINDOW: TYPEAHEAD str_var
```

EXAMPLE:

```
  do
    do_process
    ask window: typeahead z$
    if pos(ucase$(z$), 'STA') > 0  then show_status
    if  pos(z$, chr$(26)) > 0  then exit do
  loop
  stop

  routine do_process
    delay 1                     //<--- simulated processing
    print '.';
  end routine

  routine show_status
    print
    print 'Showing status'
    set window: typeahead ''
  end routine
  end


.......                                      <---- type 'STA' while dots are printing
showing status
.......       [Ctrl/Z]                  <---- to stop, press [Ctrl] and [z]
```

DESCRIPTION:

ASK WINDOW: TYPEAHEAD gets data from the typeahead buffer. This statement can be used to determine, for example, whether the user has typed `[Ctrl/Z]` or other special keystrokes. Asking for typeahead data does not lose what is already in the typeahead buffer.

FORMAT:

```
        SET WINDOW: TYPEAHEAD str_expr
```

EXAMPLE:

```
  set window: typeahead 'FRED' + chr$(13)
  input 'Name': name$
  print name$
  end


Name? FRED
FRED
```

DESCRIPTION:

SET WINDOW: TYPEAHEAD puts data into the typeahead buffer as though the user had typed the data in from the terminal.

FORMAT:

```
        ASK ZONEWIDTH num_var
```

EXAMPLE:

```
  ask zonewidth x
  print 'The current print zone width is'; x
  end


The current print zone width is 20
```

DESCRIPTION:

ASK ZONEWIDTH finds the print zone width of the device specified and assigns the value to the numeric variable, *num_var*.

FORMAT:

```
        SET ZONEWIDTH num_expr
```

EXAMPLE:

```
  print 1,2,3
  set zonewidth 10
  print 1,2,3
  set zonewidth 20
  end


1                   2                   3
1         2         3
```

DESCRIPTION:

SET ZONEWIDTH sets the print zone width of the device specified to the number designated. *num_expr* indicates the width to set the device's print zones.

FORMAT:

```
SET SYSTEM, SYMBOL '@' + str_expr1: VALUE str_expr2
ASK SYSTEM, SYMBOL '@' + str_expr1: VALUE str_expr2

where str_expr1 is: @program_name, data_for_program
```

EXAMPLE:

```
// Note:  Excel's DDE interface is a bit odd in that
//        to reference cell B1, we use "R1C2"  (row 1, column 2)
// This example is also found in \sheerpower\samples folder

dde$ = '@excel,ddetest.xls,'

xlsfile$ = filespec$("sheerpower:samples\ddetest.xls")
pass url: xlsfile$  // open our spreadsheet

cell$ = 'r1c1'
data$ = 'Nifty'
set system, symbol dde$+cell$: value data$

cell$ = 'r2c1'
data$ = day$
set system, symbol dde$+cell$: value data$

for x = 1 to 50
  set system, symbol dde$ + 'r'+str$(x)+'c2': value str$(x)
next x

ask system, symbol '@': value st$
z0 = pos (st$, '0')
if z0 > 0 then
  message error: 'Got an error on command# '; z0
end if

end
```

Note

Accessing Excel through SheerPower's DDE interface has only been tested on English versions of Excel.

PURPOSE:

2 or more programs running simultaneously that support DDE can exchange information and commands. For example, SheerPower 4GL can exchange information and commands with Microsoft Word or Excel.

Each application controlled by SheerPower's DDE interface has its own unique way of being controlled. You must be familiar with the application's DDE commands that you want to control with SheerPower. These commands are usually outlined in the application's documentation on DDE.

DESCRIPTION:

The **set system, symbol** statement sends a command or data to the receiving application.

The **ask system, symbol** statement asks the application for data to be returned.

The use of the **PASS NOWAIT** statement is the best way to start an application to control.

The **PASS URL** statement opens the application if it is not already open. Use the **PASS NOWAIT** statement to start or open an application that doesn't have a document associated with it.

If you receive an error message that reads:

```
Bad syntax for execute command
```

check to make sure the syntax follows these rules:

- Each command must be enclosed in square brackets and consist of an opcode and an optional parameter list enclosed in parenthesis.
- The opcode cannot contain spaces, commas, parenthesis, brackets or quotes.

For example, the following syntax to make a DDE connection to Quattro spreadsheets is incorrect:

```
a$ = "{FileNew}"
set system, symbol "@QPW,System": value a$
```

The correct syntax would be:

```
a$ = "[FileNew]"
set system, symbol "@QPW,System": value a$
```

Below are some examples of accessing Microsoft Word via DDE using SheerPower:

```
// bookmarks.spsrc
// written by Mark S. Johnson, CMH

ask system: user user$
pass url: "c:\sheerpower\myname.doc"
delay 1
// PASS URL statement opens up the Word document

ask system, symbol "@WINWORD,System,Topics": value topics$
//Get the available Word topics for the current document

selection$ = piece$(topics$,2,chr$(9))
//The name of the current document is the second topic

print "The current Word document is ";selection$

set system, symbol "@WINWORD," & selection$ & ",user": value user$
//Store the user name into the bookmark named "user"

stop
```

```
//Open a word file whose name in stored in new_word_file$
word_command$ = '[FileOpen("' & new_word_file$ & '")]'
set system, symbol "@WINWORD,System": value word_command$

//Commands enclosed in brackets [] are Word Basic commands

//Close a Word document whose name is in selection$
set system, symbol "@WINWORD," & selection$: value "[FileClose 1]"
//Close and save the file is set by parameter "1". Using "2" as the
//parameter closes, but does not save the file. A parameter of "0" prompts
//the user.

//Exit Word
set system, symbol "@WINWORD," & selection$: value "[FileExit 2]"
//Here the parameter "2" says quit and don't save. A parameter of "1"
//says quit and save. If the parameter is left out or is "0", the user is
//prompted.
```

FORMAT:

```
  ASK SYSTEM, SYMBOL "\" + REGISTRY_KEY
  SET SYSTEM, SYMBOL "\" + REGISTRY_KEY
```

EXAMPLE:

```
// Fetch the Product ID from the registry
  rkey$ = '\hkey_local_machine\Software\Microsoft\Windows' +
                  '\CurrentVersion\ProductId'
  ask system, symbol rkey$: value kvalue$
  print 'Product ID: '; kvalue$
  end


Product ID: 55555-OEM-1113333-44444
```

Exception handling routines intercept runtime exceptions and execute a block of code which handles them. If there is no exception handler, SheerPower returns an exception message specifying what the exception was and where it occurred. SheerPower stops program execution or tries the offending statement again.

There are two types of exception handlers: attached handlers and detached handlers.

For an **attached handler**, a WHEN EXCEPTION IN statement is used. WHEN EXCEPTION IN puts the handler (the block of code to execute) right after the block of code it protects.

For a **detached handler**, the statement WHEN EXCEPTION USE is used. When an exception occurs in the protected block, WHEN EXCEPTION USE calls a handler routine located in some other part of the program. The same handler routine can be used by any number of WHEN EXCEPTION USE statements and can be placed anywhere in a program.

This section explains exception handlers. It also describes the handling statements--RETRY, CONTINUE and EXIT HANDLER.

The following functions are also related to exception handling:

- EXTYPE
- EXLABEL$
- EXTEXT$
- SYSTEXT$

For a full description of these functions, see Section 6.8, Debugging and Exception Handling Functions.

FORMAT:

```
        CAUSE EXCEPTION exception_number
```

EXAMPLE:

```
  do
    input 'Select a number between 1 and 10': no
    if  no < 1 or no > 10  then cause exception 1001
    repeat do
  end do
  end


Select a number between 1 and 10? 8
Select a number between 1 and 10? 99
Illegal number at 10.2
```

PURPOSE:

CAUSE EXCEPTION is used to generate an exception under specific conditions.

DESCRIPTION:

CAUSE EXCEPTION causes the specified exception to occur. *exception_number* can be any integer expression. When SheerPower executes the CAUSE EXCEPTION statement, it generates the exception specified. See Section C.2, Exceptions for a list of exception messages.

FORMAT:

```
        WHEN EXCEPTION IN
          ---
          ---  protected block
          ---
        USE
          ---
          ---  handler
          ---
        END WHEN
```

EXAMPLE:

```
  input 'Your name, please': name$
  when exception in
    input 'How old are you': age
  use
    print 'Not a valid age'
    retry
  end when
  print
  print name$; ' is'; age
  end


Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

PURPOSE:

WHEN EXCEPTION IN is used to catch runtime exceptions and resolve them within a program when the code handling the exception is to be next to the code being protected.

DESCRIPTION:

WHEN EXCEPTION IN begins the protected block of code. Everything between the WHEN EXCEPTION IN and **USE** statements constitute the section of code protected by the handler---the *protected block*.

USE begins the handler routine. Everything between the USE and the **END WHEN** statements constitutes the handler. If an exception occurs in the protected block, the handler code is executed. END WHEN ends the routine.

FORMAT:

```
        WHEN EXCEPTION USE handl_name
          ---
          ---  protected block
          ---
        END WHEN
                 .
                 .
                 .
        HANDLER handl_name
          ---
          ---  handler
          ---
        END HANDLER
```

EXAMPLE:

```
  input 'Enter total sales amount': tsales
  input 'Enter number of sales': nsales
  when exception use fix_average
    average = tsales/nsales
  end when
  print 'The average is:'; average

  handler fix_average
    average = 0
    continue
  end handler
  end


Enter total sales amount? 25.00
Enter number of sales? 0
The average is: 0
```

PURPOSE:

WHEN EXCEPTION USE catches runtime exceptions and resolves them within a program when the same handler is needed for several protected blocks, or when all the handlers are to be in one place in a program.

DESCRIPTION:

WHEN EXCEPTION USE begins the protected block of code and specifies the **HANDLER** routine to use. *handl_name* is the name of the handler routine. The handler name must meet the specifications for variable names. The protected block is everything between the WHEN EXCEPTION USE and the **END WHEN** statements. If an exception occurs in this block of code, SheerPower calls the handler specified. If the handler does not exist, an error is generated.

HANDLER begins the handler routine. Everything between the HANDLER and the **END HANDLER** constitutes the handler routine. END HANDLER returns control to the statement following the END WHEN. When the handler is called, this block of code is executed. In the example above, SheerPower would normally return an exception when it tried to divide 25.00 by 0. The exception handler FIX_AVERAGE intercepts this exception and sets AVERAGE equal to 0.

The handler routine can occur before or after the protected block. For example:

```
handler fix_average
  average = 0               //<--- handler routine
  continue
end handler

input 'Enter total sales amount': tsales
input 'Enter number of sales': nsales
when exception use fix_average
  average = tsales/nsales
end when
print 'The average is'; average
end
```

One of the advantages of WHEN EXCEPTION USE is that the same handler routine can be called by any number of WHEN EXCEPTION USE statements. For example:

```
when exception use numbers
  input 'How old are you': age      //<--- first protected block
end when

input 'Your name, please': name$

when exception use numbers
  input 'Your birthdate': birth     //<--- second protected block
end when

print name$; ' was born on'; birth

handler numbers
  print 'Enter numbers only, please.'    //<--- handler routine
  retry
end handler
end


How old are you? 31       <---- type in your age
Your name, please? Sunny  <---- type in your name
Your birthdate? 10181969  <---- type in your birthdate (no spaces)
```

FORMAT:

```
        USE
          ---
          ---  RETRY
          ---
        END WHEN

           or

        HANDLER handl_name
          ---
          ---  RETRY
          ---
        END HANDLER
```

EXAMPLE:

```
  input 'Your name, please': name$
  when exception in
    input 'How old are you': age
  use
    print 'Not a valid age'
    retry
  end when
  print
  print name$; ' is'; age
  end


Your name, please? Tester
How old are you? 3x
Not a valid age
How old are you? 35

Tester is 35
```

PURPOSE:

RETRY is used after an exception to re-execute the statement that generated the exception.

DESCRIPTION:

RETRY can be used *only* in a HANDLER routine. RETRY causes SheerPower to leave the handler and re-execute the statement that generated an exception.

FORMAT:

```
        USE
          ---
          ---  CONTINUE
          ---
        END WHEN

           or

        HANDLER handl_name
          ---
          ---  CONTINUE
          ---
        END HANDLER
```

EXAMPLE:

```
  input 'Enter total sales amount': tsales
  input 'Enter number of sales': nsales
  when exception use fix_average
    average = tsales / nsales
  end when
  print 'The average is:'; average

  handler fix_average
    average = 0
    continue
  end handler
  end


Enter total sales amount? 18.00
```

```
Enter number of sales? 0
The average is: 0
```

PURPOSE

CONTINUE is used to continue normal program execution at the statement following the one that generated the exception.

DESCRIPTION:

CONTINUE causes SheerPower to exit the exception handler and continue program execution at the first statement following the statement which generated the exception. CONTINUE can *only* be used in a HANDLER routine.

FORMAT:

```
        USE
          ---
          ---  RESUME target
          ---
        END WHEN

           or

        HANDLER handl_name
          ---
          ---  RESUME target
          ---
        END HANDLER
```

EXAMPLE:

```
  input 'Enter total sales amount': tsales
  input 'Enter number of sales': nsales
  when exception use fix_average
    average = tsales / nsales
  end when
  print 'The average is:'; average

  handler fix_average
    average = 0
    print 'Invalid numbers.  Try again.'
    resume 1
  end handler
  end


Enter total sales amount? 75.00
Enter number of sales? 0
Invalid numbers.  Try again.
Enter total sales amount? 75.00
Enter number of sales? 3
The average is: 25
```

PURPOSE:

RESUME is used to resume normal program execution at the routine name or line number that you specify.

DESCRIPTION:

RESUME causes SheerPower to exit the exception handler and resume program execution at the routine name specified.

RESUME can *only* be used in a handler routine.

FORMAT:

```
        USE
          ---
          ---  EXIT HANDLER
          ---
        END WHEN

           or

        HANDLER handl_name
          ---
          ---  EXIT HANDLER
          ---
        END HANDLER
```

EXAMPLE:

```
  when exception use mistake
    input 'Enter your age': age
  end when
  print 'You are'; age; 'years old'

  handler mistake
    print 'Oops...'
    delay 2
    exit handler
  end handler
  end


Enter your age? 3x
Oops...
Non-numeric input when number expected at 10.1
Enter your age? 35
You are 35 years old
```

PURPOSE:

EXIT HANDLER is used to exit a handler routine and pass the exception up to the next level of exception handling.

DESCRIPTION:

EXIT HANDLER exits the current HANDLER routine. If the current handler is nested within another handler, SheerPower jumps to the outside handler and executes the code for that handler. If there is no other exception handler, control returns to the SheerPower system. SheerPower prints the exception message and takes whatever action it normally would.

EXIT HANDLER can *only* be used within an exception handler.

Files are places where information is stored. Files can be accessed from within SheerPower programs, but files exist outside of the program. Therefore, when a program ends, the file and the information in it still exists. The next time the program is run, the file can be accessed and old information removed from it, and new information stored in it.

Files are stored on devices: disks, tapes, etc. They are stored under file specifications, which include a device name.

The following pages describe the SheerPower statements used to manipulate files.

Note

See Section 6.7, File and Structure Access Functions for more on working with files in SheerPower. For example Section 6.7.5, FINDFILE$(str_expr [, int_expr]) to process a series of files, or batch files.

FORMAT:

```
OPEN #chnl_num: NAME 'file_spec'
       [, ACCESS INPUT| OUTPUT | OUTIN ] [, UNFORMATTED]
       [, UNIQUE] [, OPTIMIZE OFF]
```

EXAMPLE:

```
  open  #1: name 'test_file.txt', access output
  print #1: 'This is the first line of text.'
  print #1: 'This is the second line of text.'
  close #1

  open  #1: name 'test_file.txt'
  line input #1: line_1$, line_2$
  print line_1$
  print line_2$
  close #1


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

PURPOSE:

OPEN opens a file so it can be read and written to. OPEN can also create a file.

DESCRIPTION:

OPEN either opens an existing file or creates a new one. *#chnl_num* is the channel number associated with the file. *chnl_num* can be any integer number in the range of 1 to 95. (0 is the channel number associated with the terminal. Channel number 0 cannot be opened or closed.) The channel number is used to refer to the file. The channel number must be unique. If a channel number is already associated with an open file, an exception is generated.

file_spec gives the file specification of the file being opened. The file specification can be any string expression.

FORMAT:

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

EXAMPLE:

```
open file text_ch: name 'test_file.txt', access output
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.
```

PURPOSE:

The **OPEN FILE** statement is used to open existing files or create new files. You can also access webpage data using the **html://** file open option, or send emails with the **mailto://** file open option. See Chapter 19, Writing Network Applications and Accessing Devices for complete details.

DESCRIPTION:

OPEN either opens an existing file or creates a new one. The OPEN FILE syntax looks up a free channel, stores the channel into varname, and then opens the file.

file_spec gives the file specification of the file being opened. The file specification can be any string expression.

In SheerPower, the @ sign is a **LOGICAL** that is translated from being just an "@" to the full PATH of the folder that the current application is being run from. For example:

```
open #1: name '@filename', access outin
```

sheerpower: is also a logical. If the file to be opened is inside the SheerPower folder, it can be opened as:

```
open #1: name 'sheerpower:filename', access outin
```

You can also open a file using the full path of where the file is located. For example:

```
open #1: name 'c:\foldername\filename', access outin
```

In SheerPower you can make your own logicals. See Section 11.14.6, SET SYSTEM, LOGICAL: VALUE.

The **OPEN** statement has several options. Multiple options are separated with commas.

The **ACCESS** option specifies one of three input/output options. These options tell SheerPower whether you want to input (read) data, output (store) data or input and output data.

ACCESS INPUT

- The file is opened for input only; data can only be read from the file.
- If the file does not exist, an exception is generated.
- Access INPUT is the default if no ACCESS is specified.

ACCESS OUTPUT

- The file is opened for output only. Data can be stored in the file, but it cannot be read from the file.
- When a text file is opened, the default line length is 40960; for line lengths other than 40960 the SET #chnl_num: MARGIN num_expr statement must be used.
- If the file exists, SheerPower creates a new version of the file and uses that version. Old versions of the file are left untouched.
- If the file does not exist, SheerPower creates a file using the specifications provided.

ACCESS OUTIN

- The file is opened for input and output. Data can be read, deleted and added to the file.
- If the file exists, SheerPower opens the latest version.
- If the file does not exist, SheerPower creates a file using the specifications provided.

When writing to a file opened as **UNFORMATTED**, the writes are done without any carriage control. This allows various character sequences to be sent to the channel without having CR/LF (carriage return/line feed) sequences sent as well.

Important note for the following example:

This example will create a NEW file in your SheerPower folder called 'unformatted.txt'. To view the result of this example, you need to open and view 'unformatted.txt' in SPDEV *after* you run the program. See Section 1.2, Creating a New Program in SheerPower for instructions on opening files in SPDEV.

```
open #1: name 'sheerpower:unformatted.txt', access output, unformatted
for i = 1 to 10
  print #1: i;
next i
print #1:
end


1  2  3  4  5  6  7  8  9  10
```

Read and Write Binary Files Using the UNFORMATTED Option

SheerPower reads and writes binary files using the UNFORMATTED option of the OPEN statement. When reading unformatted data, up to 40960 bytes are read at a time. The following example gets a chunk of binary data from a file, and then takes the first 121 bytes of that file and stores it into a variable.

```
open file myfile: name 'somefile.xxx', unformatted
line input #myfile: rec$  // read a "chunk" of binary data
firstbytes$ = left(rec$, 121)  // Just the first 121 bytes
//Where 'somefile.xxx' is the name of your binary file.
```

When **OPTIMIZE OFF** is specified, the file is opened without any of the SheerPower I/O optimizations.

```
open #2: name 'report.txt', optimize off, access output
end
```

When the **UNIQUE** option is used, the file is created with a unique name. These are usually temporary work or holding files for listings, etc. The temporary files by default are located in the Windows temporary folder for the current user. If you specify your own path for the file name, then that path will be used instead when creating the temporary file.

```
open file temp_ch: name 'c:\sheerpower\somfolder\myfile.tmp', unique, access output
ask #temp_ch: name filename$
print filename$
```

```
  open  #12: unique, access output
  ask   #12: name x$
  print x$
  close #12
  end


c:\windows\temp\_fd56639b.tmp
```

The following example illustrates how to create uniquely named files based on *file_spec*.

```
  open  #12: name 'payroll', unique, access output
  ask   #12: name x$
  print x$
  close #12
  end


c:\windows\temp\payroll_fd5aefaf.tmp
```

When writing to a file opened with the **LOCK** option, the file cannot be accessed by any other process. When a file is opened with LOCK, the following error message is returned:

```
  Privilege or security level insufficient for operation
```

```
  open #1: name 'sheerpower:unformatted.txt', access output, unformatted, lock
  open #2: name 'sheerpower:unformatted.txt', access output
  end


  18:25:50  Privilege or security level insufficient for operation at MAIN.0001
```

FORMAT:

```
        CLOSE [#chnl_num | ALL]
```

Important note for the following example:

Running the following example will create a file called 'test_file.txt' in your SheerPower folder.

EXAMPLE:

```
  open  #1: name 'test_file.txt', access output
  print #1: 'This is the first line.'
  print #1: 'Here is the second line.'
  close #1
  end
```

PURPOSE:

CLOSE #chnl_num closes a file. **CLOSE ALL** closes all files. Files should be closed before the program ends.

DESCRIPTION:

CLOSE closes a file from further access. Once a file is closed, data cannot be stored in it or read from it. *chnl_num* is the channel number associated with the file. (The channel number is assigned in the OPEN statement.) The channel number can be given as any numeric expression.

The **PRINT #chnl_num** statement writes data to a file so the data can be referenced at a later time.

FORMAT:

```
        PRINT #chnl_num [, USING print_mask]: [TAB(col){, | ;}] [expr {, | ;}
                                              [TAB(col){, | ;}] expr...]
```

EXAMPLE:

```
  open  #1: name 'test.txt', access output
  print #1, using '{ucase}?': 'first line'
  print #1: tab(5); 'second line'
  close #1
  open  #1: name 'test.txt'
  line input #1: record_1$, record_2$
  print record_1$
  print record_2$
  close #1
  end


FIRST LINE
    second line
```

DESCRIPTION:

PRINT writes data to a file. The file must be open and have a channel number associated with it. *chnl_num* is the channel number and it can be any numeric expression. *expr* is the expression being stored in the file. When SheerPower executes a PRINT statement, it evaluates the expression and stores its value in the file. The expression is optional. A PRINT statement without an expression writes a blank line to the file.

Note

See Section 6.7, File and Structure Access Functions for more on working with files in SheerPower.

FORMAT:

```
        INPUT #chnl_num: var, var...
```

EXAMPLE:

```
open  #1: name 'number.ars', access output
print #1: 1; 2; 3
close #1
open  #1: name 'number.ars', access input
input #1: a, b, c
print a; b; c
close #1
end


1  2  3
```

DESCRIPTION:

The **INPUT #chnl_num** statement is used to read the data stored in a file. The file must be open and have a channel number associated with it. The simplest version of the INPUT statement is:

```
INPUT #chnl_num: var
```

chnl_num is the channel number associated with the file. The channel number can be any integer expression. *var* is the variable the data is assigned to. Each time data is input from a file, it must be assigned to a variable. The data input must match the variable's data type. SheerPower inputs data sequentially starting at the beginning of the file.

One **INPUT** statement can be used to input data into a number of variables. The variables in the INPUT list must be separated by commas.

```
INPUT #chnl_num: var, var, var...
```

SheerPower inputs data sequentially, starting from the beginning of the file. SheerPower continues inputting data until all the variables in the list have values.

```
dim num(10)
open  #1: name 'number.ars', access output
print #1: 1
print #1: 2
print #1: 3
close #1
open  #1: name 'number.ars', access input
for i = 1 to 3
  input #1: num(i)
  print num(i);
next i
close #1
end


1  2  3
```

If the variable and data types do not match, an exception will be generated.

If an attempt is made to input more data than the file contains, an exception is generated.

FORMAT:

```
LINE INPUT #chnl_num [, EOF num_var]: str_var, str_var...
```

EXAMPLE:

```
open #1: name 'test_file.txt', access input
line input #1: line_1$, line_2$
print line_1$, line_2$
close #1
end

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

DESCRIPTION:

LINE INPUT #chnl_num reads a line of text from a file. Everything on the line including commas, quotation marks, semicolons, etc., is accepted as part of the input line.

The file must be open and have a channel number associated with it. The simplest version of the LINE INPUT statement is:

```
LINE INPUT #chnl_num: str_var
```

chnl_num is the channel number associated with the file. The channel number can be any integer expression. *str_var* is the variable to which data is being assigned. When SheerPower executes the LINE INPUT statement, it reads a line from the file and assigns it to the string variable specified.

LINE INPUT accepts as the input data everything from the beginning of the line up to the first line terminator. The contents of the line---including commas, quotation marks, tabs, leading and trailing spaces, etc.---are assigned to the string variable specified. A string variable must be specified. If a numeric variable is specified, an error will result. If the line being read is empty, SheerPower assigns a null string to the string variable.

The **EOF** option of LINE INPUT causes SheerPower to return a TRUE/FALSE value indicating when the end-of-file has been reached. This eliminates the need for an error handler when reading files. The format is:

```
LINE INPUT #chnl_num, EOF num_var: str_var
```

```
ven_ch = _channel
open #ven_ch: name 'test_file.txt'
do
  line input #ven_ch, eof endfile?: datafile$
  if endfile? then exit do
  print 'line was: '; datafile$
loop
close #1
end

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

To append data to a file, put the PRINT statement after the LOOP. For example:

```
open file myfile_ch: name 'myfile.txt', access outin
do
  line input #myfile_ch, eof eof?: z0$
  if eof? then exit do
loop
print #myfile_ch: 'This is a new line added to the file'
close #myfile_ch
```

_CHANNEL is the next available channel number.

LINE INPUT can be used to read several lines of data from a file. To read more than one item, separate the string variables with commas. Lines are read sequentially, starting from the beginning of the file, and assigned to the variables listed.

```
open #1: name 'test_file.txt', access input
line input #1: line_1$, line_2$
print '1  '; line_1$
print '2  '; line_2$
close #1
end


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

If an attempt is made to input more data than the file contains, SheerPower generates an exception.

FORMAT:

```
ASK #chnl_num: [NAME str_var][, ZONEWIDTH num_var] [, MARGIN num_var]
               [, CURRENT str_var]
```

EXAMPLE:

```
open #3: name 'storage.ars', access output
ask  #3: zonewidth x
print 'The current print zone width is'; x
close #3
end


The current print zone width is 20
```

PURPOSE:

ASK #chnl_num is used to find what various characteristics a device is set to.

DESCRIPTION:

ASK returns the characteristic of the device specified and stores the value in a *num_var* or *str_var*. *chnl_num* is an optional channel number. If no channel number is specified, SheerPower checks the default device. If a channel number is specified, SheerPower checks the device associated with that channel number.

An option must be included in the **ASK #chnl_num** statement. The ask options currently available are described below.

ZONEWIDTH num_var

ASK #chnl_num: ZONEWIDTH finds the print zone width of the device specified and assigns the value to the numeric variable *num_var*.

```
  open #3: name 'storage.ars', access output
  ask  #3:  zonewidth x
  print 'The current print zone width is'; x
  close #3
  end


The current print zone width is 20
```

MARGIN num_var

ASK MARGIN finds the right margin of the device specified and assigns its value to the numeric variable *num_var*.

```
  open #3: name 'test_file.txt', access output
  ask  #3: margin marg
  print 'The current margin is'; marg
  close #3
  end


The current margin is 132
```

CURRENT str_var

ASK #chnl_num: CURRENT is used to store a current record value into the str_var.

```
  open #1: name 'temp.txt', access output
  for z = 1 to 20
    print #1: 'This is line number '; z
  next z
  close #1
  open #1: name 'temp.txt'
  for i = 1 to 5
    line input #1: a$
  next i
  ask #1: current c$
  print '5th item was: '; a$
  for i = 1 to 5
    line input #1: a$
  next i
  print '10th item was: '; a$
  set #1: current c$
  line input #1: a$
  print 'Back to 5th item again: '; a$
  close #1
  end


5th item was: This is line number  5
10th item was: This is line number  10
Back to 5th item again: This is line number  5
```

NAME str_var

ASK #chnl_num: NAME asks the SheerPower operating system for the file specification of the file open on channel *#chnl_num* and stores the value into *str_var*.

```
  out_ch = 12
  open #out_ch: name 'sheerpower:minutes.txt', &
      access output
  ask #out_ch: name x$
  print x$
  close #out_ch
  end


c:SHEERPOWER\minutes.txt
```

FORMAT:

```
        SET # chnl_num: [ ZONEWIDTH num_expr ] [, MARGIN int_expr]
                        [, CURRENT str_expr]
```

EXAMPLE:

```
  open #3: name 'storage.ars', access output
  ask  #3: zonewidth x
  print 'The current print zone width is'; x
  set #3: zonewidth 35
  ask #3: zonewidth x
  print 'The new print zone width is'; x
  close #3
  end


The current print zone width is 20
The new print zone width is 35
```

DESCRIPTION:

SET #chnl_num sets various device characteristics. *chnl_number* is a channel number. If no channel number is specified, SheerPower sets the default device (the terminal). If a channel number is specified, SheerPower sets the device associated with that channel number.

When a device characteristic is SET, it remains set until the device is closed. Therefore, if the terminal is SET, it will remain SET until you exit from the SheerPower environment or the SET statement is used again.

An option must be included with the **SET #chnl_num** statement. The set options currently available are described below:

ZONEWIDTH num_expr

SET #chnl_num: ZONEWIDTH sets the print zone width of the device specified to the number designated. *num_expr* indicates the width to set the device's print zones. See above example.

MARGIN int_expr

SET #chnl_num: MARGIN sets the right margin on the device specified to the number indicated. *int_expr* specifies the column to set the margin to. The margin must be greater than the zone width.

```
open  #3: name 'storage.ars', access output
set   #3: margin 45
print #3: repeat$('1234567',10)
close #3
open  #3: name 'storage.ars', access input
do
  line input #3, eof endfile?: item$
  if  endfile?  then  exit do
  print item$
loop
close #3
end


1234567123456712345671234567123456712345671234567123
4567123456712345671234567
```

CURRENT str_expr

SET #chnl_num: CURRENT sets the current record to that specified by *str_expr*. The *str_expr* contains the information for the record you want to make current.

```
open #1: name 'temp.txt', access output
for z = 1 to 20
  print #1: 'This is line number '; z
next z
close #1
open #1: name 'temp.txt'
for i = 1 to 5
  line input #1: a$
next i
ask #1: current c$
print '5th item was: '; a$
for i = 1 to 5
  line input #1: a$
next i
print '10th item was: '; a$
set #1: current c$
line input #1: a$
print 'Back to 5th item again: '; a$
close #1
end


5th item was: This is line number  5
10th item was: This is line number  10
Back to 5th item again: This is line number  5
```

FORMAT:

```
        KILL str_expr
```

EXAMPLE:

```
! DANGER -- Executing this program will DELETE a file
  input 'What file do you want to delete': file$
  kill file$
  end


 What file do you want to delete?
```

PURPOSE:

KILL is used to delete a file from within your program.

DESCRIPTION:

KILL searches for and deletes the file specified in *str_expr*, the string expression. The full file name must be included with extension. If no version number is included, KILL will delete the most recent version only.

FORMAT:

```
  open #u: name "textwindow://title?width=nn&height=mm&max=qqqq"
```

EXAMPLE:

```
  open file text_ch: name 'textwindow://my text window?noclose&max=30', access output
  for i=1 to 30
    print #text_ch: i, sqr(i)
  next i
  delay
  stop
```

PURPOSE:

Occasionally there is a need to output text to a window that is not the main SheerPower 4GL Debug Window. To do this, you need to open a separate **TEXTWINDOW**.

When the channel to the textwindow is closed, the actual textwindow is also closed.

DESCRIPTION:

Below is a table of textwindow attributes.

Table 14-1 TEXTWINDOW Attributes

Attribute	Description
title	the default title is "text window n" where "n" is the number of untitled windows created so far.
width	the width of the window in characters. The default is the width of the main console window.
height	the height of the window in characters. The default is the height of the main console window.
max	the maximum lines the user can scroll back. (the default is 10000)
noclose	no [x] close button in the text window

The **PASS** statement can be used to print output from SheerPower.

```
//  Create your output text file:
outfile$ = 'myfile.txt'
open file out_ch: name outfile$, access output
for i=1 to 100
  print #out_ch: i, sqr(i)
next i
close #out_ch

// Now print it out to the default printer
pass print: outfile$
end
```

Printing HTML Output

You can output HTML code and then envoke the browser to display and print it using the PASS statement as illustrated in the following example:

```
//  Create your output html file
outfile$ = 'myfile.html'
open file out_ch: name outfile$, access output
print #out_ch: '<html><body>'
print #out_ch: '<table border=3 bgcolor=lightblue>'
for i=1 to 100
  print #out_ch: '<tr>'
  print #out_ch: '<td>'; i; '<td>'; sqr(i)
  print #out_ch: '</tr>'
next i
print #out_ch: '</table>'
print #out_ch: '</body></html>'
close #out_ch

// Now have the browser handle it
pass url: outfile$
end
```

FORMAT:

```
media str_var
```

EXAMPLE:

```
// Play some sounds
for i = 1 to 3
  media '@yes.wav'
next i
media '@sorry.wav'
end
```

PURPOSE:

The **MEDIA** statement allows you to open media files with SheerPower.

DESCRIPTION:

Currently the following media files are supported by the MEDIA statement in SheerPower:

- .WAV

The MEDIA statement also has as features/options:

```
media loop: string_expr$  // play this over and over again
```

such as:

```
media loop: '@yes.wav'
```

This will continue to play as the program continues execution. This can be used to provide "background music" to an application.

To stop playing looped media:

```
media ''  // stop any background media that is playing
```

Note

See Section 6.7, File and Structure Access Functions for more on working with files in SheerPower.

This chapter explains the SheerPower data structure statements. One of the major features of SheerPower is its ability to perform database operations as part of the language. The data structure statements allow you to manipulate data structures in your own programs.

List of Data Structure Statements:

OPEN STRUCTURE
CLOSE STRUCTURE
EXTRACT STRUCTURE
REEXTRACT STRUCTURE
CANCEL EXTRACT
ADD STRUCTURE
DELETE STRUCTURE
INCLUDE
EXCLUDE
SORT
FOR EACH
ASK STRUCTURE
SET STRUCTURE
LOCK STRUCTURE
UNLOCK STRUCTURE
UNLOCK STRUCTURE: COMMIT
UNLOCK ALL: COMMIT

In all data structure statements, the word TABLE can be used interchangeably with STRUCTURE. For example:

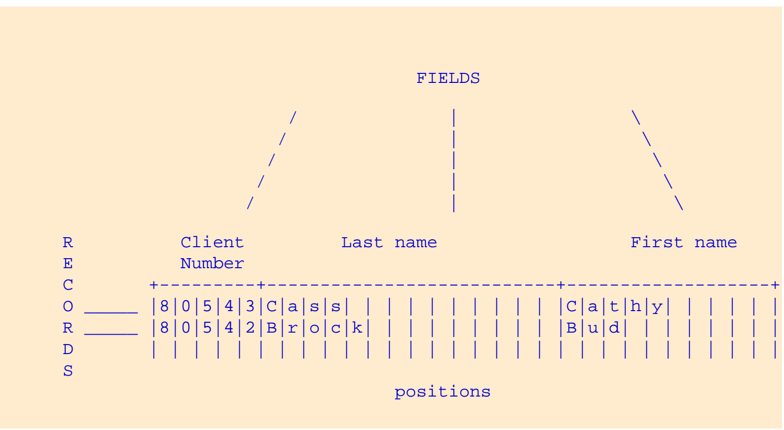
```
  OPEN STRUCTURE

is the same as

  OPEN TABLE
```

SheerPower's data management system stores data file information in **structures**. (Chapter 16, Database Setup tells how structures are created.) A structure file contains the following information:

- name of the database engine (ARS, etc.)
- dataset name (data file name)
- data dictionary name (definition file name)
- file security and other miscellaneous information

In SheerPower, you address the structure file and not the dataset directly.

A SheerPower structure is made up of **records** *and* **fields**. A structure looks something like this:



In the above example, the records are the horizontal lines of information. In the CLIENT structure above, there is a record for each customer.

Each record consists of fields. For example, the customer records alone might contain a field for the customer's ID number, last name, first name, address, phone number, company name, etc. Each of these pieces of data is stored in its own field---the name field, address field, phone number field, etc. The fields appear as columns in the diagram above.

To reference a field, indicate the structure and the desired field. The field name is enclosed in parentheses.

```
struc_name(field_name)
```

struc_name is the name associated with the structure. *field_name* is the name of a field in the structure. SheerPower searches for the field specified in the current record and reads its contents.

```
open structure cl: name 'sheerpower:samples\client'
extract structure cl: key id = '80522'
  print cl(last), cl(first) // print last and first
                            // fields from the CL structure
end extract
close structure cl
end


Errant            Earl
```

The remainder of this chapter describes the data structure statements and how to use them.

FORMAT:

```
OPEN STRUCTURE struc_name: NAME 'struc_filename'
  [, ACCESS INPUT | OUTIN] [, LOCK] [, DATAFILE filename]
  [, OPTIMIZE OFF]
```

EXAMPLE:

```
open structure cl: name 'sheerpower:samples\client', &
  access input, lock
extract structure 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 structure cl
end


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

DESCRIPTION:

The **OPEN STRUCTURE** statement is used to open a data structure. The structure must be open in order to reference data in the structure (i.e. change field data, add or delete records). *struc_name* is a name (i.e. nickname) you assign to the structure. You use this name to refer to the structure within the program. The structure name must be unique within the program or program system. If the structure name is associated with another open structure, an exception is generated. The structure name must meet the requirements for variable names.

After the keyword **NAME**, is the file specification of the structure file being opened. (See Chapter 16, Database Setup for information on legal structure file names.) The file specification can be any valid string expression.

The **ACCESS** option tells SheerPower whether to open the structure for input (reading field data) or for input and output (reading, changing and adding field data). ACCESS *input* is the default open mode for structures, meaning, that if no ACCESS option is provided, SheerPower opens the structure for INPUT.

When SheerPower executes the OPEN statement, it searches for the structure file specified. SheerPower opens the structure and assigns it the structure name. If the structure file does not exist, an exception is generated.

The **ACCESS** option determines how you can access the structure.

The access options are:

- ACCESS INPUT
  The structure is opened for input only. Fields can be referenced but not assigned new values. This is the default access method.
- ACCESS OUTIN
  The structure is opened for input and output. Fields can be referenced and assigned new values.

The **LOCK** option causes SheerPower to lock the structure from write access by others. As long as the structure is open, the structure cannot be modified by others. This can speed up SheerPower's structure statements (EXTRACTs especially). The particular effect depends on the file management system being used.

The **DATAFILE** option overrides the default datafile as specified by the structure.

FORMAT:

```
DATAFILE file_spec
```

file_spec is the file specification of the data file you want to open. The file_spec can be any string expression.

FORMAT:

```
CLOSE STRUCTURE struc_name
```

EXAMPLE:

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


List of California Clients
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

DESCRIPTION:

CLOSE STRUCTURE closes a structure from further access. Once the structure is closed, you cannot reference records or add them, and you cannot change field data. *struc_name* is the name associated with the structure, the name assigned with the OPEN statement.

You can use the statement **CLOSE ALL** to close *all* open structures and other files.

FORMAT:

```
ADD STRUCTURE struc_name
      ---
  [LET] struc_name(field) = expr
      ---
END ADD
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access outin
  input 'Enter ID number': id%
  input 'Enter last name': last$
  input 'Enter first name': first$
  input 'Enter state': state$
  input 'Enter phone': phone$
  add structure cl
    print
    print 'Adding '; last$; ', '; first$
    let cl(id) = id%
    let cl(last) = last$
```

```
    let cl(first) = first$
    let cl(state) = state$
    let cl(phone) = phone$
  end add
  close structure cl
  end


Enter ID number? 12233
Enter last name? Jones
Enter first name? Tom
Enter state? NV
Enter phone? 2345556161

Adding Jones, Tom
```

DESCRIPTION:

ADD STRUCTURE adds a record to a structure. The ADD STRUCTURE statement begins the add block. *struc_name* is the name associated with the structure that the record is going to be added to. **END ADD** marks the end of the block.

When ADD STRUCTURE is executed, SheerPower executes the block of code between the ADD STRUCTURE statement and END ADD. This block of code with **LET** statements is used to put data into the fields.

LET assigns a value to the field specified. *struc_name(field)* specifies a field in the structure. *expr* is an expression. When SheerPower executes the LET statement, it evaluates the expression and assigns the value of this expression to the field specified. END ADD writes out the new record.

FORMAT:

```
        CANCEL ADD
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access outin
  add structure cl
    input 'Client ID': cl(id)
    if  _exit  then cancel add
    input 'Last name': cl(last)
    if  _exit  then cancel add
    input 'First name': cl(first)
    if  _exit  then cancel add
    print 'Adding client'
  end add
  end


Client ID? 14422
Last name? White
First name? exit
```

DESCRIPTION:

CANCEL ADD immediately transfers control to the next statement after the END ADD statement, *without adding* any of the current record to a structure.

This statement can be used *only* within an ADD block---that is, between an ADD STRUCTURE and an END ADD pair of statements.

FORMAT:

```
EXIT ADD
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access outin
  add structure cl
    input 'Client ID ': cl(id)
    input 'Last name ': cl(last)
    input 'First name': cl(first)
    input 'Contact   ': cl(contact)
    if  _exit  then exit add
    input 'Salesman  ': cl(salesman)
    input 'Mother    ': cl(mother)
    input 'Comment   ': cl(comment)
  end add
  print 'Client added'
  end


Client ID ? 11111
Last name ? Hollerith
First name? Herman
Contact   ? exit
Client added
```

DESCRIPTION:

EXIT ADD transfers control immediately to the corresponding END ADD statement and **ADDS the record to the structure**.

This statement is useful when you want to add a record but do not have all the data. You can enter data into the first few fields and skip the rest of the fields.

FORMAT:

```
DELETE STRUCTURE struc_name
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access outin
  add structure cl
    input 'Client ID ': cl(id)
    input 'Last name ': cl(last)
    input 'First name': cl(first)
    input 'Contact   ': cl(contact)
    input 'State     ': cl(state)
  end add
  print 'Client added'
  delay
  extract structure cl
    include cl(state) = 'PA'
  end extract
  // delete all clients from Pennsylvania
  for each cl
    print 'Deleting '; cl(first); ' '; cl(last) ;'...';
    delete structure cl
    print 'record deleted'
```

```
  next cl
  close structure cl
  end


Client ID ? 87433
Last name ? Shores
First name? Sandy
Contact   ? 8435557373
State     ? PA
Client added
Deleting Sandy Shores...record deleted
```

DESCRIPTION:

DELETE STRUCTURE deletes the current record from the specified structure. *struc_name* is the structure name associated with the structure that the record is going to be deleted from.

FORMAT:

```
        [LOCK | UNLOCK] STRUCTURE struc_name
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access outin
  extract structure cl
    include cl(state) = 'CA'
  end extract
  for each cl
    print
    print cl(first); ' '; cl(last)
    lock structure cl   // give us exclusive access
    line input default cl(phone), prompt 'Enter new phone ': phone$
    if  _exit  then exit for
    cl(phone) = phone$
    unlock structure cl // put the record out to disk
                        // and release it
  next cl
  close structure cl
  end


Keith Kent
Enter new phone 6199675021

Paul Johnson
Enter new phone EXIT
```

DESCRIPTION:

LOCK/UNLOCK STRUCTURE can be used to lock or unlock the data record currently being accessed. SheerPower automatically locks and unlocks data when you work with it. LOCK and UNLOCK override SheerPower's automatic locking features allowing you to do it manually.

LOCK STRUCTURE locks the data currently being accessed, giving the program exclusive access to the current record. No one else can access the data until it is unlocked.

UNLOCK STRUCTURE unlocks the current record or data. The record is put to disk (if needed) and can again be accessed by others.

struc_name is the structure name associated with the structure.

FORMAT:

```
        [UNLOCK] STRUCTURE struc_name: COMMIT
        [UNLOCK] ALL: COMMIT
```

EXAMPLE:

```
  unlock structure payroll: commit
```

EXAMPLE:

```
  unlock all: commit
```

DESCRIPTION:

COMMIT tells both ARS and the underlying operating system to write all cached data to disk. Using **UNLOCK ALL** with COMMIT causes the data on all structures in the program to be written to disk.

FORMAT:

```
        EXTRACT STRUCTURE struc_name
                ---  block of code
          [INCLUDE | EXCLUDE] cond_expr
          [SORT [ASCENDING | DESCENDING] BY expression]
                ---
        END EXTRACT
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  print 'List of Clients'
  print
  extract structure cl
    print cl(first); ' '; cl(last), cl(phone)
  end extract
  close structure cl
  end


List of Clients

Earl Errant          (408) 844-7676
Al Abott             (202) 566-9892
Bud Brock            (218) 555-4322
Cathy Cass           (619) 743-8582
Dale Derringer       (818) 223-9014
Fred Farmer          (305) 552-7872
```

DESCRIPTION:

When SheerPower does an extract, it examines each record in the structure. For each record, SheerPower executes the code between the **EXTRACT STRUCTURE** and **END EXTRACT** statements. For instance, here is a structure with client information:

```
          ID #     LAST       FIRST      CITY      STATE  PHONE
        +------+-----------+--------+--------------+--+----------+
        |80543 |Cass       |Cathy   | San Diego    |CA|6197438582|
        |80542 |Brock      |Bud     | Duluth       |MN|2185554322|
        |80522 |Errant     |Earl    | Monterey     |CA|4088447676|
        |80561 |Derringer  |Dale    | Los Angeles  |CA|8182239014|
        |80531 |Abott      |Al      | New York     |NY|2025669892|
        |80573 |Farmer     |Fred    | Miami        |FL|3055527872|
```

EXTRACT creates a list of clients. The above program example extracts information from each record in the structure and displays each client's name and phone number.

When SheerPower does an extract, it examines each record of the structure in order. If the **KEY** option is specified, only those records with a key matching the KEY expression are examined. For each examined record, the following is done:

1. Each **INCLUDE** and **EXCLUDE** statement is checked in turn. The examined record is *not* extracted if an INCLUDE statement evaluates to *FALSE*, or an EXCLUDE statement evaluates to *TRUE*.
2. If any **SORT** specifications are given, a sort key is built using the SORT expression as the key. If no SORT specifications are given, the record is immediately extracted.
3. When all records have been examined, the sort keys, if any, are sorted.

SheerPower then builds a list of extracted records. This list can be accessed later with a **FOR EACH** loop. You can have up to 16 sort keys and 32 extract criteria.

FORMAT:

```
INCLUDE cond_expr
```

EXAMPLE:

```
open structure cl: name 'sheerpower:samples\client'
extract structure cl
  include cl(state) = 'CA'
end extract
print 'List of California Clients'
print
for each cl
  print cl(first); ' '; cl(last), cl(state)
next cl
close structure cl
end


List of California Clients

Keith Kent          CA
Paul Johnson        CA
Wayne Waters        CA
Earl Errant         CA
Cathy Cass          CA
Pete Porter         CA
Dale Derringer      CA
```

DESCRIPTION:

The **INCLUDE** statement includes records depending on the value of a conditional expression.

cond_expr is a conditional expression that SheerPower evaluates. If the expression is **TRUE**, SheerPower **includes** the record in the extract list. If the expression is **FALSE**, SheerPower excludes the record

from the list. For example, the program above creates an extract list containing only those clients located in California.

NOTE: The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

FORMAT:

```
        EXCLUDE cond_expr
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
    exclude cl(phone)[1:3] = '619'
  end extract
  print 'List of Clients'
  print
  for each cl
    print cl(first); ' '; cl(last), cl(phone)
  next cl
  close structure cl
  end


List of Clients

Earl Errant          (408) 844-7676
Al Abott             (202) 566-9892
Bud Brock            (218) 555-4322
Dale Derringer       (818) 223-9014
Fred Farmer          (305) 552-7872
```

DESCRIPTION:

The **EXCLUDE** statement excludes records from the extract list, depending on the value of a conditional expression.

cond_expr is a conditional expression. SheerPower evaluates this expression. If the expression is **TRUE**, SheerPower **excludes** the current record from the extract list. For example, the program above creates an extract list of all the clients in the client structure---except those with an area code of 619.

NOTE: The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

FORMAT:

```
        SORT [ASCENDING | DESCENDING] BY expr
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
    sort ascending by cl(last)
  end extract
  print 'List of Clients'
  print
  for each cl
    print cl(first); ' '; cl(last), cl(phone)
  next cl
  close structure cl
  end


List of Clients

Al Abott            (202) 566-9892
Bud Brock           (218) 555-4322
Cathy Cass          (619) 743-8582
Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
Fred Farmer         (305) 552-7872
```

DESCRIPTION:

The **SORT** statement sorts the records in an extract list in either **ASCENDING** or **DESCENDING** order. *expr* is an expression whose value determines how to order the list. SheerPower evaluates the expression for each record and stores the value. When all the records have been extracted, SheerPower orders the list according to these values.

You can sort in either ASCENDING or DESCENDING order. ASCENDING creates a list in ascending order (lowest to highest). DESCENDING creates a list in descending order (highest to lowest). The default is ascending order. String values are sorted according to the ASCII character set.

SheerPower does sorts in order. Therefore, you can use multiple sorts to order the list more and more specifically. For example, the following program creates a list of clients. The clients are sorted first by state and within each state by last name.

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
    sort ascending by cl(state)
    sort ascending by cl(last)
  end extract
  print 'List of Clients'
  print
  for each cl
    print cl(last); ', '; cl(first), cl(state)
  next cl
  close structure cl
  end


List of Clients

Cass, Cathy            CA
Derringer, Dale        CA
Errant, Earl           CA
Farmer, Fred           FL
Brock, Bud             MN
Abott, Al              NY
```

When you sort fields that are filled with nulls (no data was ever stored in them), the fields are sorted as though they were space-filled.

FORMAT:

```
FOR EACH struc_name
        ---
        ---  block of code
        ---
NEXT struc_name
```

EXAMPLE:

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


List of California Clients

Dale Derringer      (818) 223-9014
Earl Errant         (408) 844-7676
```

DESCRIPTION:

The **FOR EACH** statement can be used to execute a block of code for each record in the extract list. This allows for manipulation of structure information in programs.

The FOR EACH statement begins a loop that executes a block of code for each record in the extract list. *struc_name* is the structure name associated with the structure. **NEXT struc_name** marks the end of the loop.

The **REPEAT**, **ITERATE** and **EXIT FOR** statements can be used in the FOR EACH loop.

FORMAT:

```
EXTRACT STRUCTURE struc_name: KEY field = expr1 [TO expr2]
          ---
          ---  block of code
          ---
END EXTRACT
```

struc_name is the structure name associated with the structure. *field* is the name of the field that contains the key. *expr* is an expression that tells what key(s) to extract. SheerPower evaluates the expression, checks the structure's index for records with matching keys, and extracts these records (if any records are found).

KEY Option

The **KEY** option includes records using the record's *key*. The key is a field which has an index for fast access. The key option can considerably speed up extractions.

The conditional expression must match the field's data type. For instance, if the field has a CHARACTER data type, the expression must be a string expression.

For example, we have a structure with the following client information and the ID field is a key field:

```
      ID #     LAST       FIRST       CITY      STATE  PHONE
    +------+-----------+--------+--------------+--+----------+
    |80543 |Cass       |Cathy   | San Diego    |CA|6197438582|
    |80542 |Brock      |Bud     | Duluth       |MN|2185554322|
    |80522 |Errant     |Earl    | Monterey     |CA|4088447676|
    |80561 |Derringer  |Dale    | Los Angeles  |CA|8182239014|
    |80531 |Abott      |Al      | New York     |NY|2025669892|
    |80573 |Farmer     |Fred    | Miami        |FL|3055527872|
```

In the program below, the KEY option is used to extract the client with the ID number 80561.

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl: key id = 80561
    print 'Client:',
    print cl(first); ' '; cl(last), cl(id)
  end extract
  close structure cl
  end


Client:            Dale Derringer        80561
```

TO expr Option

Records can be extracted with keys in a certain range with the **TO** option. *expr1* is the lowest key to check. *expr2* is the highest. SheerPower extracts any records whose keys are within the range specified.

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  input 'Enter the lowest ID to check': lowest
  input 'Enter the highest ID to check': highest
  extract structure cl: key id = lowest to highest
    print cl(id); tab(10); cl(last)
  end extract
  close structure cl
  end


Enter the lowest ID to check? 80540
Enter the highest ID to check? 80570
80542    Brock
80543    Cass
80561    Derringer
```

FORMAT:

```
        EXTRACT STRUCTURE struc_name, FIELD field_expr: PARTIAL KEY str_expr
```

EXAMPLE:

```
open structure cl: name 'sheerpower:samples\client'
extract structure cl, field last: partial key 'Rod'
end extract
print 'List of clients with last name starting with Rod'
print
for each cl
  print cl(first); ' '; cl(last)
next cl
close structure cl
end

Homero Rodrigues
```

DESCRIPTION:

The **PARTIAL KEY** option will search in the EXTRACT STRUCTURE for part of a key value.

The above example program creates an extract list containing only those clients with a last name starting with "ROS".

Below is a structure with the following client information with the ID as a key field:

```
   ID #      LAST      FIRST      CITY      STATE  PHONE
+------+-----------+--------+--------------+--+----------+
|80543 |Roberts    |Cathy   | San Diego    |CA|6197438582|
|80542 |Roske      |Bud     | Duluth       |MN|2185554322|
|80522 |Rost       |Earl    | Monterey     |CA|4088447676|
|80561 |Rosty      |Dale    | Los Angeles  |CA|8182239014|
|80531 |Abott      |Al      | New York     |NY|2025669892|
|80573 |Farmer     |Fred    | Miami        |FL|3055527872|
```

FORMAT:

```
CANCEL EXTRACT
```

EXAMPLE:

```
open structure cl: name 'sheerpower:samples\client'
extract structure cl
  print 'Client: '; cl(last)
  line input 'Press enter to continue': z$
  if  _exit  then cancel extract
end extract
print 'Records extracted:'; _extracted
close structure cl
end

Client: Smith
Press enter to continue? EXIT
Records extracted: 0
```

DESCRIPTION:

CANCEL EXTRACT cancels the current extract of a record and transfers control to the next statement after the END EXTRACT statement.

This statement can *only* be used within an EXTRACT block---that is, between an EXTRACT STRUCTURE and an END EXTRACT pair of statements.

FORMAT:

```
        EXIT EXTRACT
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
    print 'Client: '; cl(last)
    line input 'Press enter to continue': z$
    if  _exit  then exit extract
  end extract
  print 'Records extracted:'; _extracted
  end


Client: Smith
Press enter to continue? <ENTER>
Client: Kent
Press enter to continue? EXIT
Records extracted: 1
```

DESCRIPTION:

EXIT EXTRACT passes control to the corresponding END EXTRACT statement, performs final sorting (if any), and creates the extracted collection.

FORMAT:

```
        REEXTRACT STRUCTURE struc_name
                  ---
           [INCLUDE | EXCLUDE] cond_expr...
           [SORT [ASCENDING | DESCENDING] BY expression...
                  ---
        END EXTRACT
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access input
  extract structure cl
    include cl(state) = 'CA'
  end extract
  reextract structure cl
    exclude cl(phone)[1:3] <> '619'
    sort ascending by cl(last)
  end extract
  print 'List of California Clients in Area Code 619'
  for each cl
    print cl(first); ' '; cl(last), cl(phone)
  next cl
  close structure cl
```

```
  end

List of California Clients in Area Code 619
Cathy Cass          (619) 743-8582
Paul Johnson        (619) 489-5551
Keith Kent          (619) 967-5021
Pete Porter         (619) 778-6709
Wayne Waters        (619) 564-1231
```

DESCRIPTION:

REEXTRACT STRUCTURE can be used to do a second extract on a list of structure records you previously extracted. This allows for increasingly specific records to be chosen through a series of REEXTRACTs.

REEXTRACT does an extract on the list of records previously extracted. *struc_name* is the structure name associated with an open structure.

END EXTRACT marks the end of the REEXTRACT construct. REEXTRACT operates the same as EXTRACT. However, REEXTRACT operates on a previously extracted list.

Note on EXTRACT

Extract operations by key cannot be performed with REEXTRACT.

FORMAT:

```
        EXTRACT STRUCTURE struc_name: APPEND
```

EXAMPLE:

```
  open structure detail: name 'sheerpower:samples\detail'
  set structure detail: extracted 0
  extract structure detail, field lineid : &
      key '10301001' to '10302000', append
    sort by detail(prodnbr)
    sort by detail(invnbr)
  end extract
  extract structure detail, field lineid : &
      key '10311001' to '10312000', append
    sort by detail(prodnbr)
    sort by detail(invnbr)
  end extract
  print 'Prod'; tab(7); 'Line ID'; tab(17); 'Quantity'
  for each detail
    print detail(prodnbr); tab(7); detail(lineid); &
        tab(17); detail(qty)
  next detail
  end

Prod  Line ID   Qty
22800 10301-002      2
22800 10301-004      1
22800 10301-006      2
24100 10311-003      1
24200 10301-003      1
24200 10311-009      1
28400 10311-001      2
28800 10301-009      2
28800 10311-002      9
28800 10311-005      1
28800 10311-006      1
```

```
31020 10301-005      1
31040 10311-010      2
31150 10301-001      1
31150 10301-008      8
31150 10311-004      1
31150 10311-008      1
33090 10301-007      2
33090 10311-007      1
```

DESCRIPTION:

The **EXTRACT STRUCTURE: APPEND** statement adds records to the last collection of extracted records rather than creating a new collection.

FORMAT:

```
        ASK STRUCTURE struc_name: struc_option [num_var | str_var]
```

DESCRIPTION:

The **ASK STRUCTURE** statement is used to ask about various device and structure characteristics from within your programs. *struc_name* is the name of a structure whose characteristics are being asked about. *struc_option* can be any of the structure options available. The options are explained in the following sections.

FORMAT:

```
        ASK STRUCTURE struc_name, FIELD field_expr: item var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  ask structure cl: fields num_fields
  for i = 1 to num_fields
    clear
    ask structure cl, field #i: description b$
    ask structure cl, field #i: prompt a$
    ask structure cl, field #i: position a%
    ask structure cl, field #i: length b%
    ask structure cl, field #i: heading f$
    ask structure cl, field #i: printmask c$, &
                                screenmask d$, &
                                help e$
    print at 5,5: ''
    print 'Description   : '; b$
    print 'Prompt        : '; a$
    print 'Position      : '; a%
    print 'Field length  :' ; b%
    print 'Rpt. heading  : '; f$
    print 'Print mask    : '; c$
    print 'Screen mask   : '; d$
    print 'Help          : '; e$
    delay
  next i
  close structure cl
  end

Description   : Client ID number
```

```
Prompt        : Client ID number
Position      :  1
Field length  : 5
Rpt. heading  : CLNT,ID
Print mask    : >####
Screen mask   : digits:#####
Help          : Client ID number
```

DESCRIPTION:

The **FIELD** option allows you to get information about a specific *field* in a structure. *struc_name* is the name of the structure. *field_expr* is the field you are inquiring about. *item* specifies what type of information you are asking. The information is stored in the variable specified.

The following sections provide more information on what you can use for *field_expr* and on the various field *items*.

For more information:

The *ITEM* information is created when the SETUP routine is used to define the field. You can refer to Chapter 16, Database Setup for information on defining fields.

The **field_expr** used in the **ASK STRUCTURE FIELD: item** statement can be either a *constant* or a *string or numeric expression*.

A string constant can be used to specify the field name. To use a string constant, just enter the field name, *without quotes*. SheerPower will then use the string constant as the field name:

```
        ASK STRUCTURE sheerpower:samples\CLIENT, FIELD LAST: DESCRIPTION A$
                                                 /
                  the field is specified by its field name
```

You can also specify a field name with an expression. To use an expression, precede the expression with a pound sign (#). The pound sign tells SheerPower that the following characters are an expression, not the field name. If a pound sign is not included SheerPower will interpret the characters as a field name.

```
        ASK STRUCTURE CL, FIELD #FIELDNAME$: DESCRIPTION A$
                                         /
          the field is specified by the value of the variable FIELDNAME$

        ASK STRUCTURE CL, FIELD #FIELDNUM: DESCRIPTION A$
                                         /
          the field is specified by the value of the variable FIELDNUM
```

EXAMPLE:

This example shows how to access the actual field data using field expressions.

```
open structure cl: name 'sheerpower:samples\client'
do
clear
print at 1, 1:;
  ask_fields
  if  _back or _exit  then exit do
  show_data
loop
close structure cl
stop

routine ask_fields
do
```

```
    if  _error  then  set error off
    print 'Field List:  '; &
        'ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE'
    print
    line input 'Select fields to display': field_list$
    if  _back or _exit  then exit do
    for f = 1 to elements(field_list$)
      field$ = element$(field_list$, f)
      ask structure cl, field #field$: number z
      if  z = 0  then
        message error: 'Illegal field: '; field$
      end if
    next f
  loop while _error
  end routine

  routine show_data
  print
  extract structure cl
    for f = 1 to elements(field_list$)
      field$ = element$(field_list$, f)
      print cl(#field$),
    next f
    print
  end extract
  delay
  end routine
  end


Field List:  ID, LAST, FIRST, MIDDLE, STREET, CITY, STATE, ZIP, PHONE

Select fields to display? last,first,phone

Smith               Sam                 (809) 555-8789
Kent                Keith               (619) 967-5021
Johnson             Paul                (619) 489-5551
Waters              Wayne               (619) 564-1231
Rodrigues           Homero              (   )    -   0
Donaldson           George              (   )    -   0
Errant              Earl                (408) 844-7676
Abott               Al                  (202) 566-9892
Brock               Bud                 (218) 555-4322
Cass                Cathy               (619) 743-8582
Porter              Pete                (619) 778-6709
Derringer           Dale                (818) 223-9014
Farmer              Fred                (305) 552-7872
```

```
        ASK STRUCTURE struc_name, FIELD field_name: ACCESS str_var
```

ACCESS retrieves the access (read and write) rules for the specified field. This information tells if the field can be read and written to. **N** is normal--meaning the field can be read and written to if the structure is also "N"ormal. Depending on whether security levels have been set on the structure and/or field, the letter can be in the range of A thru Z. SheerPower defaults to N when the structure is created and fields are defined.

```
  open structure inv: name 'sheerpower:samples\invoice', access input
  ask structure inv, field custnbr: access x$
  print x$
  close structure inv
  end


READ:N, WRITE:N
```

```
ASK STRUCTURE struc_name, FIELD field_expr: APPLICATION str_var
```

APPLICATION returns a name of an application for the specified field in a string variable. This is optional information the user provides when the field is defined.

```
  open structure cl : name 'sheerpower:samples\client'
  ask structure cl, field id: application str$
  print str$
  end


REPORTING_ID
```

```
ASK STRUCTURE struc_name, FIELD field_expr: ATTRIBUTES str_var
```

ATTRIBUTES returns the SheerPower field semantics (NUM - number, UC - upper-case, etc.) for the specified field in a string variable.

Refer to Section 16.3.1.9, Semantics for detailed information on field attributes.

```
ASK STRUCTURE struc_name, FIELD field_expr: CHANGEABLE num_var
```

CHANGEABLE returns a value of TRUE or FALSE. If the field specified by *field_expr* can be changed, the value is TRUE. If the field cannot be changed, the value is FALSE.

```
  open structure cl: name 'sheerpower:samples\client'
  ask structure cl, field id: changeable z
  ask structure cl, field city: changeable z1
  print z
  print z1
  close structure cl
  end


0
1
```

```
ASK STRUCTURE struc_name, FIELD field_expr: DATATYPE str_var
```

DATATYPE returns the field data type, such as CH (character), IN (integer), etc., in a string variable.

Refer to Section 16.3.1.4, Data Type for detailed information on field data types.

```
ASK STRUCTURE struc_name, FIELD field_expr: DESCRIPTION str_var
```

DESCRIPTION returns the description for the specified field in a string variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: HEADING str_var
```

HEADING returns the report column heading for the specified field in a string variable. This is the heading that would appear in a Guided Query Language (GQS) report column.

```
ASK STRUCTURE struc_name, FIELD field_expr: HELP str_var
```

HELP returns the help text for the specified field in a string variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: KEYED num_var
```

KEYED returns a value of TRUE or FALSE in a numeric variable. If the specified field is a key field, the value is TRUE. Otherwise, the value is FALSE.

```
ASK STRUCTURE struc_name, FIELD field_expr: LENGTH num_var
```

LENGTH returns the length of the specified field in a numeric variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: NAME str_var
```

NAME returns the name of the specified field in a string variable.

```
ASK STRUCTURE struc_name, FIELD field_name: NULL int_var
```

If the specified field is **NULL** (i.e. contains no data), this statement returns TRUE. If the field is not NULL, the statement returns FALSE.

```
ASK STRUCTURE struc_name, FIELD field_expr: NUMBER num_var
```

NUMBER returns the field number of the specified field in a numeric variable. Fields are numbered sequentially. If the field does not exist, SheerPower returns a value of 0.

```
ASK STRUCTURE struc_name, FIELD field_expr: OPTIMIZED num_var
```

OPTIMIZED returns a value of TRUE or FALSE in a specified numeric variable. If the key field in *field_expr* is optimized, the value is TRUE. Otherwise, the value is FALSE.

```
open structure cl: name 'sheerpower:samples\client'
ask structure cl, field id: optimized z
print z
close structure cl
end


1
```

```
ASK STRUCTURE struc_name, FIELD field_expr: POSITION num_var
```

POSITION returns the starting position for the specified field in a numeric variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: PRINTMASK str_var
```

PRINTMASK returns the print mask for the specified field in a string variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: PROMPT str_var
```

PROMPT returns the prompt for the specified field in a string variable.

```
ASK STRUCTURE struc_name, FIELD field_expr: SCREENMASK str_var
```

SCREENMASK returns the screen mask for the specified field in a string variable. This option is not currently used.

```
ASK STRUCTURE struc_name, FIELD field_expr: VRULES str_var
```

VRULES returns the validation rules for the specified field in a string variable.

Refer to the Section 6.6.6, VALID(text_str, rule_str) for information on validation rules.

```
open structure cl : name 'sheerpower:samples\client'
ask structure cl, field bday: vrules str$
print str$
end


date ymd; minlength 8
```

FORMAT:

```
ASK STRUCTURE struc_name: CURRENT str_var
```

EXAMPLE:

```
dim a$(100)
let i = 0
open structure cl: name 'sheerpower:samples\client'
extract structure cl
end extract
for each cl
  print cl(last); ', '; cl(first)
  input 'Would you like to see this record (Y/N)': yn$
  if  yn$ = 'Y'  then
    let i = i + 1
    ask structure cl: current a$(i)
  end if
next cl
print
for j = 1 to i
  set structure cl: current a$(j)
  print cl(last); ','; cl(first), cl(state), cl(phone)
next j
end


Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

DESCRIPTION:

ASK STRUCTURE: CURRENT assigns the current record value to the *str_var*. Once the current record has been assigned with ASK STRUCTURE: CURRENT, the SET STRUCTURE: CURRENT statement can be used to get this record.

SheerPower returns a zero-length string if there is no CURRENT record.

FORMAT:

```
ASK STRUCTURE struc_name: DATAFILE str_var
```

EXAMPLE:

```
open structure cust: name 'sheerpower:samples\customer'
open structure cl: name 'sheerpower:samples\client'
ask structure cust: datafile z1$
ask structure cl: datafile z2$
print z1$
print z2$
end


C:\SHEERPOWER\samples\CUSTOMER.ARS
C:\SHEERPOWER\samples\CLIENT.ARS
```

DESCRIPTION:

The **ASK STRUCTURE: DATAFILE** statement returns the full file name/file specification of a specified structure. *struc_name* is the specified structure name. *str_var* contains the returned file specification.

FORMAT:

```
        ASK STRUCTURE struc_name: FIELDS num_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access input
  ask structure cl: fields z
  print z
  end


18
```

DESCRIPTION:

The number of fields in a structure can be determined with the **ASK STRUCTURE: FIELDS** statement. The number is assigned to the numeric variable *num_var*.

FORMAT:

```
        ASK STRUCTURE struc_name: KEYS num_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access input
  ask structure cl: keys z
  print z
  end


2
```

DESCRIPTION:

The **ASK STRUCTURE: KEYS** statement returns the number of keys that are accessible by SheerPower. It returns the value of 0 if no keys are available.

FORMAT:

```
        ASK STRUCTURE struc_name: CAPABILITY str_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client', access input
  ask structure cl: capability z$
  print z$
  end


INDEXED,INPUT
```

DESCRIPTION:

Given a structure expression, **ASK STRUCTURE: CAPABILITY** sets *str_expr* to a comma delimited string containing one or more of the following: INDEXED, INPUT, OUTPUT

Table 15-1 Structure Types

Structure Type	Description
INDEXED	The structure is indexed; it can be accessed by key with statements such as SET STRUCTURE...KEY.
INPUT	You can read from the structure.
OUTPUT	You can write to the structure.
null string	The structure is not currently open.

FORMAT:

```
        ASK STRUCTURE struc_name: EXTRACTED num_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
  end extract
  ask structure cl: extracted z
  print 'Records found: '; z
  end


Records found:  13
```

DESCRIPTION:

ASK STRUCTURE: EXTRACTED asks the operating system for the last extracted count for the structure specified.

FORMAT:

```
        ASK STRUCTURE struc_name: ID str_var
```

EXAMPLE:

```
  declare structure str
  open structure cl: name 'sheerpower:samples\client'
  ask structure cl: id cl_id$
  set structure str: id cl_id$
  extract structure str
  end extract
  for each str
    print str(#1); '  '; str(#2)
  next str
  end


20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

The **ASK STRUCTURE: ID** statement asks the operating system for the ID of a structure and returns it in the string variable *str_var*.

FORMAT:

```
        ASK STRUCTURE struc_name: POINTER num_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
  end extract
  for each cl
    ask structure cl: pointer ptr
    print ptr, cl(last)
  next cl
  end


1                 Smith
2                 Jones
3                 Kent
4                 Johnson
5                 Waters
6                 Errant
7                 Brock
8                 Cass
9                 Porter
10                Derringer
11                Farmer
```

DESCRIPTION:

From within a FOR EACH...NEXT STRUCTURE_NAME block, **ASK STRUCTURE: POINTER** asks the structure for the number of the current record pointer.

FORMAT:

```
        ASK STRUCTURE struc_name: RECORDSIZE int_var
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  ask structure cl: recordsize recsize
  print 'Logical recordsize: '; recsize
  end


Logical recordsize:   400
```

DESCRIPTION:

The **ASK STRUCTURE: RECORDSIZE** statement returns the record size of the structure data file.

FORMAT:

```
        ASK STRUCTURE struc_name: ACCESS str_var
```

EXAMPLE:

```
  open structure inv: name 'sheerpower:samples\invoice', access input
  ask structure inv: access x$
  print x$
  close structure inv
  end


SECURITY:N, READ:N, WRITE:N, UPDATE:N, DELETE:N
```

DESCRIPTION:

The **ASK STRUCTURE: ACCESS** statement retrieves the access rules for the specified structure. Security level, data file read, write, update, and delete rules are returned.

FORMAT:

```
        ASK STRUCTURE #string_expr. . .
        SET STRUCTURE #string_expr. . .
```

EXAMPLE:

```
open structure cl: name 'sheerpower:samples\client'
str$ = 'CL'
fld$ = 'ID'
do_work
stop

routine do_work
  ask structure #str$, field #fld$: description dsc$
  print 'Description is: '; dsc$
end routine
end

Description is: Client ID number
```

DESCRIPTION:

A string expression for the structure name can be used in an **ASK STRUCTURE #string_expr** or **SET STRUCTURE #string_expr** statement. This allows generalized code to be written to work for several structures.

FORMAT:

```
ASK STRUCTURE struc_name : ENGINE str_var
```

EXAMPLE:

```
open structure cl : name 'sheerpower:samples\vendor'
ask structure  cl : engine ename$
print 'DATABASE ENGINE is: '; ename$
end

DATABASE ENGINE is: ARS
```

DESCRIPTION:

The **ASK STRUCTURE: ENGINE** statement returns the name of the database engine (record management system) being used for the specified structure in *str_var*.

The **SET STRUCTURE** statement is used to change various device and structure characteristics from within your programs.

FORMAT:

```
SET STRUCTURE struc_name: struc_option [num_var | str_var]
```

SET STRUCTURE sets characteristics of structures. *struc_name* is the name of the structure whose characteristics are being set. *struc_option* is the option being set. The options are explained in the following sections.

FORMAT:

```
SET STRUCTURE struc_name: CURRENT str_expr
```

EXAMPLE:

```
dim a$(100)
let i = 0
open structure cl: name 'sheerpower:samples\client'
extract structure cl
end extract
for each cl
  print cl(last); ', '; cl(first)
  input 'Would you like to see this record (Y/N)': yn$
  if  yn$ = 'Y'  then
    let i = i + 1
    ask structure cl: current a$(i)
  end if
next cl
print
for j = 1 to i
  set structure cl: current a$(j)
  print cl(last); ','; cl(first), cl(state), cl(phone)
next j
end


Errant, Earl      Would you like to see this record (Y/N)? Y
Abott, Al         Would you like to see this record (Y/N)? Y
Brock, Bud        Would you like to see this record (Y/N)? N
Cass, Cathy       Would you like to see this record (Y/N)? N
Derringer, Dale   Would you like to see this record (Y/N)? Y
Farmer, Fred      Would you like to see this record (Y/N)? Y
Errant, Earl       CA       (408) 844-7676
Abott, Al          NY       (202) 566-9892
Derringer, Dale    CA       (818) 223-9014
Farmer, Fred       FL       (305) 552-7872
```

DESCRIPTION:

The **CURRENT** option sets the current record to that specified by the *str_expr*. The str_expr contains the information for the current record for the record management system being used. ASK STRUCTURE: CURRENT is used to store a current record value into a string variable.

When SheerPower executes a SET STRUCTURE: CURRENT statement, it uses the structure name and sets the current record to the value specified by the string variable. The structure must be open and str_expr must contain a value stored with the **ASK STRUCTURE: CURRENT** statement.

If a null string is used for the value, SheerPower sets the structure to no current record and sets _EXTRACTED to zero.

```
SET STRUCTURE struc_name: CURRENT ''
```

FORMAT:

```
SET STRUCTURE struc_name, FIELD field_expr: KEY str_expr
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  line input 'Enter an ID': id$
  set structure cl, field id: key id$
  if  _extracted = 0  then
    message error: 'Not found'
  else
    print cl(id), cl(last)
  end if
  end


Enter an ID? 80561
80561               Derringer
```

DESCRIPTION:

The **FIELD** option allows for record retrieval using a key field in a structure. SETUP's "SHOW FIELDS" menu option (see Chapter 16, Database Setup) displays the field names. The FIELD option is currently used only with the **KEY** or **PARTIAL KEY** option. The KEY option specifies the key to look for. The key is contained in *str_expr*.

The above example shows how to look in the CLIENT structure for an ID.

_EXTRACTED contains the number of records extracted. If the operation fails, _EXTRACTED will be 0 and an error message will be displayed.

FORMAT:

```
        SET STRUCTURE struc_name, FIELD field_expr: PARTIAL KEY str_expr
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  input 'Name': name$
  set structure cl, field last: partial key name$
  print cl(id); ' '; cl(last)
  end


Name? D
80561 Derringer
```

DESCRIPTION:

This statement retrieves the first record matching the partial key in *str_expr*.

FORMAT:

```
        SET STRUCTURE struc_name: ID str_expr
```

EXAMPLE:

```
  declare structure str
  open structure cl: name 'sheerpower:samples\client'
  ask structure cl: id cl_id$
  set structure str: id cl_id$
  extract structure str
  end extract
  for each str
    print str(#1); ' '; str(#2)
  next str
  end


20000 Smith
20001 Jones
20002 Kent
23422 Johnson
32001 Waters
43223 Errant
80542 Brock
80543 Cass
80544 Porter
80561 Derringer
80573 Farmer
```

DESCRIPTION:

SET STRUCTURE: ID sets a structure to a structure ID that has been stored previously into a string variable with the **ASK STRUCTURE: ID** statement. Once the SET STRUCTURE: ID statement has been used, the structure with the new structure name (STR in the example) can be accessed. By using these statements, generalized routines can be written when the structure that will be accessed until runtime is not known.

FORMAT:

```
        SET STRUCTURE struc_name: POINTER num_expr
```

EXAMPLE:

```
  open structure cl: name 'sheerpower:samples\client'
  extract structure cl
  end extract
  set structure cl: pointer 3
  print cl(id); ' ' ; cl(last)
  end


23422 Johnson
```

DESCRIPTION:

This statement sets the structure to the *nth* record extracted. The statement is useful after an extract has been done because it provides random access to any record extracted. There is no error message if there are no records extracted, or if the number given is out of range. If the number is valid, _EXTRACTED is set to 1; otherwise, it is set to 0.

FORMAT:

```
SET STRUCTURE struc_name: EXTRACTED 0
```

EXAMPLE:

```
open structure vend: name 'sheerpower:samples\vendor'
set structure vend: extracted 0
end
```

DESCRIPTION:

Setting the number of records extracted to zero causes a new collection to be started. The **SET STRUCTURE struc_name : EXTRACTED 0** statement is used in conjunction with the **EXTRACT STRUCTURE struc_name: APPEND** statement.

Below is an example of opening an existing structure, extracting a record from it and updating the information in the record.

```
// Simple customer query
open structure cust: name 'sheerpower:\samples\customer', access outin
do
  line input 'Customer number', default '12513': cust$
  if _exit or _back or cust$ = '' then exit do // get out if nothing to do
  set structure cust, field custnbr: key cust$  // do the search
  if _extracted = 0 then
    message error: 'Cannot find '; cust$
    repeat do
  end if
  print cust(custnbr), cust(name)
  line input 'New name', default cust(name): newname$
  if _exit or _back then repeat do
  if newname$ = cust(name) then repeat do // nothing to do
  cust(name) = newname$  // update the name
  print 'Name changed to: '; cust(name)
loop
end
```

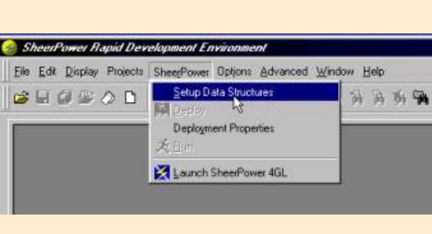
SheerPower works with data in any database engine. SETUP tells SheerPower where data files are located and what database engines are used. Structures and definitions are the description of the data layout and it’s characteristics. Once structures have been defined with SETUP, SheerPower's structure statements can be used to manipulate the data.

SETUP supports the following database engines:

ARS
ODBC
FASTFILE

See Chapter 17, SheerPower and ODBC for instruction on using ODBC with SheerPower.

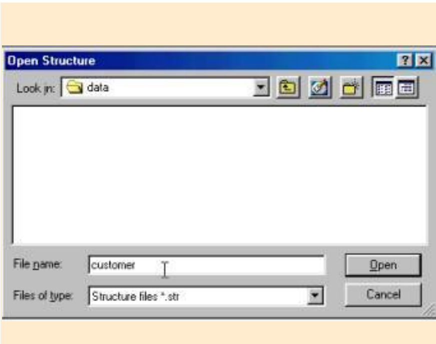
SETUP is entered by selecting **SheerPower** on the toolbar, then selecting **Setup Data Structures**.



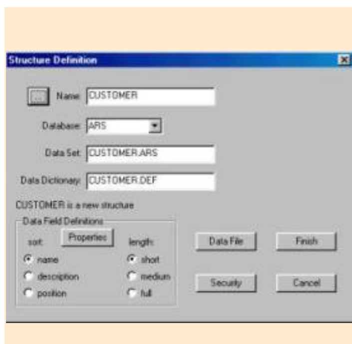
The **OPEN STRUCTURE** dialog box will open. If this is an existing structure, select the folder where it is stored, then select the file name.

If you are creating a new structure, choose the folder where it is to be stored and type in the file name.

Click on `[Open]` to open the structure.



Once the structure has been opened, the **STRUCTURE DEFINITION** window will be displayed.



This section describes the SETUP structure definition fields and options for answering them. Data fields inside the structure can be defined or modified. Field definitions can also be displayed in multiple formats.

Default answers are provided for many of the structure definition fields. You may either accept the default answer, or type in different information.

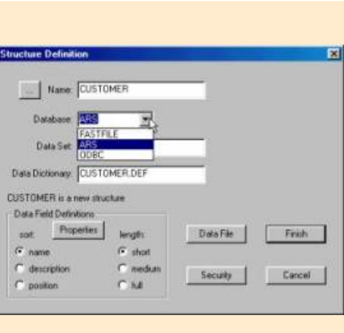
The **NAME** of the structure will default to be the same name as the file that you opened or created. The default extension for the file is **.STR**. The structure file contains:

- the name of the database engine (record management system)
- the name of the data set (data file)
- the name of the data dictionary

The structure name can be any Windows file specification.

Clicking on the `[...]` button to the left of the structure **NAME** field will reopen the **Open Structure** dialog box. This allows you to choose a different structure name.

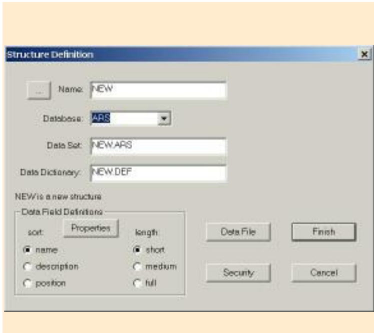
Click on the down arrow beside the **DATABASE** field to select the **DATABASE ENGINE** used by this structure.



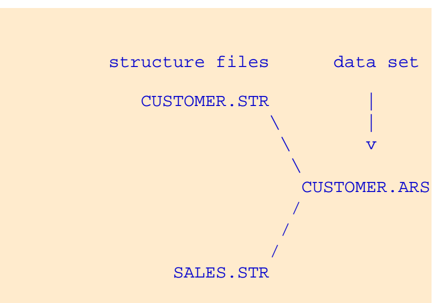
If this is a new structure, the default database engine will be ARS.

When modifying an existing data structure, the default will be the database engine specified in the structure file when it was originally setup.

Inside the **DATA SET** field, type in the name of the **DATA SET** that will be used with this structure. If this is a new structure, the default data set name will be the same as the structure name. The default extension is **.ARS**.



The same data set can be used for a number of structures. For instance, you might use the CUSTOMER data set for both a CUSTOMER and a SALES structure:

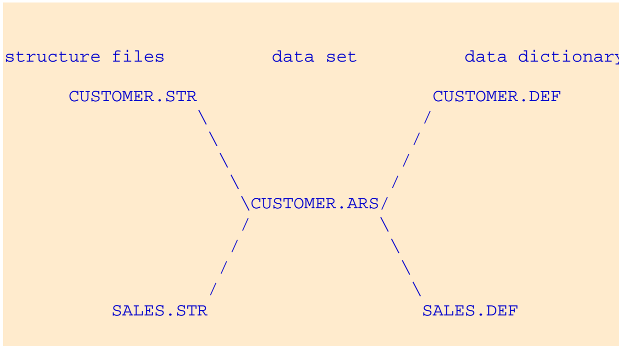


This can be useful when there are multiple uses for a set of data, but the access and definitions required should be different for each structure.

The data set name can be any Windows file specification.

Type in the name of the **DATA DICTIONARY** to be used with this structure. If a new structure is being created, the default data dictionary name will be the same as the structure name. The default extension is **.DEF**. This is illustrated in Section 16.2.2, Database Engine.

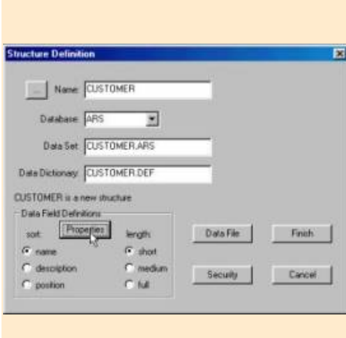
As with the data set, the same data dictionary can be used for a number of structures. In fact, any combination of definition, data and structure files can be created:



The data dictionary name can be any Windows file specification.

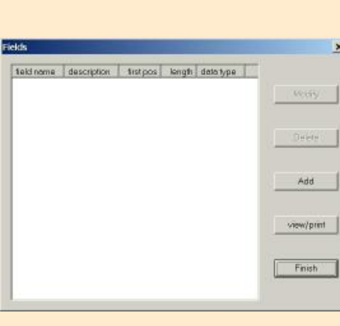
After you have selected the structure and data dictionary that you will be working with, the other options in the Structure Definition window can be used to perform SETUP procedures.

Inside the **STRUCTURE DEFINITION WINDOW** click on `[Properties]`. The **FIELDS** window will appear.



The options inside the Fields window will enable you to perform the following procedures in SETUP on the selected data dictionary:

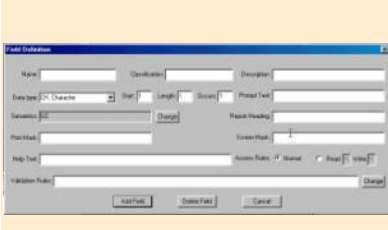
- Modify
- Delete
- Add
- View/Print
- Finish.



Clicking on `[ADD]` will open the **FIELD DEFINITION** window. This selection is used to add data fields to new or existing data dictionaries. Here you can enter in the following information for each data field:

- name
- classification

- description
- data type
- starting position
- length
- occurances
- prompt
- text
- semantics
- report heading
- print mask
- screen mask
- help text
- access rules
- validation rules



When you have completed adding all of the field information inside the Field Definition window, click on `[DONE]`. The field definitions will then be added to the **DATA DICTIONARY**. You will be returned to the Fields window where you can either perform other functions or choose `[Finish]`.

If this is a new structure, the new structure file will be created *after* you click on `[Finish]`.

Enter the data field **NAME** inside the **NAME** input field. For example, name a data field "CUST_ID" that will store customer identification information.

The field name:

- must begin with a letter
- can consist of numbers, letters, underscores and dollar signs
- can be up to 32 characters long
- the last character of the field name can be a "$" or "%"

Note that the **Description**, **Prompt Text** and **Report Heading** fields will default to be the same as the data field name. This default may be changed.

The **CLASSIFICATION** input field should be left blank at this time.

The **Description** input field will default to the same as the data field NAME that was entered. Any "$", "%", "_" characters will be changed to spaces.

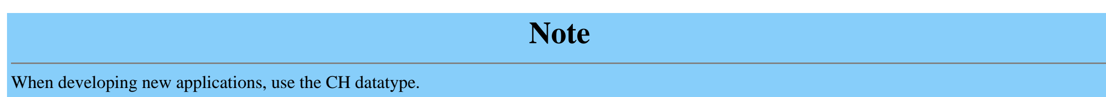
You may type in any description you want for this data field. This description is shown when viewing the data dictionary.

The description can consist of any characters on the keyboard and be up to 30 characters long.

Beside the **DATA TYPE** input field, click on the down arrow to select a data type. The data type is the internal representation of the data.

The most common data type is **CH** (character) which is the default. Use CH as the data type when defining names, addresses, money, date/times.

The other datatypes that follow such as IN (integer), IU (integer unsigned), ... are provided for special cases where there is pre-existing binary data that must be defined.

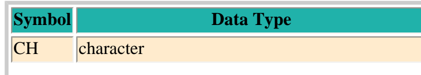


Note

When developing new applications, use the CH datatype.

The data type will be one of the following:

Table 16-1 Data Types in SETUP



Symbol	Data Type
CH	character

AC	ASCII counted string
AP	Pointer to ASCII counted
C3	COBOL comp-3, also known as packed decimal
DS	Date stamp (length=8)
EB	EBCDIC
FL	Floating point
GF	G-Float
IN	Integer (signed)
IU	Integer (unsigned)
PF	Packed floating
PZ	Packed zipcode
QS	Quadword (signed)
RO	Right overpunch
RS	Right sign separate
UN	Undefined
ZE	Zoned EBCDIC
ZN	Zoned numeric; used by DIBOL

Note

The DATE (DS) attribute is currently only used by applications and has no effect within SheerPower.

Enter the **STARTING POSITION** for this data field. The first field starting position will be defaulted as position 1.

Fields generally follow each other directly. Therefore, the first position of a new data field is usually one more than the last position used. If the last position used was ten, the next data field would start at eleven. SheerPower automatically defaults the starting position of each data field to the correct sequential position.

Defining Existing ARS File Fields

When defining data fields for an existing ARS file, keep in mind that the starting position of an ARS file record is 0 and in SheerPower, the starting position is 1. (ARS is 0 oriented, whereas SheerPower is 1 oriented.) For example, if the ARS file data field starts in position 0, when you define this data field in SheerPower, you must define the data field as starting in position 1. If the ARS data field starts in position 20, you would use 21 as the starting position, etc.

Enter the **LENGTH** required for the field. The default value shown for every data field added is 1 character. The length should be the maximum number of character positions that data in this data field may contain. Data fields can be any length. The length must be given as an integer number.

Variable Field Lengths

If a CH or UN data type field has a first position that starts at or before the end of a record (row), that field will have its length adjusted upon reference. The new length will reflect the actual end of the record (row).

For example, if you made a data field called **ALL**, and it started in position one with a length of 5000 bytes---and the actual record length was 1200 bytes, referencing the **ALL** field would return just 1200 bytes.

Inside the **OCCURS** input field, type in the number of times that this data field occurs in the data record. The default number of occurrences is **1**. If the data field being defined is an array, enter the number of elements in the array. Otherwise, leave the default value of **1**.

Enter the **PROMPT TEXT** for this data field. The prompt text will be used when data is entered for this field. The prompt text:

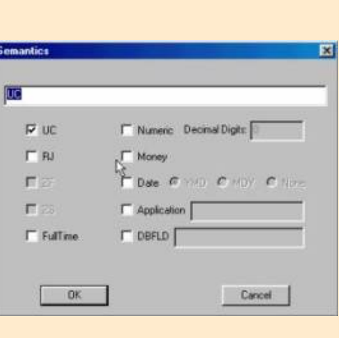
- can be any characters on the keyboard.
- can be up to 30 characters long

The default prompt text will be the data field **NAME** you entered for this field.

Enter the **SEMANTICS (attributes)** that apply to this data field. The semantics describe what type of data will be in the field.

The default semantic is UC (uppercase). To change semantics, or to add/remove from the list, click on `[Change]`. The semantics window will open. Place a checkmark beside a semantic you want to add, and remove the checkmark from beside any semantic you want to remove.

For example, if a dollar amount will be stored into this field you would select **MONEY**. This will set the semantics for this field to **NUM:2** (numeric field with two decimal places) and **RJ** (right justified).



Setting/defining semantics is optional. If you do not want any pre-defined semantics for the field, remove the UC semantic and click okay to leave the semantics list blank.

The semantics options are:

Table 16-2 Semantics Options in SETUP

Semantic	Description
NUMERIC	Indicates a numeric field which allows only numbers. The semantic RJ, right-justified, is automatically appended when NUMERIC is selected. If you select this you will also fill in Decimal Digits to define how many decimal places the field contains.
MONEY	Specifically sets the field up as a numeric dollar field with two decimal places and right-justified.
RJ	The data will be right-justified in the field.
UC	All letters will be stored in upper-case format. Additionally, field comparison data will be upper-cased prior to any comparison.
ZF	The data will be zero filled.
ZS	Causes all leading zeros to be suppressed.
DATE[:YMD|MDY]	This field contains a date. The field must be six or eight digits long. The default date format is YMD. You can enter DATE or optionally, specify the date format. The date format is currently only used by applications. SheerPower does NOT automatically switch date values into or out of the format specified.
FULLTIME	Indicates that the field contains a date-time stamp and the time value is to be preserved. By default, SheerPower stores/retrieves only the date portion of a date stamp field.
APPLICATION:name	This is a user defined semantic and is not used by SheerPower. A user can assign a name to indicate that the field is related to a specific application (i.e. accounting). Application programs can then ask for the application name and perform some action based on the data.
DBFLD:fieldname	This is a special semantic for augmented definition files only. The DBFLD semantic is used to associate an augmented field definition with an actual database field. Fieldname is the actual name of the field in the database.

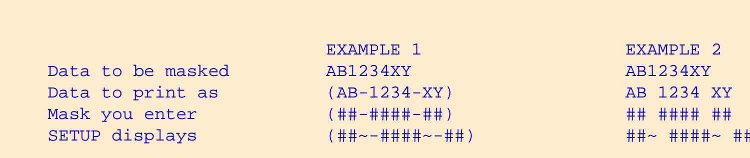
The **REPORT HEADING** is the heading that you want to appear on reports. This heading will appear at the top of the report column containing this field's data. If you are adding a new record, the default report heading is the **NAME** you entered.

The heading can be up to 30 characters long.

To make a multi-line heading, separate each line of the heading with a comma (i.e., Customer,Information).

Enter a **PRINT MASK** for this data field. The print mask is used to display information entered into this field. The print mask will be used whenever you use SheerPower's PRINT statement to print this field's information.

Print masks are created with the #, @ and % characters. SETUP will automatically provide the tilde (~) character for any literal (non-masking) characters found in the entered mask. Some examples:



```
                         EXAMPLE 1                  EXAMPLE 2
Data to be masked        AB1234XY                   AB1234XY
Data to print as         (AB-1234-XY)               AB 1234 XY
Mask you enter           (##-####-##)               ## #### ##
SETUP displays           (##~-####~-##)             ##~ ####~ ##
```

You can refer to Chapter 7, Printing and Displaying Data for more information on print masks.

The **SCREEN MASK** is used by applications to enter data into this field.

A program can ask for the screen mask and use it in whatever manner that is desired. The format should be the same as for the print mask.

HELP TEXT can be entered for this data field. The help text can be displayed during data entry if the user types "HELP".

Custom applications can also display this help text.

The help text:

- can be up to 60 characters long.
- can consist of any characters on the keyboard.

Enter the **ACCESS RULES** for this data field. The access rules determine who will have access to this field. **NORMAL** allows users to read, add and change data field information.

Access rules have the following formats:

```
READ:read_access, WRITE:write_access

NORMAL
```

where *read_access* and *write_access* are each represented by a single letter from (A-Z).

NORMAL denotes READ:N, WRITE:N.

A is the most restricted access level. Z is the least restricted access level.

The access_rules are used in conjunction with the structure security level to determine whether or not a user can access the data field.

When a user attempts to access a data field, the structure security level is compared to the read or write access level defined for the data field. If the data field's access level is lower than the structure security level, access is allowed. For example:

If the structure READ access is J and the field READ access is T, the field data CAN be read. However, if the structure WRITE access is J and the field WRITE access is E, the field CANNOT be written to or changed.

You can enter **VALIDATION RULES** for this field. The validation rules will be used to validate data entered by a data entry program.

The data entry program can access these validation rules to make sure that the data entered is valid prior to storing the data into the data field.

To add or change the validation rules, click on `[CHANGE]` beside the **Validation Rules** field. The **VALIDATION RULES** window will open.

In the example above, the field has been set with validations that it is a REQUIRED field and can contain only LETTERS.

The validation options are:

```
LETTERS         UCASE         LCASE          REQUIRED
YES/NO          INTEGER       INTEGERWORD    DIGITS
NUMBER          FULLTIME      VRULES         PRINTMASK
EXPRESSION      CODE          MENU           ROUTINE
DECIMALS        LENGTH        CHARACTERS     NOCHARACTERS
ALLOW           DISALLOW      PATTERN        FILTER
DATE            MINLENGTH     MAXLENGTH
```

When you have completed your list of validation rules, click on `[OK]`.

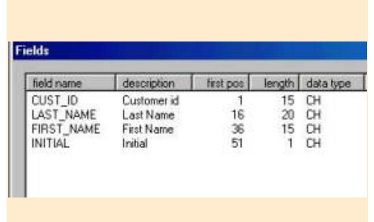
Refer to Section 6.6.6, VALID(text_str, rule_str) for detailed information on the **VALID function**.

Normally, when you define the record fields, each defined field represents a single piece of data in the record. This allows your programs access to all of the record data.

There are times when you want to process two or more adjoining fields. For example, if you have a last name field, first name field and initial field and you want to sort the records by name, you would sort the three separate fields--last name, first name and initial.

To simplify this process, you can define an additional data field, FULL_NAME, which can consist of the three fields. When you define the FULL_NAME field, the starting position would be the first position of the last name and the length would be the total number of characters in the three fields. Then, when you want to sort or print the full name, you need only refer to one field, the FULL_NAME field.

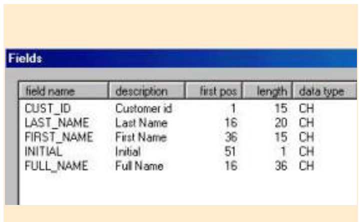
For example, if you have the following fields defined:



Fields

field name	description	first pos	length	data type
CUST_ID	Customer id	1	15	CH
LAST_NAME	Last Name	16	20	CH
FIRST_NAME	First Name	36	15	CH
INITIAL	Initial	51	1	CH

You would define the field FULL_NAME as in the example below:



Fields

field name	description	first pos	length	data type
CUST_ID	Customer id	1	15	CH
LAST_NAME	Last Name	16	20	CH
FIRST_NAME	First Name	36	15	CH
INITIAL	Initial	51	1	CH
FULL_NAME	Full Name	16	36	CH

Redefining fields becomes very useful when you need to set up unique key fields for ARS indexed files.

KEY FIELDS

If you are creating an ARS indexed file, you will be designating one or more data fields as **KEY FIELDS**. ARS indexed files require at least one key field.

When you use the CREATE procedure to create an ARS indexed file, you will be prompted for a primary key field and alternate key fields.

SheerPower supports keys of 4096 bytes in length.

Key fields contain unique data which enables programs to locate records very quickly. For example, the customer number field could be a key field because all the customer numbers in all the records would be different. This type of key field is called a **UNIQUE KEY FIELD**.

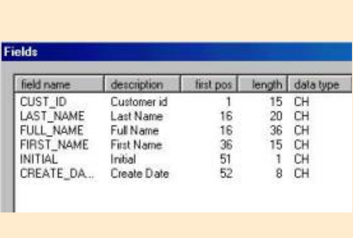
DUPLICATE KEY FIELDS contain unique types of data but the data might be found in more than one record in the file. Any defined or redefined field can be designated as a key field. However, it is best to use fields that normally will contain unique data as key fields. Common practice is to set up the first field as a key field.

A key field can be defined which consists of two or more fields. If the fields which make up this key field are not adjoining (i.e. one after the other), this is called a **SEGMENTED KEY FIELD**. Key fields

consisting of two or more fields, whether segmented or not, must have a length that equals the total length of all the segments that make up the key field.

SheerPower can extract records using a segmented key.

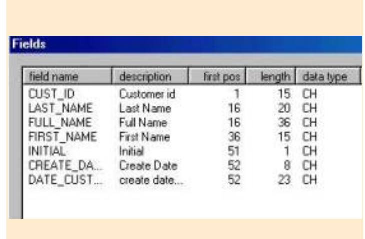
A record has the following field definitions:



Fields

field name	description	first pos	length	data type
CUST_ID	Customer id	1	15	CH
LAST_NAME	Last Name	16	20	CH
FULL_NAME	Full Name	16	36	CH
FIRST_NAME	First Name	36	15	CH
INITIAL	Initial	51	1	CH
CREATE_DA...	Create Date	52	8	CH

To define a **SEGMENTED KEY** that consists of CREATE_DATE plus CUSTOMER_ID, you would define another field, DATE_CUST_KEY, that starts at the first position of the first segment of the key with the length being the sum of the lengths of all of the segments:



Fields

field name	description	first pos	length	data type
CUST_ID	Customer id	1	15	CH
LAST_NAME	Last Name	16	20	CH
FULL_NAME	Full Name	16	36	CH
FIRST_NAME	First Name	36	15	CH
INITIAL	Initial	51	1	CH
CREATE_DA...	Create Date	52	8	CH
DATE_CUST...	create date...	52	23	CH

To extract using this key:

```
EXTRACT STRUCTURE customer,  FIELD date_cust_key: partial key "20020101"
        .
        .
 END EXTRACT
```

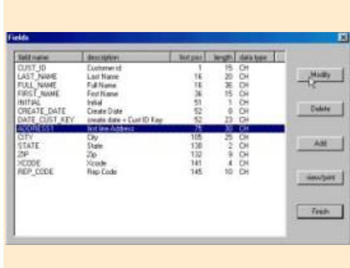
The above statement would extract all CUSTOMER records created on 1/1/2002.

To modify an existing data field in a structure definition, enter SETUP and select the structure to be modified. See Section 16.1.1, Entering SETUP for details on how to enter SETUP.

Click on `[Properties]`.

The currently defined data fields will be displayed.

Click on the data field that you want to modify . The `[MODIFY]` button will then become available.



When you click on `[MODIFY]` the Field Definition window will appear. All input fields, except name, can be modified. See Section 16.3.1, Data Field Definitions for details on the various data field information options.

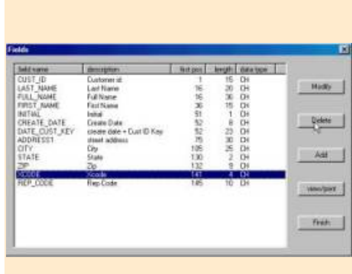
Caution must be taken if you modify the field starting position and field length as your new starting position and length may overlap other data in the record. Other fields may also need to be modified for correct starting positions and lengths.

If you need to delete an existng data field in a structure definition, enter SETUP and select the structure to be modified. See Section 16.1.1, Entering SETUP for details on how to enter SETUP.

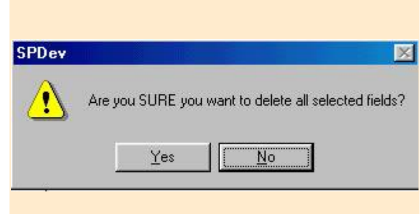
Click on `[Properties]`.

The currently defined data fields will be displayed.

Click on the data field that you want to delete and the `[Delete]` button will become available.



You will then be prompted to confirm the deletion of this data field:



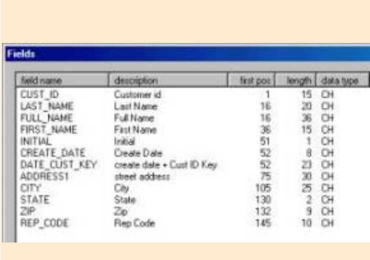
Choosing **NO** will return you to the Field Definition window.

Choosing **YES** will immediately delete the selected field.

Note

Deleting a data field contained in the file does not reposition following data fields.

In the example we deleted the field XCODE (see example above). We now have a record layout like this:



Fields

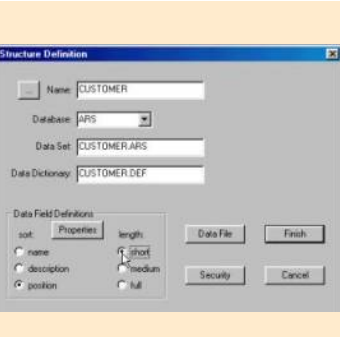
field name	description	first pos	length	data type
CUST_ID	Customer id	1	15	CH
LAST_NAME	Last Name	16	20	CH
FULL_NAME	Full Name	16	36	CH
FIRST_NAME	First Name	36	15	CH
INITIAL	Initial	51	1	CH
CREATE_DATE	Create Date	52	8	CH
DATE_CUST_KEY	create date + Cust ID Key	52	23	CH
ADDRESS1	street address	75	30	CH
CITY	City	105	25	CH
STATE	State	130	2	CH
ZIP	Zip	132	9	CH
REP_CODE	Rep Code	145	10	CH

Note that field ZIP starts at position 132 for a length of 9 and field REP_CODE starts at position 145 for a length of 10. There is a gap between the two fields where the deleted field had been.

When you need to view or print the record layout/definition you have several options.

Enter SETUP and select the structure to be viewed. See Section 16.3.1, Data Field Definitions for details on how to enter SETUP.

How the data fields will be presented to you is controlled by selections in the Structure Definition window, under the **Data Field Definitions** heading. You have the options to sort by name, description, position and the length of what is to be shown.



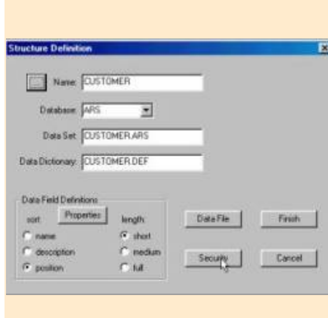
```
Sort By:        Description

Name            sorts by the field names

Description     sorts by the descriptions that have been entered
                for the fields

Position        sorts the fields in record layout by their starting
                position in the record

Length          Description

Short           Displays field name, description, first position, length, data type

Medium          Displays field name, description, first position, length, occurances,
                last position, data type, print mask, semantics

Full            Displays field name, description, first position, length, occurances,
                last position, data type, print mask, semantics, prompt,
                heading, help, screen mask, validations
```

Clicking on `[Properties]` will bring up the Fields window where you can see the report on screen.

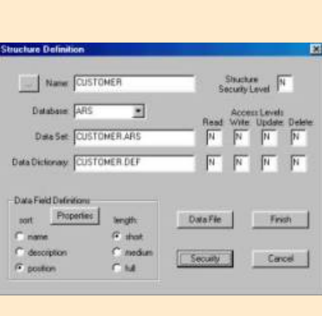
Clicking on `[View/Print]` in the Fields window will bring the field information report up in Notepad where you can then do normal Notepad functions, including printing.

Structure security controls access levels for read, write, update and delete activities.

Enter SETUP, select the structure (see 15.1.1 Entering SETUP and 15.2 SETUP Structure Definition) and click on `[Security]`.



The following information will then be displayed on the window:



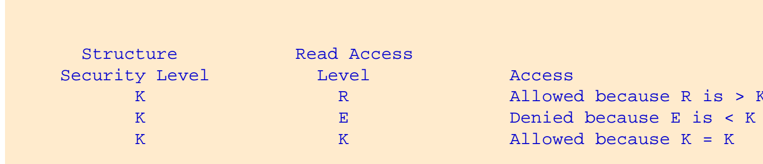
The security level is a single letter in the range of A to Z. Security level A is the highest level of security. Security level Z is the lowest level of security.

The default value for the security and access levels is ‘N’, normal. You may optionally assign a different security level to the structure and then for both the definition and the dataset you can assign various security levels for read, write, update and delete accesses.

Read/ Write/ Update/ Delete Access Levels

The access level is a single letter in the range A to Z. A is the most restricted access level. Z is the least restricted. When an attempt is made to read, write, update or delete a record from this structure, SheerPower will compare the structure security level to the corresponding access level. If the access level is lower than the structure security level, the request will not be allowed.

For example:

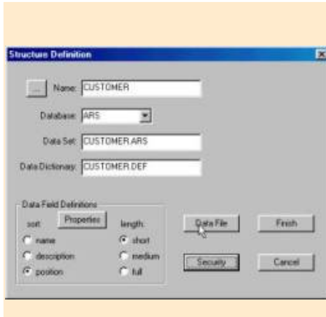


```
   Structure          Read Access
 Security Level         Level            Access
       K                  R              Allowed because R is > K
       K                  E              Denied because E is < K
       K                  K              Allowed because K = K
```

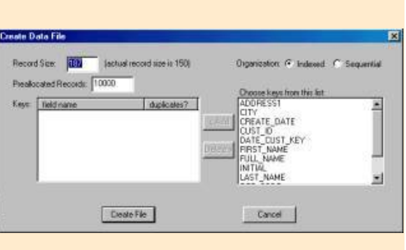
Clicking on `[Security]` again will remove the security information from the window.

After you have finished defining the record layout, you will need to create the data file and define it’s characteristics.

Enter SETUP , select the structure (see 15.1.1 Entering SETUP and 15.2 SETUP Structure Definition’) and click on `[Data File]`.



The following will be displayed:



The actual record size you have defined will be displayed. Sheerpower will automatically extend the record size for future use. The default record size is 25% larger than the actual length of the fields defined. You may accept the defaulted record size or modify it for additional characters. It is recommended that some additional character positions be set in the record size of new file definitions to allow for additional fields if they become necessary. This allows room for adding new fields to the record without having to expand the record size.

Sheerpower will default a value for the number of records preallocated for the file creation. This is not a limit to the number of records but for size of initial file creation. You may accept the defaulted value or modify as needed.

Select how you want your file to be organized. The options are:

```
INDEXED    - the data file record contains one or more key fields
SEQUENTIAL - the data file record contains no key fields
```

If you are going to be sorting and/or extracting data from the ARS file you are creating, the indexed file organization is more efficient than the sequential organization. This is especially true if you have large files.

An indexed file contains one or more indexed key fields which allow programs to locate and access file data very quickly without having to read through the file to locate records.

A sequential file has no key fields and accessing file data can be very slow and time consuming. Locating data in a sequential file requires reading every record in the file to find all occurrences of the data.

If you select Indexed file organization, you will need to enter the key field information.

The list of fields defined is shown on the right side of the window. Click on the field that should be a key field and then click `[Add]`.

You will be asked if duplicates should be allowed or not. If the data to be stored in the key field will be unique (only one occurrence of the data in the key field) then duplicates should not be allowed. If the data could occur many times (ie: a salesrep code) then allow duplicates (multiple occurrences of the data in the key field).

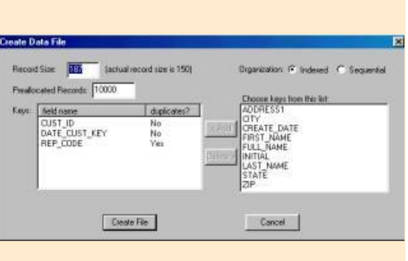
Data access works most efficiently with unique key fields.

If you want more than one key field, select the fields you want to use as key fields.

All key fields selected will be shown on the left side of the window.

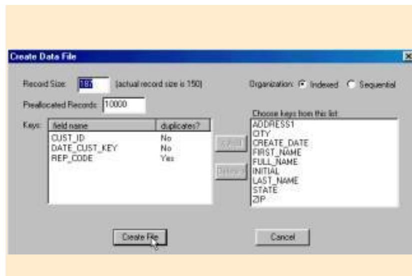
In the following example three (3) keys have been selected:

```
  CUST_ID              no duplicates
  DATE_CUST_KEY        no duplicates
  REP_CODE             duplicates
```



If you need to delete one of the fields selected as a key field, click on the field to highlight it, then click on `[Delete]`.

After you have defined all fields and the file organization click on `[Create File]`.



When the data file has been created, a notification window will appear.

Click on `[OK]`. The **Create Data File** window will close and bring you back to the **Field Definition** window. Clicking on `[FINISH]` will create the structure file if it is new. If you modified an existing structure, clicking on `[FINISH]` will save your modifications.

SheerPower 4GL supports **all ODBC compatible database engines** such as Microsoft ACCESS and Oracle. ODBC stands for "Open Database Connectivity". ODBC is a universal database interface used to access a wide range of databases.

To access ANY ODBC database, the following are required:

- The appropriate ODBC database driver.
- A data source file that associates that driver and database.

All of the normal database structure statements are supported when accessing an ODBC database. See Chapter 15, Data Structure Statements.

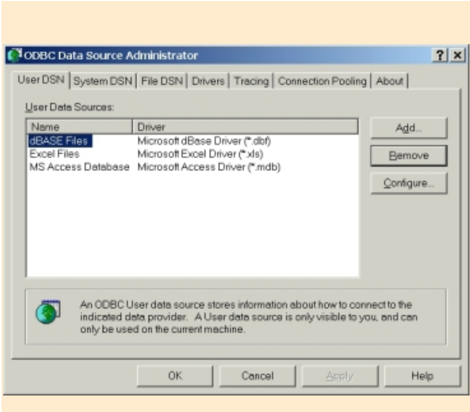
To use ODBC in SheerPower 4GL, using Windows, you must first setup an ODBC data source. This must be done for **every database** that you want SheerPower to have access to.

The following example will illustrate how to setup an ODBC data source where the source is a Microsoft Access database. This example uses the "Contacts" table in a "MyContacts.mdb" sample database found inside the SheerPower\samples folder.

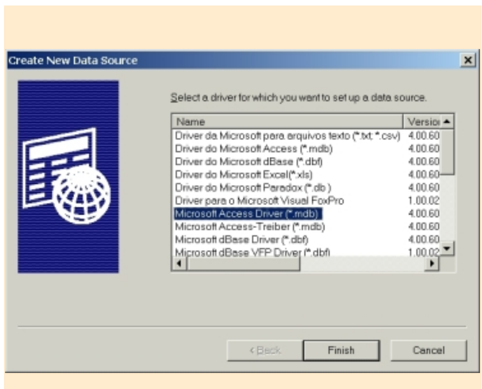
On Windows XP and Windows 2000, click on the **Start** menu button, then choose **Settings**, then go into the **Control Panel**.

Next, double-click on the **Administrative Tools** icon inside the Control Panel window.

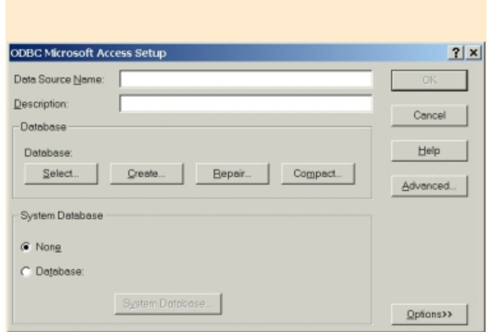
Then double-click on the **Data Sources (ODBC)** shortcut inside the Administrative Tools window. The **ODBC Data Source Administrator** window will appear:



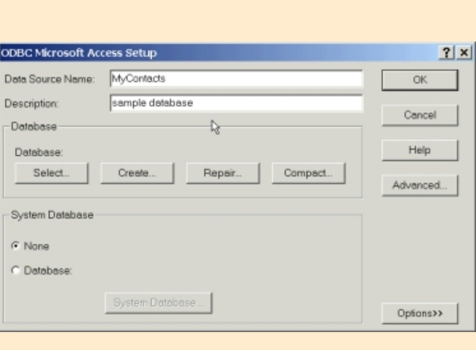
Inside the User DSN tab, click on the **Add** button on the right. The **Create New Data Source** window will appear:



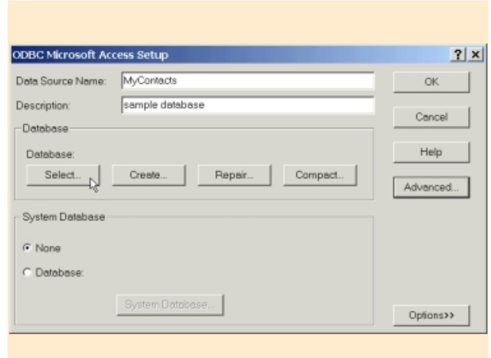
Select the **Microsoft Access Driver (*.mdb)** driver from the data source list, then click on the **Finish** button. The **ODBC Microsoft Access Setup** window will appear:



Inside the **Data Source Name** field, type in the name of the data source you are using. In this example, the name is **MyContacts**. You can then enter a description for the database inside the **Description** field:

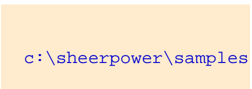


Next, click on the **Select** button to select the database you want SheerPower to access.



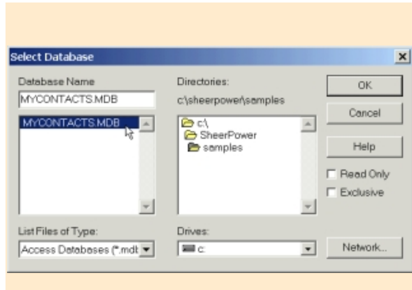
The **Select Database** window will appear.

For this example, we will select the **MYCONTACTS.MDB** sample database inside the SheerPower **Samples** folder. The default location is:



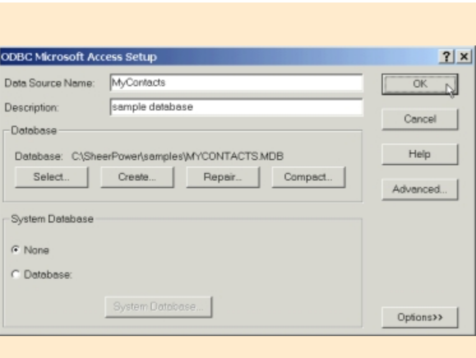
```
c:\sheerpower\samples
```

Use the **Directories** window on the right to browse for this database. Once the SheerPower Samples folder is open, **MyContacts.mdb** will appear in the **Database Name** window on the left:

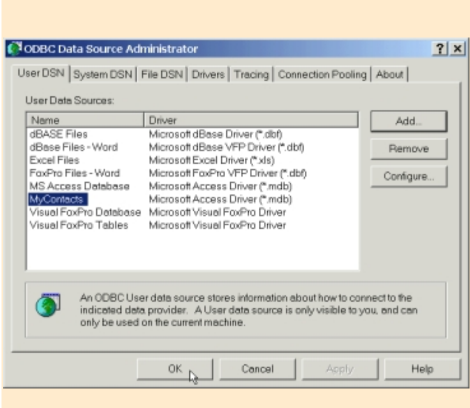


Highlight **MyContacts.mdb**, then click on the **OK** button.

Inside the **ODBC Microsoft Access Setup** window, click on the **OK** button:



The **MyContacts** database will now appear in the list of User Data Sources inside the ODBC Data Source Administrator window. Click on the **OK** button to close out the window:



SheerPower will now be able to access this database.

To access an ODBC database with SheerPower, you just specify the **table** to access inside the **database** in the **OPEN STRUCTURE statement** of the source code.

For example, to access the **Contacts** table inside the **MyContacts.mdb** database:

```
open structure con: name 'contacts in mycontacts'
extract structure con
  sort ascending by con(LastName)
end extract
print 'Contact List'
print
for each con
  print con(FirstName); ' '; con(LastName), con(MobilePhone), con(EmailName)
next con
close structure con
end


Contact List

Steven Buchanan     (206) 555-1856     StevenB@hfs.com
Nancy Davolio       (425) 555-9811     nancy@anywhere.com
Andrew Fuller       (206) 555-6666     andrewf@anywhere.com
```

```
Janet Leverling     (206) 555-7777      janetl@anywhere.com
Margaret Peacock    (206) 555-5555      margiep@anywhere.com
```

If the ODBC database you are using requires a username and password to access it, the following format is used in SheerPower:

```
?user=xxx&password=yyy
```

The code could look like the following:

```
dbuser$ = 'dbadmin'
dbpass$ = 'dbpassword'
mycontacts$ = 'Contact in MyContact?user=' + dbuser$ + '&password=' + dbpass$
open structure d: name mycontacts$
```

SheerPower Internet Services (SPINS) Webserver comes bundled with SheerPower 4GL. The SPINS webserver allows anyone to make web-based applications, even on their own local computers not connected to the Internet, without having to purchase a webserver license. And, of course, the SPINS webserver is EASY to install and use!

For more on writing network and web-based applications with SheerPower 4GL, see Section 19.3, Webserver CGI Interface.

By default, the SPINS webserver (spins_webserver.exe) is installed to the following directory:

```
\sheerpower\sphandlers\
```

This will assume that the root folder is:

```
\sheerpower\sphandlers\wwwroot\
```

SPINS_webserver expects the directory structure to be:

```
[wherever SPINS_webserver.exe is]
          |
       [wwwroot]--> things like INDEX.HTML
        |      |
   [images] [scripts]
```

If you are replacing IIS with the SPINS_webserver, you would do the following inside the Command Prompt program:

```
c:>  \sheerpower\sphandlers\spins_webserver.exe -wwwroot "c:\inetpub\wwwroot"
```

The **-wwwroot** option tells the SPINS webserver where the root folder is.

To run SPINS webserver, the Microsoft IIS webserver must be stopped or a different port specified for either webserver. Before running SPINS webserver for the first time:

- Stop Microsoft IIS webserver:

Start --> Settings --> Control Panel --> Administrative Tools --> Internet Information Services

Right-click on your Web Site from the list on the left (or Default Web Site) and choose **Stop**

The web site name will now display in the list with a red circle with a white "x" through it to show that it's stopped.

- Delete the old **/sheerpower/sphandlers/spiis.gblpool** file.
- Open the Command Prompt program and type in the following command:

  c:> **iisreset**

  The Command Prompt window will display:

```
Attempting stop...
Internet services successfully stopped
Attempting start...
Internet services successfully restarted
```

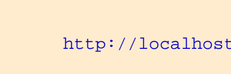
  But the IIS webserver will remain stopped.

- Run SPINS_webserver by double-clicking on \sheerpower\sphandlers\spins_webserver.exe. Currently SPINS runs in a Command Prompt window.

To specify a different port number for SPINS to use, see Section 18.1.3, Specify a Different Port Number.

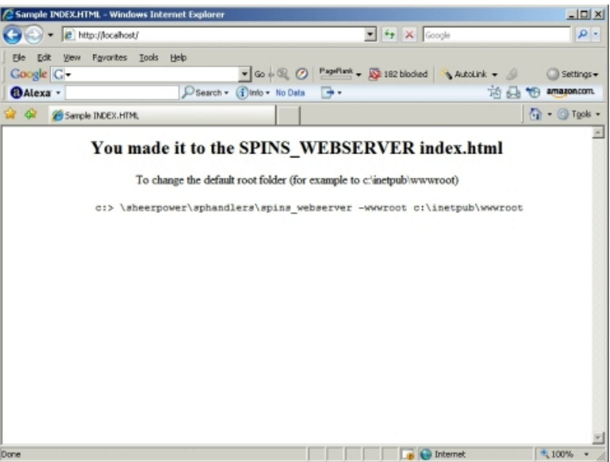
To test SPINS webserver:

- Make sure SPINS_webserver.exe is running (default location is \sheerpower\sphandlers\spins_webserver.exe).
- Open a browser window and go to:



This will open up the .HTML file located in:

```
sheerpower/sphandlers/wwwroot/index.html
```



To tell the SPINS webserver to use a different port number from the default port 80, start it from the **Command Prompt** program or with a **PASS NOWAIT** statement within your program using the following syntax:

```
spins_webserver -port nn
```

For example, to change from the default port 80 to port 8080:

```
spins_webserver -port 8080
```

to use **port 8080**.

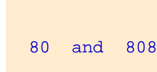
Running SPINS and IIS Simultaneously

If you need to utilize some of the IIS facility, you can set the IIS webserver to use port 8080 and the SPINS websever to use port 80 (or the other way around). Both webservers can co-exist this way.

To run web-based applications on multiple ports, the following syntax is used:

```
spins_webserver -port 80 -wwwroot c:\root1 -port 8080 -wwwroot c:\root2
```

Each **-port nn** lets you specify a port. In the above example, we are listening on TWO ports:



```
80  and  8080
```

You can tell the SPINS webserver to use a specific root folder by performing the following command inside the Command Prompt program:

```
spins_handler -wwwroot "c:\inetpub\wwwroot"
```

Another example would be:

```
SPINS_webserver -wwwroot "c:\myplace\stuff\"
```

This would mean than an **INDEX.HTML** file would be in:

```
c:\myplace\stuff\index.html
```

For a list of options (like specifying the wwwroot folder), type the following command inside the Command Prompt program:

```
C:\sheerpower\sphandlers>  spins_webserver -?
```

The command will return the following options (subject to change):

```
C:\Documents and Settings\User>c:\sheerpower\sphandlers\spins_webserver -?
SheerPower InterNet Services Web Server SPINS_WEBSERVER V00.00.060
Copyright (c) 2005 Touch Technologies, Inc. - Sat, 17 Sep 2005 19:05:23

spins_webserver [-option value] [-nextoption value] ...
   -? or -help                FOR this display
   -ipaddr xxx.xxx.xxx.xxx    TO specify AP address to listen on
   -port  ##                  TO specify port to listen on
   -https ##                  TO specify port to listen on
   -cafile filespec           FOR Certificate Authority File
   -certfile filespec         FOR Certificate File
   -keyfile filespec          FOR Private Key File (default = certfile)
   -keypass password          FOR Private Key File Password
   -recvbuf_def ###           TO specify receive buffersize
   -recvbuf_max ###           TO specify MAXIMUM receive buffersize
   -sendbuf_def ###           TO specify send buffersize
   -threads ##                TO specify number of threads to use
   -virtual hostname pathname TO specify a virtual host and wwwroot
   -wwwroot pathname          TO specify the wwwroot directory
   -flags [NOEXCLUSIVE][:DISPLAY_GET][:DISPLAY_PUT]
          [:DISPLAY_RECV][:STATUS_CONN][:SSL_ERRORS]
          [:DEBUG][:MISC_DEBUG]
```

-port ## and -https

Any parameters **before** a -port or -https will become the **global defaults**.

Those parameters **after** the -port or -https will be **specific** to that port.

SheerPower 4GL includes a number of network-based extensions for accessing data from webpages and sending email.

FORMAT:

```
open ch #num: name 'http://url'
```

EXAMPLE:

```
open #1: name 'http://www.ttinet.com/sheerpower/sample.txt'
for count = 1 to 100
  line input #1, eof eof?: rec$
  if eof? then exit for
  print count; tab(10); rec$
next count
close #1
end
```

PURPOSE:

To access raw HTML data from anywhere across the Internet.

DESCRIPTION:

The **html:// file open** option allows programs complete access to raw HTML data. This data can then be used to do things like get stock quotes, read news headlines, and fetch data from SheerPower handlers. For more see Section 19.3, Webserver CGI Interface.

FORMAT:

```
  email$ = 'mailto://'       + sendto$       +
           '?subject='       + subject$      +
           '&mailfrom='      + mailfrom$     +
           '&replyto='       + replyto$      +
           '&friendlyname='  + friendlyname$ +
           '&cc='            + cc$           +
           '&server='        + servername$   +
           '&username='      + username$     +
           '&password='      + password$     +
           '&mime_type=html'

  open file email_ch: name email$, access output
  print #email_ch: 'Text of email.'
  print #email_ch: 'More text of email.'
  print #email_ch: 'End of email.'
  close #email_ch

Where:

  sendto$        = email recipient
  subject$       = subject line
  mailfrom$      = who this email will claim it is from
  replyto$       = the "reply" email address if different from the "mailfrom$"
address
  friendlyname$  = the display name seen by the recipient (i.e. From: "Support"
<support@ttinet.com>)
  cc$            = recipient to be copied on the email
  servername$    = SMTP server that handles outgoing emails
  username$      = SMTP server username (if outgoing authentication is required)
  password$      = SMTP server password (if outgoing authentication is required)
  mime_type=html = Defines the mime type of the email to allow sending HTML formatted
emails
```

EXAMPLE:

```
  mailform$ = '<form>'
  mailform$ = mailform$ + 'From: <input type=text name=from ' +
    'value="Sender email address"><br><br>'
  mailform$ = mailform$ + 'Reply To: <input type=text name=reply ' +
    'value="Reply To email address"><br><br>'
  mailform$ = mailform$ + 'Friendly Name: <input type=text name=display ' +
    'value="Friendly (display) Name"><br><br>'
  mailform$ = mailform$ + 'To: <input type=text name=to ' +
    'value="Recipient email address"><br><br>'
  mailform$ = mailform$ + 'Server: <input type=text name=server ' +
    'value="SMTP server name"><br><br>'
  mailform$ = mailform$ + 'Subject: <input type=text name=subject ' +
   'value="Subject line"><br><br>'
  mailform$ = mailform$ + 'Text: <br><textarea name=body rows=5 cols=60>'
  mailform$ = mailform$ + 'Type in your text here.</textarea><br><br>'
  mailform$ = mailform$ + '<input type=submit name=submit value="Send Email">'
  mailform$ = mailform$ + '<input type=submit name=exit value="Cancel">'
  mailform$ = mailform$ + '</form>'
  line input dialogbox mailform$: data$

  for item = 1 to pieces(data$, chr$(26))
    z0$    = piece$(data$, item, chr$(26))
    name$  = element$(z0$, 1, '=')
    value$ = element$(z0$, 2, '=')

  select case name$
    case 'from'
      mailfrom$ = value$
    case 'reply'
      replyto$ = value$
    case 'display'
      friendlyname$ = value$
    case 'to'
      sendto$ = value$
    case 'server'
      servername$ = value$
    case 'body'
      text$ = value$
    case 'subject'
      subject$ = value$
    case else
  end select
  next item

  email$ = 'mailto://' + sendto$ + '?subject=' + subject$ +
           '&mailfrom=' + mailfrom$ + '&replyto=' + replyto$ +
           '&friendlyname=' + friendlyname$ + '&server=' + servername$ +
           '&wait'

  message 'Sending...'
  open #1: name email$, access output
    print #1: text$
  close #1
  message 'Sent!'
end
```

PURPOSE:

To send emails.

DESCRIPTION:

When sending email, a number of optional parameters are supported.

Table 19-1 Sending Email - Optional Parameters Supported

	cc	to copy another recipient to receive the email. To cc more than 1 recipient, use multiple CC options: &cc=someone@somewhere.com&cc=someone_else@somewhere.com
	emailfrom	default to the email address in Outlook Express or the value of sheerpower$emailfrom
	replyto	to specify a "reply to" email address that is different from the emailfrom email address
	friendlyname	the name that is displayed to the recipient beside your email address
	server	default to the SMTP server used by Outlook Express or the value of the logical sheerpower$server
	wait	wait for email send to be completed
	nowait	queue the email for sending as time permits
	timeout	default is 30 second timeout. If we cannot reach the email server in this amount of time, abort the email sending.
	username and password	when the SMTP server requires authentication, use these parameters to enter the username and password.
	attach=file_to_attach.xxx	attach a file to this email. To attach more than one file, use multiple ATTACH options: attach=file1.xxx&attach=file2.xxx
	mime_type=html	specifies the email can be created in HTML format. If this parameter is not defined, the default of plain text email will be used.

Note

The start of the parameter list begins with a **?** (question mark), and each parameter is separated by an *&* (ampersand).

Emails are queued for sending when the file is closed. If WAIT was specified, then SheerPower waits until the email is delivered before continuing processing. The default is NOWAIT.

For high reliability on sending emails with SheerPower 4GL, the SMTP server should be either on the same server as SheerPower or at least on the same LAN. This is because the SheerPower 4GL email handler is not a full email system that endlessly tries to send out emails even if the application has terminated.

The best use for the SheerPower email facility is:

SheerPower --> SMTP server (local one) --> Internet for delivery

This way there will be very few delays and reliability will be highest.

Preparation

SheerPower applications can easily be "web-enabled" through its simple CGI Interface. The CGI interface works with SheerPower's own webserver program, **SPINS webserver**. For more information on SPINS webserver, see Chapter 18, SheerPower Internet Services (SPINS) Webserver.

The sample CGI program **eval_handler.spsrc** is located in:

```
c:\sheerpower\sphandler
```

The webpage instructions that go along with the sample program is located in the same folder, and is called **cgi.html**.

To use the CGI interface in SheerPower, **SPINS Webserver must be running**. If the Microsoft IIS webserver is running, it needs to be stopped, or ports configured to run both webservers at the same time. Instructions are found here - Section 18.1.3, Specify a Different Port Number.

To continue with this test, double-click on the EVAL_HANDLER.SPSRC file. This runs the EVAL_HANDLER program. This program file is located in:

c:\sheerpower\sphandlers\eval_handler.spsrc

After the EVAL_HANDLER has been started, you can try the FORM below.

After entering an expression, press the **ENTER key**. To get back to this webpage, click on the browser's **BACK button**.

Enter an expression, like **sin(355/113)** then press the **ENTER key**:

If the form does not work..

If this form did not work correctly for you, see Appendix L, Troubleshooting the CGI Interface.

How this form works

Each time anyone enters an expression to evaluate (like 2+3), their browser sends the data to your SPINS webserver along with a handler name. In this example, the handler name is EVAL. The SPINS websever then passes the data to the EVAL_HANDLER.SPSRC SheerPower program. The EVAL_HANDLER handles the request, figures out the result, and sends the result back to the SPINS server. The SPINS server then sends the result back to the browser.

Important Note on the Handler Name

The **handler name** defined in the source code MUST be defined in **UPPER CASE**.

In order to handle CGI requests, the following steps need to be taken:

- Open the CGI connection to the SPINS webserver
- Wait for a request
- Handle timeouts
- Process the request
- Open our CGI connection

First we open our CGI connection to the SPINS server.

```
handler_name$ = 'cgi://EVAL'
open file cgi_ch: name handler_name$
```

In this example, EVAL is the HANDLER NAME.

Note that the **handler name** defined in the source code MUST be defined in **UPPER CASE**.

For higher performance, you can run as many copies of this handler as you wish. The SheerPower CGI interface will queue all requests to these handlers that have the form:

```
http://www.ttinet.com/scripts/spiis.dll/EVAL
```

Next we set up a logic loop to wait for requests, handle timeouts, and process each request.

```
do
  line input #cgi_ch: method$
  if method$ = '' then repeat do
  ..
  ..
loop
```

The LINE INPUT waits for a request from the SPINS server. When the LINE INPUT completes, the variable METHOD$ will contain one of three values.

- A null string (""), meaning that there was no request from the SPINS server for at least five seconds.
- The string "POST" if a form used the POST method.
- The string "GET" if a form or URL used the "GET" method.

In this program, when there is no request from the SPINS server (a timeout), we just go back and try again. In complex applications a program might instead unlock databases, write out statistics, and then go back and try again.

Now that we have a request from the SPINS server, we have to process the request.

```
ask #cgi_ch, symbol 'EXPR': value expr$
if  expr$ = '' then
  print #cgi_ch: '<h2>Thank you for using the Evaluator!</h2>'
  repeat do
end if

when exception in
  answer = eval(expr$)
use
  answer = extext$
end when
if dtype(answer) <> 1 then answer = str$(answer)
print #cgi_ch: '<h2>'; expr$; ' --> '; answer; '</h2>'

ask #cgi_ch, symbol 'env:REMOTE_ADDR': value ipaddr$
print #cgi_ch: '(Your IP address is '; ipaddr$; ')'
```

This form returns one form variable---the expression to be evaluated (EXPR). We use the SheerPower ASK instruction to ask for its value.

```
ask #cgi_ch, symbol 'EXPR': value expr$
```

If they didn't enter any expression, we just tell them "Thank you...". This is done using the PRINT instruction.

print #cgi_ch: '<h2>Thank you for using the Evaluator!</h2>'

Using EXPR$, we calculate the ANSWER and PRINT the result back to the SPINS server.

print #cgi_ch: '<h2>'; expr$; ' --> '; answer; '</h2>'

Finally, we ask the SPINS server for the "REMOTE_ADDR". This is the IP address of the requestor. Since "REMOTE_ADDR" is an environmental variable, we must put "env:" in front of the symbol name. Once we get the IP address, we PRINT it to the SPINS server---which, in turn, sends the data back to the browser.

```
ask #cgi_ch, symbol 'env:REMOTE_ADDR': value ipaddr$
print #cgi_ch: '(Your IP address is '; ipaddr$; ')'
```

By default, SheerPower automatically builds the CGI required HTTP headers. But, sometimes there is a need to write out your own custom HTTP headers. For example, you might need to write out binary data, or perhaps send a cookie. Here is a sample routine that writes out a HTTP header:

```
  routine start_http_output
    ask #cgi_ch, symbol 'env:PATH_INFO': value path_info$
    print #cgi_ch: "HTTP/1.1 200 OK"
    print #cgi_ch: 'Cache-Control: max-age=2, must-revalidate'
    print #cgi_ch: "Content-type: text/html"
    print #cgi_ch: 'Referer: ' + path_info$
    print #cgi_ch: "Pragma: no-cache"
    print #cgi_ch:
    print #cgi_ch:
  end routine
```

Note

When printing out binary data, include a trailing semi-colon at the end of the PRINT instruction:

print #cgi_ch: mybinary$**;** // notice the trailing semi-colon

The SheerPower CGI interface was designed for very high performance. If the server it is running on has multiple CPUs, full advantage of the CPUs can be taken by running multiple copies of the same HANDLER. A good "rule of thumb" is to run at least two HANDLERS for each CPU on the server. Idle handlers consume very little cpu-time---less than 1/10th of 1% of available CPU-time for each idle handler.

FORMAT:

```
ASK #cgi_ch, SYMBOL 'ENV:environment_variable : VALUE x$
```

EXAMPLE:

```
// This sample program can be found in
// SheerPower/sample/sphandlers/eval_handler.spsrc
// It is also the sample program used to illustrate
// and test the CGI Interface in SheerPower, also
// in this chapter (Webserver CGI Interface)

declare object answer
handler_name$ = 'cgi://EVAL'

print 'Eval Handler started '; date$(days(date$), 4);' at '; time$

open file cgi_ch: name handler_name$
do
  line input #cgi_ch: method$
  if method$ = '' then repeat do
  ask #cgi_ch, symbol 'EXPR': value expr$
  if  expr$ = '' then
    print #cgi_ch: '<h2>Thank you for using the Evaluator!</h2>'
    repeat do
  end if

  when exception in
    answer = eval(expr$)
  use
    answer = extext$
  end when
  if dtype(answer) <> 1 then answer = str$(answer)
  print #cgi_ch: '<h2>'; expr$; ' --> '; answer; '</h2>'

  ask #cgi_ch, symbol 'env:REMOTE_ADDR': value ipaddr$
  print #cgi_ch: '(Your IP address is '; ipaddr$; ')'

loop
close #cgi_ch
stop

end
```

PURPOSE:

The SheerPower CGI interface allows full access to the SPINS webserver "environment variables". For example, to ask for the QUERY_STRING (the data that follows a "?" in a URL) you would use:

```
ask #cgi_ch, symbol 'env:QUERY_STRING': value qstring$
```

The "env:" symbol prefix tells SheerPower that you are requesting an environment variable and not a form variable.

To PARSE the QUERY_STRING, you would use:

```
ask #nn, symbol 'query:VARNAME': value vvalue$
```

This parses the QUERY_STRING (the data after the ?) for the value of the given VARNAME.

DESCRIPTION:

Table 19-2 CGI Environment Variables

ALL_HTTP	Retrieves all HTTP headers that were received. These variables are of the form HTTP_header field name. The headers consist of a null-terminated string with the individual headers separated by line feeds.
ALL_RAW	Retrieves all headers in raw form. The header names and values appear as the client sends them. Currently, proxy servers and other similar applications primarily use this value.
APPL_MD_PATH	Retrieves the metabase path of the application for the ISAPI DLL or the script.
AUTH_PASSWORD	Specifies the value entered in the client's authentication dialog. This variable is only available if Basic authentication is used.
AUTH_TYPE	Specifies the type of authentication used. If the string is empty, then no authentication is used. Possible values are Kerberos, user, SSL/PCT, Basic, and integrated Windows authentication.
AUTH_USER	Specifies the value entered in the client's authentication dialog box.
CERT_COOKIE	Specifies a unique ID for a client certificate. Returned as a string. Can be used as a signature for the whole client certificate.
CERT_FLAGS	If bit0 is set to 1, a client certificate is present. If bit1 is set to 1, the certification authority (CA) of the client certificate is invalid (that is, it is not on this server's list of recognized CAs).
CERT_ISSUER	Specifies the issuer field of the client certificate. For example, the following codes might be O=MS, OU=IAS, CN=user name, C=USA, and so on.
CERT_KEYSIZE	Specifies the number of bits in the Secure Sockets Layer (SSL) connection key size.
CERT_SECRETKEYSIZE	Specifies the number of bits in the server certificate private key.
CERT_SERIALNUMBER	Specifies the serial-number field of the client certificate.
CERT_SERVER_ISSUER	Specifies the issuer field of the server certificate.
CERT_SERVER_SUBJECT	Specifies the subject field of the server certificate.
CERT_SUBJECT	Specifies the subject field of the client certificate.
CONTENT_LENGTH	Specifies the number of bytes of data that the script or extension can expect to receive from the client. This total does not include headers.
CONTENT_TYPE	Specifies the content type of the information supplied in the body of a POST request.
LOGON_USER	The Windows account that the user is logged into.
HTTP_ACCEPT	A variable used to advise the server what types of content your browser can handle.
HTTP_ACCEPT_ENCODING	Defines the type of encoding that may be carried out on content returned to the client.
HTTP_ACCEPT_LANGUAGE	Defines the language or locale to use for: the UI, formatting or collation preferences, about which to provide content or information.
HTTP_CONNECTION	Classes used to manage a HTTP connection to a webserver.
HTTP_COOKIE	A client variable that stores a cookie on the server passed by a HTTP header.
HTTP_HOST	Displays the current domain name.
HTTP_REFERER	A meta-variable used to identify the page where the URL obtained was requested (i.e. the source of the link that was followed).
HTTP_USER_AGENT	This is used to get the user agent string, such as browser name and version , bot, or other program. The syntax is: ask #cgi_ch, symbol 'env:HTTP_USER_AGENT : value agent$
HTTPS	Returns on if the request came in through secure channel (with SSL encryption), or off if the request is for an unsecure channel.
HTTPS_KEYSIZE	Specifies the number of bits in the SSL connection key size.
HTTPS_SECRETKEYSIZE	Specifies the number of bits in server certificate private key.
HTTPS_SERVER_ISSUER	Specifies the issuer field of the server certificate.
HTTPS_SERVER_SUBJECT	Specifies the subject field of the server certificate.
INSTANCE_ID	Specifies the ID for the server instance in textual format. If the instance ID is 1, it appears as a string. This value can be used to retrieve the ID of the Web-server instance, in the metabase, to which the request belongs.
INSTANCE_META_PATH	Specifies the metabase path for the instance to which the request belongs.
PATH_INFO	Specifies the additional path information, as given by the client. This consists of the trailing part of the URL after the script or ISAPI DLL name, but before the query string, if any.
PATH_TRANSLATED	Specifies this is the value of PATH_INFO, but with any virtual path expanded into a directory specification.
QUERY_STRING	Specifies the information that follows the first question mark in the URL that referenced this script.
REMOTE_ADDR	Specifies the IP address of the client or agent of the client (for example gateway, proxy, or firewall) that sent the request.

REMOTE_HOST	Specifies the host name of the client or agent of the client (for example, gateway, proxy or firewall) that sent the request if reverse DNS is enabled. Otherwise, this value is set to the IP address specified by REMOTE_ADDR.
REMOTE_USER	Specifies the user name supplied by the client and authenticated by the server. This comes back as an empty string when the user is anonymous.
REQUEST_METHOD	Specifies the HTTP request method verb.
SCRIPT_NAME	Specifies the name of the script program being executed.
SERVER_NAME	Specifies the server's host name, or IP address, as it should appear in self-referencing URLs.
SERVER_PORT	Specifies the TCP/IP port on which the request was received.
SERVER_PORT_SECURE	Specifies a string of either 0 or 1. If the request is being handled on the secure port, then this will be 1. Otherwise, it will be 0.
SERVER_PROTOCOL	Specifies the name and version of the information retrieval protocol relating to this request.
SERVER_SOFTWARE	Specifies the name and version of the Web server under which the ISAPI extension DLL program is running.
URL	Specifies the base portion of the URL. Parameter values will not be included. The value is determined when SPINS parses the URL from the header.

TCP/IP and UDP protocols are specific methods used to send data back and forth from a server to a client computer.

The difference between TCP/IP and UDP is:

- **TCP/IP** is a *handshake* protocol
- **UDP** is a *no-handshake* protocol.

With **TCP/IP**, the **handshake** means that if one machine sends another some data, the receiving machine sends back a data packet that tells the sending machine the data was received. If the receiving machine requests that the sending machine then confirm that their confirmation packet was received, the sending machine sends a data packet back, and so on.

If the receiving machine receives garbled or otherwise messed up data, it will automatically re-request the data from the sending machine.

UDP is not a handshake protocol... there is no guarantee of data delivery. The data gets sent, and that is it. The receiving machine may or may not successfully receive the data, but the sender doesn't get any confirmation. The programmer has to create the rules for UDP protocol to get responses back from the recipient machine.

FORMAT:

```
  open file tcp_ch: name
'tcp://[ip]?[param1=value1]&[param2=value2]&[paramn=valuen]', access outin
```

EXAMPLE:

```
// Very simple TELNET client
line input 'IP address of the telnet server': ip$

open file tcp_ch: name 'tcp://' + ip$ + '?port=23', access outin
print 'Ready for data...'

do
  // check for data from the server
  // but not forever... just a few iterations
  for idx = 1 to 100
    line input #tcp_ch: ip$
    if ip$ = '' then exit for
    ask #tcp_ch, symbol 'data': value rec$
    print rec$;
  next idx

  // check for data from this client
  when exception in
    key input timeout 1, prompt '': dat$
    print #tcp_ch: dat$
  use
  end when
```

```
loop
end
```

PURPOSE:

TCP/IP allows the programmer to transmit data back and forth from one computer to another with the capability of receiving responses and data transmission confirmation.

DESCRIPTION

If an **IP address** is specified, then it denotes the receive source IP and the send destination IP for a **client**.

If no IP address is specified, the channel is a **server**.

The start of the options list begins with a ? (question mark), and each parameter is separated by an & (ampersand). The options are **NAMED** as outlined in the table below. For example, if you wanted to specify "port 23", you would use:

```
  open file tcp_ch: name 'tcp://' + '?port=23', access outin
```

Table 19-3 TCP/IP Protocol Parameters

Parameter	Description	Default
Sockets	specifies the number of sockets to use	100
port	specifies the port number to be used for transmitting data	31111
ipaddress	the IP address of the CLIENT machine receiving data	none
max_receive_queue	specifies the max number of requests able to be received at once	200
max_send_queue	specifies the max number of requests allowed in the send queue	200
maxdupconnections	specifies the maximum duplicate connections allowed	3
timeout	specifies the length of timeout in seconds	50

FORMAT:

```
  open file udp_ch: name
'udp://[ip]?[param1=value1]&[param2=value2]&[paramn=valuen]', access outin
```

DESCRIPTION

If an **IP Address** is specified, then it denotes the receive source IP and the send destination IP for a **client**.

If no IP address is specified, the channel is a **server**.

The start of the options list begins with a ? (question mark), and each parameter is separated by an & (ampersand). The options are **NAMED** as outlined in the table below. For example, if you wanted to specify a "timeout of 20 seconds", you would use:

```
  open file tcp_ch: name 'tcp://' + '?timeout=20', access outin
```

Table 19-4 UDP Protocol Parameters

Parameter	Description	Default
Sockets	specifies the number of sockets to use	100
port	specifies the port number to be used for transmitting data	31111

ipaddress	the IP address of the CLIENT machine receiving data	none
max_receive_queue	specifies the max number of requests able to be received at once	200
max_send_queue	specifies the max number of requests allowed in the send queue	200
maxdupconnections	specifies the maximum duplicate connections allowed	3
timeout	specifies the lenght of timeout in seconds	50

FORMAT:

```
open #n: name "com://port=p?options", access outin
```

EXAMPLE:

Not a "runnable example"

The example below is not a 'runnable' example. It is intended only to illustrate how to use the communication support in SheerPower 4GL.

```
// use com4 port for talking to some device
open file com_ch: name 'com://port=4', access outin
print #com_ch: 'AT'
line input #com_ch: cdata$
if cdata$ <> '' then print 'Response was: '; cdata$
end
```

PURPOSE:

Communication (COM) Port Support is available to access the serial communication ports on a computer.

DESCRIPTION

The **port number** *p* is required. i.e., 1, 2, 3, etc.

The start of the options list begins with a ? (question mark), and each parameter is separated by an & (ampersand). The options are **NAMED** as outlined in the table below. For example, if you wanted to specify "7-bits", you would use:

```
com://port=4?speed=9600&bits=7
```

Table 19-5 Communication Port Options

Option	Description	Default	Value
SPEED	specifies baud rate	9600	110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 56000, 128000, 256000
BITS	specifies number of bits	8	5, 6, 7, 8
PARITY	specifies parity used	off	off, even, odd, mark, space
STOPBITS	specifies number of stop bits	1	1, 1.5, 2
DSRDTR	specifies dsr/dtr protocol	off	on, off
RTSCTS	specifies rts/cts protocol	off	on, off
XONXOFF	specifies xon/xoff protocol	off	on, off

If the baud or bit value is invalid, the default value is used.

SheerPower 4GL can be used as a very powerful web scripting language. SheerPower allows you to write simple to complex interactive web applications---fast---very fast!

SheerPower web scripting programs produce dynamic webpages, and can be embedded into HTML code, similar to PHP, ASP and Perl languages.

Programs written in other languages can be called from a SheerPower web scripting program (using the PASS Statement Section 10.8), and the database capabilities are extremely powerful (Chapter 15, Data Structure Statements).

The full power and simplicity of the entire SheerPower 4GL language is available for use when web scripting.

SheerPower Web Scripting: Immune to SQL Injection Attacks

SheerPower web scripting is immune to SQL injection attacks. This is because SheerPower does not mix data and code when doing database operations. With SheerPower 4GL, the data and the code are always separate and well defined---eliminating the possibility of SQL injection attack methods.

For more on SQL injection attacks, see http://en.wikipedia.org/wiki/SQL_injection

This chapter assumes you have experience using HTML code to develop webpages. The contents of this chapter is in the following structure:

- Running two different sample script programs to get a feel for the output that can be generated, and how SheerPower script programs are run using SPINS Webserver
- A breakdown of how the matrix.spsrc sample program works to generate identical browser results using either SCRIPT areas or CODE areas
- A breakdown of how the more advanced test.spsrc sample interactive web form application works
- Sections on all of the SheerPower web scripting tags (with code examples), along with information on where script programs are saved and the steps to run them:
    - SCRIPT and CODE area tag syntax (including "short tags") (Section 20.2)
    - displaying complex expression results (Section 20.4.1)
    - embedding code into script areas using the %include directive (Section 20.4.2)
    - using getsymbol$() to retrieve CGI environment symbols (Section 20.4.3)
    - [[%persist]] to enhance the performance of your web application (Section 20.5)
    - the [[%once]] tag used to cause specific code that you want to run only once per application instance. (Section 20.5.1)
    - where script programs are located (Section 20.6)
    - how to invoke a SheerPower script program in the browser (Section 20.6.1)

Supported Script Tag Formats

Script and Code Area tags are defined by [[xxx]]. Using the [[xxx]] format allows created HTML files to be run through "pretty" programs and other HTML analysis tools without error.However, SheerPower also supports the <<xxx>> format of these tags as an alternative.

To see a SheerPower 4GL Web Script program in action, follow the steps below to run the following example script program located in **C:\SheerPower\sphandlers\scripts\sp4gl\test.spsrc**.

1. Start the SPINS_WEBSERVER (if not yet started) by clicking on the Windows **Start** menu and select **Run**. Type the following path into the "Open" field:

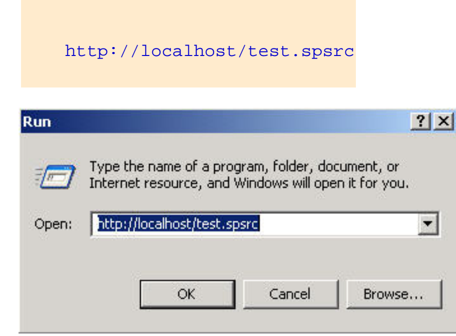
```
\sheerpower\sphandlers\spins_webserver.exe
```

Then click on the **OK** button. SPINS_WEBSERVER will start up in a Command Prompt window.

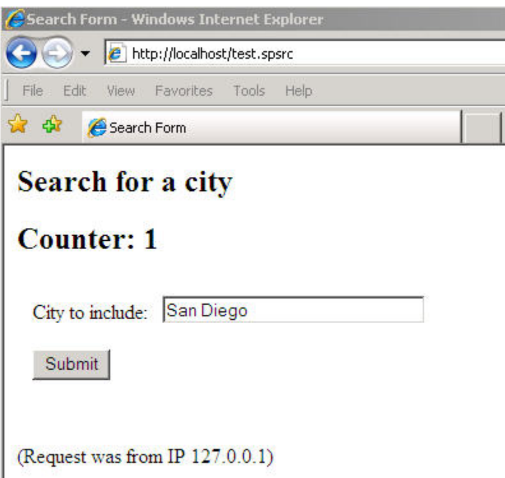
```
C:\sheerpower\sphandlers\spins_webserver.exe
SheerPower InterNet Services Web Server SPINS_WEBSERVER V01.00.032
Copyright (c) 2005-2008 Touch Technologies, Inc. - Sun, 22 Jun 2008 09:45:51
https port 443 is disabled
Listening IP address is 0.0.0.0
       Port    80, wwwroot is C:\sheerpower\sphandlers\wwwroot\
```

2. Open the Windows **Run** program again ("Start" menu, then select "Run") and type the following into the "Open" field:

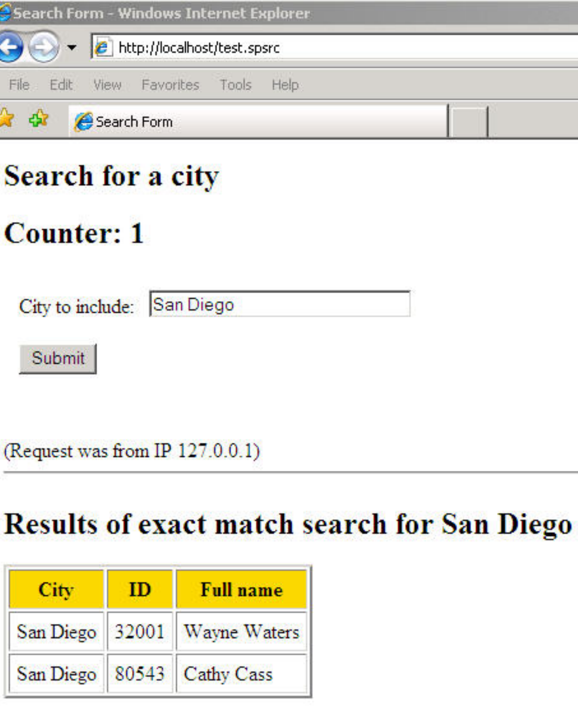
```
http://localhost/test.spsrc
```



The browser window will display the following form where a visitor can interactively search a database for information relating to a city name.



When the **Submit** button is pressed, the program retrieves the results and displays them in a table below the form.



Run the next sample web scripting program using the following steps:

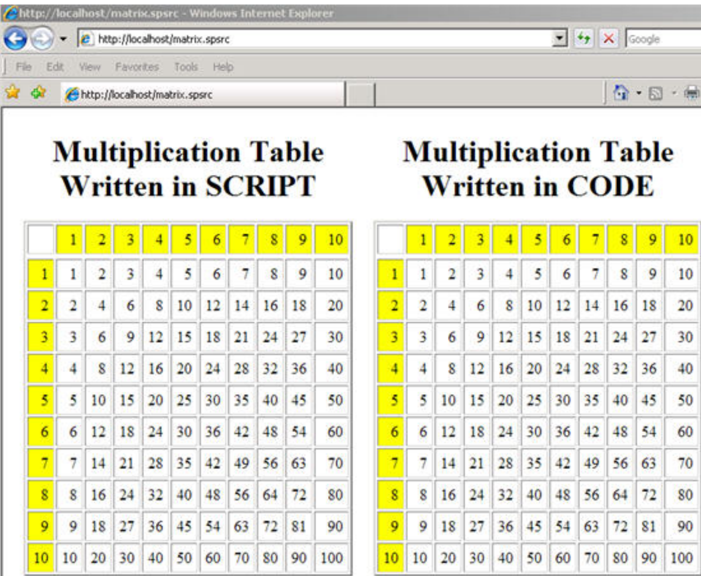
1. Start the SPINS_WEBSERVER (if not yet started) by clicking on the Windows **Start** menu and select **Run**. Type the following path into the "Open" field:

```
\sheerpower\sphandlers\spins_webserver.exe
```

   Then click on the **OK** button. SPINS_WEBSERVER will start up in a Command Prompt window.
2. Open the Windows **Run** program again ("Start" menu, then select "Run") and type the following into the "Open" field:

```
http://localhost/matrix.spsrc
```

This sample program creates a two identical Multiplication Tables displayed in a web browser window; one table is generated with a **SCRIPT AREA**, and the other is generated with a **CODE AREA**.



Multiplication Table Written in SCRIPT

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Multiplication Table Written in CODE

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Next, in the browser address bar, change the URL to be:

```
http://localhost/matrix.spsrc?highvalue=25
```

This changes the multiplication tables to display 25 x 25 instead of the default 10 x 10 display. The program file is located in **C:\SheerPower\sphandlers\scripts\sp4gl\matrix.spsrc**.

When you invoke matrix.spsrc with the browser, two identical multiplication tables are generated and displayed. The table on the left is generated by a SCRIPT AREA, and the table on the right is generated by a CODE AREA. We'll first take a look at how the Script Area works.

When writing a SheerPower web scripting program it is important to note that you are always in either a SCRIPT area or a CODE area.

In matrix.spsrc, the "Script Area" begins with HTML code to setup the table. Because neither the [[%script]] or [[%spcode]] tags were used to designate the area as script or code, SPINS Webserver sees the HTML code and outputs it directly to the browser, processing it as SCRIPT.

```
<h1><center>Multiplication Table<br>Written in SCRIPT</center></h1>

<table border=2 cellpadding=5>
  <tr>
  <td> 
```

The next lines illustrate how SheerPower code can be embedded inside a Script Area to process and display in the webpage dynamically.

```
  [[for j=1 to highvalue]]
    <td bgcolor=yellow align=right>[[j]]
  [[next j]]
```

To embed the SheerPower code in a Script area, surround the code with the **[[ ]]** tags. SPINS Webserver sees the code between the [[ ]] tags and processes it accordingly.

The code above says tells SPINS Webserver to take the value of "j", display it inside a table cell, then increment the value of "j" by one and display the new value into the next table cell - until the value of "highvalue" is reached.

The following embedded SheerPower code shows how to display the result of an evaluated expression:

```
  <td align=right> [[=i*j]]
```

The "=" in front of the expression causes the result to be displayed in the browser. This is how the values of "i" and "j" are multiplied together to complete the multiplication table.

The complete Script Area used to create the multiplication table on the left side of the browser window is below:

```
<h1><center>Multiplication Table<br>Written in SCRIPT</center></h1>

<table border=2 cellpadding=5>
  <tr>
  <td> 
  [[for j=1 to highvalue]]
    <td bgcolor=yellow align=right>[[j]]
  [[next j]]
  <tr>
  [[for i= 1 to highvalue]]
    <tr>
    <td bgcolor=yellow align=right> [[i]]

    [[for j=1 to highvalue]]
      <td align=right> [[=i*j]]
    [[next j]]
  [[next i]]
</table>
```

Next, let's look at the CODE Area used to generate the multiplication table displayed on the right side of the browser window.

```
[[
print '<td>'

print '<h1><center>Multiplication Table<br>Written in CODE</center></h1>'
```

The first line with the short tag "[[" by itself begins the Code Area. This ended the previous Script Area. Note that the tag is on a line all by itself. Alternatively, the **[[%spcode]]** tag could have been used.

The next lines in the Code Area are standard SheerPower PRINT statements used to send the HTML code directly to the browser.

```
[[
print '<td>'

print '<h1><center>Multiplication Table<br>Written in CODE</center></h1>'
```

Code used to perform the calculations required is written in standard SheerPower code format. SPINS Webserver processes it, and the PRINT statements are used to display the results in the browser.

```
for j=1 to highvalue
  print '<td bgcolor=yellow align=right>'; j;
next j
```

After the last PRINT statement, the ending tag **]]** is used on a line all by itself to signify the end of the Code Area:

```
print '</table>'
]]
```

The complete Code Area used to generate the multiplication table displayed on the right side of the browser window is below:

```
[[
print '<td>'

print '<h1><center>Multiplication Table<br>Written in CODE</center></h1>'

print '<table border=2 cellpadding=5>'
print '<tr>'
print '<td> '
for j=1 to highvalue
  print '<td bgcolor=yellow align=right>'; j;
next j
print '<tr>'
for i= 1 to highvalue
  print '<tr>'
  print '<td bgcolor=yellow align=right>'; i
  for j=1 to highvalue
    print '<td align=right>'; i*j
  next j
next i
print '</table>'
]]
```

At the beginning of the matrix.spsrc program, a Code Area was used to define "highvalue" as a **symbol** using the **GETSYMBOL$()** function.

```
// Change highvalue  to change the table size
[[
highvalue=val(getsymbol$('highvalue'), false)
if highvalue = 0 then highvalue = 10
highvalue = min(25, highvalue)  // limit to 25x25
]]
```

This allows the size of the multiplication tables to be changed according to the optional **highvalue** parameter entered in the browser URL.

```
http://localhost/matrix.spsrc?highvalue=25
```

More on **GETSYMBOL$()** can be found at Section 6.4.13, GETSYMBOL$( ).

The [[tag all by itself on a line indicates the start of a Code Area. The end of the Code Area is defined by the **]]** tag all by itself on a line.

When the Code Area is completed with the **]]** tag by itself on a line, SPINS Webserver processes the next area of code as Script until another Code Area is defined.

The entire matrix.spsrc sample program is shown below:

```
<html>
<head>
</head>
<body>

// Change highvalue  to change the table size
[[
highvalue=val(getsymbol$('highvalue'), false)
if highvalue = 0 then highvalue = 10
highvalue = min(25, highvalue)  // limit to 25x25
]]

<table cellpadding=10>
// lets do the LEFT SIDE first
<tr>
<td>

<h1><center>Multiplication Table<br>Written in SCRIPT</center></h1>

<table border=2 cellpadding=5>
  <tr>
  <td> 
  [[for j=1 to highvalue]]
    <td bgcolor=yellow align=right>[[j]]
  [[next j]]
  <tr>
  [[for i= 1 to highvalue]]
    <tr>
    <td bgcolor=yellow align=right> [[i]]

    [[for j=1 to highvalue]]
      <td align=right> [[=i*j]]
    [[next j]]
  [[next i]]
</table>

// And, now the right-side -- but this time in a CODE area
[[
print '<td>'

print '<h1><center>Multiplication Table<br>Written in CODE</center></h1>'

print '<table border=2 cellpadding=5>'
print '<tr>'
print '<td> '
for j=1 to highvalue
  print '<td bgcolor=yellow align=right>'; j;
next j
print '<tr>'
for i= 1 to highvalue
  print '<tr>'
  print '<td bgcolor=yellow align=right>'; i
  for j=1 to highvalue
    print '<td align=right>'; i*j
  next j
next i
print '</table>'
]]

</table>

</body>
</html>
```

To write web scripting programs in SheerPower, it is important to understand how CODE and SCRIPT areas are defined.

In a SheerPower web scripting program, one is always either in a CODE AREA or a SCRIPT AREA. Stopping one area automatically starts the other. It is implied that when a new area is started (for example, a new CODE area) that the previous area has ended (for example, a SCRIPT area).

A **CODE AREA** in a web scripting program can be started with either of the following tags **alone on a single line**:

```
[[%spcode]]
[[
```

A CODE area can be ended by any one of the following three tags alone on a single line:

```
[[/%spcode]]
]]
[[%spscript]]  ! a beginning Script area tag will also end a code area
```

A SCRIPT area can be started with with the following tag by itself on a single line:

```
[[%spscript]]
```

A SCRIPT area can be ended by one of the following tags placed on a single line by itself:

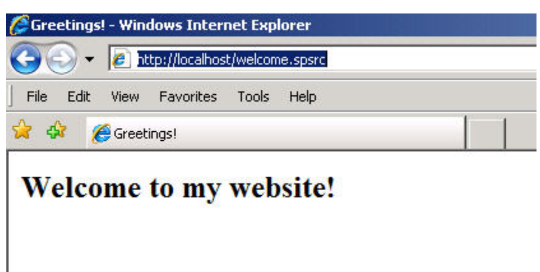
```
[[/%spscript]]  ! closing script tag
[[  ! the start of a CODE area
[[%spcode]] ! the start of a CODE area
```

FORMAT:

```
[[%spscript]]
...
...
[[/%spscript]]
```

EXAMPLE:

```
// The code below illustrates the %SPSCRIPT tag
// causing the HTML to be displayed in a browser window

[[%spscript]]

<html>
<head>
  <title>Greetings!</title>
</head>
<h2>Welcome to my website!</h2>

[[/%spscript]]
```



PURPOSE:

The tag [[%spscript]] tells SheerPower that the script area that follows is to be output to the browser. The script area ends when you use the [[/%spscript]] tag all by itself on a line.

The [[%spcode]] tag or the short tag "[[" all by itself on a line will also end a script area by starting a code area. It is implied that the Script Area is now complete.

Code comments in SheerPower Script Areas

Within SheerPower script areas, the "//" is used for inserting comments.

DESCRIPTION:

When web scripting, each text line inside a SheerPower script area becomes one or more PRINT statements at compile-time. For example:

```
[[%spscript]]
<h1>Your name is [[name$]]</h1>
```

At compile-time this generates:

```
print '<h1> Your name is ';
print name$;
print '</h1>'
```

Another example:

```
[[for idx = 1 to 5]]
<h1> idx is [[idx]] </h1>
[[next idx]]
```

Generates at compile-time:

```
for idx = 1 to 5
 print '<h1> idx is ';
 print idx;
 print ' </h1>'
next idx
```

While in a SheerPower script area, text inside of the **[[ ]]** tags is treated as **embedded SheerPower code**. For example:

```
[[%spscript]]
  [[for idx = 1 to 5]]
  <h1> idx is [[idx]] </h1>
  [[next idx]]
[[/%spscript]]
```

The following is output to the browser:

```
idx is 1
idx is 2
idx is 3
idx is 4
idx is 5
```

If the embedded code consists of an expression, then the expression is evaluated and the results are sent to the browser. For example:

```
[[%spcode]]
a=45
b=10
[[%spscript]]
The answer is [[=a+b]]
[[/%spscript]]
[[/%spcode]]
```

The following is output to the browser:

```
The answer is 55
```

FORMAT:

```
  [[%spcode]]
  ...
  ...
  [[/%spcode]]
```

EXAMPLE:

```
[[%spcode]]
city$ = getsymbol$("city")
print "<h1>"
select case city$
case "Calgary"
  print "Calgary is a great city!"
  case else
    print "You chose a different city!"
end select
print "</h1>"
```

```
[[/%spcode]]
```

To run the above example, copy and paste the code into a new program file called **spcode_city.spsrc** and save it in **sheerpower/sphandlers/scripts/sp4gl/**. Open a browser window and enter the URL **http://localhost/spcode_city.spsrc?city=Calgary**. The following is output to the browser:

```
Calgary is a great city!
```

PURPOSE:

A SheerPower Code Area starts with a [[%spcode]] tag. It designates a section of code within a web scripting program that contains SheerPower code. A code area is ended by a [[/%spcode]] tag on a line all by itself.

Code Areas can also be started with the [[by itself on a line. This is referred to as a "short tag"... a shortened version of the full [[%spcode]] tag. The short tag used to end a Code Area is the]] by itself on a line.

If a [[%spscript]] tag is used after the [[%spcode]] tag it implies that the code area has ended and a script area has started.

DESCRIPTION:

Within a SheerPower web scripting program, one is always in either a **CODE** area or a **SCRIPT** area. If a script area has ended, the next section of code is a code area.

When in a SheerPower code area, printing to channel zero (a "normal" print statement) causes the printed text to be sent to the browser. For example, to send HTML to the browser that displays a client's city in bold text:

```
  [[%spcode]]
  print '<b>'; client(city); '</b>'
```

Short tags can be used to start and end Code Areas. If a line begins with the **[[** tag all by itself, then all following lines are treated as a Code Area.

An ending line consisting of only the **]]** tag ends the Code Area.

Comments in SheerPower Code Areas

Within SheerPower Code Areas, the "//" is used to insert comments.

When you need to display the results of complex expressions contained inside the embedded SheerPower code within a script area (surrounded by [[ ]]), use an "=" at the beginning of the expression. For example:

```
// start a SheerPower code area
[[%spcode]]
  a=45
  b=10

// end the code area and start the SheerPower script area
[[%spscript]]

// embed the SheerPower code within the script area between [[ ]]
// the "=" in front of the expression causes the results to display in the browser
<h2><font color=darkgreen>The answer is [[=a+b]].</font></h2>

[[/%spscript]]
// end the script area and begin the code area
```

While in a SheerPower Code Area, you can embed any other code or script, including an .HTM or .HTML file, by using the **%INCLUDE** directive. SheerPower will recognize the HTML code and output it all to the browser. This makes it trivial to embed large blocks of HTML within your program. To include an HTML file:

```
%include "@sample_page.html"
```

If the file type is .SPSRC, .SPINC or .SPRUN, then the entire include file is assumed to be code. Otherwise it is assumed to be script.

See Section 3.9.4,%INCLUDE for more on the %INCLUDE directive in SheerPower.

FORMAT:

```
variable_name$ = getsymbol$('cgi_symbol')
```

EXAMPLE:

```
  // get the CGI symbol value of "password" and store it into
  // the variable mypassword$
[[%spcode]]
  mypassword$ = getsymbol$('password')
  print 'My Password is:'; mypassword$
[[/%spcode]]
```

PURPOSE:

The function GETSYMBOL$() is used to get the CGI symbol value from the webserver. If the symbol does not exist, then a null string is returned.

CGI Symbol Names

CGI Symbol Names are case sensitive.

DESCRIPTION:

Standard CGI environment symbols must be prefixed with "env:". For example:

```
[[%spcode]]
myip$ = getsymbol$('env:REMOTE_ADDR')
print 'My IP address is:'; myip$
[[/%spcode]]
```

For a list of CGI environment symbols supported, see Section 19.3.8, Summary of CGI Environment Variables.

If embedded code inside a SheerPower script area starts with a **$** (dollar sign) then what follows the $ is treated as a CGI symbol lookup. The result of that lookup is then sent to the browser. For example, to send the HTML to the browser the value of CITY (where CITY was received from a browser form submit action):

```
The city was: [[$CITY]]
```

CGI Symbol Names

CGI Symbol Names are case sensitive.

FORMAT:

```
  [[%persist]]
  ...
  ...
  [[/%persist]]
```

EXAMPLE:

```
[[%persist]]
[[%spscript]]
<html>
<head>
  <title>Search Form</title>
</head>
<body onload="document.forms[0].city.select()">
<h2>Search for a city</h2>
[[/%spscript]]
counter++
print '<h2>Counter: '; counter;'</h2>'
process$ = getsymbol$('process')
select case process$
case ''
  display_form
case 'show_results'
  display_form
  do_show_results
case else
  print 'Unknown process: '; process$
end select

print '</body>'
print '</html>'

[[/%persist]]
```

PURPOSE:

The **[[%persist]]** and **[[/%persist]]** tags will provide your program with a higher performance.

DESCRIPTION:

For high performance, a script can remain running longer than initially required using the **[[%persist]]** and **[[/%persist]]** tags.

To use the tags, place the [[%persist]] tag at the beginning of the main logic area, and the [[/%persist]] at the end of the main logic area.

The process included within the tags "goes away" after 10 seconds of inactivity.

In addition to the [[%persist]] tag, find any code that you want to run just once per persistance and surround it with the **[[%once]]** and **[[/%once]]** tags. Good candidates are:

- data structures that can be opened once and then left open
- any constants that you might want statically initialized before your main logic starts

```
routine do_show_results
  [[%once]]
  open structure client: name 'c:\sheerpower\samples\client'
  [[/%once]]
```

SheerPower scripting programs must be located in the following directory:

```
SheerPower\sphandlers\scripts\sp4gl
```

The scripting programs are not directly in the wwwroot folder purposely to increase security. They are parallel to it. The actual location of the program when it runs is:

```
wwwroot\..\scripts\sp4gl\
```

If you move wwwroot to a different location, then the location of the scripting program moves as well. For example, if you change wwwroot as follows:

```
spins_webserver -wwwroot d:\mystuff\
```

Then the new location that the scripting program should reside is:

```
d:\scripts\sp4gl
```

And the parallel location the program will run from is:

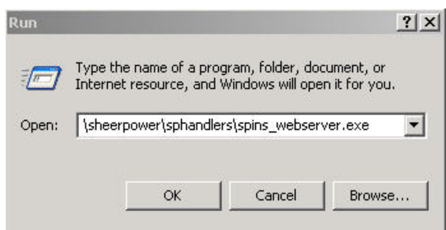
```
d:\wwwroot\..\scripts\sp4gl
```

See Section 18.1.4, Specify Any Root Folder on how to change the location of wwwroot in SPINS Webserver.

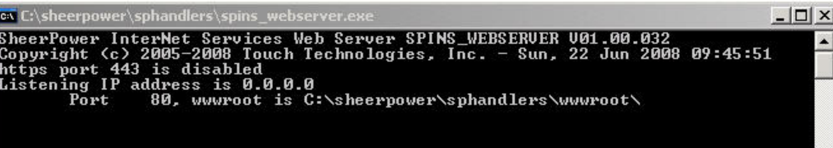
To invoke a SheerPower script program in your browser:

1. Start the SPINS_WEBSERVER (if not yet started) by clicking on the Windows **Start** menu and select **Run**. Type the following path into the "Open" field:

   ```
   \sheerpower\sphandlers\spins_webserver.exe
   ```

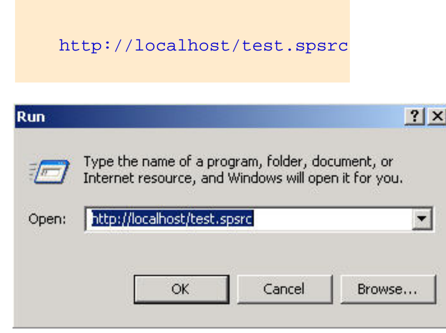


Then click on the **OK** button. SPINS_WEBSERVER will start up in a Command Prompt window.



2. Open the Windows **Run** program again ("Start" menu, then select "Run") and type the URL of your server:

```
http://localhost/test.spsrc
```



Routines written in other languages can be called and run from a SheerPower program.

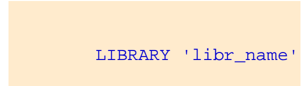
Callable routines are stored in libraries. The LIBRARY statement tells SheerPower what library a routine is located in. The LIBRARY statement must be used to specify where routines are located they are called in a program.

The CALL statement calls routines and executes them. Any routine in a shared Windows library can be called and executed. The CALL and LIBRARY statements make programs more powerful, more versatile, and provide more programming options.

Note

SheerPower 4GL can also run other programs from within SheerPower using the PASS instruction. See Section 10.8, Pass Commands to the Operating System.

FORMAT:



```
LIBRARY 'libr_name'
```

DESCRIPTION:

The **LIBRARY** statement is used to specify the libraries that will be used in a program.

The LIBRARY statement specifies libraries from which to CALL routines. The routines can be written in any Windows language that supports the standard calling interface (FORTRAN, BASIC, COBOL, etc.).

libr_name is the file specification of a library. The library can be one of the Windows supplied libraries, or a user-defined library. A library must be named with the LIBRARY statement before a routine is called from it. The library name must be a string constant in quotes.

FORMAT:

```
CALL routine_name(arg [BY pass_mech], arg...)
```

EXAMPLE:

```
  library 'msvcrt.dll'
  a$    = space$(40)
  text$ = 'Hello there!'
  call 'strcpy' (a$, text$)
  print a$
  end

Hello there!
```

DESCRIPTION:

The **CALL** statement is used to call and execute library routines. These routines can perform procedures so the programmer does not have to write the code from scratch.

The library to which the routine belongs must have been specified in a LIBRARY statement.

routine_name is the name of the routine being called. Some routines take arguments. *arg* is an argument that passes data to the routine or retrieves data from the routine. If more than one argument is passed, separate the arguments with commas:

```
CALL routine_name(arg, arg, arg...)
```

pass_mech refers to the mechanism by which values are passed to the routine. The default passing mechanism for integers is by **value**. The default passing mechanism for strings is by **reference**. Here is an explanation of the passing mechanisms available:

BY REF	By reference. This is the default passing mechanism for strings and integer arrays. Arguments passed by reference can be changed by the routine they are passed to.
BY VALUE	By value. This is the default passing mechanism for integer data. Arguments passed by value cannot be changed by the routine they are passed to.

Note

All INTEGER VALUES AND REFERENCES are to LONG WORD (4 byte) INTEGERS.

To pass integer arrays to external routines from SheerPower, you specify the name of the array along with "()". For example:

```
  dim abc%(100)
  call thestuff (abc%() by ref)
```

The system function _INTEGER can be used in conjunction with the library statements. This function returns resulting data associated with the CALL.

As defined by Rick Cadruvi

rick@rdperf.com

To program well you should not have to think too much about the actual code. Making the code simple and easy to understand is where your thoughts should be. It is the logic/problem-solving part of programming that requires 90% of the thinking.

Always choose to write code as simply as possible. It's often tempting to do "clever" things, but simple code is what pays off in the end. At some point in the future, you or someone else will need to look at the code and understand it... quickly! Too much time is wasted by programmers trying to interpret what another programmer (or they, themselves) did in a program or routine.

Writing the programming code is just creating the flow of logic needed to achieve the purpose of a program. So keep it simple and on track.

Think ahead when programming. Assume that the code you're writing will be useful in the future--then make it easily reusable and therefore somewhat generic (including re-entrant). It's worthwhile to take a few extra minutes to ask yourself: "What should this routine look like to make it easily used in the future for similar projects?" When working on a project, separate out dependencies such as calling something "operating system specific" or limiting it to a certain file specification. It is guaranteed that you will want to run this code on a different platform, or use a different file specification, at some point in the future.

To create professional looking code, here are some basic guidelines:

The code must be kept SIMPLE and CLEAN.

Code should always be neat in appearance and well-documented in each routine header.

Always take extra time to write the initial code. 95% of code should work the first time if it's written with care.

Put each small "concept" within the program into a separate routine. This is the single most important point about program structure. Use subroutines as a packaging mechanism to organize the individual thoughts and concepts in the program. The size of a concept will be subjective, but the smaller the thoughts or concepts you break a program into, the more successful you will be writing code.

Each routine should be under 25 lines. If the number of lines exceeds 25, the routine is becoming too complex. If your code no longer fits on one screen, you really need to ask yourself: "WHY?" There should be many routines and subroutines written.

Note: See Appendix M, SheerPower and Program Segmentation for more on program segmentation.

There are exceptions, however. CASE/SWITCH-type statements where the number of cases is very large can make a routine much longer than normal. In such instances, the individual cases should translate to routine calls to do the work rather than inline code.

Keep columns short (up to 80 characters). Many people use Notebook computers or have smaller monitors. Keeping the columns to 80 characters or fewer guarantees that all of the code will appear in the screen area. This will keep the viewer from having to scroll horizontally to see the ends of the lines.

Avoid more than three levels of indentation (nested LOOPS and IFs, etc.). Three levels or fewer keeps the code in a routine from becoming too complex. If there are more, then an inner loop should be moved out to a new routine. This keeps the code looking neat, clean, and easy for the reader to follow.

Be consistent with names within the program. Making up different names for the same thing is too difficult to follow. Variables and routines should be named in a manner that explains their meaning. Additionally, strive to keep names short and simple.

Always line up "=" signs. When declaring variables, list them alphabetically to make them easy to find. Initialize variables at the top of the routine.

Avoid doing IF ELSEs. IF ELSEs complicate code. For example, a well-written program that contains 25,000 lines of code (including comments) might have approximately 20 ELSEs compared to over a thousand IFs. This illustrates how rare an IF ELSE should be. If an IF ELSE is necessary, then the IF should be the short line of code, and the ELSE the long line.

Use RETURN statements as soon as possible within a routine. Do not wait until you get to the bottom of the routine. This will eliminate a LOT of ELSEs and funky loops and BUGS!

Always make the call, then test the result in the conditional expression statement. In general, never set variables to something, or test the result of a function call in an IF, DO or FOR expression.

Use parentheses around every set of operations. This makes it easy to determine the beginning and end of an operation. It also makes it absolutely clear what you intended, without depending on the order of precedence of operators. If someone reading the code can know with absolute certainty how you intended it to be executed via the use of parentheses, then they will not have to wonder if the language you used created code other than what you intended it to create. For example:

```
if a > b * c or d + 4 = f then x = x + 1

if ((a > (b * c)) or ((d + 4) = f)) then x = x + 1
```

Code should be language-independent. Code should be written so that programmers of all kinds of different languages can understand it. All languages have more or less the same constructs. There are language-specific items that will be unavoidable, but they should be few and far between. They should not get in the way of someone understanding the code, even if that person has never programmed in that language before.

Make code easy to understand quickly. It should take no more than 30 seconds to understand a line of code, and no more than 10 minutes to understand a routine. If the reader has to sit down and figure it out, the code is poorly written. Poorly written code is guaranteed to have bugs!

Avoid writing extraneous code. When writing the code, ask yourself: "Is this code really necessary?" Extraneous code makes for a complicated routine and is difficult to understand later. Once in a while, however, it is necessary to include some extra code purely for the sake of clarity.

Make code understandable and concise. Think about the code just written and see if there's a shorter, more concise and understandable way of writing it. However, sometimes taking a shortcut to write the code smaller can make it LESS understandable--so keep in mind that understandability is the main goal!

Always do a thorough code review before you compile it. A code review should enable you to find the logic flaws even before the first compile. The majority of errors will then be compile errors--and generally trivial ones at that. When you start your code review, ask yourself: "What did I do incorrectly?" Get into the mindset that your code has errors/bugs as opposed to thinking, "Of course it's right---I wrote it!" It's easier to find your errors when you're expecting them rather than using *selective vision* during your review.

Focus on writing good code, not debugging bad code. **A good programmer is one who spends very little time working with debuggers. Don't focus on debugging your code--focus on writing code that doesn't contain any bugs!**

Test the smallest parts of code. Take the routine and write a program to test it. Later when you go back to that code, you will know that routine will always work unless some changes were made to it. This can be particularly handy when having to debug, since the already-tested routine can be ruled out right away.

Always document code thoroughly in each routine header. Ideally, commenting should rarely be done inside the actual routine. If the routine is written simply and kept short, then no comments are needed within the code. Keep the size and scope of the routine limited and obvious for future reference.

Do not be afraid to rewrite or modify your code! When rewriting or modifying code, whether it's your own or someone else's, it's essential to have a backup copy of the ORIGINAL kept in a safe place. This way, if you need to scrap your changes, the original working copy won't be lost. You may lose the time that you spent doing the modifications--but at least you will still have what you started out with before making any changes. A second hard drive is recommended to store all programs for safekeeping.

Organize your routines in logical or alphabetical order. In shorter programs, the routines should follow each other in a logical order within the program. For longer programs with many routines, putting the routines in alphabetical order makes it easier to find a routine when needed.

Group routines into small modules. Every attempt should be made to group similar routines into separate source modules to keep module size smaller, which makes it easier to locate things. GLOBAL routines should be grouped together alphabetically, as should LOCAL routines.

The key to creating excellent code is in writing small routines that express individual thoughts/concepts.

Carefully writing your code and performing regular, thorough code checks can virtually eliminate the need for debugging--saving much time and frustration.

Most of the time spent on programming is NOT on the initial code creation itself, but in getting it to work, future modifications, and continuing to keep it working.

A good understanding of the purpose of the specific piece of code being worked on would help to create professional-looking code.

A programmer's creativity comes from the manner in which the problem is solved, NOT in the actual coding.

Always remember---the very best solutions expressed in code are the ones that any beginning programmer can understand.

Reserved words are words that cannot be used as identifiers. SheerPower's reserved words are:

DATE
DATE$
ELSE
EXLINE
EXTYPE
MAXNUM
NOT
PI
PRINT
REM
RND
TIME
TIME$

The following words will be reserved in future versions:

CON
IDN
NUL$
TRANSFORM
ZER

In SheerPower 4GL Debug Window, errors are returned either immediately after typing a syntactically incorrect line, or when the RUN command is given and your program is syntactically incorrect.

Errors happen when the program is compiling. Exceptions happen as the program is running.

In the case that the error happens outside of a routine (in the main logic area) the first digit is always a **1** followed by a period. Following the period is the source code line number.

I.E., **1.45** means the error occurred in the 45th line from the top of the file.

I.E., **do_totals.3** means the error occurred in 3 lines from the definition of the routine called **do_totals**.

In SPDEV the Alt + UP or DOWN arrow key can be used to move up or down a specific number of lines in your file. For more specialized programming keystrokes in SPDEV, see Appendix F, Keystrokes for SheerPower Rapid Development Environment.

ACCESS mode not INPUT, OUTPUT, or OUTIN
ADD without matching END ADD
Already declared as other than array
Array name expected
Array not yet dimensioned
Array or structure reference is missing "()"
Bad line number
CANCEL ADD without matching ADD
CANCEL EXTRACT without matching EXTRACT
Can't find relation in database
Can't GO
Can't store into an internal field
Can't use this option with FIELD clause
CASE expression is of different type than SELECT CASE
CASE statement missing 'TO'
CASE without matching SELECT CASE
CASE OF not within SELECT block
Compilation errors
Compiler doesn't handle this yet
Database engine discovered bad binary information buffer
Datatype mismatch
DO without matching LOOP
Dots not allowed in variable names
Dynamic variables may not be used as arrays
ELSE without matching IF

END ADD without matching ADD
END EXTRACT without matching EXTRACT or REEXTRACT
END IF without matching IF
END HANDLER without matching HANDLER
END SELECT not within SELECT block
END WHEN without a previous USE
EXIT ADD without matching ADD
EXIT EXTRACT without matching EXTRACT
EXPECTED KEY keyword
EXTRACT without matching END EXTRACT
Expected a line number or label
Expected a relational operator
Expected an expression
Expected ELSE clause
Expected end-of-line
Expected EXCEPTION keyword
Expected FIELD keyword
Expected file name expression
Expected IN or USE keywords
Expected INPUT keyword
Expected 'ON' or 'OFF'
Expected quoted string
Expected RECORD keyword
Expected STEP keyword
Expected STRUCTURE keyword
Expected THEN clause
Expected ','
Expected ':' in substring expression
Expected ',' or ':'
Expected ',' or ';'
Expected ',' or ')'
Expected '('
Expected ')'
Expected '='
Expression poorly formed
Expression too complex
EXTRACT without matching END EXTRACT
INPUT missing variable list
Invalid file name
Invalid relation name
Invalid routine name
Invalid workspace name
FOR EACH may not appear within and EXTRACT
FOR index must be real or integer simple variable
FOR without matching NEXT
Function or array not declared
GOTO or GOSUB expected
HANDLER name not found
HANDLER or FUNCTION name cannot be used as a label
HANDLER without matching END HANDLER
IF without matching END IF
Illegal ADD STRUCTURE nesting
Illegal character in source code
Illegal delimiter
Illegal EXTRACT nesting
Illegal integer
Illegal structure field name
Illegal use of dynamic variable/structure in CASE
Illegally nested error handlers
Illegally nested PROGRAM statement
INCLUDE, EXCLUDE, or SORT outside EXTRACT block
Internal END-ON without preceding ON
Invalid NEXT variable
Leading white space not within a multiple statement line
Line number or label not found
LOOP without matching DO
LSET, CSET, and RSET require string assignments
Missing operand
Missing '=' in FOR statement
Missing ':' in substring expression
Missing '('

Missing ')'
More than 32 dimensions not allowed
Multiply defined HANDLER
Multiply defined label
Nested FOR loops with same index variable
NEXT without matching FOR
No END statement terminating PROGRAM block
Not existing HANDLER name
Only one OPTION BASE allowed per program
Only one OPTION integer/real allowed per program
OPEN statement missing NAME keyword
PARTIAL only allowed for character fields
PROGRAM statement not first in program
Relation name expected
REPEAT DO or ITERATE DO not inside DO loop
REPEAT FOR, ITERATE FOR, or EXIT FOR not in FOR loop
Reserved word
RESUME, RETRY, CONTINUE, or EXIT HANDLER not within handler
SELECT CASE without matching END SELECT
SELECT has no matching END SELECT
Statement following LOOP is not WHILE or UNTIL
Statement is only legal in immediate mode
Statement must be inside a program
Structure name expected
Syntax error
Syntax error in specified expression
This option needs a FIELD clause
Too many relations declared
Too many structures in the relation
Tried to DECLARE an illegal or typed variable
Unexpected number of dimensions
Unknown special variable
Unrecognized COMPILE option
Unrecognized keyword
Unrecognized OPEN keyword
Unrecognized relational operator
Unrecognized SET/ASK keyword
Unrecognized statement
Unterminated ELSEIF statement
USE block has no END WHEN
Use of '()' not valid in this context
USE without preceding WHEN EXCEPTION IN...
Using '()' with array name not valid in this context
Variable already declared
Variable currently exists with a different data type
Variable names may not contain embedded dots
Valid OPTION BASE parameters are 0 and 1
Warning: Missing quote assumed
WHEN EXCEPTION IN has no USE block
WHEN EXCEPTION USE has no END WHEN
WHEN may not be inside of a HANDLER or USE block
WHILE/UNTIL or EXIT DO not inside DO loop
Wrong number of arguments
Wrong number of subscripts in array reference

Exceptions only occur during execution. The runtime exceptions are:

	Too many GOSUBs
	User CAUSED exception
	Improper entry to error handler detected
	Dynamic variable used but not assigned
	Upper bound is less than lower bound
	File already exists (use REPLACE)
	Datatype mismatch
	Statistics are not enabled
-1001	Program interrupted
-1002	Source code changed

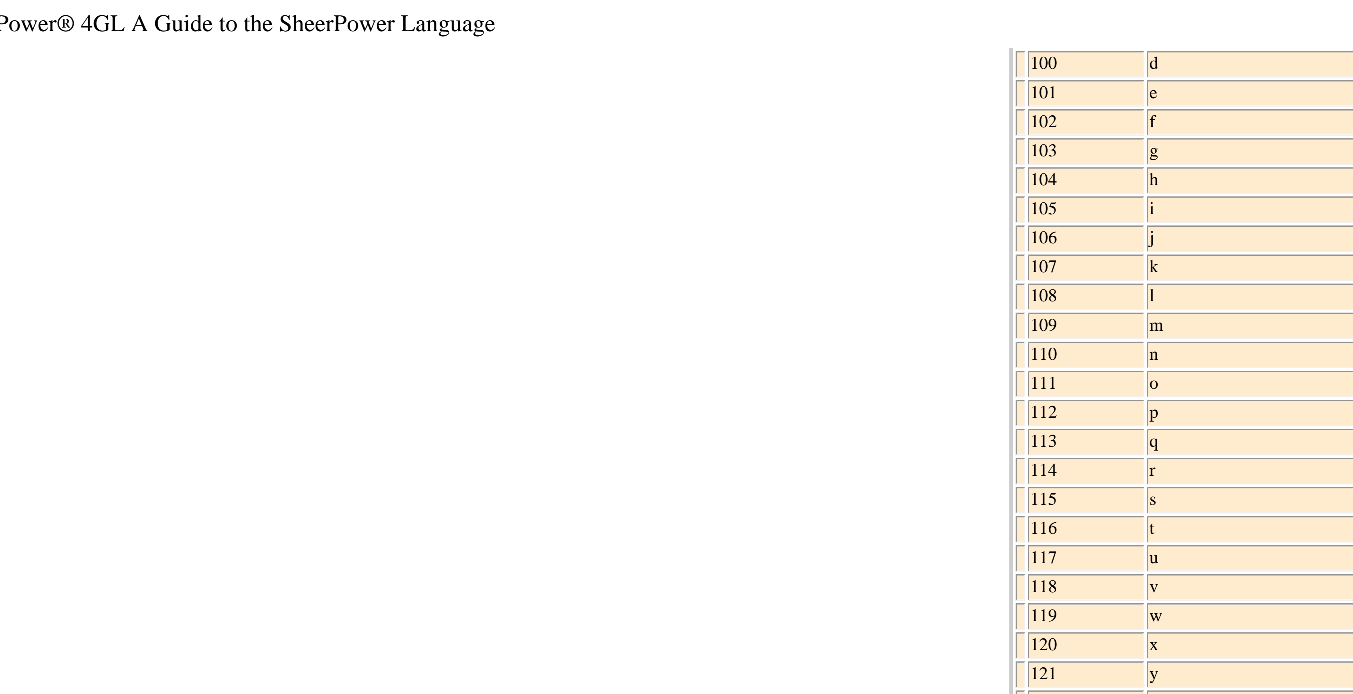
-3101	Dynamic variable datatype mismatch
-3102	Nonexistent structure field referenced
-4001	Illegal digit in decoded string
-4002	Numeric base in DECODE/ENCODE$ not between 2 and 16
-4003	Illegal argument to ORDNAME$()
-4004	Illegal argument to date$
-4005	Illegal argument to DAYS
-4006	Invalid logical value
-4007	Invalid input screen format
-4008	Illegal memory address to MEM function
-4009	Illegal argument to ENCODE$()
-4010	Illegal argument to CHARSET$()
-4016	Illegal length argument
-4017	Expression or statement invalid
-4021	Illegal validation rule
-7011	Must close structure before reusing structure name
-7012	Error in directory name
-7013	Invalid attempt to update a key field
-7014	Deadlock detected
-7015	Database engine rejected add or update
-5001	Program interrupted
-5002	Exceeded virtual memory
-7001	Miscellaneous file error
-7002	Error in structure file
-7003	Error in data dictionary
-7004	Error in data file
-7005	No privilege for operation
-7006	No such record
-7007	Not a key field
-7008	Unknown DBMS in structure
-7009	Duplicate key
-7010	File full
-8001	Structure reference of nonexistent field
-8002	Numeric conversion error
-8003	Unsupported field data type
-8004	Disk quota exceeded
-8102	Syntax error in DATA
-10001	Illegal passing mechanism
-10002	Can't call external routine
-10003	Error in external routine
-10004	Dispatch routine not found
-10005	Illegal symbol name
-10006	Setting symbol failed
-10007	Ambiguous symbol definition
1001	Illegal number
1002	Integer overflow
1002	Floating overflow
1003	Overflow in built-in function
1004	Overflow in evaluating VAL
1006	Overflow on READ number
1007	Overflow in numeric input from terminal
1051	Overflow in evaluating string expression

1053	Overflow on READ string
1054	String too long on input
2001	Subscript out of bounds
3001	Division by zero
3002	Negative number raised to a nonintegral power
3003	Zero raised to negative power
3004	LOG() of zero or negative number
3005	SQRT() of negative number
3006	MOD or REM had zero divisor
3007	Argument to ACOS or ASIN not in range -1<=X<=1
3008	Attempt to evaluate ANGLE(0,0)
4001	Illegal argument to VAL()
4002	Illegal argument to CHR$()
4003	Illegal argument to ORD()
4004	Illegal argument to SIZE
4005	Index in TAB less than one
4006	SET MARGIN value less than zonewidth
4007	SET ZONEWIDTH value greater than margin
4008	LBOUND index out of range
4009	UBOUND index out of range
4010	Second argument to REPEAT$ < 0
5001	REDIM overflow
6005	REDIM first index greater than second
7001	Bad channel number
7002	Attempt to open or close channel zero
7003	Channel number already in use
7004	File not open
7100	Invalid file attribute
7101	Too many files open
7110	File not found
7112	File locked by another user
7120	Illegal file name
7305	No current record
7322	Delete or update not allowed
8001	Out of DATA
8002	Too few input items
8003	Too many input items
8011	End of file on input
8012	Too few data items in record
8013	Too many data items in record
8101	READ a non-number
8102	Syntax error in input
8103	Non-numeric input when number expected
8105	Invalid file input
8201	Bad format$() or using string
8301	Record is larger than record size
8401	Timeout on input
8402	Illegal numeric value for time-expression
10001	ON index out of range
10002	RETURN without GOSUB
10004	SELECT found no matching CASE
10005	Chained program unavailable

Table D-1 ASCII Character Set

Decimal Value	ASCII character equivalent
0	NUL
1	SOH
2	STX
3	ETX (CTRL/C)
4	EOT
5	ENQ
6	ACK
7	BEL (bell)
8	BS (backspace)
9	HT (horizontal tab)
10	LF (linefeed)
11	VT (vertical tab)
12	FF (form feed)
13	CR (carriage return)
14	SO
15	SI (CTRL/O)
16	DLE
17	DCL (CTRL/Q XON)
18	DC2
19	DC3 (CTRL/S XOFF)
20	DC4
21	NAK (CTRL/U)
22	SYN
23	ETB
24	CAN
25	EM
26	SUB (CTRL/Z)
27	ESC (escape)
28	FS
29	QS
30	RS
31	US
32	SP (space)
33	!
34	"
35	#
36	$
37	%
38	&
39	' (apostrophe)
40	(
41	)
42	*
43	+
44	, (comma)
45	-
46	.
47	/

48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\ (backslash)
93	]
94	^ (or up arrow)
95	_ (or back arrow)
96	` (grave accent)
97	a
98	b
99	c



100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124		(vertical line)
125	}
126	~ (tilde)
127	del (rubout)

Multiple database engines (record management systems) can be used with SheerPower. The following database engines are currently supported:

- ARS
- FASTFILE
- ODBC

The SETUP routine tells SheerPower where data files are located and what record management systems are used. Once you have defined your structures using SETUP, you can use SheerPower's structure statements to manipulate the data.

For detailed information on SheerPower interfaces to the various database engines, you can refer to Chapter 16, Database Setup, in this manual.

SheerPower Rapid Development Environment has many keystrokes mapped to simplify and speed up program creation and file editing. A 'mapped keystroke' means that the regular function of a certain key may not perform the same function while in SheerPower Rapid Development Environment. For example, KP0 - means the number 0 (zero) in the numeric keypad of a keyboard. KP stands for keypad. In other programs, KP0 will print a 0 (zero). In SPDEV, KP0 moves the cursor to the beginning of the next line. At the end of a file, it will create a new line at the bottom of the file.

The keystrokes are grouped in tables by usage for quick reference.

The **GOLD Key** referred to in these tables is a special key used to create many of the keystrokes within SPDEV. Both the `[Esc]` (escape key - top left corner of the keyboard) and the `[Num-Lock]` (numbers lock key in the numeric keypad) are GOLD keys in SheerPower Rapid Development Environment.

To use the GOLD key, press the GOLD key, let go, then continue with the rest of the keystroke to complete the function.

For example, to use gold/F3 to bring up the FIND dialog box, you would press once on a GOLD key, either the **escape key** or the **num-lock key**. Let go and look in the bottom right hand corner of the SPDEV window. SheerPower tells you if your GOLD key is activated by highlighting a small square in the bottom frame in black, with gold letters that say 'Gol'.

Gol|CAP|NUM|OVR

Now that the GOLD key is activated, press `[F3]` to complete the keystroke. The FIND dialog box will pop open on the screen.

When using the Ctrl or Alt in SPDEV keystrokes, the Ctrl or Alt key depressed while completing the keystroke. Keystrokes are shown in the order of key pressed.

In SPDEV you can custom map any keystroke to suit your programming and editing style. For details on custom keymapping in SPDEV, see Section H.6.3, Keystroke Function Mappings Option.

By default, **keypad editing** is turned on---where the numeric keypad is used for rapid editing. To turn off this high-speed editing feature:

- In SPDEV toolbar click on Options.
- Select "Change System Settings".
- Then select the checkbox beside "Disable Keypad Editing"

1. Section F.1, Most Popular Keystrokes

2. Section F.2, Programming Keystrokes

3. Section F.3, Keystroke Combinations

4. Section F.4, Editing Keystrokes

5. Section F.5, Keystrokes for Movement Within a File

6. Section F.6, Search, Find and Replace Keystrokes

7. Section F.7, Keypad Editing Keystrokes

8. Section F.11, Alt Keystrokes

9. Table F-14, Ctrl Keystrokes

10. Table F-15, F Keystrokes

11. Section F.12, Shift Keystrokes

12. Section F.13, Learn and Macros Keystrokes

13. Section F.14, All Keystrokes

Certain keystrokes are used more often than others. The following table contains the most popular keystrokes for SheerPower users.

Table F-1 Most Popular SPDEV Keystrokes

Keystroke	Function Performed
ESCAPE	GOLD key
NUMLOCK	GOLD key
gold/alt/n	bring up NEW FILE dialog box
gold/alt/o	bring up OPEN FILE dialog box
gold/alt/p	purge previous file versions
gold/alt/l	launch a new instance of SPDEV
gold/alt/4	launch console window
gold/alt/s	save current file

gold/f	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/f	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/KP Enter	bring up the REPLACE dialog box
gold/F3	bring up FIND dialog box
gold/KP *	bring up FIND dialog box
F3	find next
F11	go to definition-place the cursor on a reference to a routine, then press F11 to go directly to that routine.
F12	show definition-place cursor on a reference to a routine, then press F12 to display the contents of the routine header in a new window.
KP *	find next
ctrl/g	find next
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
ctrl/p	move cursor to previous bookmark
ctrl/a	select all
ctrl/KP +	cut current character
gold/ctrl/KP +	uncut current character [paste]
KP 5+	cut current word
gold/KP +	uncut current word [paste]
KP 4	select current line
gold/KP 4	select current paragraph
ctrl/c	copy (end selection)
ctrl/x	cut selection
ctrl/v	paste
gold/KP 6	paste
delete	delete current selection or current character
gold/end	delete from cursor to end of line
ctrl/u	delete from cursor to beginning of line
gold/home	delete from cursor to beginning of line
gold/F2	delete from cursor to end of word [including any white space]; deleted characters saved in word paste buffer
ctrl/y	redo
ctrl/z	undo
ctrl/r	restore previous selection
alt/r	runs the current program
gold/KP 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/KP 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
home	move cursor to the beginning of current line
ctrl/e	move cursor to end of line (or end of next line if already at end of line)
end	move cursor to end of line (or beginning of line if already at end of line, and visa versa)
gold/page down	move cursor to bottom of file
gold/page up	move cursor to top of file
left arrow	move cursor left one character
ctrl/left arrow	move cursor left one word
right arrow	move cursor right one character
ctrl/right arrow	move cursor right one word
down arrow	move cursor down one line
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
up arrow	move cursor up one line
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines
gold/down arrow	convert selection to lower case, skips quoted text

gold/up arrow	convert selection to upper case, skips quoted text

These keystrokes are extremely helpful and time-saving particularly for programmers.

Table F-2 Programming Keystrokes

Keystroke	Function Performed
gold/alt/n	bring up NEW FILE dialog box
gold/alt/o	bring up OPEN FILE dialog box
gold/p	generate a program template
gold/r	generate a routine template
gold/o	organize routines within source code
gold/d	document a routine in a program
gold/a	align the "=" and ":" in a group of variable assignments
gold/c	generate debug comment line of !++ xxx date
gold/forward slash	comment/uncomment the selected text or current line
gold/f	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/f	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
alt/i	open .SPINC (include) files
gold/alt/l	launch a new instance of SPDEV
gold/alt/4	launch console window
gold/alt/z	save open files and exit SPDEV
gold/F3	bring up FIND dialog box
gold/KP *	bring up FIND dialog box
F3	find next
F11	go to definition-place the cursor on a reference to a routine, then press F11 to go directly to that routine.
F12	show definition-place cursor on a reference to a routine, then press F12 to display the contents of the routine header in a new window.
KP *	find next
ctrl/f	brings up the FIND dialog box
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
ctrl/p	move cursor to previous bookmark
gold/l	start/stop recording macro
gold/n	run the macro associated with the number (n = number)
alt/r	runs the current program
F9	expansion key
KP /	expansion key
KP 0	move cursor to the beginning of the next line. If at EOF, a new line is started.
gold/KP 0	insert a blank line before the current line; moves cursor to the blank line.
gold/KP 2	delete from cursor to end of line
gold/kp 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/kp 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/home	delete from cursor to beginning of line
gold/page down	move cursor to bottom of file
gold/page up	move cursor to top of file
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines
gold/alt/s	save current file

The following table describes keystroke combinations that allow the user to do particular functions with ease. For example, `[Ctrl/C]` and `[Ctrl/V]` perform the common "copy and paste" combination.

Table F-3 Keystroke Combinations

Keystroke	Combinations
ctrl/c copy (ends selection)	**ctrl/v** paste
KP 5 copy	**KP Enter** paste
ctrl/x cut selection	**ctrl/v** paste
gold/F3 bring up FIND dialog box	**F3** search for next occurence
F5 cut current line	**gold/F5** uncut current line [pastes it back]
F6 cut current word	**gold/F6** uncuts current word [pastes it back]
home move to the beginning of current line	**gold/end** deletes from cursor to end of line
KP 4 select line	**KP 6** cut
gold/KP 4 select paragraph	**gold/KP 5** copy paragraph
ctrl/p go to previous bookmark	**ctrl/n** go to next bookmark

The following tables describe the special SheerPower keystrokes designed to make editing in SPDEV fast and easy.

Table F-4 Select

Keystroke	Function Performed
KP4	selects current line
gold/KP4	selects current paragraph
ctrl/a	select all
shift/ctrl/end	selects text from cursor to end of file
shift/ctrl/home	selects text from cursor to beginning of file
shift/right arrow	selects text one character at a time

Table F-5 Copy

Keystroke	Function Performed
ctrl/c	copy (end selection)
KP 5	copy [or copy CURRENT LINE if no text selected]
gold/KP 5	copy current paragraph without selecting

Table F-6 Cut

Keystroke	Function Performed
KP 6	cut
ctrl/x	cut selection
F7	cut current character without selecting
F6	cut current word without selecting
F5	cut current line without selecting
KP -	cut current line without selecting
KP +	cut current word without selecting
ctrl/KP +	cut current character without selecting
shift/delete	cut selected text to the clipboard

Table F-7 Paste

Keystroke	Function Performed
gold/KP 6	paste
ctrl/t	trim whitespace and paste
ctrl/v	paste
KP Enter	paste
gold/ctrl/KP +	uncut current character [paste]
gold/F7	uncut character [paste]
gold/KP +	uncut current word [paste]
gold/F6	uncut current word [paste]

gold/F5	uncut current line [paste]

Table F-8 Delete

Keystroke	Function Performed
delete	delete the current selection or current character
backspace	delete previous character
gold/F2	delete from cursor to end of word
gold/KP 3	delete from cursor to end of word [including any whitespace]; deleted characters saved in word paste buffer
gold/KP 2	delete from cursor to end of line
gold/ctrl/e	delete from cursor to end of line
gold/end	delete from cursor to end of line
gold/home	delete from cursor to beginning of line
ctrl/u	delete from cursor to beginning of line

Table F-9 Formatting

Keystroke	Function Performed
gold/a	align the "=" and ":" in a group of variable assignments
gold/f	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/f	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/KP 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/KP 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/o	organize routines within source code
gold/p	generate program template
gold/r	generate routine template
gold/s	sort current column

SheerPower has keystrokes which allows quick and easy movement within files. This saves much time when large files are being worked with.

Table F-10 Keystrokes for Movement within a File

Keystroke	Function Performed
KP 1	move cursor to next word
F2	move cursor to beginning of next word
ctrl/KP4	move cursor left one word
KP 2	move cursor to the end of line (or next line if already at end of line)
ctrl/e	move cursor to end of line (or end of next line if already at end of line)
end	move cursor to end of line (or beginning of line if already at end of line, and visa versa)
ctrl/end	go to the bottom of the file
ctrl/h	move cursor to beginning of line (or beginning of previous line if already at beginning)
home	move cursor to the beginning of current line
ctrl/home	go to the top of the file
KP 0	move cursor to the beginning of the next line. If at EOF, a new line is started.
F1	move cursor to the beginning of the next line. If at EOF, a new line is started.
KP 7	move cursor one page up
KP 8	move cursor one page down
page down	move cursor one page down
gold/page down	move cursor to bottom of file
page up	move cursor one page up
gold/page up	move cursor to top of file
right arrow	move cursor right one character
ctrl/right arrow	move cursor right one word

left arrow	move cursor left one character
ctrl/left arrow	move cursor left one word
down arrow	move cursor down one line
up arrow	move cursor up one line
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines
gold/F3	bring up FIND dialog box
gold/KP *	bring up FIND dialog box
ctrl/g	find next
F3	find next
F11	go to definition-place the cursor on a reference to a routine, then press F11 to go directly to that routine.
F12	show definition-place cursor on a reference to a routine, then press F12 to display the contents of the routine header in a new window.
KP *	find next
ctrl/b	toggle between current cursor position and previous cursor position
ctrl/f	brings up the FIND dialog box
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
ctrl/p	move cursor to previous bookmark
gold/KP 0	insert a blank line before the current line. Move cursor to the blank line

If you want to find certain words or phrases within a file, and even replace it throughout an entire file, the following keystrokes will speed up that task enormously.

Table F-11 Search, Find and Replace Keystrokes

Keystroke	Function Performed
gold/F3	bring up FIND dialog box
gold/KP *	bring up FIND dialog box
F3	find next
gold/ctrl/f	puts data into the "find buffer". F3 performs the "find".
ctrl/g	find next
KP *	find next
ctrl/f	brings up the FIND dialog box
Insert	toggle between insert and overstrike modes
gold/KP Enter	bring up the REPLACE DIALOG BOX
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
ctrl/p	move cursor to previous bookmark
ctrl/b	toggle between current cursor position and previous cursor position
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines
gold/down arrow	convert selection to lower case, skips quoted text
gold/up arrow	convert selection to upper case, skips quoted text
F9	expansion key
gold/. (period)	toggle select [SEL] mode
ctrl/r	restore previous selection

In order to speed up the writing of programs and editing of text, high frequency use tasks have been mapped to the numeric keypad located on the right side of most personal computer keyboards. This allows for extremely fast one hand editing.

KP = keypad. To perform a KP keystroke, just press the keypad number or character indicated for the function you wish to perform. Some of the keystrokes use the 'GOLD' key, which is the `[Esc]` or `[NumLock]` key.

Disable Keypad Editing

While using SPDEV, the numeric keypad defaults to the special keypad editing keystrokes outlined below. To disable or re-enable the numeric keypad editing keystrokes inside SPDEV, click on Options --> Change System Settings --> then select the checkbox beside "Disable Keypad Editing".

Table F-12 Keypad Editing Keystrokes

Keystroke	Function Performed
Alt/Num-Lock	disables or re-enables the embedded keypad special functions.
KP 0	move cursor to the beginning of the next line. If at EOF, a new line is started.
gold/KP 0	insert a blank line before the current line; moves cursor to the blank line.
KP 1	move cursor to the next word
gold/KP 1	reverse case on selected text---including quoted text
KP 2	move cursor to the end of line (or next line if already at end of line)
gold/KP 2	delete from cursor to end of line
gold/KP 3	delete from cursor to end of word [including any white space] ; deleted characters saved in word paste buffer
KP 4	select current line
gold/KP 4	select current paragraph
KP 5	copy current line
gold/KP 5	copy current paragraph
KP 6	cut
gold/KP 6	paste
KP 7	move cursor one page up
KP 8	move cursor one page down
gold/KP 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/KP 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
KP . (period)	toggle select [SEL] mode
KP +	cut current word
gold/KP +	uncut current word [pastes it back]
ctrl/KP +	cut current char
gold/ctrl/KP +	paste [uncut current word]
KP /	expansion key
KP *	find next
gold/KP *	bring up FIND dialog box
KP -	cut the current line
gold/KP Enter	bring up the REPLACE DIALOG BOX

These are the `[GOLD]` keystrokes. The **GOLD Key** referred to in this tables is a special key used to create many of the keystrokes within SPDEV. Both the `[Esc]` (escape key - top left corner of the keyboard) and the `[Num-Lock]` (numbers lock key in the numeric keypad) are GOLD keys in SheerPower Rapid Development Environment.

To use the GOLD key, press the GOLD key, let go, then continue with the rest of the keystroke to complete the function.

For example, to use gold/F3 to bring up the FIND dialog box, you would press once on a GOLD key, either the **escape key** or the **num-lock key**. Let go and look in the bottom right hand corner of the SPDEV window. SheerPower tells you if your GOLD key is activated by highlighting a small square in the bottom frame in black, with gold letters that say 'Gol'.

Gol|CAP|NUM|OVR

Now that the GOLD key is activated, press `[F3]` to complete the keystroke. The FIND dialog box will pop open on the screen.

Table F-13 GOLD Keystrokes

Keystroke	Function Performed
gold/alt/4	launch console window
gold/a	align the "=" and ":" in a group of variable assignments
gold/c	generate debug comment line of !++ xxx date
gold/d	document a routine in a program

gold/down arrow	convert selection to lower case, skips quoted text
gold/ctrl/e	delete from cursor to end of line
gold/end	delete from cursor to end of line
gold/ctrl/f	puts data into the "find buffer". F3 performs the "find".
gold/f	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/f	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/F2	delete from cursor to end of word
gold/F3	bring up FIND dialog box
gold/F5	uncut current line [pastes it back]
gold/F6	uncut current word [pastes it back]
gold/F7	uncut character [pastes it back]
gold/home	delete from cursor to beginning of line
gold/KP 0	insert a blank line before the current line. Move cursor to the blank line.
gold/KP 1	reverse case on selected text---including quoted text
gold/KP 2	delete from cursor to end of line
gold/KP 3	delete from cursor to end of word [including any white space] ; deleted characters saved in word paste buffer
gold/KP 4	select current paragraph
gold/KP 5	copy current paragraph without selecting
gold/KP 6	paste
gold/KP 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/KP 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
gold/KP *	bring up FIND dialog box
gold/KP +	uncut current word [pastes it back]
gold/ctrl/KP +	uncut current character [paste]
gold/KP Enter	bring up the REPLACE dialog box
gold/=	repeat next keystroke NNN times [for macros see gold/l]
gold/l	start/stop recording macro
gold/alt/l	launch a new instance of SPDEV
gold/alt/n	bring up NEW FILE dialog box
gold/n	run the macro associated with the number (n = number)
gold/o	organize routines within source code
gold/alt/o	bring up OPEN FILE dialog box
gold/p	generate program template
gold/alt/p	purge previous file versions
gold/. (period)	toggle select [SEL] mode
gold/page down	move cursor to bottom of file
gold/page up	move cursor to top of file
gold/r	generate routine template
gold/s	sort current column
gold/alt/s	save current file
gold/alt/t	open a temporary file
gold/forward slash	comment/uncomment the selected text or current line
gold/up arrow	convert selection to upper case, skips quoted text
gold/alt/z	save open files and exit SPDEV

These are the `[Ctrl]` (Control) keystrokes. The `[Ctrl]` is generally located to the left and sometimes to the right of the space bar. When using the Ctrl key in SPDEV keystrokes, keep the Ctrl key depressed while completing the keystroke. Keystrokes are shown in the order of key pressed.

Table F-14 Ctrl Keystrokes

Keystroke	Function Performed

ctrl/a	select all
ctrl/b	toggle between current cursor position and previous cursor position
ctrl/c	copy (ends selection)
ctrl/e	move cursor to end of line (or end of next line if already at end of line)
gold/ctrl/e	delete from cursor to end of line
ctrl/f	brings up the FIND dialog box
gold/ctrl/f	puts data into the "find buffer". F3 performs the "find".
ctrl/g	find next
ctrl/h	move cursor to beginning of current line (or beginning of previous line if already at beginning)
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
ctrl/p	move cursor to previous bookmark
ctrl/r	restore previous selection
ctrl/t	trim whitespace and paste
ctrl/u	delete from cursor to beginning of line
ctrl/v	paste
ctrl/x	cut selection
ctrl/y	redo
ctrl/z	undo
ctrl/right arrow	move cursor right one word
ctrl/left arrow	move cursor left one word
ctrl/KP 4	select current line

These are the `[F]` keystrokes. (F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, and F12). These keys are usually located in the very top row of keys on a personal computer keyboard. To perform the gold/F keystrokes, press the gold key and let go, then press the specific F key to complete the keystroke.

Table F-15 F Keystrokes

Keystroke	Function Performed
F1	move cursor to the beginning of the next line. If at EOF, a new line is started.
F2	move cursor to beginning of next word
gold/F3	bring up FIND dialog box
F5	cut current line
gold/F5	uncut current line [pastes it back]
F6	cut current word
gold/F6	uncut current word [pastes it back]
F7	cuts/delete a character
gold/F7	uncut character [pastes it back]
F9	expansion key
F11	go to definition-place the cursor on a reference to a routine, then press F11 to go directly to that routine.
F12	show definition-place cursor on a reference to a routine, then press F12 to display the contents of the routine header in a new window.

These are the `[Alt]` keystrokes. The Alt key is usually located on either side of the space bar on a personal computer keyboard. When using the `[Alt]` key in SPDEV keystrokes, keep the `[Alt]` key depressed while completing the keystroke. Keystrokes are shown in the order of key pressed.

Table F-16 Alt Keystrokes

Keystroke	Function Performed
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
gold/alt/l	launch a new instance of SPDEV
gold/alt/4	launch console window
alt/i	open .SPINC (include) files
gold/alt/n	bring up NEW FILE dialog box
gold/alt/o	bring up OPEN FILE dialog box

gold/alt/p	purge previous file versions
alt/r	runs the current program
gold/alt/s	save current file
gold/alt/t	open a temporary file
alt/x	clear search bookmarks
gold/alt/z	save open files and exit SPDEV
alt/back	undo
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines

These are the `[Shift]` keystrokes. The Shift key is usually located on either side of a personal computer keyboard. When using the `[Shift]` key in SPDEV keystrokes, keep the `[Shift]` key depressed while completing the keystroke. Keystrokes are shown in the order of key pressed.

Table F-17 Shift Keystrokes

Keystroke	Function Performed
shift/backspace	delete previous character
shift/Caps Lock	toggle Caps Lock key
shift/delete	cut selected text to the clipboard
shift/ctrl/end	selects text from cursor to end of file
shift/ctrl/home	selects text from cursor to beginning of file
shift/Insert	toggle between insert and overstrike modes
shift/right arrow	selects text one character at a time

The keystrokes below are especially helpful to programmers who are writing and documenting code.

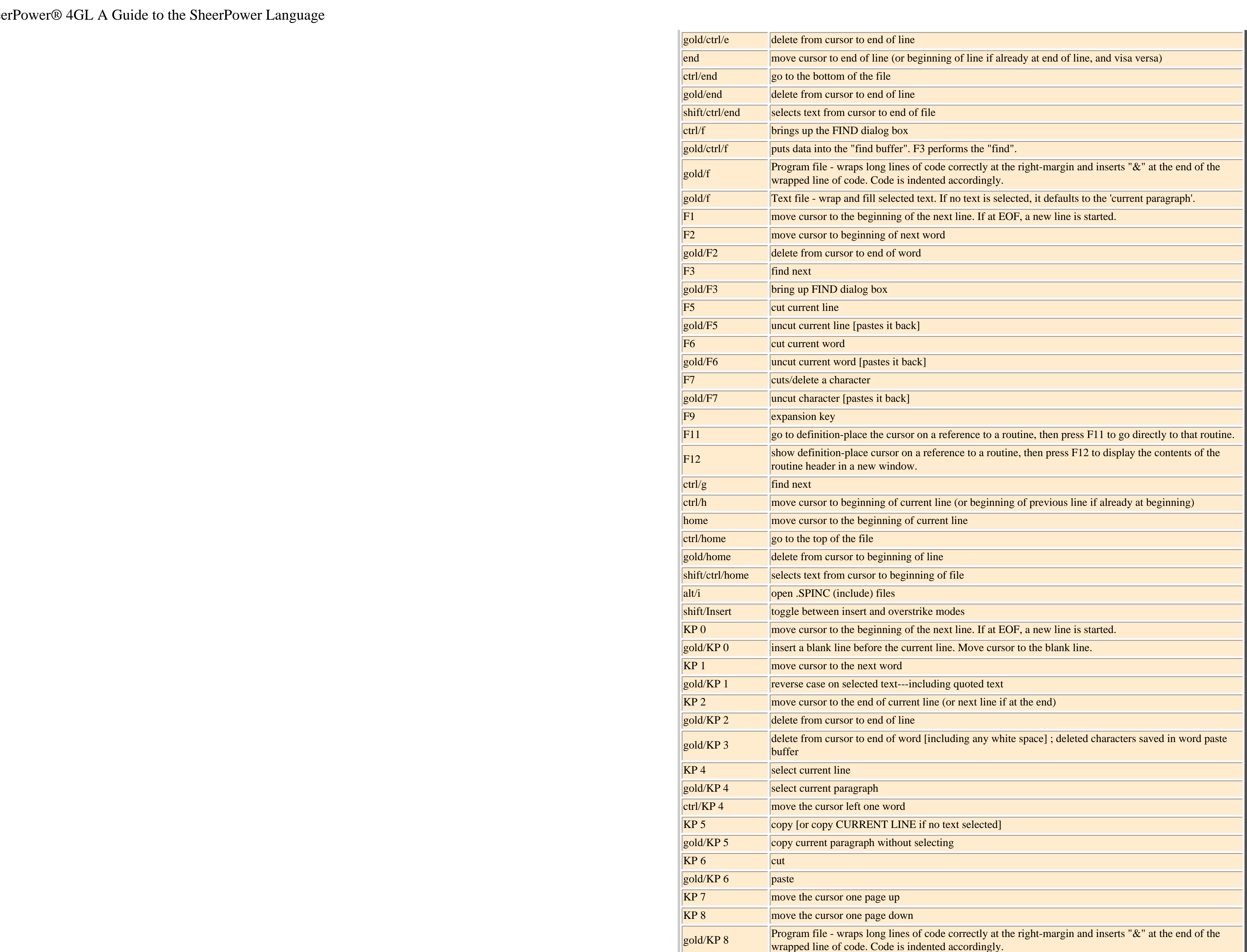
Table F-18 Learn and Macros Keystrokes

Keystroke	Function Performed
gold/l	start/stop recording macro
gold/n	run the macro associated with the number (n = number)
gold/=	repeat next keystroke NNN times [for macros see gold/l]

The following table contains all of the SheerPower Rapid Development keystrokes in alphabetical order.

Table F-19 All Keystrokes - Alphabetical Order

Keystroke	Function Performed
ESCAPE	GOLD key
NUMLOCK	GOLD key
alt/back	undo
gold/alt/4	launch console window
ctrl/a	select all
gold/a	align the "=" and ":" in a group of variable assignments
ctrl/b	toggle between current cursor position and previous cursor position
shift/backspace	delete previous character
ctrl/c	copy (end selection)
gold/c	generate debug comment line of !++ xxx date
shift/Caps Lock	toggle Caps Lock key
gold/d	document a routine in a program
delete	delete the current selection or current character
shift/delete	cut selected text to the clipboard
down arrow	move the cursor one line down
gold/down arrow	convert selection to lower case, skips quoted text
alt/down arrow	prompts for # of lines to move down; moves cursor down that # of lines
ctrl/e	move cursor to end of line (or end of next line if already at end of line)



gold/ctrl/e	delete from cursor to end of line
end	move cursor to end of line (or beginning of line if already at end of line, and visa versa)
ctrl/end	go to the bottom of the file
gold/end	delete from cursor to end of line
shift/ctrl/end	selects text from cursor to end of file
ctrl/f	brings up the FIND dialog box
gold/ctrl/f	puts data into the "find buffer". F3 performs the "find".
gold/f	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/f	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.
F1	move cursor to the beginning of the next line. If at EOF, a new line is started.
F2	move cursor to beginning of next word
gold/F2	delete from cursor to end of word
F3	find next
gold/F3	bring up FIND dialog box
F5	cut current line
gold/F5	uncut current line [pastes it back]
F6	cut current word
gold/F6	uncut current word [pastes it back]
F7	cuts/delete a character
gold/F7	uncut character [pastes it back]
F9	expansion key
F11	go to definition-place the cursor on a reference to a routine, then press F11 to go directly to that routine.
F12	show definition-place cursor on a reference to a routine, then press F12 to display the contents of the routine header in a new window.
ctrl/g	find next
ctrl/h	move cursor to beginning of current line (or beginning of previous line if already at beginning)
home	move cursor to the beginning of current line
ctrl/home	go to the top of the file
gold/home	delete from cursor to beginning of line
shift/ctrl/home	selects text from cursor to beginning of file
alt/i	open .SPINC (include) files
shift/Insert	toggle between insert and overstrike modes
KP 0	move cursor to the beginning of the next line. If at EOF, a new line is started.
gold/KP 0	insert a blank line before the current line. Move cursor to the blank line.
KP 1	move cursor to the next word
gold/KP 1	reverse case on selected text---including quoted text
KP 2	move cursor to the end of current line (or next line if at the end)
gold/KP 2	delete from cursor to end of line
gold/KP 3	delete from cursor to end of word [including any white space] ; deleted characters saved in word paste buffer
KP 4	select current line
gold/KP 4	select current paragraph
ctrl/KP 4	move the cursor left one word
KP 5	copy [or copy CURRENT LINE if no text selected]
gold/KP 5	copy current paragraph without selecting
KP 6	cut
gold/KP 6	paste
KP 7	move the cursor one page up
KP 8	move the cursor one page down
gold/KP 8	Program file - wraps long lines of code correctly at the right-margin and inserts "&" at the end of the wrapped line of code. Code is indented accordingly.
gold/KP 8	Text file - wrap and fill selected text. If no text is selected, it defaults to the 'current paragraph'.

KP /	expansion key
KP *	find next
gold/KP *	bring up FIND dialog box
KP -	cut current line without selecting
KP +	cut current word without selecting
gold/KP +	uncut current word [pastes it back]
ctrl/KP +	cut current character without selecting
gold/ctrl/KP +	uncut current character [paste]
KP Enter	paste
gold/KP Enter	bring up the REPLACE dialog box
KP . (period)	toggle select [SEL] mode
gold/=	repeat next keystroke NNN times [for macros see gold/l]
gold/l	start/stop recording macro
gold/alt/l	launch a new instance of SPDEV
left arrow	move cursor left one character
ctrl/left arrow	move cursor left one word
ctrl/m	insert/remove bookmark
ctrl/n	move cursor to next bookmark
gold/alt/n	bring up NEW FILE dialog box
gold/n	run the macro associated with the number (n = number)
alt/numlock	numbers lock
gold/o	organize routines within source code
gold/alt/o	bring up OPEN FILE dialog box
ctrl/p	move cursor to previous bookmark
gold/p	generate program template
gold/alt/p	purge previous file versions
gold/. (period)	toggle select [SEL] mode
page down	move cursor down one page
gold/page down	move cursor to bottom of file
page up	move cursor one page up
gold/page up	move cursor to top of file
alt/r	runs the current program
ctrl/r	restore previous selection
gold/r	generate routine template
right arrow	move cursor right one character
ctrl/right arrow	move cursor right one word
shift/right arrow	selects text one character at a time
gold/alt/s	save current file
gold/s	sort current column
ctrl/t	trim whitespace and paste
gold/alt/t	open a temporary file
gold/forward slash	comment/uncomment the selected text or current line
ctrl/u	delete from cursor to beginning of line
up arrow	move the cursor one line up
gold/up arrow	convert selection to upper case, skips quoted text
alt/up arrow	prompts for # of lines to move up; moves cursor up that # of lines
ctrl/v	paste
alt/x	clear search bookmarks
ctrl/x	cut selection
ctrl/y	redo
ctrl/z	undo
gold/alt/z	save open files and exit SPDEV

- <form>...</form>
- <input>
    - type
        - text
        - password
        - checkbox
        - radio
        - submit
        - reset
    - name
    - value
    - checked
    - size
    - maxlength
- <select>...</select>
    - name
    - size
    - multiple
- <textarea>...</textarea>
    - name
    - rows
    - cols

- <center>...</center>
- <div>...</div>
- <p>...</p>
-
- ...
- ...
- <hr>
- ...
- <h1>...</h1>
- <h2>...</h2>
- <h3>...</h3>
- <h4>...</h4>
- <h5>...</h5>
- <h6>...</h6>
- ...
- <i>...</i>
- ...
- <pre>...</pre>
- <address>...</address>
- <blockquote>...</blockquote>
- <table>...</table>
- <th>...</th>
- <tr>...</tr>
- <td>...</td>

Table Tag Attributes

- align
- alt
- bgcolor
- border
- bordercolor
- bordercolorlight
- bordercolordark
- cellpadding
- cellspacing
- checked
- color
- cols
- disabled
- height
- img
- maxlength
- multiple
- name

- nowrap
- rows
- selected
- size
- src
- type
- valid
- valign
- value
- wrap
- width
- align (left or right for horizontal alignment, top, texttop, middle, center, bottom and baseline for vertical)
- border
- height
- src
- width
- white
- red
- lime
- blue
- yellow
- fuchsia
- aqua
- silver
- gray/grey
- olive
- purple
- teal
- maroon
- green
- navy
- black
- darkblue

File Edit Display Projects SheerPower Options Advanced Window Help

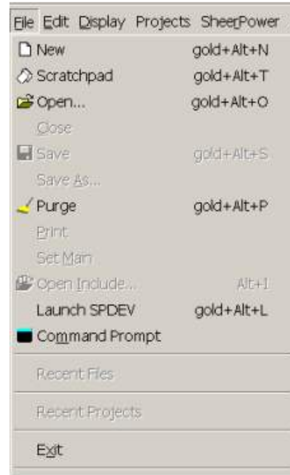


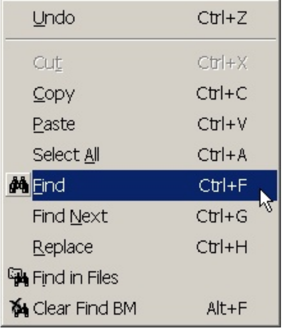
Table H-1 File Menu - Functions

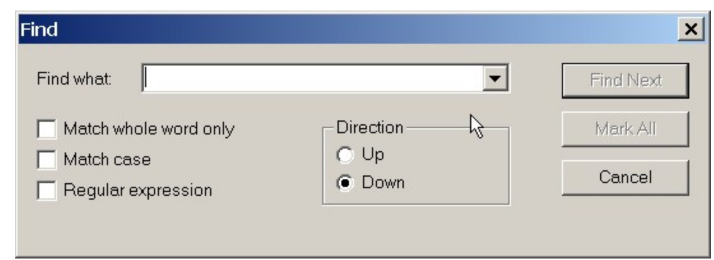
Function	Description
New (GOLD+Alt+N)	Open a new file.
Scratchpad (GOLD+Alt+T)	Open a temporary file.
Open (GOLD+Alt+O)	Open an existing file.
Close	Close an opened file.

Save (GOLD+Alt+S)	Save the current file.
Save As	Save the current file under a new name and/or file type.
Purge (GOLD+Alt+P)	Deletes (purges) individual files and folders containing files of backed up file versions.
Print	Opens the printer software allowing you to print the current file.
Open Include (Alt+I)	Lists all the .SPINC (SheerPower Include) files to choose from to open.
Launch SPDEV (GOLD+Alt+L)	Open a new instance of SPDEV.
Command Prompt	Opens the Command Prompt program.
Recent Files	List of recently opened files.
Recent Projects	List of recently opened projects.
Exit	Close SPDEV.

Table H-2 Edit Menu

Function	Description
Undo (Ctrl+Z)	Undo the last edit in the current file.
Cut (Ctrl+X)	Cuts the selected text to the clipboard.
Copy (Ctrl+C)	Copies the selected text to the clipboard.
Paste (Ctrl+V)	Pastes the contents of the clipboard.
Select All (Ctrl+A)	Selects the entire file.
Find (Ctrl+F)	Find a word or a phrase within the current file.
Find Next (Ctrl+G)	Find next instance of word or phrase in **find**.
Replace (Ctrl+H)	Replace a character, word or phrase with another.
Find In Files	Search multiple files for a word or phrase. You can choose which FIND window to display the results in (Find 1 or Find 2). When the result is displayed in the Find tab you can then double-click on the line displayed and it will open the corresponding file to that exact line.
Clear Find BM (Alt+F)	Clears all find bookmarks (binocular icons).





Inside the **FIND** dialog box you will find the following options:

Table H-3 Find Menu

Function	Description
Find what:	type in the word or phrase you want to find in the current document. You can also click on the down arrow to see a list of previously "found" words/phrases to choose.
Match whole word only	find only the exact specified word/phrase in the file.
Match case	search word/phrase in the exact case specified.
Regular expression	see http://etext.lib.virginia.edu/helpsheets/regex.html for an explanation of regular expressions.
Direction Up/Down	searches the current file either up or down from where the cursor is currently placed.
Find Next	find the next instance of the search word/phrase
Mark All	places bookmark icons in the left margin on each line where the search word/phrase is found.
Cancel	Cancel the current find.

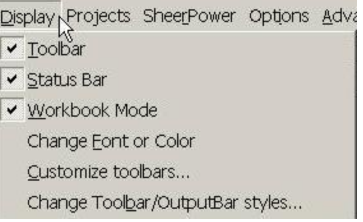


Table H-4 Display Menu

Function	Description
Toolbar	Choose which icons to display and the look of them.
Status Bar	Choose to show the status bar or not.
Workbook Mode	Change the look of the working SPDEV file.
Change Font or Color	Change the font size, font color, background and foreground colors.
Customize Toolbars	Customize what options to display in the toolbar.
Change Toolbar/OutputBar Styles	Change the look of the toolbar.

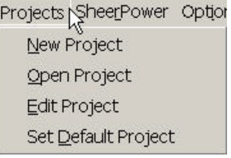


Table H-5 Projects Menu

Function	Description
New Project	Create a new project file (.SPPRJ).
Open Project	Open a project. This will open in SPDEV all the files associated with that project.

Edit Project	Edit the name, files and description of a project file.
Set Default Project	Specify which project is the default, or remove default project. The default project will open in SPDEV each time SPDEV is started.

Table H-6 SheerPower Menu

Function	Description
Setup Data Structures	Create or open/edit an existing data structure file. See Chapter 16, Database Setup for instructions on how to setup a data structure in SheerPower.
Deploy	Create an .SPRUN program file (like an .EXE) where the source code is hidden from the end-user
Deployment Properties	Disable or re-enable the Deployment Password dialog box to appear each time you click on Deploy.
Run	Run the current SheerPower program file.
Launch SheerPower 4GL	Launch a new instance of the console window.

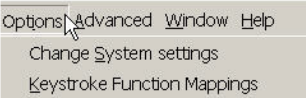


Table H-7 Options Menu

Function	Description
Change System Settings	Options to change a variety of settings within SPDEV. See Section H.6.1, Change System Settings for details.
Keystroke Function Mappings	Customize the keystroke mappings of your keyboard when in SPDEV. See Appendix F, Keystrokes for SheerPower Rapid Development Environment for a complete listing of all SheerPower special editing keystrokes.

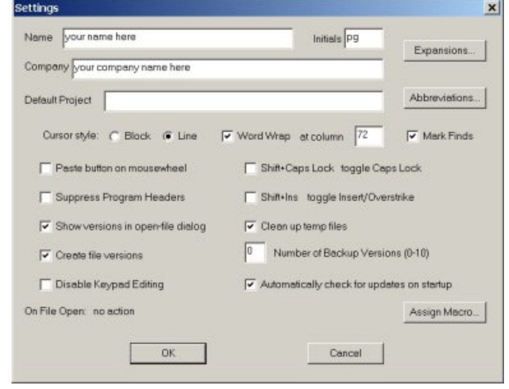
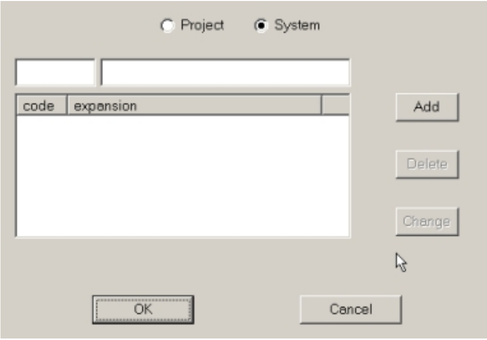


Table H-8 Change System Settings Option

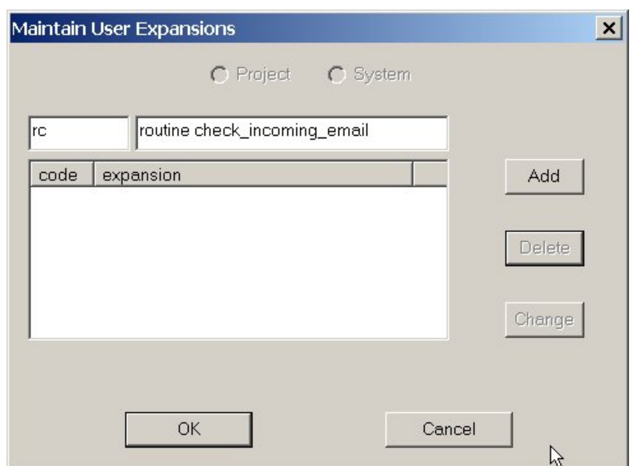
System Setting	Description
Name, Initials and Company	Enter your Name, Initials and Company name inside these fields. This information will be used when creating new programs using the SPDEV program template (GOLD/P).
Default Project	Enter in the name of the **PROJECT** you want set to be the default.
Cursor style	Choose either block or line style cursor.
Word Wrap	You can enable SPDEV to wrap lines of code/text at a specific number of characters. The default is set to 78 characters.
Paste button on mousewheel	Enable the mousewheel to act as a paste button.
Suppress Program Headers	Disables the automatic creation of a program template when creating a new .SPSRC file
Show versions in open-file dialog	Display the backup file versions in the **Open-File** dialog box. SPDEV backs up the last 3 saved versions of your files by inserting a **-1, -2, or -3** behind the file name. For example, program.spsrc would have 3 saved versions called program.spsrc-1, program.spsrc-2 and program.spsrc-3 stored. You can go back to a prior version should there be any errors or unwanted changes in the current file.
Create file versions	Enable/Create backup file versions.
Ctrl+Ins toggle Insert/Overstrike	Cause `[Ins]` (Insert) to be active only when `[Ctrl]` is pressed first.
Automatically check for updates on startup	Enable SheerPower to check for program updates automatically when you start the program SPDEV.
Mark Finds	Marks the lines where words or phrases are "found" in a file (using the FIND function) with a binocular icon.
Ignore Caps Lock	Disable Caps Lock on your keyboard when editing files in SPDEV.
Clean up temp files	An advanced setting. Always leave this checkbox "checked" unless you are working with SheerPower technical support and they require you uncheck this setting.
Expansions	Open dialog box to enter in custom expansions.
Abbreviations	Open dialog box to enter in custom abbreviations.



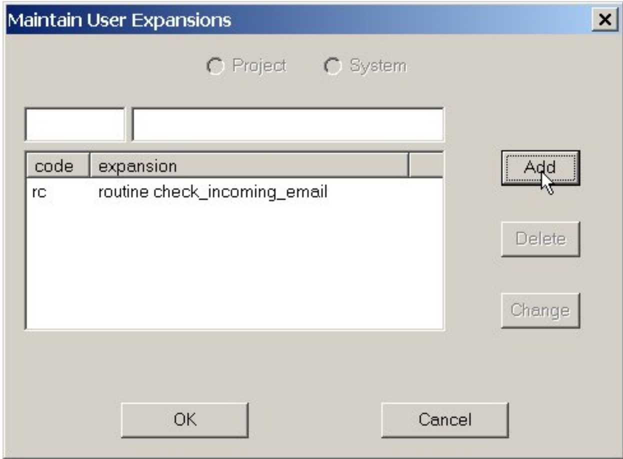
Expansions can be used to speed up programming in SheerPower. To create an expansion, click on **Options** in the SPDEV toolbar, then choose **Change System Settings**. The **Settings** dialog box will appear. Click on the **Expansions...** button.

Inside the **Maintain User Expansions** dialog box in the top empty fields you can type in the expansion code and what it 'expands' to. For example:

```
rc     routine check_incoming_email
```



Then click on the [Add] button.



Once the expansion is added in you can **Delete** or **Change** it by clicking on either the [Delete] button or the [Change] button.

To use the expansion inside any type of file in SPDEV, simply type in the expansion code, then press the [F9] key on your keyboard.

```
rc --> press F9 --> routine check_incoming_email
```

You can insert "\n" inside the expansion if you define the entire expansion as a literal string. For example, to insert:

```
print '1'
print '2'
```

The expansion would be:

```
print '1'\nprint '2'
```

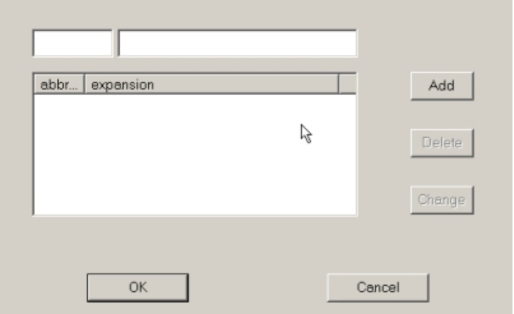
To insert new line codes outside of literals, use the token **newline$**. For example, to insert:

```
print '1'
print '2'
```

The expansion string would be:

```
"print '1'" + newline$ + "print '2'"
```

Similarly, tab$ inserts tabs.

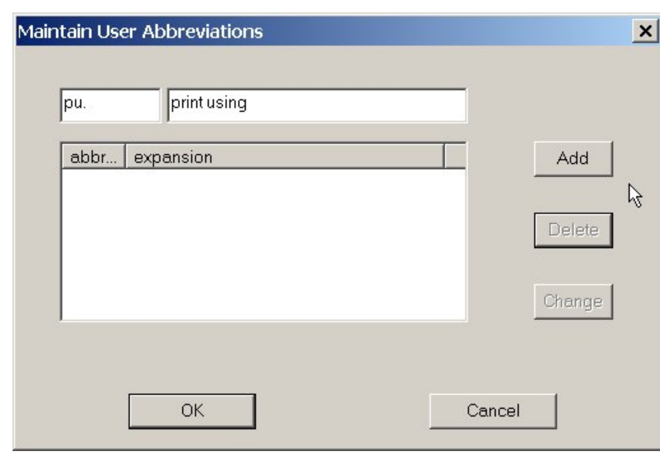


Custom **abbreviations** can also be programmed into SPDEV to make programming repititious code or text faster. To add a custom abbreviation, click on **Options** in the SPDEV toolbar, then select **Change System Settings**. The **Settings** dialog box will appear. Click on the **Abbreviations...** button. The **Maintain User Abbreviations** dialog box will appear.

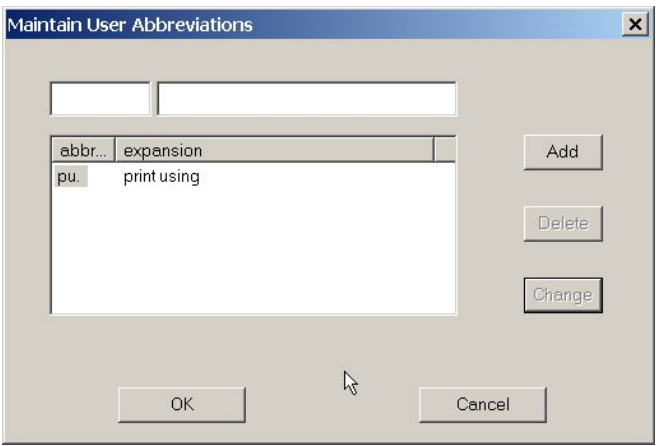
Inside the empty top fields of the dialog box you can enter in the abbreviation code on the left.

For example:

```
pu   print using
```



Then click on the [Add] button.



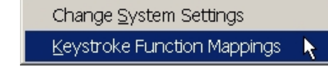
Once the abbreviation is added in you can **Delete** or **Change** it by clicking on either the [Delete] button or the [Change] button.

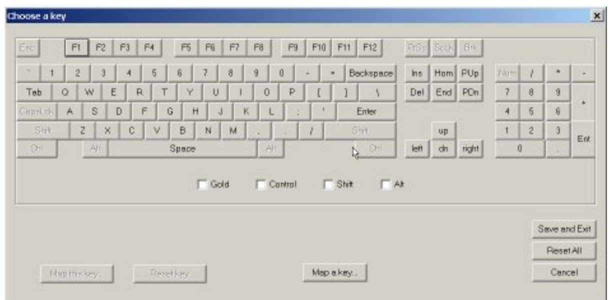
To use the abbreviation inside any type of file in SPDEV, simply type in the abbreviation, **followed by a period "."**, then press the [Tab] key on your keyboard.

```
pu. --> press Tab --> print using
```

SheerPower Rapid Development Environment has many keystrokes mapped to simplify and speed up program creation and file editing. A 'mapped keystroke' means that the regular function of a certain key may not perform the same function while in SheerPower Rapid Development Environment. For example, KP0 - means the number 0 (zero) in the numeric keypad of a keyboard. KP stands for keypad. In other programs, KP0 will print a 0 (zero). In SPDEV, KP0 moves the cursor to the beginning of the next line. At the end of a file, it will create a new line at the bottom of the file.

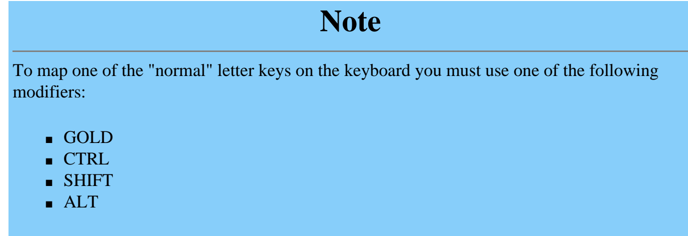
For a complete listing of specially mapped keystrokes in SPDEV, see Appendix F, Keystrokes for SheerPower Rapid Development Environment.





SPDEV allows you to customize your keyboard map to suit your specific needs and style of programming and editing. To edit the keymap, click on **Options** in the SPDEV toolbar, then select **Keystroke** Function Mappings. The **Choose a key** dialog box will appear.

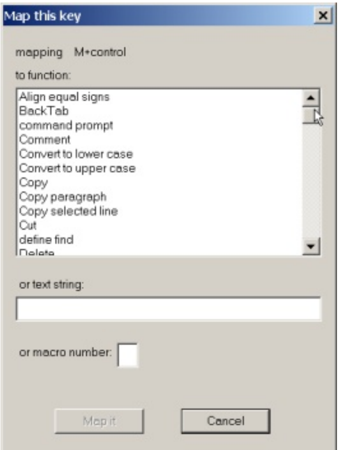
To map a key, click on the key you want to map inside the keyboard image. Then click on the **Map this key** button.



Note

To map one of the "normal" letter keys on the keyboard you must use one of the following modifiers:

- GOLD
- CTRL
- SHIFT
- ALT



The **Map this key** dialog box will appear. You can choose to map the keystroke to a particular function from the list displayed, or enter a **text string** or **macro number**.

When you have chosen what you want to map the keystroke to, click on the **Map it** button. You can then map a different keystroke, or click on the **Save and Exit** button to save your keymap changes and exit the dialog box.

At anytime you can reset a single keystroke or the entire keymap back to the default keymap by clicking on the **Reset key** or **Reset All** button.

You must always click on the **Save and Exit** button to save any changes that you made.

In SPDEV you can assign a **BYTE VALUE** to any keystroke. Click on **Options** in the SPDEV toolbar, then select **Keystroke Function Mappings**.

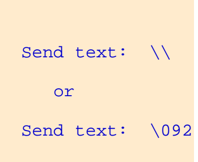
Place a checkbox beside the modifier (GOLD/ALT/CTRL) you want to use, then click on the letter that you want to map the

byte value to. Now click on the **Map this key** button. The **Map this key** dialog box will appear.

Inside the **Send text:** field, type in the byte value that you want to assign to the keystroke. For example, to generate a form feed:

```
Send text: \012
```

If you want to generate a BACKSLASH, you would:



```
Send text:  \\
   or
Send text:  \092
```

Below is a list of all the different functions that you can map a keystroke to:

Table H-9 Keystroke Functions

Function
Align equal signs
Back Tab
Command prompt
Comment
Convert to lower case
Convert to upper case
Copy
Copy paragraph
Copy selected line
Cut
Define find
Delete
Delete current character
Delete current line
Delete current word
Delete from caret (cursor) to next word
Delete to beginning of line
Delete to end of line
Delete to end of word
Document routine
Find
Find next
Fixup Right Margin (default: current paragraph)
Fixup Right Margin (default: current paragraph, no fill)
Fixup Right Margin (default: current line)
Fixup Right Margin (default: current line, no fill)
Generate debug trace
Go down
Go to definition
Go up
Gold
Highlight line
Highlight paragraph

Insert blank line prior
Launch console window
Launch SPDEV
Macro:
Move down one line
Move left one character
Move left one word
Move right one character
Move right one word
Move to beginning of line
Move to beginning of next line
Move to beginning of this or previous line
Move to bottom of file
Move to end of line
Move to end or beginning of line
Move to top of file
Move up one line
New file
Next bookmark
Open file
Open temp file
Organize code
Page Down
Page Up
Paste
Paste current character
Paste current line
Paste current word
Previous bookmark
Program template
Purge files
Redo
Repeat
Replace
Restore selection
Reverse case
Routine template
Save file
Select all
Show definition
Sort text
Start recording a macro
Substitute expansion
Tab
Text
Toggle bookmark
Toggle select mode
Trim paste
Undo
Where was I?

Note

These advanced functions should only be used when assisting technical support with problem solution. Advanced features are not intended to be used outside of resolving technical support issues.

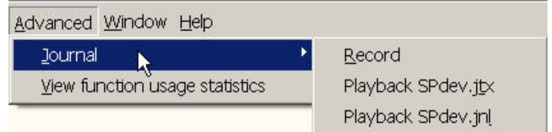


Table H-10 Advanced Menu

Function	Description
Journal	Records all mouse and keystroke movements within SPDEV. This is used for SheerPower development debugging.
View Function Usage Statistics	Displays the number of times each special keystroke function has been used during the current session in SPDEV.

Note

Maximize All is the default setting for the window display.

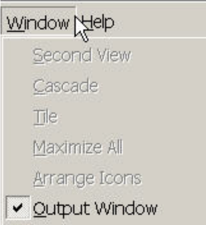


Table H-11 Window Menu

Function	Description
Second View	View the current file inside a new window in SPDEV.
Cascade	View all open files in SPDEV in smaller cascading windows.
Tile	View all open files in tile format inside SPDEV.
Maximize All	View all open files in maximized view. This is the default window display setting.
Arrange Icons	Arranges minimized icons.
Results Window	Hide or display the Results Window at the bottom of the SPDEV window. The Results Window contains the following tabs: Find 1 and Find 2: When using the **Find in Files** feature (Edit-->Find in Files) you can choose which Find tab to display the results in. Build: Displays the build status of the last .SPSRC program file run or deployed during the current session in SPDEV. Purge: Displays the names of the backed up file versions successfully purged.

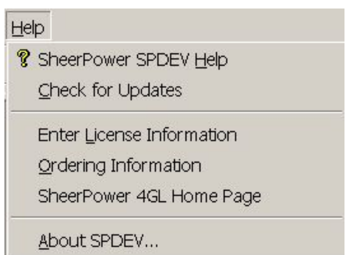


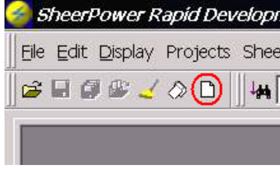
Table H-12 Help Menu

Function	Description
SheerPower SPDEV Help	Opens a new browser window containing links to the online SheerPower 4GL documentation and downloadable .PDF version.
Check for Updates	Checks if you are running the latest version of SheerPower.
Enter License Information	Copy and paste your SheerPower 4GL License Information into this dialog box.
Ordering Information	Opens a new browser window containing SheerPower 4GL GOLD License purchase information.
SheerPower 4GL Home Page	Opens the SheerPower 4GL website in a new browser window.
About SPDEV	Displays the current version and build number of SPDEV.

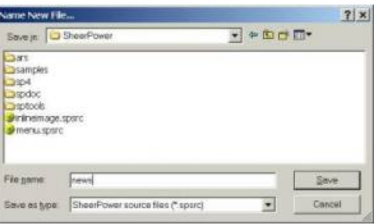
The purpose of this example is to illustrate the special keystrokes and other features of SheerPower Rapid Development Environment that enable a programmer to easily and quickly create professional programs.

To start SheerPower Rapid Development Environment (SPDEV), double click the **SheerPower** shortcut icon located on your desktop---a circle with a lightning bolt through it.

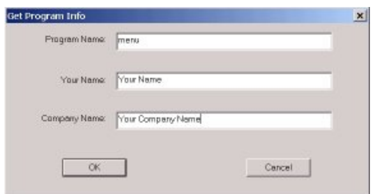
To create a **new program** in SheerPower Rapid Development Environment, click once on the **New** icon in the toolbar---a white paper with one corner folded.



Name your new program file **news.spsrc** and click on `[SAVE]` inside the **Name New File...** dialog box.



The **Get Program info** dialog box will also appear prompting you for your name, company name and program name. The program name will already be filled in for you.



When you fill in this information and click on `[OK]`, a **program template** will automatically be inserted into your new program file.

> ## Note
>
> You can change your name and/or company name at anytime by clicking on **Options** in the SPDEV toolbar, then selecting **Change System Settings**.

You are now ready to write a professional application in SheerPower.

A **PROGRAM TEMPLATE** can also be created *instantly* inside a program file by using the **GOLD/P** keystroke inside SheerPower Rapid Development Environment (SPDEV).

The **GOLD Key** referred to in this section is a special key used to create many of the specialized keystrokes within SPDEV. Both the `[Esc]` (escape key - top left corner of the keyboard) and the `[Num-Lock]` (numbers lock key in the numeric keypad) are GOLD keys in SheerPower Rapid Development Environment.

To use the GOLD key, press either GOLD key (`[Esc]` or `[Num-Lock]`), let go, then continue with the rest of the keystroke to complete the function.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! Program: Name of program
! System :
! Author : Your name
! Company: Company name
! Date   : June 30, 2002
! Purpose:
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!         I n i t i a l i z a t i o n
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!         M a i n   l o g i c   a r e a
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



stop


!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!         R o u t i n e s
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



end
```

Your cursor will be positioned at the top section of the template beside **Purpose:**. You can type in the purpose of the program being written here. The purpose of **news.spsrc** (this example program) is: **To present the top news story from CNN.COM**.

SPDEV automatically fills in the name of the program, the author and company names, and the date for you when the template is created.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! Program: news
! System :
! Author : Your name
! Company: Company name
! Date    : June 30, 2002
! Purpose: To present the top news story from CNN.COM
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Copy and paste the following SheerPower code underneath the **Initialization** heading.

```
// Note:  From time to time CNN changes its format, so the
//        main$  and  end_main$  sometimes have to be changed.

main$ = '<div class="CNN_homeBox">'
end_main$ = '</div>'

begin_form$ = '<form>' +
              '<h1><font color=green>' +
              'Top News from CNN' +
              '</font></h1>' +
              '<br><h2>'
```

Copy and paste the following code underneath the **Main logic area** heading above the word **stop**.

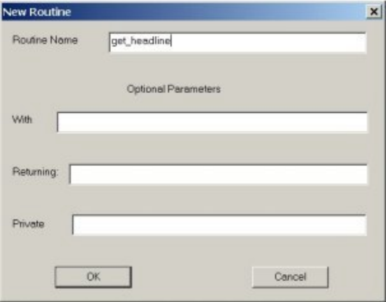
```
news_ch = _channel
open #news_ch: name 'http://www.cnn.com'
get_headline
```

Now you are ready to write the **routines** that create **news. spsrc**.

Programs are made up of **ROUTINES**. Each routine has its own heading with details on what the purpose of the routine is, and how it works.

Creating a Routine Header Template

Place your cursor underneath the **Routines** heading. Press the `[GOLD]` key (either `[Esc]` or `[Num-Lock]`), and let go. Then press `[R]`. You will see the following prompt:



Type **get_headline** inside the empty field beside **Routine Name:**.

Note

All routine names must contain at least one underscore (_).

The **Optional Parameters** section contains the following fields:

Table I-1 Routine Template-Optional Parameters

Parameter	Description
With:	parameters being passed into the routine.
Returning:	parameters returned as a result of running the routine.
Private:	variables that can only be accessed from inside this routine.

You can leave the Optional Parameters fields blank for the purpose of this tutorial.

Note

See Section 3.5, Passing Optional Parameters to Routines for more on passing parameters in SheerPower.

Click on [OK]. The **ROUTINE HEADER TEMPLATE** will appear.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!
! Brief description:
!
!
! Expected on entry:
!
!
! Locals used:
!
!
! Results on exit:
!
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
routine get_headline

end routine
```

There are four fields to complete inside the Routine Header Template:

- **Brief description** - describe the purpose of the routine.
- **Expected on entry** - list variable names and the data they store that are *coming into the routine* (already defined).
- **Locals used** - list any variables defined in this routine only and the data they store.
- **Results on exit** - list any variables defined in this routine that will be used elsewhere in the program. Include what the result of the routine will be on exit.

Here is the completed routine header for this routine:

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! g e t _ h e a d l i n e
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!
! Brief description:
!   Go out to www.cnn.com and extract the headline of the top
!   news story.
!
! Expected on entry:
!   main$       = the starting point of the headline
!   begin_form$ = the top HTML code for the dialogbox form
!   end_main$   = the end point of the headline
!
! Locals used:
!   text$  = the HTML code on the CNN website
!   crlf$  = contains the character line feeds
!   state$ = tells which action needs to be
!            performed next on the incoming data
!
! Results on exit:
!   dbox$  = stores the code to create the dialogbox
!
!   The headline is found and the form is created.
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Copy and paste the following SheerPower code into the **get_headline** routine on a blank line in between **routine get_headline** and **end routine**. The first line of code is indented 2 spaces under the routine statement with the rest of the code indented accordingly below.

```
  crlf$  = chr$(13) + chr$(10)
  state$ = 'find_main'
  do
    line input #news_ch, eof eof?: text$
    if eof? then exit do
    select case state$
    case 'find_main'
      z0 = pos(text$, main$)
      if z0 = 0 then repeat do
      z0$ = mid(text$, z0+len(main$))
      z0  = pos(z0$, '>')
      if z0 = 0 then repeat do
      z0$ = mid(z0$, z0+1)
      dbox$ = begin_form$ +z0$
      state$ = 'gather_news'
    case 'gather_news'
      z0 = pos(text$, end_main$)
      if z0 > 0 then
        dbox$  = dbox$ + left(text$, z0-1)
        state$ = 'show_news'
        repeat do
      end if
      dbox$ = dbox$ + text$ + crlf$
      repeat do
    case 'show_news'
      // first add in the  http://www.cnn.com  to any anchors
      dbox$ = replace$(dbox$, '<a href="==<a href="http://www.cnn.com/',',', '==')
      // now finish off the input form with a single "submit" button
      dbox$ = dbox$ + '</h2><input type=submit></form>'
      line input dialogbox dbox$: ans$
      exit do
    end select
  loop
```

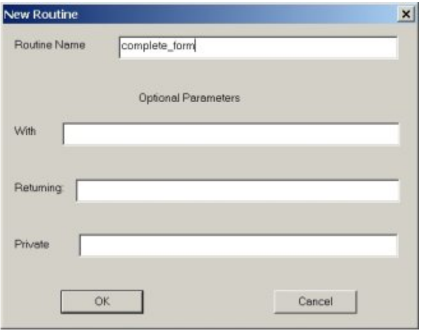
Often routines can get to be too long and complicated. SheerPower makes it easy to make subroutines from within a routine. Just highlight the code you want to move into a subroutine and press `[GOLD][R]`. You will be prompted to type in a name for the subroutine. Type in the name and press `[OK]`. SheerPower will move the highlighted code into a new subroutine below the original routine instantly.

In the news.spsrc example, we will illustrate how this is done.

Inside the news.spsrc program file, highlight the following section of SheerPower code near the bottom of the program:

```
    // first add in the  http://www.cnn.com  to any anchors
    dbox$ = replace$(dbox$, '<a href="==<a href="http://www.cnn.com/',',', '==')
    // now finish off the input form with a single "submit" button
    dbox$ = dbox$ + '</h2><input type=submit></form>'
```

Now press `[GOLD][R]`. The **New Routine** dialog box will appear. Type the name of the subroutine being created as **complete_form**.



You can leave the Optional Parameters fields blank for the purpose of this tutorial.

Click on `[OK]`. SheerPower will take the highlighted code from the original routine, and create a new routine below the first one containing the highlighted code.

SheerPower will also insert the **CALL** for the subroutine inside the original routine.

The completed routine header for **complete_form** is as follows:

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! c o m p l e t e _ f o r m
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!
! Brief description:
!   Add the CNN URL to anchors and complete the form with the submit
!   button.
!
! Expected on entry:
!   dbox$ = stores the code to create the dialogbox
!
! Locals used:
!   none
!
! Results on exit:
!   Any anchors are now complete with the CNN URL and the form
!   is completed.
!
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

RUN this program by clicking on the **Run** icon in the SPDEV toolbar---the running man.

The program goes out to www.cnn.com and grabs the latest headline for the top news story, then displays it in a form window with a clickable link to the complete story.

To make it easier to locate routines and find out what they do inside large programs, SPDEV has two specially mapped keys to assist:

F12: Show definition

F11: Go to definition

The **F12** key displays the routine header information inside a new window. This feature especially benefits maintenance programmers.

You can place your cursor on the reference to *any routine name* inside a program, press `[F12]` and a new window will appear containing the routine definition (header information).

Inside the example program **news.spsrc** place your cursor on the reference to the routine **get_headline** in the **Main logic area**.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!        M a i n   l o g i c   a r e a
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
news_ch = _channel
open #news_ch: name 'http://www.cnn.com'
get_headline    //<--- place your cursor on this routine reference

stop
```

Now press the `[F12]` key. The following new window will appear:

```
get_headline
Brief description:
  Go out to www.cnn.com and extract the headline of the top
  news story.
Expected on entry:
  main$       = the starting point of the headline
  begin_form$ = the top HTML code for the dialogbox form
  end_main$   = the end point of the headline
Locals used:
  text$  = the HTML code on the CNN website
  crlf$  = contains the character line feeds
  state$ = tells which action needs to be
  performed next on the incoming data
Results on exit:
  dbox$  = stores the code to create the dialogbox
  The headline is found and the form is created.
```

The special **F12** keystroke allows you to find the details about any routine in a program without having to go to the actual routine.

The **F11** key takes you directly to a routine when you place your cursor on the reference to that routine.

Inside the example program **news.spsrc** place your cursor on the reference to the routine **get_headline** in the **Main logic area**.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!         M a i n   l o g i c   a r e a
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
news_ch = _channel
open #news_ch: name 'http://www.cnn.com'
get_headline    //<--- place your cursor on this routine reference

stop
```

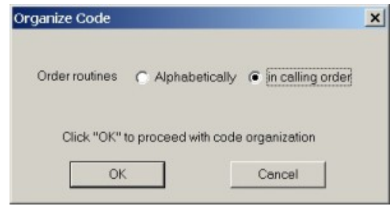
Now press the [F11] key. You will automatically be taken to the start of the get_headline routine.

To get back to where you were simply press **GOLD/B**. See Appendix F, Keystrokes for SheerPower Rapid Development Environment for more details on all the special programming keystrokes built into the SheerPower Rapid Development Environment.

Routines can become logically out of order as a program is being written, making it hard to follow program logic and hard to find a given routine.

In SPDEV, the **GOLD/O** keystroke lets you automatically order your routines in either **calling order** or **alphabetical order**.

Inside the example program **news.spsrc** place your cursor anywhere and press the **GOLD** key ([Esc]), let go, then press the **O** key. The **Organize Code** dialog box will appear.



The routines in news.spsrc are currently organized in calling order. Select **Alphabetically**.

Routine complete_form will now be first in the program, and get_headline will be last. Press GOLD/O again and choose **in calling order**. Then click on [OK]. The routines will now be organized in their calling order.

See Appendix F, Keystrokes for SheerPower Rapid Development Environment for more details on all the special programming keystrokes built into the SheerPower Rapid Development Environment.

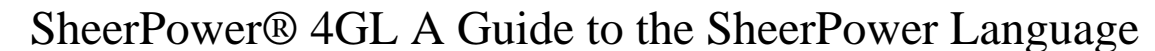
The following programs have been included in the distribution of SheerPower. They can be found in the **Samples** folder inside your SheerPower folder---c:\sheerpower\samples folder. (The default install location for SheerPower 4GL is c:\sheerpower.)

We encourage you to:

- Create the same program in other languages and compare programming ease, functionality and results to those of SheerPower!
- Make modifications to the programs to see how the SheerPower commands and statements work. (You may want to make a working copy first and save the original!)
- Use these sample programs as 'how-to' tools.

Table J-1 Program File List and Description

Program Name	Description
CDPLAYER.SPSRC	This program illustrates SheerPower making Windows API calls. In this case it is controlling the PC CD player.
CLIENT_SCAN.SPSRC	This program scans through a database of clients looking for names similar to what is being searched for.
CQUERY.SPSRC	Shows how to do a simple query on a database using INPUT DIALOGBOX to display the form, allowing for key lookups and updating data fields.



DDETEST.SPSRC	Have SheerPower control an Excel spreadsheet!
EVAL.SPSRC	In this program, the user enters a complex mathematical expression. SheerPower evaluates the expression and returns the solution.
KBTUTOR.SPSRC	This is a complete program to develop keyboard skills and touch typing methods. It tracks accuracy and skill levels providing instant feedback to the user. Try it and find your keyboard skill level!
MAKE_HTML.SPSRC	This program generates simple HTML and opens up a new browser window upon completion.
MULTI_TABLE_DIALOGBOX_STATEMENT.SPSRC	This sample program illustrates accessing multiple tables in a database, then putting the data into a form using INPUT DIALOGBOX. This program can be found in multi-table.zip.
MULTI_TABLE_HTML_STATEMENT.SPSRC	This program illustrates how to access data from multiple tables in a database, then printing the data to an .HTML file... then opens the HTML file using PASS URL. This program can be found in multi-table.zip.
MULTI_TABLE_SCREEN_STATEMENT.SPSRC	This program shows how to access multiple tables in a database, then print out the data to the screen. This program can be found in multi-table.zip.
NEWS.SPSRC	In this program, SheerPower accesses CNN.COM and returns back the current top news story with a clickable link that opens a new browser window to the URL containing the rest of the story.
PLAY_SOUND	This program illustrates SheerPower playing .wav sound files using the MEDIA statement.
PRECISION.SPSRC	An example of SheerPower's perfect precision. Try this with your current programming language.
QUIZ.SPSRC	A simple quiz program showcasing SheerPower's strengths in dynamic form generation and database access. Includes feedback response to both correct and incorrect responses.
SEND_EMAIL.SPSRC	This program generates a form that allows the user to create and send an email.
SET_ICON.SPSRC	This program shows you how to dynamically change the taskbar icon.
SIEVE.SPSRC	This is a benchmark program that generates a count of prime numbers.
STOCKQUOTE.SPSRC	Enter a stock symbol or a list of stock symbols. SheerPower will return the current stock price quote for each symbol entered.

Table K-1 What is a 4GL?

GL	Description
GL	"Generation Language"
1GL	No one uses this anymore to program in. To program in 1GL a person would be physically at a computer with a bunch of switches literally programming into the computer 1's and 0's by pressing the switches manually.
2GL	Also known as "Machine Code", "Compute Instructions", and "Assembler". A 2GL language tells the computer precisely what to do. Instructions are given to the actual CPU.
3GL	Java, Cobol, Basic, C++ are all 3GL languages. The instructions given to the computer are more generalized. More words are used than instructions. The type of computer or memory location are not given using a 3GL language. Compiling and linking converts the code to 2GL which then converts to the 1GL to run the program.
4GL	Is not a precise programming language. A 4GL programming language makes assumptions on the command given. A 4GL programming language is faster, but the results very generalized. "Sure is easy to do things, but I can't do anything with it!" If a programmer wants to override the assumption and get a specific result, they need to use a 3GL language.

SheerPower 4GL is both a 3GL and a 4GL---that one is of the reasons why SheerPower 4GL is so powerful!!

SheerPower applications can easily be "web-enabled" through its simple CGI Interface. The CGI interface works with SheerPower's SPINS_WEBSERVER. SPINS_webserver.exe is included in the download of SheerPower 4GL. It is automatically installed to:

```
\sheerpower\sphandlers\
```

See Chapter 18, SheerPower Internet Services (SPINS) Webserver for the complete chapter on SPINS webserver.

The following is a checklist to go through if you experience problems with the SheerPower CGI Interface and SPINS webserver.

- Section L.1.1, Stop Microsoft IIS Webserver
- Section L.1.2, Test SPINS_Webserver
- Section L.1.3, Specify a Different Port Number
- Section L.2, Testing the CGI Interface

To run SPINS webserver, the Microsoft IIS webserver must be stopped or a different port specified for either webserver. Before running SPINS webserver for the first time:

- Stop Microsoft IIS webserver:

  Start --> Settings --> Control Panel --> Administrative Tools --> Internet Information Services

  Right-click on your Web Site from the list on the left (or Default Web Site) and choose **Stop**

  The web site name will now display in the list with a red circle with a white "x" through it to show that it's stopped.
- Delete the old **/sheerpower/sphandlers/spiis.gblpool** file.
- Open the Command Prompt program and type in the following command:

  c:> **iisreset**

  The Command Prompt window will display:

  ```
  Attempting stop...
  Internet services successfully stopped
  Attempting start...
  Internet services successfully restarted
  ```

  But the IIS webserver will remain stopped.
- Run SPINS_webserver by double-clicking on \sheerpower\sphandlers\spins_webserver.exe. Currently SPINS runs in a Command Prompt window.

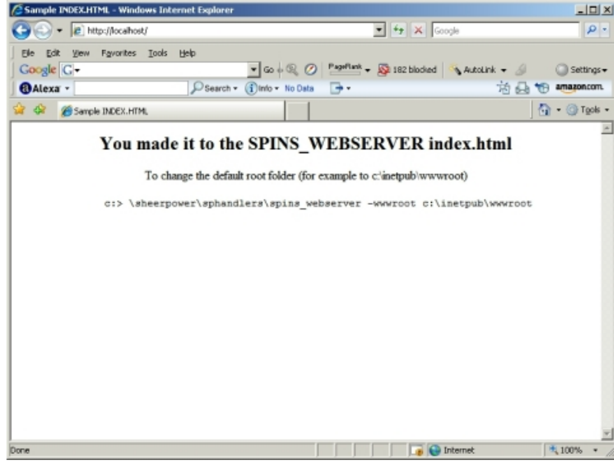
To test SPINS webserver:

- Make sure SPINS_webserver.exe is running (default location is \sheerpower\sphandlers\spins_webserver.exe).
- Open a browser window and go to:

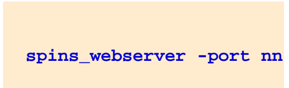
  ```
  http://localhost
  ```

This will open up the .HTML file located in:

```
sheerpower/sphandlers/wwwroot/index.html
```



To tell the SPINS webserver to use a different port number from the default port 80, start it from the **Command Prompt** program or with a **PASS NOWAIT** statement within your program using the following syntax:



```
spins_webserver -port nn
```

For example, to change from the default port 80 to port 8080:

```
spins_webserver -port 8080
```

to use **port 8080**.

Running SPINS and IIS Simultaneously

If you need to utilize some of the IIS facility, you can set the IIS webserver to use port 8080 and the SPINS websever to use port 80 (or the other way around). Both webservers can co-exist this way.

The sample CGI program **eval_handler.spsrc** is located in:

```
c:\sheerpower\sphandler
```

The webpage that goes along with the sample program is located in the same folder, and is called **cgi.html**.

First, using Windows Explorer, double-click on the EVAL_HANDLER.SPSRC file. This runs the EVAL_HANDLER program. This program by default is located in:

```
c:\sheerpower\sphandlers\eval_handler.spsrc
```

After the EVAL_HANDLER has been started, you can try the FORM below.

After entering an expression, press the **ENTER key**. To get back to this webpage, click on the browser's **BACK button**.

Enter an expression, like **sin(355/113)** and press the **ENTER key**

Each time anyone enters an expression to evaluate (like 2+3), their browser sends the data to your SPINS webserver along with a handler name. In this example, the handler name is EVAL. The SPINS websever then passes the data to the EVAL_HANDLER.SPSRC SheerPower program. The EVAL_HANDLER handles the request, figures out the result, and sends the result back to the SPINS server. The SPINS server then sends the result back to the browser.

In order to handle CGI requests, the following steps need to be taken:

- Open the CGI connection to the SPINS server
- Wait for a request
- Handle timeouts
- Process the request

First we open our CGI connection to the SPINS server:

```
handler_name$ = 'cgi://EVAL'
open file cgi_ch: name handler_name$
```

In this example, **EVAL** is the **HANDLER NAME**. For higher performance, you can run as many copies of this handler as you wish. The SheerPower CGI interface will queue all requests to these handlers that have the form:

```
http://www.ttinet.com/scripts/spiis.dll/EVAL
```

Next we set up a logic loop to wait for requests, handle timeouts, and process each request:

```
do
  line input #cgi_ch: method$
  if method$ = '' then repeat do
  ..
  ..
loop
```

The **LINE INPUT** waits for a request from the SPINS server. When the **LINE INPUT** completes, the variable **METHOD$** will contain one of three values.

- A null string (""), meaning that there was no request from the SPINS server for at least five seconds.
- The string "POST" if a form used the POST method.
- The string "GET" if a form or URL used the "GET" method.

In this program, when there is no request from the SPINS server (a timeout), we just go back and try again. In complex applications a program might instead unlock databases, write out statistics, and then go back and try again.

Now that we have a request from the SPINS server, we have to process the request.

```
ask #cgi_ch, symbol 'EXPR': value expr$
if  expr$ = '' then
  print #cgi_ch: '&lt;h2&gt;Thank you for using the Evaluator!&lt;/h2&gt;'
  repeat do
end if

when exception in
  answer = eval(expr$)
use
  answer = extext$
end when
if dtype(answer) <> 1 then answer = str$(answer)
print #cgi_ch: '&lt;h2&gt;'; expr$; ' --> '; answer;
'&lt;/h2&gt;'
ask #cgi_ch, symbol 'env:REMOTE_ADDR': value ipaddr$
print #cgi_ch: '(Your IP address is '; ipaddr$; ')'
```

This form returns one form variable---the expression to be evaluated (EXPR). We use the SheerPower **ASK** instruction to ask for its value.

```
ask #cgi_ch, symbol 'EXPR': value expr$
```

If they didn't enter any expression, we just tell them "Thank you...". This is done using the **PRINT** instruction.

```
  print #cgi_ch: '&lt;h2&gt;Thank you for using the Evaluator!&lt;/h2&gt;'
```

Using **EXPR$**, we calcluate the **ANSWER** and **PRINT** the result back to the SPINS server.

```
print #cgi_ch: '&lt;h2&gt;'; expr$; ' --> '; answer; '&lt;/h2&gt;'
```

Finally, we ask the SPINS server for the "REMOTE_ADDR". This is the IP address of the requestor. Since "REMOTE_ADDR" is an **environmental variable**, we must put "env:" in front of the symbol name. Once we get the IP address, we **PRINT** it to the SPINS server---which, in turn, sends the data back to the browser.

```
ask #cgi_ch, symbol 'env:REMOTE_ADDR': value ipaddr$
print #cgi_ch: '(Your IP address is '; ipaddr$; ')'
```

By default, SheerPower automatically builds the CGI required HTTP headers. But, sometimes there is a need to write out your own custom HTTP headers. For example, you might need to write out binary data, or perhaps send a cookie. Here is a sample routine that writes out a HTTP header:

```
  routine start_http_output
    ask #cgi_ch, symbol 'env:PATH_INFO': value path_info$
    print #cgi_ch: "HTTP/1.1 200 OK"
    print #cgi_ch: 'Cache-Control: max-age=2, must-revalidate'
    print #cgi_ch: "Content-type: text/html"
    print #cgi_ch: 'Referer: ' + path_info$
    print #cgi_ch: "Pragma: no-cache"
    print #cgi_ch:
    print #cgi_ch:
  end routine
```

Note

When printing out binary data, include a trailing semi-colon at the end of the **PRINT** instruction:

print #cgi_ch: mybinary$; // notice the trailing semi-colon

The SheerPower CGI interface was designed to very high performance. If the server it is running on has multiple CPUs, full advantage of the CPUs can be taken by running multiple copies of the same **HANDLER**. A good "rule of thumb" is to run at least two **HANDLERS** for each CPU on the server. Idle handlers consume very little cpu-time---less than 1/10th of 1% of available CPU-time for each idle handler.

The SheerPower CGI interface allow full acccess to the SPINS webserver "environment variables". For example, to ask for the **QUERY_STRING** (the data that follows a "?" in a URL) you would use:

```
ask #cgi_ch, symbol 'env:QUERY_STRING': value qstring$
```

The **"env:"** symbol prefix tells SheerPower that you are requesting an environment variable and not a form variable.

Table L-1 CGI Environment Variables

Variable	Function
ALL_HTTP	Retrieves all HTTP headers that were received. These variables are of the form HTTP_header field name. The headers consist of a null-terminated string with the individual headers separated by line feeds.
ALL_RAW	Retrieves all headers in raw form. The header names and values appear as the client sends them. Currently, proxy servers and other similar applications primarily use this value.
APPL_MD_PATH	Retrieves the metabase path of the application for the ISAPI DLL or the script.
APPL_PHYSICAL_PATH	Retrieves the physical path that corresponds with the metabase path. SPINS maps the namespace to the physical directory path; this allows APPL_MD_PATH to return this value. This is a costly function (in run time) compared to getting just APPL_MD_PATH.
AUTH_PASSWORD	Specifies the value entered in the client's authentication dialog. This variable is only available if Basic authentication is used.
AUTH_TYPE	Specifies the type of authentication used. If the string is empty, then no authentication is used. Possible values are Kerberos, user, SSL/PCT, Basic, and integrated Windows authentication.
AUTH_USER	Specifies the value entered in the client's authentication dialog box.
CERT_COOKIE	Specifies a unique ID for a client certificate. Returned as a string. Can be used as a signature for the whole client certificate.
CERT_FLAGS	If bit0 is set to 1, a client certificate is present. If bit1 is set to 1, the certification authority (CA) of the client certificate is invalid (that is, it is not on this server's list of recognized CAs).
CERT_ISSUER	Specifies the issuer field of the client certificate. For example,the following codes might be O=MS, OU=IAS, CN=user name, C=USA, and so on.
CERT_KEYSIZE	Specifies the number of bits in the Secure Sockets Layer (SSL) connection key size.
CERT_SECRETKEYSIZE	Specifies the number of bits in the server certificate private key.
CERT_SERIALNUMBER	Specifies the serial-number field of the client certificate.
CERT_SERVER_ISSUER	Specifies the issuer field of the server certificate.
CERT_SERVER_SUBJECT	Specifies the subject field of the server certificate.
CERT_SUBJECT	Specifies the subject field of the client certificate.
CONTENT_LENGTH	Specifies the number of bytes of data that the script or extension can expect to receive from the client. This total does not include headers.
CONTENT_TYPE	Specifies the content type of the information supplied in the body of a POST request.
LOGON_USER	The Windows account that the user is logged into.
HTTPS	Returns on if the request came in through secure channel (with SSL encryption), or off if the request is for an unsecure channel.

HTTPS_KEYSIZE	Specifies the number of bits in the SSL connection key size.
HTTPS_SECRETKEYSIZE	Specifies the number of bits in server certificate private key.
HTTPS_SERVER_ISSUER	Specifies the issuer field of the server certificate.
HTTPS_SERVER_SUBJECT	Specifies the subject field of the server certificate.
INSTANCE_ID	Specifies the ID for the server instance in textual format. If the instance ID is 1, it appears as a string. This value can be used to retrieve the ID of the Web-server instance, in the metabase, to which the request belongs.
INSTANCE_META_PATH	Specifies the metabase path for the instance to which the request belongs.
PATH_INFO	Specifies the additional path information, as given by the client. This consists of the trailing part of the URL after the script or ISAPI DLL name, but before the query string, if any.
PATH_TRANSLATED	Specifies this is the value of PATH_INFO, but with any virtual path expanded into a directory specification.
QUERY_STRING	Specifies the information that follows the first question mark in the URL that referenced this script.
REMOTE_ADDR	Specifies the IP address of the client or agent of the client (for example gateway, proxy, or firewall) that sent the request.
REMOTE_HOST	Specifies the host name of the client or agent of the client (for example, gateway, proxy or firewall) that sent the request if reverse DNS is enabled. Otherwise, this value is set to the IP address specified by REMOTE_ADDR.
REMOTE_USER	Specifies the user name supplied by the client and authenticated by the server. This comes back as an empty string when the user is anonymous.
REQUEST_METHOD	Specifies the HTTP request method verb.
SCRIPT_NAME	Specifies the name of the script program being executed.
SERVER_NAME	Specifies the server's host name, or IP address, as it should appear in self-referencing URLs.
SERVER_PORT	Specifies the TCP/IP port on which the request was received.
SERVER_PORT_SECURE	Specifies a string of either 0 or 1. If the request is being handled on the secure port, then this will be 1. Otherwise, it will be 0.
SERVER_PROTOCOL	Specifies the name and version of the information retrieval protocol relating to this request.
SERVER_SOFTWARE	Specifies the name and version of the Web server under which the ISAPI extension DLL program is running.
URL	Specifies the base portion of the URL. Parameter values will not be included. The value is determined when SPINS parses the URL from the header.

It is a proven fact that maintenance of existing software accounts for 80% of software costs. Therefore, the key to writing efficient code is to keep routines short, simple and easy to maintain.

Coding Standards and Writing Professional Code

See Appendix A, Coding Principles and Standards and Appendix I, Developing Professional Applications with SheerPower for more on writing professional code in SheerPower.

SheerPower 4GL has powerful features that make it easy to write short, simple routines:

- **Routines:** All variables in the main code are available from within the routine. This form of ROUTINE makes it easy to break code down into smaller segments. This is also known as a PUBLIC ROUTINE.
- **Private Routines:** All variables are private to the routine. Main code variables are accessible by prefixing the variable with MAIN$---for example: "main$xyz" references the main code variable called "xyz." This form of ROUTINE makes it easy to write reusable routines for any number of applications.

Routines in SheerPower

See Section 3.4, ROUTINE/END ROUTINE for more on Routines and Private Routines in SheerPower.

Each variable in a program belongs to a **NAMESPACE**. By default, they belong to a "namespace" called **MAIN**. So:

```
abc = 123
print abc
```

is the same as:

```
abc = 123
print main$abc
```

is the same as:

```
main$abc = 123
print abc
```

SheerPower 4GL supports both ROUTINES that use the MAIN "namespace," and PRIVATE ROUTINES that have their own "namespace." For example:

```
show_display

routine show_display
  abc = 123
  print main$abc
end routine
```

In this ROUTINE, the variable "abc" belongs to the "namespace" of MAIN-sharing its variable names with the main program.

However in this PRIVATE ROUTINE:

```
do_totals

private routine do_totals
  abc = 123
  print abc
end routine
```

the variable "abc" belongs to the "namespace" of "do_totals":

```
do_totals

private routine do_totals
  abc = 123
  print do_totals$abc
end routine
```

Now, let's look at a more complex example:

```
abc = 123
do_totals
stop

private routine do_totals
  abc = 999
  print 'The DO_TOTALS version: '; abc
  print 'The MAIN version     : '; main$abc
end routine

end
```

In many programming languages it is difficult to take a large segment of code and break it into separate routines or functions. This difficulty arises from determining how parameters will get passed into and out of the newly created routine. SheerPower ROUTINES make this task very easy because, by default, ROUTINES have full access to all variables in the code that calls them. For example:

```
email_image$ = "c:\sheerpower\samples\emailicon.jpg"
do
  email_form$ = '<sheerpower color=#ffaaaa persist>'
  email_form$ = email_form$ + '<form><title>Send Email</title>'
  email_form$ = email_form$ + '<img src="' + email_image$ + '"><p>'

  email_form$ = email_form$ + '<table align=center>'
  email_form$ = email_form$ + '<tr><td align=left><b>From:</b> '
  email_form$ = email_form$ + '<td><input type=text name=from ' +
    'value="Type in sender email address here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>To:</b> '
  email_form$ = email_form$ + '<td><input type=text name=to ' +
    'value="Type in recipient email address here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>Subject:</b> '
  email_form$ = email_form$ + '<td><input type=text name=subject ' +
    'value="Type in a subject here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>Body:</b> '
  email_form$ = email_form$ + '<td><br><textarea name=body rows=10 cols=30>' +
    'Type in the body of the email here</textarea><p>'
  email_form$ = email_form$ + '</textarea><p></tr>'
  email_form$ = email_form$ + '</table>'
  email_form$ = email_form$ + '<p><center>' +
                '<input type=submit name=submit value="Send Email">'
  email_form$ = email_form$ + '<input type=submit name=exit value="Quit Email">'
  email_form$ = email_form$ + '</center></form>'
  line input dialogbox email_form$: info$
  if _exit then exit do
  for item = 1 to pieces(info$, chr$(26))
    z0$    = piece$(info$, item, chr$(26))
    name$  = element$(z0$, 1, '=')
    value$ = element$(z0$, 2, '=')
    select case name$
    case 'from'
      mailfrom$ = value$
    case 'to'
      sendto$ = value$
    case 'body'
      text$ = value$
    case 'subject'
      subject$ = value$
  end select
  next item
  if pos(mailfrom$,'@') = 0 or pos(sendto$,'@') = 0 then repeat do
//  send_email  // commented out so sample program works
loop
```

This code is hard to follow because it has too much detail that the maintenance programmer is forced to read through. So, we can easily segment this code into distinct ROUTINES by:

- highlighting the segment of code
- doing a GOLD/R keystroke to create a ROUTINE that has full access to all variables

Programming Keystrokes

See Section F.2, Programming Keystrokes for a complete list of special programming keystrokes available in SheerPower.

This results in code that looks like:

```
email_image$ = "c:\sheerpower\samples\emailicon.jpg"

do
  show_email_form
  if _exit then exit do
  parse_results
  if pos(mailfrom$,'@') = 0 or pos(sendto$,'@') = 0 then repeat do
//  send_email
loop
stop

routine show_email_form
  email_form$ = '<sheerpower color=#ffaaaa persist>'
  email_form$ = email_form$ + '<form><title>Send Email</title>'
  email_form$ = email_form$ + '<img src="' + email_image$ + '"><p>'

  email_form_table

  email_form$ = email_form$ + '<p><center>' +
                '<input type=submit name=submit value="Send Email">'
  email_form$ = email_form$ + '<input type=submit name=exit value="Quit Email">'
  email_form$ = email_form$ + '</center></form>'
  line input dialogbox email_form$: info$
end routine

routine email_form_table
  email_form$ = email_form$ + '<table align=center>'
  email_form$ = email_form$ + '<tr><td align=left><b>From:</b> '
  email_form$ = email_form$ + '<td><input type=text name=from ' +
    'value="Type in sender email address here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>To:</b> '
  email_form$ = email_form$ + '<td><input type=text name=to ' +
    'value="Type in recipient email address here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>Subject:</b> '
  email_form$ = email_form$ + '<td><input type=text name=subject ' +
    'value="Type in a subject here"></tr>'
  email_form$ = email_form$ + '<tr><td align=left><b>Body:</b>'
  email_form$ = email_form$ + '<td><br><textarea name=body rows=10 col30>' +
    'Type in the body of the email here</textarea><p>'
  email_form$ = email_form$ + '</textarea><p></tr></table>'
end routine

routine parse_results
  for item = 1 to pieces(info$, chr$(26))
    z0$    = piece$(info$, item, chr$(26))
    name$  = element$(z0$, 1, '=')
    value$ = element$(z0$, 2, '=')

    select case name$
    case 'from'
      mailfrom$ = value$
    case 'to'
      sendto$ = value$
    case 'body'
      text$ = value$
    case 'subject'
      subject$ = value$
  end select
```

```
  next item
end routine
```

Notice how the program is now easy to follow and therefore easy to maintain.

Unlike the ROUTINE feature whose primary purpose is to break down program code into smaller, more manageable segments, the PRIVATE ROUTINE feature is used to write routines that will be used in a number of different applications. Here is a simple PRIVATE ROUTINE that is used to write messages to a message file:

```
private routine write_message with msg
  if msg_ch = 0 then open file msg_ch: name 'message.log', access output
  print #msg_ch: time$; ' '; msg
end routine
```

WRITE_MESSAGE has a single named parameter called "msg." The first time WRITE_MESSAGE is called, msg_ch will be zero, so a new message.log file is created and msg_ch receives the channel#. Then WRITE_MESSAGE writes out the current time and the message.

This PRIVATE ROUTINE can be called from any application without the programmer worrying about variable name conflicts.

```
write_message with msg "this is a test"

private routine write_message with msg
  if msg_ch = 0 then open file msg_ch: name 'message.log', access output
  print #msg_ch: time$; ' '; msg
end routine
```

The PRIVATE ROUTINE feature of SheerPower 4GL is designed to assist in writing routines that will be used in a number of different programs. Variable name conflicts are not an issue because all variable names in PRIVATE ROUTINES have their own private "namespace."

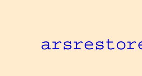
Advanced Record Systems (ARS) is a high-speed and performance database engine integrated into SheerPower 4GL. The purpose of this appendix is to outline the various utilities available for ARS.

To access the **Help** for ARS, run the Command Prompt program and type in the utility name to view its help text:

```
  ARS_UTILITY_NAME
```

UTILITY is the name of the ARS Utility to get help on. For example, to get help on ARSRESTORE, use the following command in the Command Prompt program:

```
  arsrestore
```



The help text will display inside the Command Prompt program window:

```
Administrator: Command Prompt
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\GemTech>arsrestore
ARSRESTORE V01.010

Command Format:
  ARSRESTORE infilespecs outfilepath [flag1 flag2 ...]
    infilespecs: comma seperated list of files to restore
    outfilepath: directory to store the files in
    flags:       OPTIONAL flags
      -continue:   continue record adds in spite of errors
      -stats:      output stats from the restore
      -nocrc:      Don't do CRC processing on new file
      -fastcrc:    Use FAST CRCs on new file
Alternate Command Format:
  ARSRESTORE -include filewithspecs [flag1 flag2 ...]
    -include:      tells ARSRESTORE to use next param
    filewithspecs: textfilespec with infilespecs and outfilepaths
    flags:         OPTIONAL flags
      -continue:     continue record adds in spite of errors
      -stats:        output stats from the restore
      -nocrc:        Don't do CRC processing on new file
      -fastcrc:      Use FAST CRCs on new file

C:\Users\GemTech>
```

DESCRIPTION:

This utility is used to backup ARS files and other files from one folder (directory) to another. It will perform a special backup of ARS files to a special file format with an extension of .ARS_BACKUP.

Any files that are **not ARS** will be copied to the new directory. The backup is done using many threads and is synchronized across all of the ARS files so that the data is synchronized exactly at the moment that the backup starts.

ARS also includes utilities to backup ARS tables in real time--- no need to stop or check-point applications in order to back them up and safe-keep the data. All data is synchronized and reliably backed up by ARS even while database tables are being written to. This allows for true 24/7 non-stop applications.

FORMAT:

```
Format #1:

 ARSBACKUP infilespecs outfilepath [description] [flags]
   infilespecs - comma seperated list of files to backup
   outfilepath - directory to store the files in
   description - OPTIONAL "Quoted string to save with backup"
   flags       - OPTIONAL-  flags seperated by a :
                 PROMPT to prompt user before starting jobs
                 STATS  - to display stats after backup
```

```
Format #2:

 ARSBACKUP -include filewithspecs [description] [flags]
   "-include"    - tells us to use a textfile for the filespecs
   filewithspecs - name of textfile with infilespecs and outfilepaths
   description   - OPTIONAL "Quoted string to save with backup"
   flags         - OPTIONAL flags seperated by a :
                   PROMPT - to prompt user before starting jobs
                   STATS  - to display stats after backup
```

DESCRIPTION:

This utility scans an ARS file looking for problems in the internal file structure and reports any errors as well as ARS statistics that are kept and/or derived from the scan.

FORMAT:

```
ARSCHK file-to-check [NOCRC] [log-file]
  file-to-check - File Specification of the file we want to check
  NOCRC         - OPTIONAL 2nd parameter to disable CRC checks
  log-file      - OPTIONAL 3rd parameter for filespec to log results to
```

DESCRIPTION:

This utility is designed to recover data from a corrupt ARS file. It scans an ARS file looking for records and then puts as many records as it can reasonably locate into an output file. It will report problems as if goes and tells the user the number of recovered records.

FORMAT:

```
Format #1:

 ARSFIX filespecs [flags ...]
   filespecs: wildcard filespecs to fix
   flags:     OPTIONAL flags
     -fastcrc:  Use FAST CRCs on new file
```

```
Format #2:

 ARSFIX filspec [fdlfile] [logfile]
   filespec: filespec to fix
   fdlfile:  OPTIONAL FDL Filespec to use for new file create
   logfile:  OPTIONAL LOG Filespec for logging status to
```

DESCRIPTION:

This utility is used to restore ARS files and other files from one folder (directory) to another. It will interpret the special file format with for files with an extension of .ARS_BACKUP and restore them to ARS files. It just does a copy of files that are NOT ARS file to the new directory. The restore is done using LOTs of threads.

FORMAT:

```
Format #1:

  ARSRESTORE infilespecs outfilepath [flag1 flag2 ...]
    infilespecs: comma seperated list of files to restore
    outfilepath: directory to store the files in
    flags:       OPTIONAL flags
      -continue:   continue record adds in spite of errors
      -stats:      output stats from the restore
      -nocrc:      Don't do CRC processing on new file
      -fastcrc:    Use FAST CRCs on new file
```

```
Format #2:

Alternate Command Format:
  ARSRESTORE -include filewithspecs [flag1 flag2 ...]
    -include:      tells ARSRESTORE to use next param
    filewithspecs: textfilespec with infilespecs and outfilepaths
    flags:         OPTIONAL flags
      -continue:     continue record adds in spite of errors
      -stats:        output stats from the restore
      -nocrc:        Don't do CRC processing on new file
      -fastcrc:      Use FAST CRCs on new file
```

DESCRIPTION:

This utility creates an empty ARS file from the definition specification contained in an FDL (file definition language) file.

FORMAT:

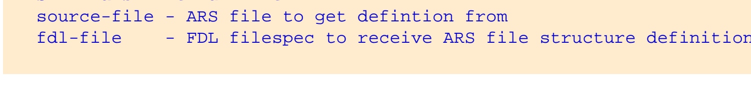
```
FDL2ARS fdl-file ars-file
  fdl-file - FDL filespec containing ARS file structure
  ars-file - ARS filespec to create
```

DESCRIPTION:

This utility takes an ARS file and creates a FDL file from it that contains the definition of how to create an equivalent ARS file from it including key structures.

FORMAT:

```
ARS2FDL ars-file fdl-file
  source-file - ARS file to get defintion from
  fdl-file    - FDL filespec to receive ARS file structure definition
```



Index | Contents

Example M-4 Correct program segmentation
Example M-3 Long code to segment into smaller routines
Example M-2 Private routines and namespace
Example M-1 Routine with default "main namespace"
Example L-1 Syntax to Change Port Number Used
Example I-2 Routine Header Template
Example I-1 Program Template
Example 21-2 Passing integer arrays to external routines
Example 21-1 LIBRARY and CALL statements
Example 20-6 GETSYMBOL$ Function
Example 20-5 %SPCODE TAG
Example 20-4 %SPSCRIPT TAG
Example 20-3 Matrix.spsrc Scripting Program
Example 20-2 Matrix.spsrc Code Area
Example 20-1 Matrix.spsrc Script Area

Example 19-8 Communication Port
Example 19-7 Communication Port
Example 19-6 UDP Protocol - Timeout Parameter
Example 19-5 TCP/IP Protocol - Port Parameter
Example 19-4 TCP/IP Protocol
Example 19-3 Environment Variables
Example 19-2 Sending Email
Example 19-1 Accessing data from webpages
Example 18-8 SPINS Webserver Options Command
Example 18-7 Specify SPINS webserver root folder
Example 18-6 Syntax to Specify Multiple Ports
Example 18-5 Syntax to Change Port Number Used
Example 18-4 Replacing IIS with SPINS
Example 18-3 Expected Directory Structure
Example 18-2 Default Root Folder Location
Example 18-1 SPINS Location
Example 17-9 Accessing ODBC Database in SheerPower OPEN STRUCTURE Statement
Example 17-8 ODBC Data Source Setup - ODBC Microsoft Access Setup Complete 2
Example 17-7 ODBC Data Source Setup - ODBC Microsoft Access Setup Complete
Example 17-6 ODBC Data Source Setup - Select MyContacts Database
Example 17-5 ODBC Data Source Setup - Select Database
Example 17-4 ODBC Data Source Setup - Data Source Name and Description
Example 17-3 ODBC Data Source Setup - ODBC Microsoft Access Setup
Example 17-2 ODBC Data Source Setup - Create New Data Source
Example 17-1 ODBC Data Source Setup - ODBC Data Source Administrator
Example 16-26 Create Data File Notification
Example 16-25 Create Data File
Example 16-24 Key Field Information
Example 16-23 Creating the Data File
Example 16-22 Creating the Data File
Example 16-21 Structure Security
Example 16-20 Structure Security
Example 16-19 Viewing Field Definitions
Example 16-18 Deleting an Existing Field in SETUP
Example 16-17 Deleting an Existing Data Field
Example 16-16 Deleting an Existing Field in SETUP
Example 16-15 Modifying an Existing Field
Example 16-14 Defining Using a Segmented Key
Example 16-13 Defining Key Fields
Example 16-12 Defining Fields
Example 16-11 Defining Fields
Example 16-10 Validation Rules
Example 16-9 Semantics
Example 16-8 Field Definition Window
Example 16-7 Fields Window
Example 16-6 Structure Definition Window
Example 16-5 Data Set
Example 16-4 Selecting Database Engine
Example 16-3 Structure Definition Window
Example 16-2 Open Structure
Example 16-1 Entering SETUP
Example 15-50 Updating a Structure

Example 15-49 SET STRUCTURE: EXTRACTED 0
Example 15-48 SET STRUCTURE: POINTER
Example 15-47 SET STRUCTURE: ID
Example 15-46 SET STRUCTURE, FIELD: PARTIAL KEY
Example 15-45 SET STRUCTURE, FIELD: KEY
Example 15-44 SET STRUCTURE: CURRENT
Example 15-43 ASK STRUCTURE: ENGINE
Example 15-42 ASK | SET STRUCTURE#string_expr . . .
Example 15-41 ASK STRUCTURE: ACCESS
Example 15-40 ASK STRUCTURE: RECORDSIZE
Example 15-39 ASK STRUCTURE: POINTER
Example 15-38 ASK STRUCTURE: ID
Example 15-37 ASK STRUCTURE: EXTRACTED
Example 15-36 ASK STRUCTURE: CAPABILITY
Example 15-35 ASK STRUCTURE: KEYS
Example 15-34 ASK STRUCTURE: FIELDS
Example 15-33 ASK STRUCTURE: DATAFILE
Example 15-32 ASK STRUCTURE: CURRENT
Example 15-31 VRULES - field definition item
Example 15-30 OPTIMIZED - field definition item
Example 15-29 CHANGEABLE - field definition item
Example 15-28 APPLICATION - field definition item
Example 15-27 ACCESS - field definition item
Example 15-26 Field expressions in ASK STRUCTURE FIELD
Example 15-25 ASK STRUCTURE FIELD: item
Example 15-24 APPEND option in EXTRACT STRUCTURE
Example 15-23 REEXTRACT STRUCTURE ... END EXTRACT
Example 15-22 EXIT EXTRACT
Example 15-21 CANCEL EXTRACT
Example 15-20 PARTIAL KEY option in EXTRACT STRUCTURE
Example 15-19 Extract a range of keys - TO expr option
Example 15-18 KEY option in EXTRACT STRUCTURE
Example 15-17 FOR EACH ... NEXT structure
Example 15-16 SORT within EXTRACT STRUCTURE - ASCENDING or DESCENDING
Example 15-15 SORT within EXTRACT STRUCTURE
Example 15-14 EXCLUDE within EXTRACT STRUCTURE
Example 15-13 INCLUDE within EXTRACT STRUCTURE
Example 15-12 EXTRACT STRUCTURE - Extracting records from a structure
Example 15-11 UNLOCK ALL: COMMIT
Example 15-10 UNLOCK STRUCTURE: COMMIT
Example 15-9 LOCK/UNLOCK STRUCTURE
Example 15-8 DELETE STRUCTURE - Delete structure record
Example 15-7 EXIT ADD - Exit when adding a structure record
Example 15-6 CANCEL ADD - Cancel adding a structure record
Example 15-5 ADD STRUCTURE/END ADD - Add structure record
Example 15-4 CLOSE STRUCTURE
Example 15-3 OPEN STRUCTURE
Example 15-2 SheerPower structure
Example 15-1 Data Structure DIAGRAM
Example 14-32 To Stop Playing Looped Media
Example 14-31 Playing Media Files in a Loop
Example 14-30 Playing MEDIA Files

Example 14-29 Printing Output from an .HTML File
Example 14-28 Printing Output using PASS
Example 14-27 TEXTWINDOW://
Example 14-26 KILL statement - deleting a file
Example 14-25 SET#chnl_num: CURRENT
Example 14-24 SET#chnl_num: MARGIN
Example 14-23 SET#chnl_num statement
Example 14-22 ASK#chnl_num: NAME
Example 14-21 ASK#chnl_num: CURRENT
Example 14-20 ASK#chnl_num: MARGIN
Example 14-19 ASK#chnl_num: ZONEWIDTH
Example 14-18 ASK#chnl_num statement
Example 14-17 Multiple variables in LINE INPUT#chnl_num
Example 14-16 Appending data to a file
Example 14-15 EOF option with LINE INPUT
Example 14-14 LINE INPUT#chnl_num statement
Example 14-13 Inputting multiple variables
Example 14-12 Input to a file - INPUT#chnl_num statement
Example 14-11 Channel number in PRINT statement - PRINT#chnl_num
Example 14-10 CLOSE#chnl_num statement
Example 14-9 LOCK option in OPEN#chnl_num
Example 14-8 UNIQUE option in OPEN#chnl_num
Example 14-7 UNIQUE option in OPEN#chnl_num
Example 14-6 Specifying Temporary File Location
Example 14-5 OPTIMIZE OFF option in OPEN#chnl_num
Example 14-4 Read/Write Binary Files using UNFORMATTED
Example 14-3 UNFORMATTED option in OPEN#chnl_num
Example 14-2 OPEN FILE statement - opening files
Example 14-1 OPEN#chnl_num statement - opening files
Example 13-9 EXIT HANDLER statement in HANDLER routine
Example 13-8 RESUME statement in HANDLER routine
Example 13-7 CONTINUE statement in HANDLER routine
Example 13-6 RETRY statement in HANDLER routine
Example 13-5 HANDLER/END HANDLER with WHEN EXCEPTION USE
Example 13-4 HANDLER/END HANDLER with WHEN EXCEPTION USE
Example 13-3 WHEN EXCEPTION USE
Example 13-2 WHEN EXCEPTION IN/USE/END WHEN
Example 13-1 CAUSE EXCEPTION
Example 12-4 Read/Write the Windows Registry
Example 12-3 More Microsoft Word DDE Commands
Example 12-2 Accessing Microsoft Word via DDE
Example 12-1 SheerPower Control an Excel Spreadsheet via DDE
Example 11-52 SET ZONEWIDTH statement
Example 11-51 ASK ZONEWIDTH statement
Example 11-50 SET WINDOW: TYPEAHEAD statement
Example 11-49 ASK WINDOW: TYPEAHEAD statement
Example 11-48 SET WINDOW: ROW statement
Example 11-47 ASK WINDOW: ROW statement
Example 11-46 SET WINDOW KEYSTROKE: VALUE statement
Example 11-45 ASK | SET WINDOW: KEYMAP
Example 11-44 SET WINDOW: DATA statement
Example 11-43 ASK WINDOW: DATA statement

Example 11-42 ASK | SET WINDOW: CURRENT
Example 11-41 SET WINDOW: COLUMN statement
Example 11-40 ASK WINDOW: COLUMN statement
Example 11-39 SET WINDOW AREA statement
Example 11-38 ASK WINDOW AREA statement
Example 11-37 ASK SYSTEM: USER statement
Example 11-36 ASK SYSTEM, SYMBOL 'OS:xxx': VALUE statement
Example 11-35 SET SYSTEM, SYMBOL: VALUE statement
Example 11-34 DNS Lookup that does not exist
Example 11-33 DNS Lookups using ASK SYSTEM, SYMBOL
Example 11-32 ASK SYSTEM, SYMBOL: VALUE statement
Example 11-31 ASK SYSTEM: RIGHTS statement
Example 11-30 ASK SYSTEM: PROGRAM Statement
Example 11-29 SET SYSTEM: PROCESS statement
Example 11-28 ASK SYSTEM: PROCESS statement
Example 11-27 ASK SYSTEM: PARAMETER statement
Example 11-26 ASK SYSTEM: MODE statement
Example 11-25 Set Logicals in SP4GL.INI
Example 11-24 SET SYSTEM, LOGICAL: VALUE statement
Example 11-23 ASK SYSTEM, LOGICAL: VALUE statement
Example 11-22 ASK SYSTEM: DIRECTORY statement
Example 11-21 ASK SYSTEM: DIRECTORY statement
Example 11-20 SET SYSTEM: COMMENT statement
Example 11-19 ASK SYSTEM: COMMENT statement
Example 11-18 ASK/SET SEED statement
Example 11-17 Randomize statement
Example 11-16 SET SCROLL statement
Example 11-15 ASK RESPONSES statement
Example 11-14 ASK PAGESIZE statement
Example 11-13 SET MARGIN statement
Example 11-12 ASK KEYSTROKES statement
Example 11-11 SET ICON statement
Example 11-10 SET HELP OFF statement
Example 11-9 SET HELP ON statement
Example 11-8 SET EXIT OFF statement
Example 11-7 SET EXIT ON statement
Example 11-6 ASK ERRORS statement
Example 11-5 SET ERROR OFF statement
Example 11-4 SET ERROR ON statement
Example 11-3 SET BACK OFF statement
Example 11-2 SET BACK ON statement
Example 11-1 SET AUTOEXIT
Example 10-38 DISPATCH statement
Example 10-37 PASS URL: Opening an .HTML file
Example 10-36 PASS Print
Example 10-35 PASS TIMEOUT:
Example 10-34 PASS NOWAIT | NORETURN, WINDOW:
Example 10-33 PASS WINDOW statement
Example 10-32 PASS NORETURN statement
Example 10-31 PASS NOWAIT statement
Example 10-30 PASS Statement
Example 10-29 CHAIN statement

Example 10-28 CASE IS statement
Example 10-27 CASE ELSE statement
Example 10-26 CASE statement - checking for a range of values
Example 10-25 CASE statement
Example 10-24 SELECT CASE/END SELECT
Example 10-23 ELSEIF option in IF/THEN
Example 10-22 ELSE option in IF/THEN
Example 10-21 END IF in IF/THEN statement
Example 10-20 EXECUTE statement
Example 10-19 EXECUTE statement
Example 10-18 ITERATE DO used in a nested loop
Example 10-17 ITERATE DO statement
Example 10-16 REPEAT DO within a nested loop
Example 10-15 REPEAT DO statement
Example 10-14 EXIT DO statement
Example 10-13 WHILE and UNTIL options in DO/LOOP
Example 10-12 UNTIL option in DO/LOOP
Example 10-11 WHILE option in DO/LOOP
Example 10-10 Infinite DO/LOOP
Example 10-9 DO LOOP
Example 10-8 ITERATE FOR used in a nested loop
Example 10-7 ITERATE FOR statement in FOR loop
Example 10-6 REPEAT FOR used within a nested loop
Example 10-5 REPEAT FOR statement in FOR loop
Example 10-4 EXIT FOR statement in FOR loop
Example 10-3 Nesting FOR/NEXT loops
Example 10-2 STEP option in FOR/NEXT loop
Example 10-1 FOR/NEXT loop
Example 9-54 Storing a select option into a variable for displaying later
Example 9-53 Storing a select option into a variable for displaying later
Example 9-52 Inserting Numeric Variable Data into a Form
Example 9-51 Inserting String Variable Data into a Form
Example 9-50 <td > tag
Example 9-49 <tr >... </tr > tag
Example 9-48 <th >... </th > tag
Example 9-47 <table > tag attributes
Example 9-46 <table >... </table > tag
Example 9-45 <blockquote >... </blockquote > tag
Example 9-44 <address >... </address > tag
Example 9-43 <pre >... </pre > tag
Example 9-42 <em >... tag
Example 9-41 <i >... </i > tag
Example 9-40 <b >... tag
Example 9-39 <h6 >... </h6 > tag
Example 9-38 <h5 >... </h5 > tag
Example 9-37 <h4 >... </h4 > tag
Example 9-36 <h3 >... </h3 > tag
Example 9-35 <h2 >... </h2 > tag
Example 9-34 <h1 >... </h1 > tag
Example 9-33 <font >... tag
Example 9-32 <div >... </div > tag
Example 9-31 <img > attributes tag

Example 9-30 <img > tag
Example 9-29 <hr > tag
Example 9-28 ... tag
Example 9-27 ... tag
Example 9-26 <br > tag
Example 9-25 <p >... </p > tag
Example 9-24 <center >... </center > tag
Example 9-23 <textarea >... </textarea > tag
Example 9-22 <select > tag and attributes
Example 9-21 <select >... </select > tag
Example 9-20 Image Submit Buttons
Example 9-19 Custom Submit Buttons
Example 9-18 <input > tag attributes
Example 9-17 <input > tag
Example 9-16 <a >... tag
Example 9-15 <form >... </form > tag
Example 9-14 <sheerpower > tag attributes - TYPE =SELECT
Example 9-13 <sheerpower > tag attributes - TYPE =SAVEAS
Example 9-12 <sheerpower > tag attributes - TYPE =OPEN
Example 9-11 <sheerpower > tag attributes - TYPE FILTER
Example 9-10 <sheerpower > tag attributes - AUTOSUBMIT
Example 9-9 <sheerpower > tag attributes - SRC
Example 9-8 <sheerpower > tag attributes - BACKGROUND
Example 9-7 <sheerpower > tag attributes - COLOR, HEIGHT, WIDTH, TITLE
Example 9-6 <sheerpower > tag
Example 9-5 %READONLY Directive
Example 9-4 %FOCUS Directive
Example 9-3 %ERROR Directive
Example 9-2 Parsing Input Dialogbox Results
Example 9-1 INPUT DIALOGBOX
Example 8-52 _TERMINATOR system function
Example 8-51 KEY INPUT statement
Example 8-50 %NOMOUSEOVER menu directive
Example 8-49 %INACTIVE menu directive
Example 8-48 %WIDTH menu directive
Example 8-47 %SPLIT menu directive
Example 8-46 %MESSAGE menu directive
Example 8-45 %MENUBAR menu directive
Example 8-44 %LOCKSTEP menu directive
Example 8-43 %ITEMS menu directive
Example 8-42 %HEADING menu directive
Example 8-41 %COLUMNS menu directive
Example 8-40 %BAR menu directive
Example 8-39 %ATTACHED menu directive
Example 8-38 MENU option in INPUT
Example 8-37 Enter key with INPUT SCREEN
Example 8-36 Screen format options - BOLD, BLINK, REVERSE
Example 8-35 Screen format options - ELAPSED
Example 8-34 Screen format options - ELAPSED
Example 8-33 Screen format options - VALID
Example 8-32 Screen format options - AT row, column
Example 8-31 Screen format options - AJ

- Example 8-30 Screen format options - DIGITS
- Example 8-29 Screen format options - NOECHO
- Example 8-28 Screen format options - LCASE
- Example 8-27 Screen format options - UCASE
- Example 8-26 Screen Format Characters - ~
- Example 8-25 Screen Format Characters -^
- Example 8-24 Screen format characters - .
- Example 8-23 Screen format characters -@
- Example 8-22 Screen format characters -#
- Example 8-21 SCREEN option
- Example 8-20 AREA Option
- Example 8-19 TIMEOUT option
- Example 8-18 VALID option
- Example 8-17 ERASE option
- Example 8-16 _STRING system function
- Example 8-15 DEFAULT option
- Example 8-14 LENGTH option
- Example 8-13 ATTRIBUTES Option
- Example 8-12 AT option
- Example 8-11 PROMPT option
- Example 8-10 Input default prompt and text
- Example 8-9 LINE INPUT statement
- Example 8-8 Validating data prior to storing it into a structure
- Example 8-7 Simple input statement
- Example 8-6 Inputting Strings
- Example 8-5 Pop-up menus
- Example 8-4 Free format multi-line text input
- Example 8-3 Formatted data entry screens
- Example 8-2 Simple input style

Example 8-1 INPUT statement
Example 7-52 CLEAR AREA BOX - BOLD, BLINK, REVERSE attributes
Example 7-51 CLEAR AREA
Example 7-50 CLEAR statement - clearing the screen
Example 7-49 DELAY statement
Example 7-48 DELAY option in MESSAGE statement
Example 7-47 ERROR option in MESSAGE statement
Example 7-46 MESSAGE statement
Example 7-45 ZIPCODE directive used with PRINT USING
Example 7-44 TIME directive used with PRINT USING
Example 7-43 ROTATE directive used with PRINT USING
Example 7-42 DATE directive used with PRINT USING
Example 7-41 HYPHEN directive used with PRINT USING
Example 7-40 LCASE directive used with PRINT USING
Example 7-39 UCASE directive used with PRINT USING
Example 7-38 $- characters in PRINT USING
Example 7-37 +$ characters in PRINT USING
Example 7-36 -$ characters in PRINT USING
Example 7-35 $+ characters in PRINT USING
Example 7-34 $ character in PRINT USING
Example 7-33 ~ character in PRINT USING
Example 7-32 - character in PRINT USING
Example 7-31 + character in PRINT USING

Example 7-30 * character in PRINT USING
Example 7-29 * character in PRINT USING
Example 7-28 % character in PRINT USING
Example 7-27 , character in PRINT USING
Example 7-26 . character in PRINT USING
Example 7-25 @ character in PRINT USING
Example 7-24 > character in PRINT USING
Example 7-23 < character in PRINT USING
Example 7-22 Numeric format characters in PRINT USING
Example 7-21 Numeric format characters in PRINT USING
Example 7-20 Numeric format characters in PRINT USING - negative and positive numbers
Example 7-19 Numeric format characters in PRINT USING
Example 7-18 Numeric format characters in PRINT USING
Example 7-17 String format characters in PRINT USING
Example 7-16 String format characters in PRINT USING
Example 7-15 PRINT USING
Example 7-14 USING option in PRINT statement - print_mask
Example 7-13 Printing Attributes--Highlighting Options
Example 7-12 Cursor positioning in PRINT statement
Example 7-11 Printing integers of 12 or fewer digits
Example 7-10 Printing negative numbers with PRINT statement
Example 7-9 Printing numbers with PRINT statement
Example 7-8 ERASE in PRINT statement
Example 7-7 AT row, column in PRINT statement
Example 7-6 TAB in PRINT statement
Example 7-5 Printing long data in records
Example 7-4 Commas and Print Zones
Example 7-3 Semicolon in PRINT statement
Example 7-2 Print expression
Example 7-1 PRINT statement
Example 6-175 UBOUND function
Example 6-174 TRUE function
Example 6-173 SIZE function
Example 6-172 MAXNUM function
Example 6-171 LBOUND function
Example 6-170 FALSE function
Example 6-169 EVAL function
Example 6-168 DTYPE function
Example 6-167 DECODE function
Example 6-166 SYSTEXT$ function
Example 6-165 EXTYPE function
Example 6-164 EXTEXT$ function
Example 6-163 EXLABEL$ function
Example 6-162 _STRING system function
Example 6-161 _STATUS system function
Example 6-160 _DEBUG system function
Example 6-159 FINDFILE$ function
Example 6-158 FINDFILE$ function
Example 6-157 Copy a file with FILEINFO$ CONTENTS
Example 6-156 FILEINFO$ function - CONTENTS
Example 6-155 FILEINFO$ function
Example 6-154 FILEINFO$ function

Example 6-153 FILEINFO$ function
Example 6-152 _EXTRACTED system function
Example 6-151 _CHANNEL system function
Example 6-150 Validation rules - FILTER - RESTORE
Example 6-149 Validation rules - FILTER - CHANGE
Example 6-148 Validation rules - ROUTINE
Example 6-147 Validation rules - MENU
Example 6-146 Validation rules - CODE
Example 6-145 Validation rules - EXPRESSION
Example 6-144 Validation rules - PRINTMASK
Example 6-143 Validation rules - VRULES
Example 6-142 Validation rules - REQUIRED
Example 6-141 Validation rules - REQUIRED
Example 6-140 Validation rules - FULLTIME
Example 6-139 Validation rules - DATE DMONCY
Example 6-138 Validation rules - DATE DMONY
Example 6-137 Validation rules - INTEGER WORD
Example 6-136 Validation rules - INTEGER
Example 6-135 Validation rules - NUMBER
Example 6-134 Validation rules - DECIMALS
Example 6-133 Validation rules - DIGITS
Example 6-132 Validation rules - UCASE
Example 6-131 Validation rules - LCASE
Example 6-130 Validation rules - LETTERS
Example 6-129 Validation rules - NOCHARACTERS
Example 6-128 Validation rules - CHARACTERS
Example 6-127 Validation rules - LENGTH
Example 6-126 Validation rules - MAXLENGTH
Example 6-125 Validation rules - MINLENGTH
Example 6-124 Validation rules - DISALLOW
Example 6-123 Validation rules - ALLOW
Example 6-122 _TERMINATOR system function
Example 6-121 _REPLY system function
Example 6-120 _HELP system function
Example 6-119 _EXIT system function
Example 6-118 _BACK system function
Example 6-117 SKIP function
Example 6-116 SCAN function
Example 6-115 POS function
Example 6-114 PATTERN function - { |directive | }
Example 6-113 PATTERN function - {(word_text) }
Example 6-112 PATTERN function - { <cc |ccc |c > }
Example 6-111 PATTERN function - { |nnn,nnn,nnn | }
Example 6-110 PATTERN function - ~
Example 6-109 PATTERN function - {^ }
Example 6-108 PATTERN function - { }
Example 6-107 PATTERN function -*
Example 6-106 PATTERN function - ?
Example 6-105 MATCH function
Example 6-104 ITEM function
Example 6-103 ITEM function
Example 6-102 COMPARE function

- Example 6-101 XLATE$ function
- Example 6-100 WRAP$ function
- Example 6-99 VAL function
- Example 6-98 URLDECODE$ function
- Example 6-97 URLENCODE$ function
- Example 6-96 UNQUOTE$ function
- Example 6-95 UCASE$ function
- Example 6-94 TRIM$ function
- Example 6-93 TAB function
- Example 6-92 STR$ function
- Example 6-91 SPACE$ function
- Example 6-90 SORT$ function
- Example 6-89 SEG$ function
- Example 6-88 RTRIM$ function
- Example 6-87 RPAD$ function
- Example 6-86 RIGHT [$] function
- Example 6-85 REPLACE$ function
- Example 6-84 REPEAT$ function
- Example 6-83 QUOTE$ function
- Example 6-82 PRETTY$ function
- Example 6-81 PIECE$ function
- Example 6-80 PARSE$ function
- Example 6-79 ORDNAME$ function
- Example 6-78 ORD function
- Example 6-77 MID [$] function
- Example 6-76 MEM function
- Example 6-75 MAXLEN function
- Example 6-74 LTRIM$ function
- Example 6-73 LPAD$ function
- Example 6-72 LEN function
- Example 6-71 LEFT [$] function
- Example 6-70 LCASE$ function
- Example 6-69 HASH$ function
- Example 6-68 GETSYMBOL$ function - Operating System Symbol
- Example 6-67 GETSYMBOL$ function - CGI Environment Symbol
- Example 6-66 GETSYMBOL$ function - HTML form submission
- Example 6-65 GETSYMBOL$ function - SheerPower Symbol & Trimming Option

Example 6-64 DATE format with FORMAT$
Example 6-63 FORMAT$ function
Example 6-62 FORMAT$ function
Example 6-61 ENCODE$ function
Example 6-60 ELEMENT$ function - separators
Example 6-59 ELEMENT$ function - separators
Example 6-58 ELEMENT$ function
Example 6-57 ELEMENTS function
Example 6-56 EDIT$ function
Example 6-55 CPAD$ function
Example 6-54 CONVERT function
Example 6-53 CONVERT$ function - supported data types
Example 6-52 CHR$ function
Example 6-51 CHARSET$ function
Example 6-50 CHANGE$ function

Example 6-49 ASCII function
Example 6-48 Pivot Date Logical
Example 6-47 TIME$ function
Example 6-46 TIME(5) function
Example 6-45 TIME function
Example 6-44 SECONDS function
Example 6-43 FULLTIME$ function
Example 6-42 DAY$ function
Example 6-41 DAYS function - integer values
Example 6-40 DAYS function
Example 6-39 DATE$ function
Example 6-38 DATE function
Example 6-37 TANH function
Example 6-36 TAN function
Example 6-35 SQR function
Example 6-34 SINH function
Example 6-33 SIN function
Example 6-32 SGN function
Example 6-31 SEC function
Example 6-30 RAD function
Example 6-29 PI function
Example 6-28 LOG10 function
Example 6-27 LOG2 function
Example 6-26 LOG function
Example 6-25 EXP function
Example 6-24 DEG function
Example 6-23 CSC function
Example 6-22 COT function
Example 6-21 COSH function
Example 6-20 COS function
Example 6-19 ATN function
Example 6-18 ASIN function
Example 6-17 ANGLE function
Example 6-16 ACOS function
Example 6-15 ABS function
Example 6-14 TRUNCATE function
Example 6-13 ROUND function
Example 6-12 RND function
Example 6-11 REMAINDER function
Example 6-10 REAL function
Example 6-9 MOD function
Example 6-8 MIN function
Example 6-7 MAX function
Example 6-6 IP function
Example 6-5 INTEGER function
Example 6-4 INT function
Example 6-3 FP function
Example 6-2 DIV0 function
Example 6-1 CEIL function
Example 5-20 OPTION BASE statement
Example 5-19 REDIM statement
Example 5-18 DIM statement

Example 5-17 Private variables in routines
Example 5-16 RESTORE statement
Example 5-15 DATA and READ statements
Example 5-14 DATA items containing commas
Example 5-13 DATA, READ, RESTORE Statements
Example 5-12 LSET, RSET, CSET FILL statements
Example 5-11 CSET statement
Example 5-10 RSET statement
Example 5-9 LSET statement
Example 5-8 RSET statement
Example 5-7 LSET, RSET and CSET
Example 5-6 Assigning numeric values with LET statement
Example 5-5 LET statement
Example 5-4 OPTION REQUIRE DECLARE statement
Example 5-3 DECLARE STRUCTURE
Example 5-2 Declaring multiple data types
Example 5-1 DECLARE statement
Example 4-12 Bit manipulation
Example 4-11 Performing relational operations on strings
Example 4-10 String expression with string variable
Example 4-9 String expressions
Example 4-8 SheerPower data structure
Example 4-7 Substrings used to change string value
Example 4-6 Substrings
Example 4-5 String constants and delimiters
Example 4-4 BOOLEAN variables
Example 4-3 String data
Example 4-2 SheerPower Exact Math
Example 4-1 SheerPower Exact Math
Example 3-28 %INCLUDE program directive
Example 3-27 %MESSAGE ERROR program directive
Example 3-26 %MESSAGE program directive
Example 3-25 %COMPILE program directive
Example 3-24 Debug comments
Example 3-23 Comments with line continuation
Example 3-22 Routine header sample
Example 3-21 Comment text
Example 3-20 Comments in programs - (//)
Example 3-19 Implied continuation with '+', 'AND'
Example 3-18 Comma-separated list continuation
Example 3-17 Continuing program lines using the AMPERSAND (&)
Example 3-16 Multiple statements on a single line
Example 3-15 Error messages when passing routine parameters
Example 3-14 Passing parameters with private variables
Example 3-13 Private variables in routines
Example 3-12 Parameter Passing Using WITH and RETURNING
Example 3-11 REPEAT ROUTINE statement
Example 3-10 EXIT ROUTINE statement
Example 3-9 Executing routines/subroutines by name
Example 3-8 Private Routine - Example
Example 3-7 Namespace
Example 3-6 ROUTINE/END ROUTINE statements

Example 3-5 STOP statement
Example 3-4 END statement
Example 3-3 SheerPower Reserved Words
Example 3-2 Program statement
Example 3-1 SheerPower program example
Example 2-22 Using STEP in DEBUG system
Example 2-21 Using BREAK in DEBUG system
Example 2-20 Listing statistics in DEBUG system
Example 2-19 STATS ON
Example 2-18 TRACE ON/OFF
Example 2-17 DEBUG ON
Example 2-16 GO command
Example 2-15 Show files after using the HALT command
Example 2-14 HALT statement
Example 2-13 Listing program lines
Example 2-12 RUN command with file specification
Example 2-11 RUN command
Example 2-10 BUILD command in SP4GL Console Window with file specification
Example 2-9 BUILD command in SP4GL Console Window
Example 2-8 Program example
Example 2-7 Spelling correction
Example 2-6 Command completion
Example 2-5 Command completion
Example 2-4 Command recall in the SP4GL Console Window
Example 2-3 Print command in the SP4GL Console Window
Example 2-2 Entering multiple commands in the SP4GL Console Window
Example 2-1 Print command in the SP4GL Console Window
Example 1-7 Presentation of command and statement information
Example 1-6 SheerPower program error message
Example 1-5 Expression Evaluator Program Example
Example 1-4 NEWS Headline Program Example
Example 1-3 Quiz Program Example
Example 1-2 Menu Program
Example 1-1 Menu Program

SheerPower® 4GL
A Guide to the SheerPower Language

Begin	Index

Contents (summary)

Preface	Preface
Chapter 1	Getting Started in SheerPower
Chapter 2	Debugging and Experimenting in SP4GL Console Window
Chapter 3	Program Elements in SheerPower
Chapter 4	Data Types in SheerPower
Chapter 5	Assigning Variables and Defining Arrays
Chapter 6	Built-in Functions
Chapter 7	Printing and Displaying Data
Chapter 8	Data Entry User Interface
Chapter 9	Input Dialogbox - Creating GUI Forms with SheerPower
Chapter 10	Loops, Conditionals, and Chaining
Chapter 11	Set and Ask Statements
Chapter 12	SheerPower and DDE
Chapter 13	Exception Handling
Chapter 14	File Handling
Chapter 15	Data Structure Statements
Chapter 16	Database Setup
Chapter 17	SheerPower and ODBC
Chapter 18	SheerPower Internet Services (SPINS) Webserver
Chapter 19	Writing Network Applications and Accessing Devices
Chapter 20	SheerPower Web Scripting
Chapter 21	Calling Routines Written In Other Languages

Appendix A	Coding Principles and Standards
Appendix B	Reserved Words
Appendix C	SheerPower's Error and Exception Messages
Appendix D	ASCII Character Set
Appendix E	SheerPower Database Interfaces
Appendix F	Keystrokes for SheerPower Rapid Development Environment
Appendix G	Input Dialogbox--supported HTML tags
Appendix H	SPDEV Menu Item Descriptions
Appendix I	Developing Professional Applications with SheerPower
Appendix J	Sample SheerPower Programs Included with SheerPower 4GL Install
Appendix K	Generation Language
Appendix L	Troubleshooting the CGI Interface
Appendix M	SheerPower and Program Segmentation
Appendix N	Advanced Record Systems (ARS) Utilities
	Index
	Examples
	Tables

Contents

Preface	
Preface	Preface
Chapter 1	
1	Getting Started in SheerPower
1.1	Getting Started
1.2	Creating a New Program in SheerPower

1.2.1	Menu Program Example
1.2.2	Saving a Program in SheerPower
1.2.3	File Backup Versions
1.2.4	Purge Old Backup Files
1.2.5	Deploying the Menu Program Example
1.3	Deploying and Running a Program in SheerPower
1.3.1	SheerPower Deployment of Portable Runnable Applications
1.3.2	Running a SheerPower Program
1.4	Program Examples
1.5	Developing Professional Applications with SheerPower
1.6	Open Existing Files in SPDEV
1.7	Default file location and file types
1.8	SheerPower Program Error Messages
1.9	SP4GL Console Window
1.9.1	SP4GL Console Window Keystrokes
1.10	Getting Help in SheerPower
1.11	Conventions Used in this Manual
1.12	Presentation of Command and Statement Information
1.13	Other SheerPower Features
Chapter 2	
2	Debugging and Experimenting in SP4GL Console Window
2.1	SheerPower Commands and Statements
2.1.1	Recalling Previous Commands in SheerPower
2.1.2	Using the `[Tab]` Key Features
2.1.2.1	Command Completion in SheerPower
2.1.2.2	Spelling Correction of Commands in SheerPower
2.1.2.3	Expanding Abbreviations
2.2	Creating a Sample Program
2.3	Errors and Exceptions

2.4	Using Commands in SP4GL Console Window
2.4.1	BUILD
2.4.2	RUN
2.4.3	LIST
2.4.4	HALT Statement
2.4.5	SHOW FILES
2.4.6	GO
2.5	Debug Facilities
2.5.1	DEBUG ON
2.5.2	DEBUG OFF
2.5.3	TRACING APPLICATION LOGIC FLOW
2.5.4	STATS ON
2.5.5	STATS OFF
2.5.6	LIST STATS
2.5.7	BREAK Statement
2.5.8	STEP
2.6	SP4GL Console Window Keystrokes
Chapter 3	
3	Program Elements in SheerPower
3.1	Storing SheerPower Programs
3.2	SheerPower Program Elements
3.2.1	PROGRAM
3.2.2	SheerPower Reserved Words
3.3	SheerPower Program Structure
3.3.1	END
3.3.2	STOP
3.4	ROUTINE/END ROUTINE
3.4.1	Understanding variable "namespace" when using routines
3.4.2	More on Routines

3.4.3	EXIT ROUTINE
3.4.4	REPEAT ROUTINE
3.5	Passing Optional Parameters to Routines
3.5.1	Parameter Passing Using WITH and RETURNING
3.5.2	Private Variables in Routines
3.5.3	Passing Parameters with Private Variables
3.5.4	Error Messages when Passing Routine Parameters
3.6	Program Lines
3.6.1	Continuing Program Lines and Implied Continuation
3.6.1.1	Implied Continuation
3.7	Comments
3.7.1	(//) comment_text
3.7.2	Routine Headers
3.7.3	The (//) with Line Continuation
3.8	Debug Comments in SPDEV
3.9	Program and Compile Directives
3.9.1	%COMPILE
3.9.2	%MESSAGE
3.9.3	%MESSAGE ERROR
3.9.4	%INCLUDE
3.9.5	%INCLUDE CONDITIONAL
Chapter 4	
4	Data Types in SheerPower
4.1	Data Types
4.1.1	Integers
4.1.2	Real Numbers
4.1.3	String Data
4.2	BOOLEAN
4.3	Expressions

4.4	Constants
4.4.1	Numeric Constants
4.4.2	String Constants
4.5	Variables
4.5.1	Arrays
4.5.2	Substrings
4.6	Structure References
4.7	Multiple Occurrence Fields
4.8	Compound Expressions
4.8.1	Numeric Expressions
4.8.2	String Expressions
4.8.2.1	Conditional Expressions
4.8.3	Conditional Numeric Expressions
4.9	Performing Relational Operations on Strings
4.9.1	Logical Operators
4.9.2	Bit Manipulation
4.10	Order of Evaluation
4.11	Using Parenthesis for Clarity

SheerPower® 4GL
A Guide to the SheerPower Language

Index

Master Alphabetic Index

4

4GL - What is it?

A

ABBREVIATIONS in SPDEV
ABS function
ACCESS - field definition item
ACCESS INPUT
ACCESS option in OPEN #chnl_num
ACCESS options in OPEN STRUCTURE
ACCESS OUTIN
Access Rules in SETUP
Accessing ODBC with SheerPower
Accessing Password Protected ODBC Databases
ACOS function
ADD STRUCTURE/END ADD
Add structure record

- cancel add
- exit add

ADDRESS tag

ADVANCED menu item in SPDEV

Advanced Record System

All Keystrokes

Alt Keystrokes

Ambiguous constant

Ampersand in line continuation

Anchor tag

ANGLE function

Append data to a file with outin

APPEND option in EXTRACT STRUCTURE

Application Example

APPLICATION - field definition item

AREA option in INPUT

AREA option of CLEAR statement

Arithmetic operators

Array
- defining
- description
- DIM statement
- dimension
- elements
- low bound
- OPTION BASE statement
- REDIM statement
- subscript

ARS #1

- ARS #2
- ARS Help Text
- ARS interface
- ARS Utilities
    - ARS2FDL
    - ARSBACKUP
    - ARSCHK
    - ARSFIX
    - ARSRESTORE
    - FDL2ARS
- ARS2FDL
- ARSBACKUP
- ARSCHK
- ARSFIX
- ARSRESTORE
- @ as a logical
- ASCII character set #1
- ASCII character set #2
- ASCII function
- ASIN function
- ASK #chnl_num options
    - CURRENT
    - MARGIN
    - NAME
    - ZONEWIDTH
- ASK #chnl_num statement
- ASK ERRORS statement
- ASK KEYSTROKES statement
- ASK MARGIN statement

ASK PAGESIZE statement
ASK RESPONSES statement
ASK/SET SEED statement
ASK/SET SYSTEM, SYMBOL 'OS:xxx': VALUE str$
ASK STRUCTURE: ACCESS
ASK STRUCTURE: CAPABILITY
ASK STRUCTURE: CURRENT
ASK STRUCTURE: DATAFILE
ASK STRUCTURE: ENGINE
ASK STRUCTURE: EXTRACTED
ASK STRUCTURE FIELD: item
- ACCESS
- APPLICATION
- ATTRIBUTES
- CHANGEABLE
- DATATYPE
- DESCRIPTION
- field expressions
- HEADING
- HELP
- KEYED
- LENGTH
- NAME
- NULL
- NUMBER
- OPTIMIZED
- POSITION
- PRINTMASK
- PROMPT

SCREENMASK
VRULES
ASK STRUCTURE: FIELDS
ASK STRUCTURE: ID #1
ASK STRUCTURE: ID #2
ASK STRUCTURE: KEYS
ASK STRUCTURE options
ASK STRUCTURE: POINTER
ASK STRUCTURE: RECORDSIZE
ASK STRUCTURE #string_expr...
ASK SYSTEM: COMMENT statement
ASK SYSTEM: DIRECTORY statement #1
ASK SYSTEM: DIRECTORY statement #2
ASK SYSTEM, LOGICAL: VALUE statement
ASK SYSTEM: MODE statement
ASK SYSTEM: PARAMETER statement
ASK SYSTEM: PROCESS statement
ASK SYSTEM: PROGRAM statement
ASK SYSTEM: RIGHTS statement
ASK SYSTEM, SYMBOL 'DNS:XXX' FOR DNS LOOKUPS
ASK SYSTEM, SYMBOL: VALUE statement
ASK SYSTEM: USER statement
ASK WINDOW AREA statement
ASK WINDOW: COLUMN statement
ASK WINDOW: CURRENT statement
ASK WINDOW: DATA statement
ASK WINDOW: KEYMAP statement
ASK WINDOW: ROW statement
ASK WINDOW: TYPEAHEAD statement

ASK ZONEWIDTH statement

Assigning Byte Values to Keystrokes

Assigning variables

AT row, column in PRINT statement

%AT row, column menu directive

AT row, column option in INPUT

ATN function

Attached attribute - input dialogbox SheerPower tag

%ATTACHED menu directive

ATTRIBUTES - field definition item

ATTRIBUTES option in INPUT

Autoscale attribute - SheerPower tag

Autosubmit attribute - SheerPower tag

B

BACKGROUND [image] attribute - SheerPower tag

Backup ARS Database

Backup Feature in SPDEV

Backups - purge files

Bad syntax for execute command with DDE

%BAR menu directive

Basic Function of Input Dialogbox

Basic program elements

Batch files and findfile$()

Benchmark program

Binary Files-read and write in SheerPower

Bit manipulation

BLINK - screen attribute

BLOCKQUOTE tag

BOLD - screen attribute

BOLD tag
BOOLEAN variables
    trailing ?
    TRUE/FALSE conditions
BOX option of CLEAR AREA
BREAK statement in debug system
Browser - Invoke SheerPower Script
Browsing files-specify root level
Build
Building CGI Custom HTTP Headers
Built-in functions
Byte Values - assigning to keystrokes

C

Call routines written in other languages
CALL statement #1
CALL statement #2
Callable routines in libraries
Called routine
CANCEL ADD
Cancel adding a structure record
CANCEL EXTRACT
CASE ELSE statement
CASE IS statement
CASE statement #1
CASE statement #2
CAUSE EXCEPTION
CD Player sample program
CEIL function
CENTER tag

Centering data, as in a heading
CGI - Building Custom HTTP Headers
CGI - Custom HTTP headers
CGI Environment Symbols
CGI Environment Variables
CGI environment variables
CGI - EVAL_HANDLER Program
CGI handler name
CGI - how eval_handler program works
CGI-how it works
CGI Interface
CGI Interface Performance Considerations
CGI Interface sample program #1
CGI Interface sample program #2
CGI Interface Test
CGI Interface testing
CGI Interface - troubleshooting
CGI - open connection
CGI Performance
CGI - Processing a request from the SPINS server
CGI - processing requests
CGI requests - waiting for
CGI Symbol lookups inside script areas
CGI Troubleshooting Checklist
CGI - waiting for requests
CHAIN statement
Chaining programs
CHANGE$ function
Change System Settings in SPDEV

CHANGEABLE - field definition item
Channel number #1
Channel number #2
Channel number in PRINT statement
CHARSET$ function
Check ARS Database
#chnl_num in PRINT statement
CHR$ function
Classification in SETUP
Clean build
CLEAR AREA #1
CLEAR AREA #2
    BOLD, BLINK, REVERSE attributes
CLEAR AREA BOX
    BOLD, BLINK, REVERSE attributes
CLEAR statement
    AREA option
        BOX option
Clearing the screen
CLOSE ALL
CLOSE ALL statement
CLOSE #chnl_num statement
CLOSE STRUCTURE
Closing a structure
Closing files
Code Area in Matrix.spsrc
Code Areas in Web Scripting
Coding principles and standards
Coding Standards Manual

Column on screen
%COLUMNS menu directive
COM Port
Comma in PRINT statement
Command completion with the TAB key
Command presentation
Command recall in the SP4GL Console Window
Command spelling correction
Command Statement
Command - what it does
Commands
- build
- go
- list
- run
- show files
- SP4GL Console Window

// comment
Comments in programs
COMMIT Statement with UNLOCK STRUCTURE
Common Math Functions
Communication Port Support
COMPARE function
Compile directives
Compile error messages
Compile errors
%COMPILE program directive
Compiling a program
Compound expressions

- concatenated string expressions
- conditional expressions
- numeric expressions

Concatenated string expressions

Conditional expressions

Conditional numeric expressions

Conditionals

console window

Console Window Commands

Console Window Errors and Exceptions

console window Keystrokes #1

console window Keystrokes #2

Constants
- description
- numeric - integer
- numeric - real
- string

Constructs - description

CONTENTS in FILEINFO$

CONTINUE statement in HANDLER routine

Continuing lines

Continuing program lines

Conventions
- other
- program examples
- user

Conventions used in manual

CONVERT function

CONVERT$ function

Coordinates on screen
Copy Keystrokes
COS function
COSH function
COT function
CPAD$ function
Create a file using OPEN #chnl_num
Create a file using OPEN FILE num_var:
Create ARS from FDL
Create Data File in SETUP
Create FDL from ARS
Creating a Routine Header Template
Creating a Routine Template
Creating a sample program for use in the console window
Creating the Data File in SETUP
CSC function
CSET FILL statement
CSET statement #1
CSET statement #2
Ctrl Keystrokes
Current file - purge backup versions
Cursor positioning in PRINT statement
Custom CGI HTTP Headers
Custom Keymapping
Custom Keymapping in SPDEV
Custom SUBMIT Buttons with Input Dialogbox
Cut Keystrokes