NATIONAL DATA PARSING CORPORATION

Data Parse™ Software Editions
Version 5.00.xx

Data Parse Users Guide

DATA PARSE SOFTWARE EDITIONS

Data Parse™ Users Guide

Revision 6.42

---

12 Barns Lane, Unit 1
East Hampton, New York 11937
Phone Toll Free 1-877-99-PARSE
Phone +1 631-482-3289

Table of Contents

Chapter 1

Introduction

What is Data Parse?

Data Parse is data processing technology from National Data Parsing Corporation It is used by programs such as **Data Parse Free Edition**, **Data Parse Basic**, **Data Parse Business** and **Data Parse Enterprise**— our programmable file-parsers.

Data Parse (all editions) is a programmable file-parser. It can help you out in countless ways. If you have a file you want to edit, manipulate, or change around, this may be just the tool you need. **Data Parse** can also speed up or automate long, repetitive editing tasks, including clipboard manipulation.

Data Parse Versus Automatic Converters

Data Parse is not an "automatic file converter". It will not, for example, convert PDF files to MS-Word format, or convert Lotus 1-2-3 Spreadsheets *directly* to Excel files — although it can read reports from one program and convert them to another format (such as comma-delimited), which can be imported by the other program.

One advantage of this method (as opposed to automatic file conversion) is that you can create an "intelligent" importing procedure, which can make decisions and modify data. You could, for example, eliminate certain types of records, tidy up names, convert case, unify fields, perform calculations, and so on.

Why You Need Data Parse — An Example

There are plenty of programs out there that have valuable data locked away inside them. How do you get that data *out* of one program and into another one?

Some programs provide a feature which "exports" a file into some kind of generic format. One of the most popular of these formats is known as "comma-delimited" (also known as CSV, which stands for "Comma-Separated Value"), which is a text file in which each data field is separated by a comma. Character strings — which might themselves contain commas — are surrounded by double quotes. So a few lines from a comma-delimited file might look something like this (an export from a hypothetical database of people who owe your company money):

```
"JONES","FRED","1234 GREEN AVENUE","KANSAS CITY","MO",293.64
"SMITH","JOHN","2343 OAK STREET","NEW YORK","NY",22.50
"WILLIAMS","JOSEPH","23 GARDEN CRESCENT","TORONTO","ON",16.99
```

Unfortunately, not all programs export or import data in this format. Even more frustrating is a program that exports data in a format that is *almost* what you need!

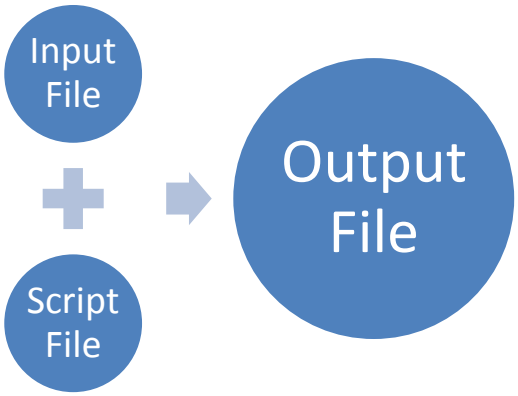
If that's the case, you might decide to spend a few hours in a text editor, modifying the export file so that the other program can understand it. Or you might write a program to do the editing for you. Both solutions are time-consuming.

An even more challenging problem arises when a program which has no export capability does have the ability to "print" reports to a file. You can write a program to read these files and convert them to something you can use, but this can be a *lot* of work!

Data Parse to the Rescue!

Data Parse reads a file, interprets the data, and outputs the result to another file. It can help you "boil down" data to its essential information. You can also use it to convert *nearly* compatible import files, or generate printable reports.

How It Works



To process data with Data Parse, you need three things:

1. The Data Parse program
2. A Data Parse script file to tell Data Parse what to do
3. The input file

The input file might be a report or data file from another program, or text captured from a communications session. Data Parse can handle many types of input. We've provided several sample input files. For example, the file `ThingsToDo.txt` is a

simple "To Do" list. If you want to modify such a file in various ways, Data Parse can help!

Data Parse works by running the entire script every time a new record is loaded from the input file. You simply need to tell Data Parse the name of the input, output and script files and click a button. (You can also automate the process by calling Data Parse from the task scheduler, a batch file, or another program.)

Advantages of Data Parse

Data Parse has evolved over more than two decades to accomplish a single task: extracting and manipulating data contained in "flat" files. Its scripts are written with a loopless, top-to-bottom rationale so that you do not have to spend time writing code to load each record from the input file — Data Parse handles that for you.

In addition, you do not have to declare variables, and the extraction commands (such as Parse and ScanPosn) are extremely powerful — designed specifically for the challenges that arise when trying to extract data from files.

Some of our clients have told us that they save hundreds of dollars in labor costs every time they write a Data Parse script instead of using a traditional programming language.

Once you have mastered Data Parse Scripting, you may find that you are regularly using it for tasks that would previously have been too time-consuming. Just about everyone has files that they would like to filter or reformat. Without the right tool these operations are sometimes too difficult to even attempt. With Data Parse, though, they can often be done in just a few minutes.

Sample Scripts

Data Parse comes with several demonstration scripts. To try one out, start up Data Parse. Then, select File, Open, Solution. You'll find the solutions in the Samples subdirectory, which was created when you initially installed the application.

Select one of the Solutions (such as `ScriptSample01`), then click on the Run button in the toolbar.

Once processing is complete, you will see the resulting output. You can also double-click the script in the Solution Explorer window to study the script that you just ran.

In addition to the sample scripts included with Data Parse, you can find additional sample scripts in the National Data Parsing Corporation Knowledge Base, available at www.Data Parse.com.

How to Contact Us

If you have any questions about Data Parse, you can contact us in the following ways:

Voice Line:	+1 631- 482-3289
Email:	support@dataparse.com
Web Site:	www.DataParse.com

You can also write to us at the following address:

National Data Parsing Corporation
12 Barns Lane, Unit 1
East Hampton, New York 11937

Chapter

2

User Interface

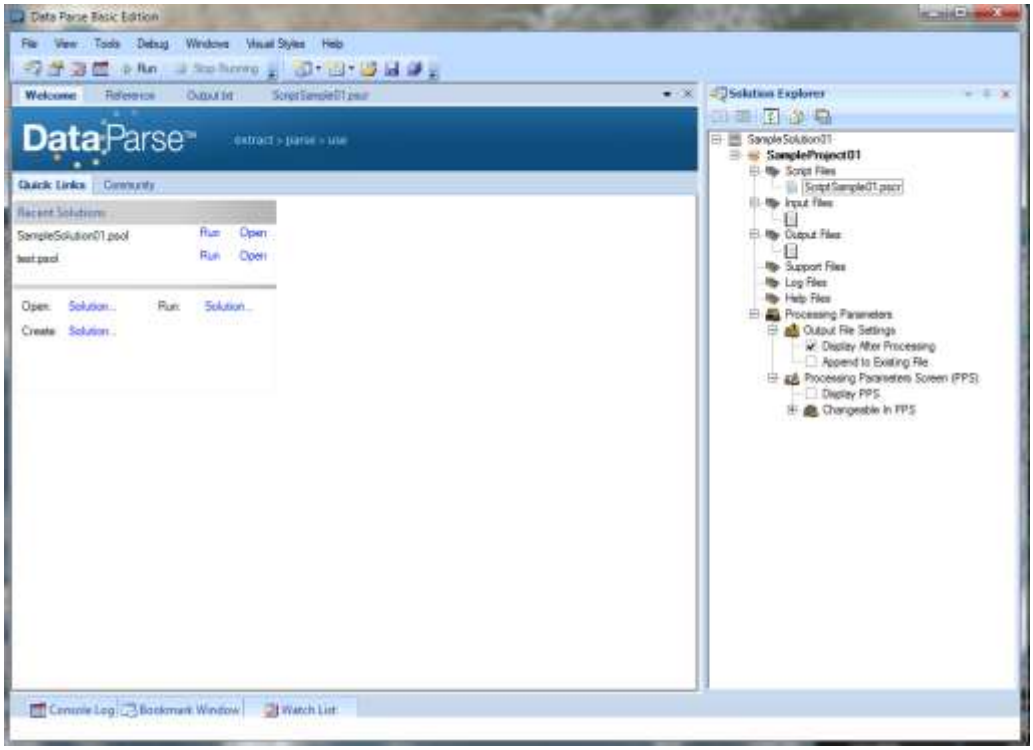
An Integrated Development Environment (IDE)

When you start Data Parse, the integrated IDE opens up a main window, a side window and bottom window.

The main window shows your most recent projects. It also shows you the options to create new projects and run existing projects.

The side window allows you to open the Solutions Explorer for a particular solution. You can view the details of the solution such as input and output file names, script file names etc. For each of the solution's objects, the properties window shows details on the solution and project properties along with the Bookmark, Breakpoint and Watch DS details. These will be explained in the respective sections of bookmark window and watch list respectively.

The bottom window has four tabs each of these showing the results of the solution, the debug console window showing errors if any, the bookmark window and the watch list.



An IDE provides the following advantages:

1. User Friendly GUI – the user interface portrays a professional theme and allows you to change the look and feel of the IDE. It is also quite easy to understand for first-time users of Data Parse, especially those who are familiar with other development environments.

2. Support of multiple, parallel user operations – the multi-window view of the IDE allows you to run a solution on the one hand, view the properties of script files on the other and also see the console log for any debugging exceptions. You can create multiple projects and also set their order of execution.

3. Color-coded development – you can handle scripts in the same way that code is handled in many commercial IDEs. The Data Parse IDE provides color-coded distinction in the various parts of the script code to differentiate between comments, actual code, code blocks, etc.

4. Deployables™: This is a feature available in the Enterprise edition of Data Parse. This is similar to an .exe file, which can be run by you without making use of the Integrated IDE of the Data Parse program. This is explained in further sections of this manual.

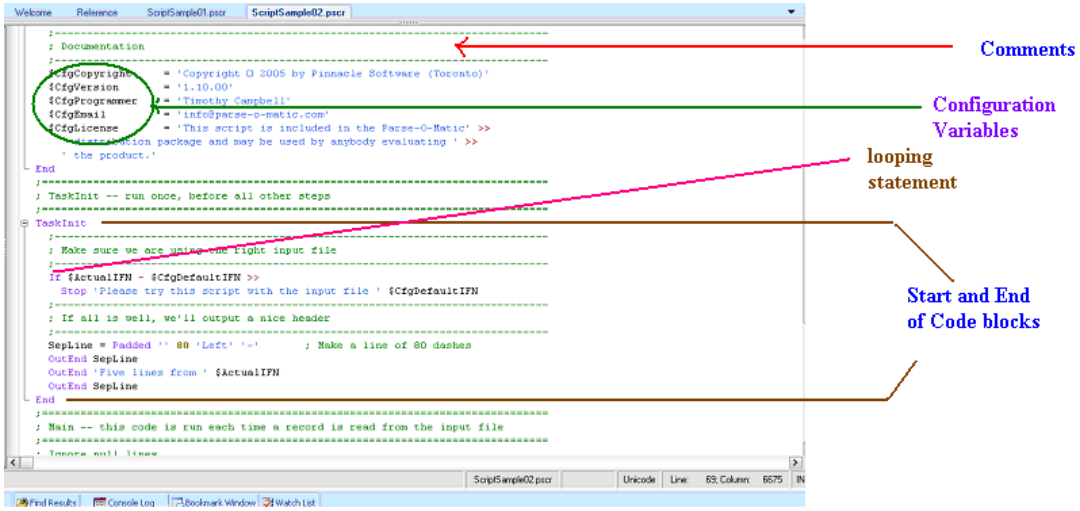
Color-coded Development

Color coded development is a feature that allows you to easily identify parts of the code depending on whether the code has comments, looping statements or variables. With color-coded development:

1. The application highlights the code in such a way that it is easy for you to identify beginning and ending of a code block, defining variables and reserved words and distinguishing between the two types and differentiating the files reference in the script from the actual code. This makes coding the script easier.

2. Another use of color-coded development is in easier maintenance of your code.

3. Color-coding also helps to prevent errors while writing the script. An example might be that as a developer you might use a reserved word as a variable in the script code but because reserved words are colored differently from user-defined variables.



Snapshot of a script

Intellisense

While editing a script file, you can press CTRL-Spacebar after typing the first few letters of a script command, and you will be shown a list of the parameters and a mini-help guide to that command.

The features of color-coding in the Data Parse IDE are:

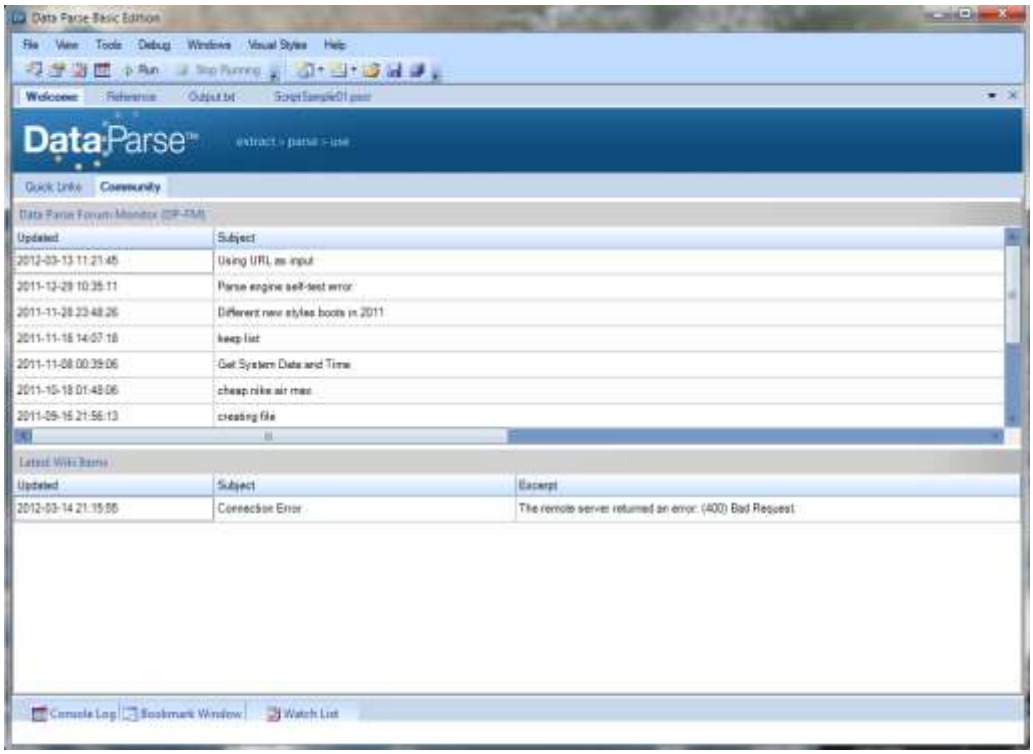
- Configuration variables always start with $ sign and are marked black
- Code comments always start with ';' and are marked green
- Looping statements such as if are marked violet in color

- User variables and printing statements are marked in blue
- Reserved words are also marked in violet color.
- Arrays are marked in maroon color
- Each script has a configuration section with Config and End statements and a TaskInit and End statement block containing script code
- Number assignments are marked in light golden color

You can expand and collapse code blocks such as the configuration code block and TaskInit code block.

Quick Links, Integrated Reference manuals and Community sections

These features are visible to you on opening the Data Parse application.



In the Quick Links section, you can see the following:

View Recent Solution Files – you can view the list of your recent solution files

Create a New Solution File – you can click on this link to create a new solution file

Run a Solution File – you can run a solution file by clicking on this link

Open a recent Solution File – you can open the most recently used solution files

More information on solution files and other components is explained in the following sections.

Integrated Reference Manual:

As a user, you can view the reference manuals and tutorials for help on various features of Data Parse. These reference manuals provide a quick reference guide to scripting, a quick start guide for first time users of Data Parse and a full-fledged user manual.

Community Section:

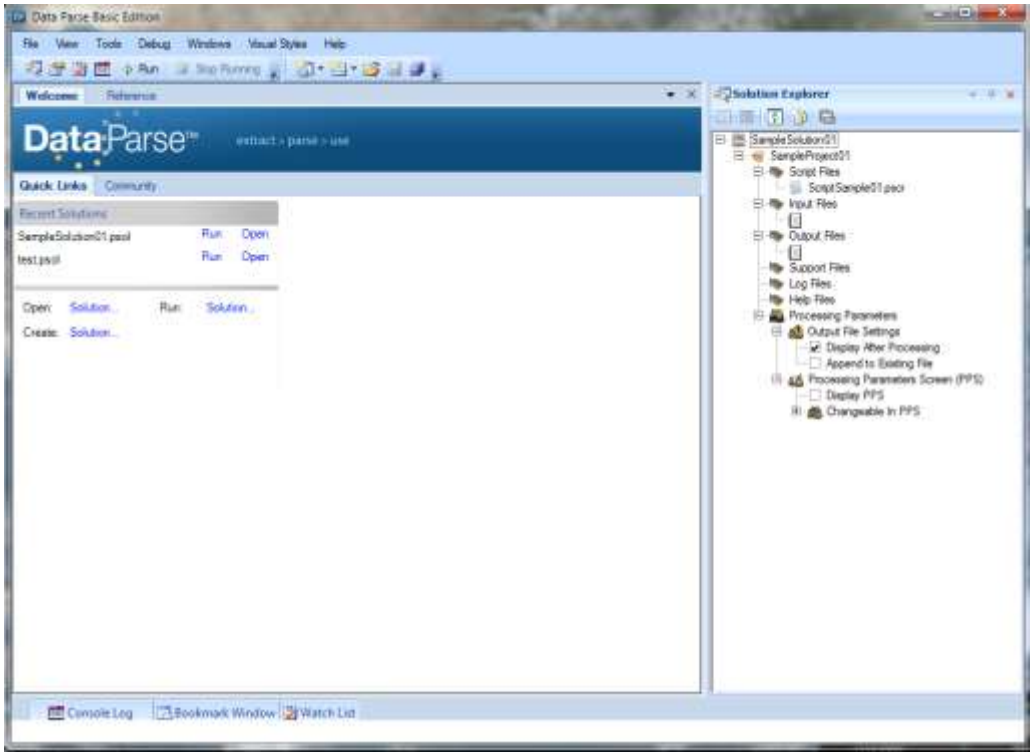
In this section, you can view forum discussions and wiki items, once you are connected to the Internet. You can see the latest forum posts and wiki items in order of their modified date.

This provides you with additional help and understanding of issues encountered during the run of Data Parse. The data is taken from live discussions on the forum and on Wiki, so this data is always up-to-date.

Solution Files, Projects and Script Files:
A solution comprises of the following:

- A project file – This forms the class files of the input, output, support and other files. This file has an extension of .ppro.
- An input and an output file – these files comprise of content, which has to be parsed and content obtained after parsing respectively.
- A script file – this specifies the actions to be performed while parsing the input file. It is usually written using the scripting features available and explained in the previous chapters. This file has an extension of .pscr.
- Support Files, Log files and help files – support files are used in addition to the input files, to run a script and view the parse results. Log files would save the log results after a solution has been run. Help files can be added to the solution for understanding and information on the script file.
- The solution file is saved with the extension of .psol and is run if parsing has to be done on a given set of input files.

The Solution Explorer on the right hand side window shows these objects, when a solution is created or selected:



In addition to these files, the solution explorer also shows 'Processing Parameters'. These are settings that you can specify just prior to running solutions.

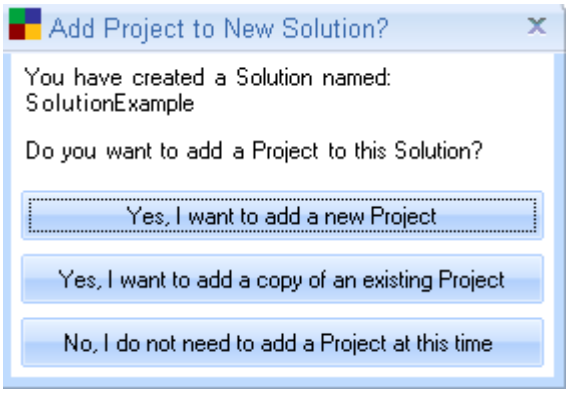
If you check "Display after processing" under Output file settings, then you can view the results of the parsing in the output file. If you select "append to existing file", then the application appends the result of the output to the input file.

The PPS or Processing Parameter Screen is only showed once you start/run/execute a project. When you check the Display PPS checkbox, Data Parse shows a separate window after the processing of script and input file is done. You can also specify what can be changed through the PPS as is seen from the diagram, namely Script File Name, Input File Name, Output File Name, Help File Name etc.

Adding a Solution

When you want to create a solution file, you should select File, click on New and select Solution. Data Parse asks you to input the solution's name, and by default the solutions are saved under the Solutions Folder. Once you save the details of the new solution, the application prompts you to add more components to the solution such as a new project, scripts etc.

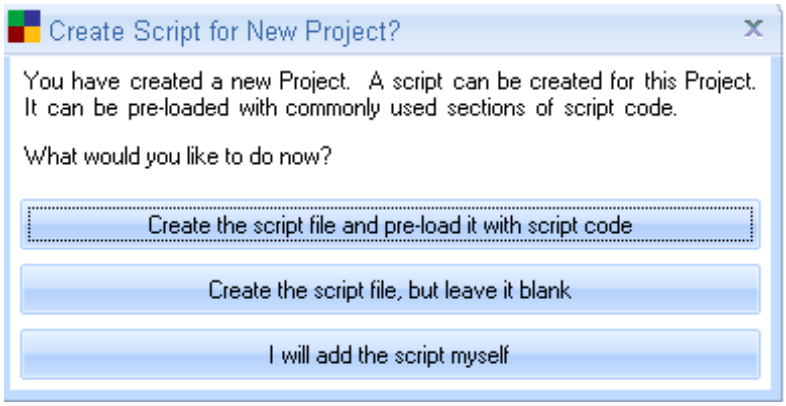
Adding a Project



When you add a new solution, the next thing to be added is a project. You will be prompted with three choices – to add a new project, add an existing project or skipping the addition of a new project. A project file is needed to compile the results of scripts. However, if you choose not to add projects but add only scripts, then the application returns back to the solutions explorer. You have to then manually add projects to the solution by right clicking on the solution name in the solution explorer.

Adding a script

When a new project has been added, Data Parse prompts you to add scripts.



You will have three choices – to create the script file with pre-loaded code from the Data Parse server, create a blank script or adding the script manually. If you choose the first option then a script file with basic code is loaded. If a blank script is created, you need to manually enter all code. If you chose to add script later, then you have to right click on the project name and add a script.

Adding input and output files

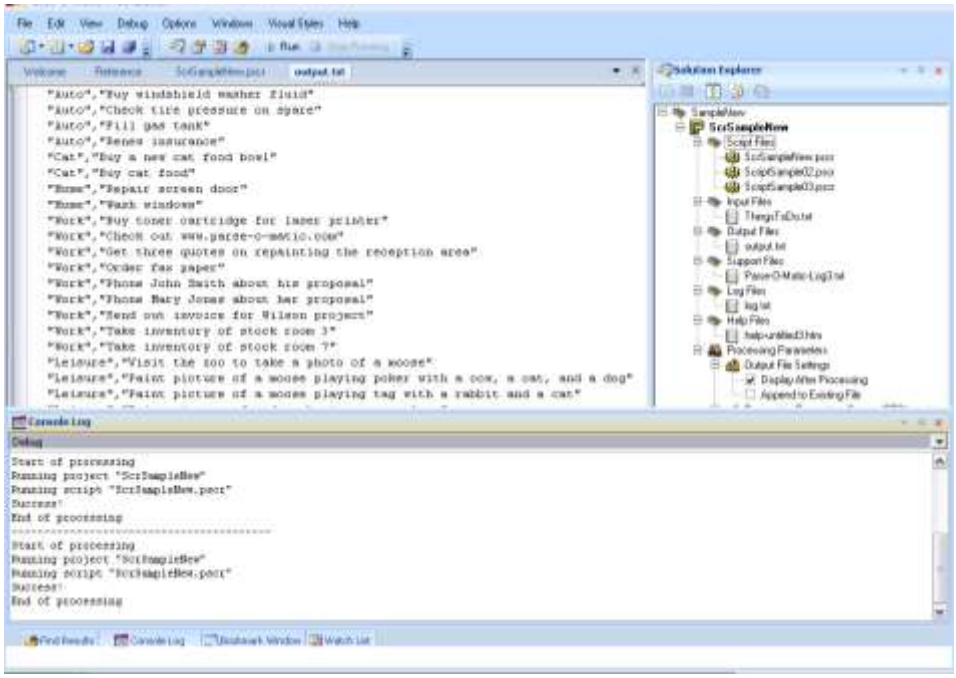
You have to point to the solution explorer to add the input file – the file which is required to be parsed and add a output file – this file would contain the results of parsing.

Both of these files can be added manually, else the application will assign an output file based on the input file specified by you.

Multi-Script Execution

You can add multiple script files to the solution and get multiple outputs at the same time. This feature allows you to parse one file in multiple ways with multiple scripts to process at the same time.

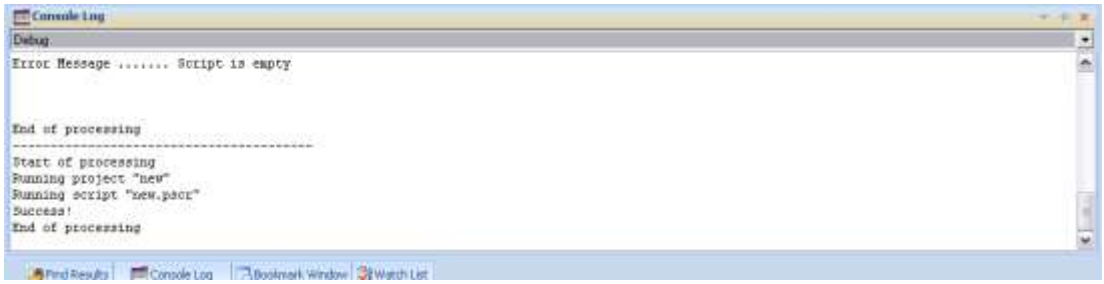
As shown below, the sample solution has more than 1 script file and generates output in more than one way.



Debugger

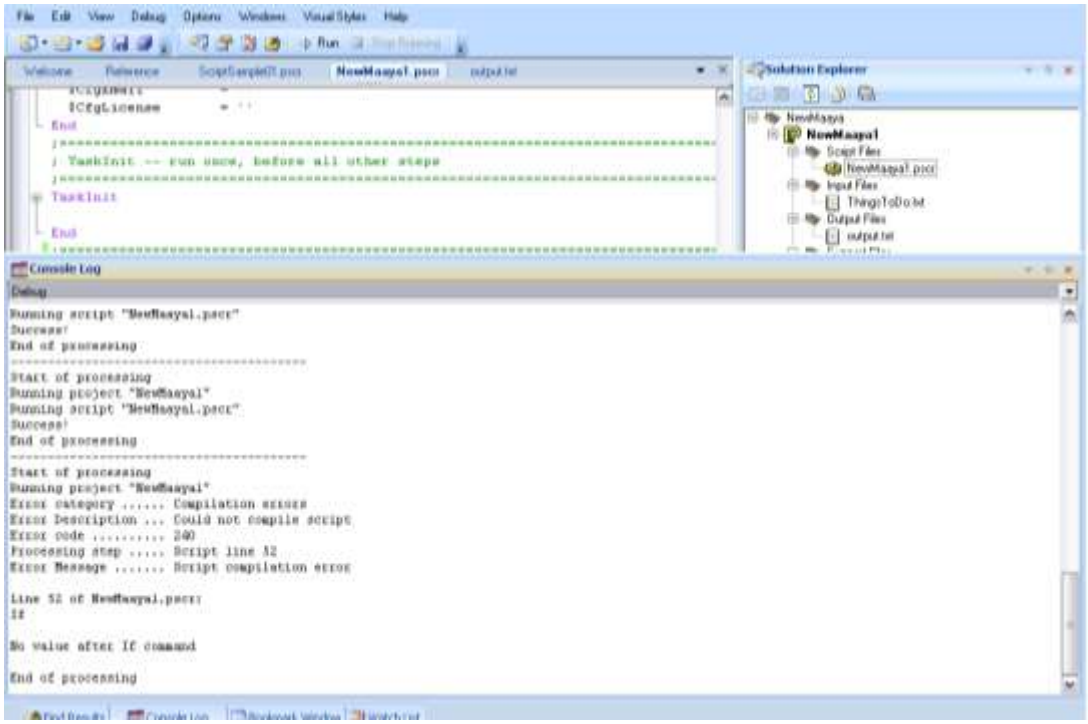
This feature allows you to debug scripts either before running the solution or individually. A script can be debugged at the time of creation if you choose to test the functionality of the script. Based on the debugging results, you may or may not make changes to the script file. It is also an option for you to write a script completely before debugging.

The debugging option is optional and it is up to you to debug scripts. You can also set break points while debugging in order to run the script one step at a time. The Step Into functionality allows you to do so. This enables you to execute script parts so that exceptions noted in the debug console window can be noted and if need be, can be rectified.

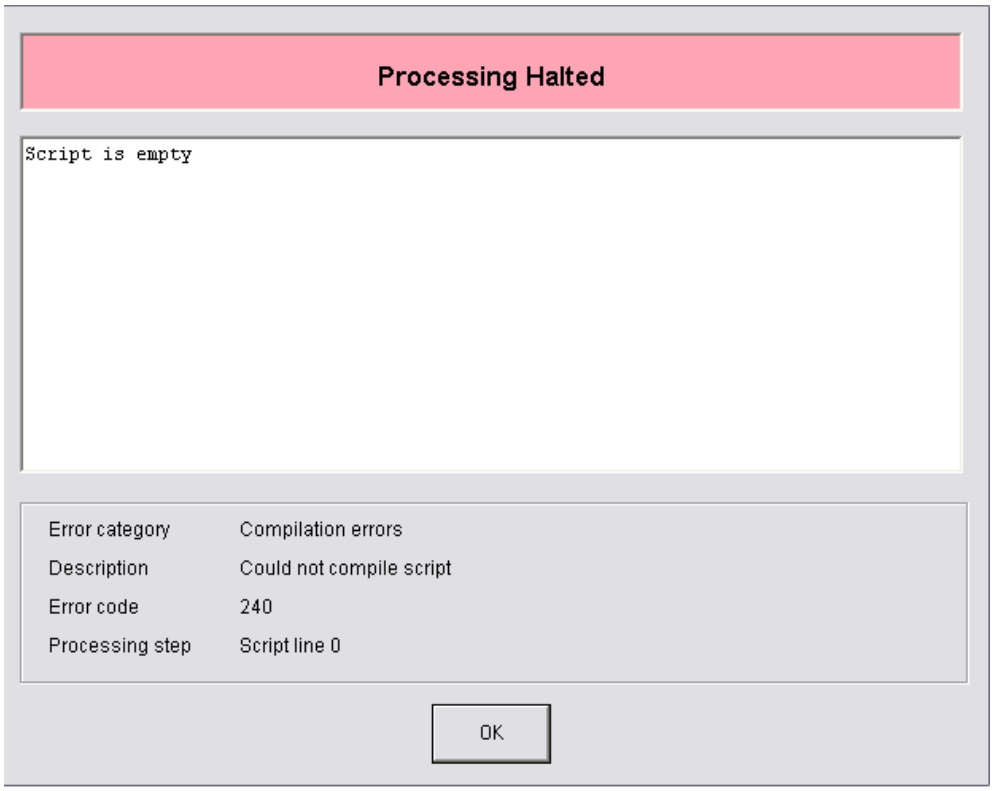


The console log window is used to show the error or success message of debugging or stepping-in the code.

This window also displays additional details, which includes a log of the projects executed, the names of the scripts executed and whether the execution was a success or failure. It shows the scripts that have errors and the scripts that have run successfully.



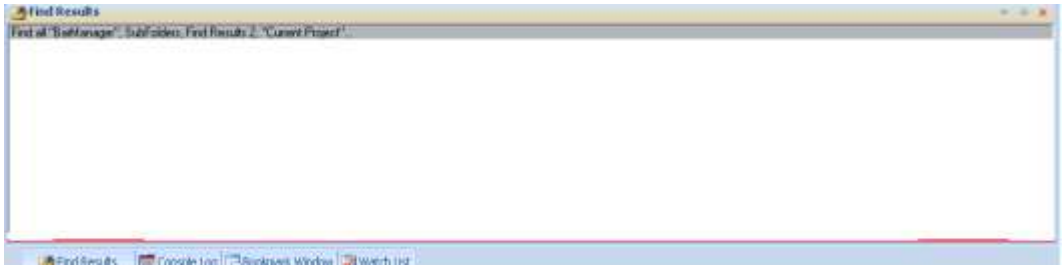
The errors are displayed both in a dialog box and in the debug console window. Errors would also be displayed in case the script file is debugged before it is completed or in case of programmatic errors:



As noted above, Data Parse shows the error category of the error, the description of the error, the error code and the line of error. This helps you to locate the cause of the error and the error itself quite easily. This design is consistent with other commercial compiler designs and IDE.

Results Log

You can view the results window to locate specific searches within the solutions folder. This window is accessible through the main toolbar and menu bar.



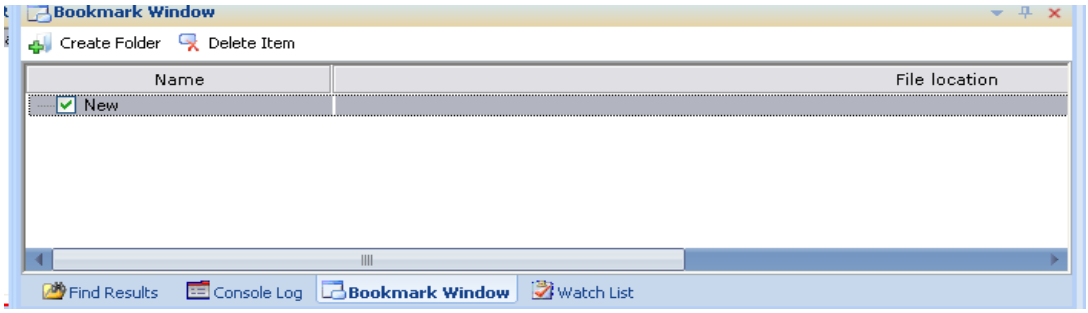
Watch List

The variables inside a script can be marked and added to the watch list, where you can see the values for these variables while running a script.

When you click on the watch list, double clicking on the name field allows you to enter the name of the variable that needs to be tracked through the watch list. When this feature is used with breakpoints, you can debug scripts effectively and note the exceptions in detail. If the checkbox for the Special Variable is checked then it means it is reserved variable like $Data.

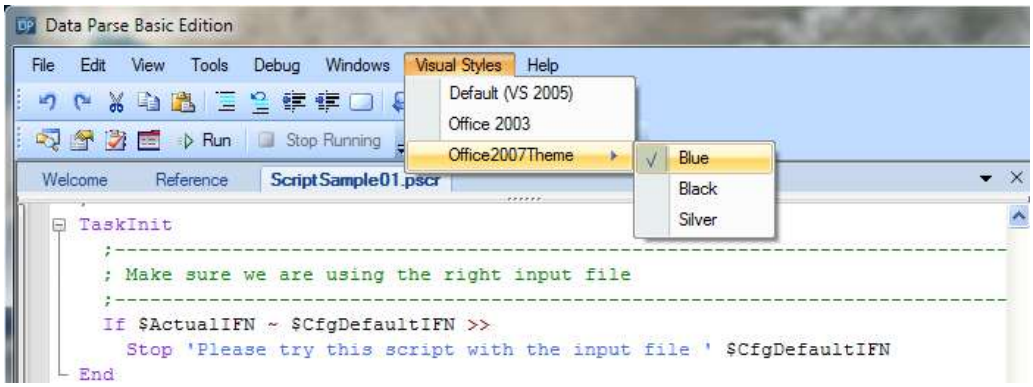
Bookmark Window

This window allows you to bookmark folders. For creating a bookmark, you just need to click on create folder and add it as a bookmark. It is a way of adding a quick reference for later uses, as you can add the file name and the line inside the file.



You can toggle bookmarks and add them to the bookmark window, by clicking on the 'Toggle bookmark icon on the right-hand side of the main window. This gets automatically added to the bookmark window. You can browse through bookmarks as well, by clicking Prev and Next Bookmark options, next to the Toggle bookmark icons.

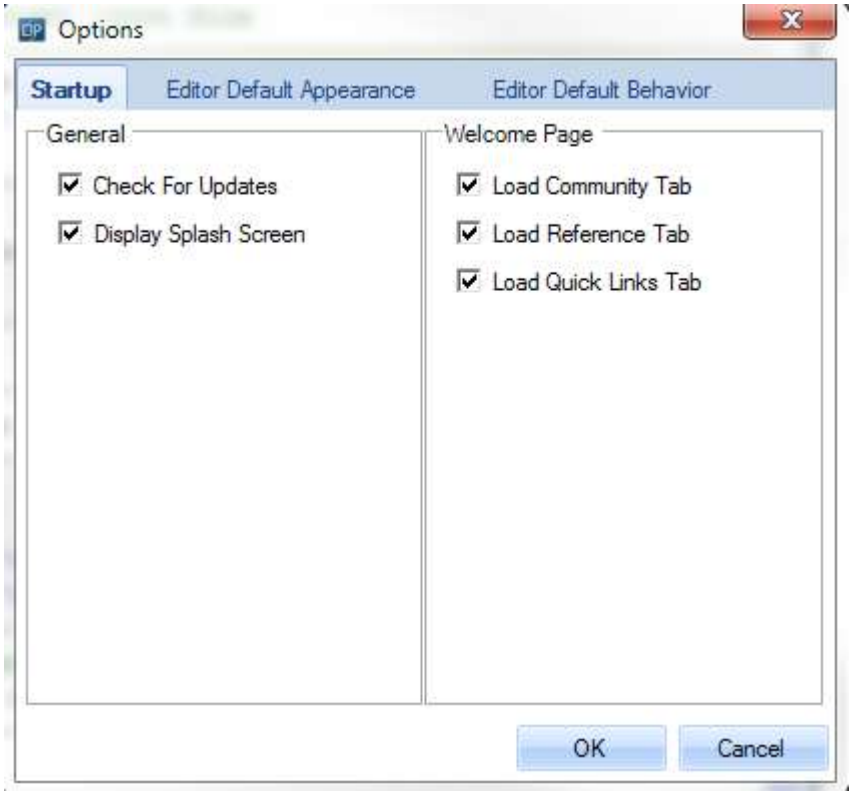
Visual Style Options



The IDE allows you to change the look and feel of the editor. You can choose from three different visual schemes – default, office 2003 and office 2007 theme.

IDE Options for tailoring the environment

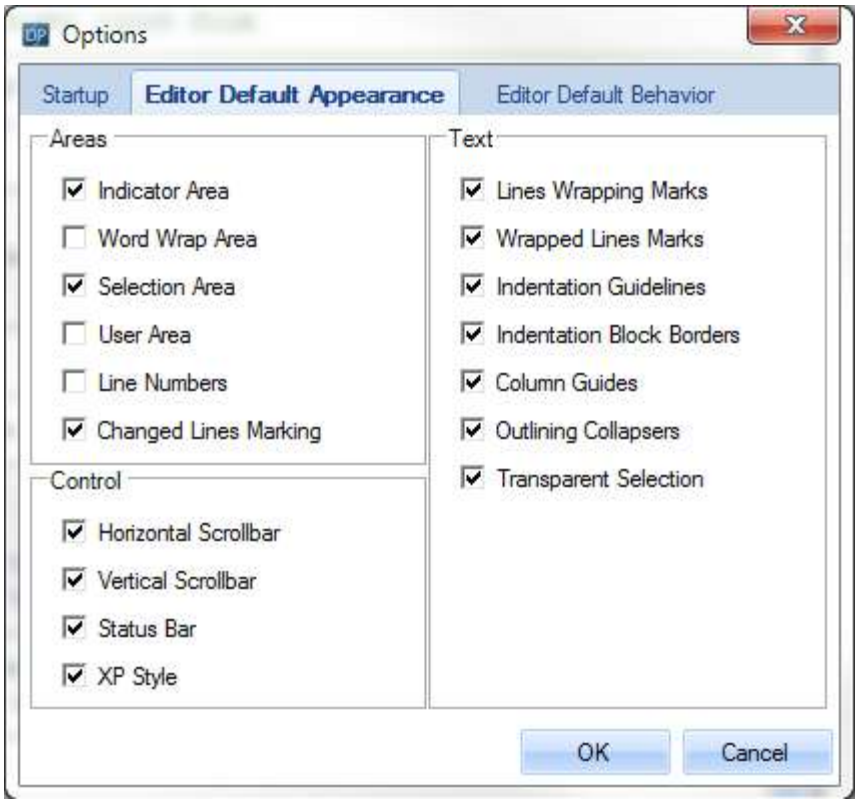
The IDE offers you with some editor options, which can be used to customize the behavior and appearance of the application when it is used.



You can set the windows to be loaded on starting up Data Parse.

If you check "Check for Updates" then the application connects to the Internet to view updates of the program. If you check " Display Splash Screen", then the application displays a splash screen with Data Parse logo when you start it.

If you check the boxes for Loading Community, Reference and Quick Links tabs, these are displayed once you open the application.



You can set appearances for areas of the script, the text options and the control toolbars and scrollbars of the editor.

If you check indicator area, then the application highlights this in the script file as to where changes have been made either as a green or golden brown strip on the left.

Checking the word wrap area, allows you to see the editor with words wrapped.

Checking the selection area, allows you to see the selections made on the editor highlighted by a maroon line on the left.

If you check user area, the editor allows you to set up the user configuration area

If you check line numbers, the editor shows the line numbers in the script code.

Checking changed lines marking, allows you to see lines which have been changed in a script

In the control tab, if you check horizontal, vertical scrollbars, status bar and XP style then the application allows these elements to appear in your editor.

In the text tab, if you check the lines wrapping marks, does not show the wrapped line marks, which appear as dots on the editor.

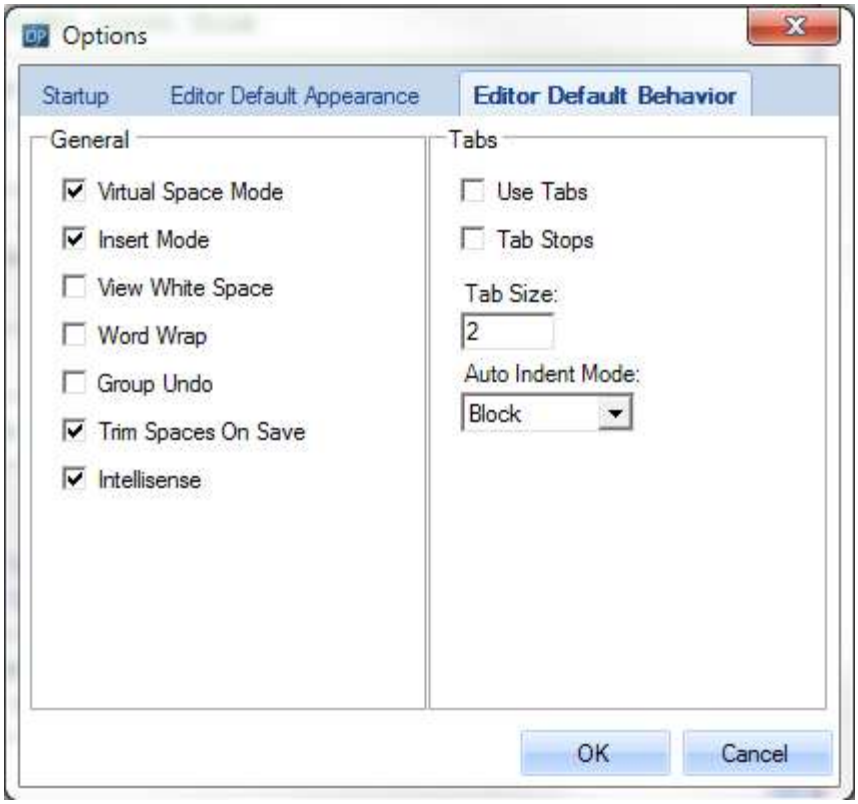
If you check the indentation block border and guidelines, then the paragraph based indentations do not appear on your editor

If the column guidelines checkbox is checked then the column wise indentation is not shown on the editor

If you check the outlining collapsers checkbox then the outline collapse and expand of the script code is disabled.

If you check the transparent selection box is checked then code indentation is not shown

You can also set the default behavior of the editor while running.



In the above options, if you check the virtual space box, then the extra space below the editor and console window is displayed

If you check insert mode, then insert mode in the editor is enabled

If you check view white space then the editor displays the white spaces in the script code

If you check the word wrap then word wrapped lines are shown on the editor

If you check the group undo option, then you can undo grouping of code blocks

If you check the trim spaces on save option then you can see that extra spaces are removed from the script code.

If you check the tabs options – tab stops and use tabs, then you can view tabbing of the script code in the editor along with the tab size set from the drop down

If you select None from the auto indent mode, then no indentation is enabled; if block is selected then the editor shows block based indentation else if the smart indenting is selected, then a space-saving editing is shown in the editor.

Deployables (Enterprise Edition only)

This feature is available only in the Enterprise Edition of the Data Parse software.

A deployable is a *stand-alone* executable file. It gets created with the Build feature. A deployable exe is the Data Parse program (with a different name).

When you run a deployable, it does the following steps.

1. Fetch the current Project's settings (such as combo and check boxes) from the project file. Even if the PPS is not going to be displayed, the name of the script file, input file, and so on, is required. Each of these is the first file listed in a list-of-files (such as the list-of-files named Input Files in Solution Explorer).

2. Show PPS (if Display PPS is true for the Project).

3. When the PPS is showing, you can make some changes and then click Start.

4. If PPS is showing, the application updates the project file so that its list-of-files (example: Input Files) and check-boxes (such as Display After Processing) match the combo boxes in the PPS, *including* each combo box's input box.

5. The application then processes the current Project.

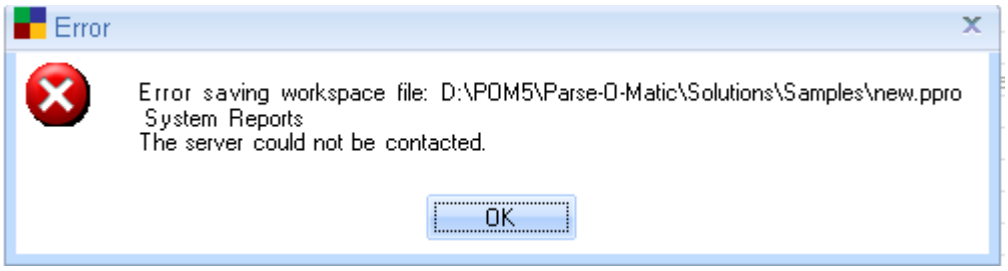
6. An Error Report window is shown if applicable. Note that at this point the PPS is *not* showing, though the progress bar *is* showing — though it is probably partially hidden by the Error Report window.

7. If Display After Processing is True, the output file is opened in a Viewer Window.

Exception Handling

Whenever an error occurs during script execution, an Exception window is displayed. When possible, the exception window will display the offending command or line of the script file in question. The IDE will also attempt to open up the offending script file and place the cursor at the location of the problem.

At other occasions, the IDE attempts to show system level errors while running a solution such as shown below:



Wildcards

Data Parse lets you process multiple input files in a single operation (i.e. clicking the Start button only once) by using "wildcards" in the Input File input box.

For example, if you set the Input File box to `*.txt` then all files with a `.txt` extension will be processed.

Here are some more examples:

Wildcard Mask	Interpretation
report??.txt	"report" followed by any two characters, .txt extension
my*.csv	"my" followed by one or more characters, .csv extension
xyz.???	"xyz" with any three-character extension

You cannot specify wildcards for the output file. All output goes to a single output file.

Stacking Wildcards

You can specify multiple wildcards by using semicolons, as in this example:

```
*.txt;*.me
```

This would process input files with the `.txt` extension (example: `xyz.txt`) and the `.me` extension (example: `read.me`).

There is almost no limit to the number of wildcards you specify, but bear in mind that when you stack wildcards you could end up processing the same file more than once. Consider this example:

```
*.txt;my*.txt
```

This would process all files with a `.txt` extension, then all files with a `.txt` extension where the file name starts with "my". Thus, a file named `myfile.txt` would be processed *twice*.

You cannot specify multiple file names for the output file. All output goes to a single output file.

Using the Windows Clipboard

Data Parse lets you process the Windows text clipboard as if it was a regular text file.

To process the clipboard as the input file, enter `Clipboard` in the Input File box.

Tip: Most Windows programs let you copy selected text into the clipboard with Ctrl-C.

You can also send output to the Windows text clipboard as if it was a regular text file. To send output to the clipboard, enter `Clipboard` in the Output File box.

Tip: Most Windows programs let you paste text from the clipboard with Ctrl-V.

It is possible to do both at once, processing input data from the clipboard and sending the resulting output to the clipboard. Of course, after processing, the original contents of the clipboard will have been overwritten.

Using a URL as input

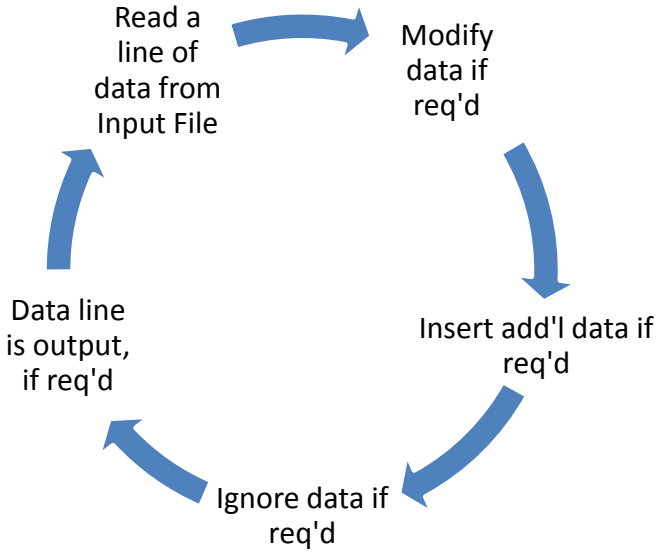
You can use standard URLs in the Input File box or within your scripts. HTTP, HTTPS and FTP, amongst others, are supported. Please note that you must make sure you have enough disk space to hold the downloaded file.

Download files are received in their entirety, before script processing proceeds.

Chapter 3

Scripting

What is a Script?



A script is a set of instructions that analyze data generated by Data Parse. Every time Data Parse has a new line of data, it sends it to the script for further processing. The script can make changes to the data before sending it to the output file, or skip the data altogether.

Here is an example of a script:

```
Change $OutData 'Cat' 'Dog'
OutEnd $OutData
```

The first line of this script changes the variable $OutData such that every instance of the word "Cat" is replaced by the word "Dog". The second line then sends the altered variable to the output file.

Here is another sample script:

```
Change $OutData 'Cat' 'Dog'
If $OutData ^ 'Dog' OutEnd $OutData
```

This is similar to the first example, but it sends data to the output file only if it contains the word "Dog".

Preparing Your Script

With only two exceptions (the If and Otherwise commands), scripts never contain more than a single scripting command on each line.

Blank lines are ignored. Lines that start with a semicolon (the `;` character) are also ignored — these are considered comments. You can also put a comment at the end of a line, following a semicolon. For example:

```
;-------------------------------------------------------------------
; This is my script file
;-------------------------------------------------------------------
If $OutData ^ 'Cow' Done   ; Ignore all lines containing the word 'Cow'
OutEnd $OutData            ; Output the line
```

It is traditional to line up end-of-line comments, as shown above. It is not mandatory, and sometimes it is not possible, but it does make the script easier to read. The horizontal lines in the example are used only as separators — these too can make a script easier to read, if used sparingly.

File Naming Conventions

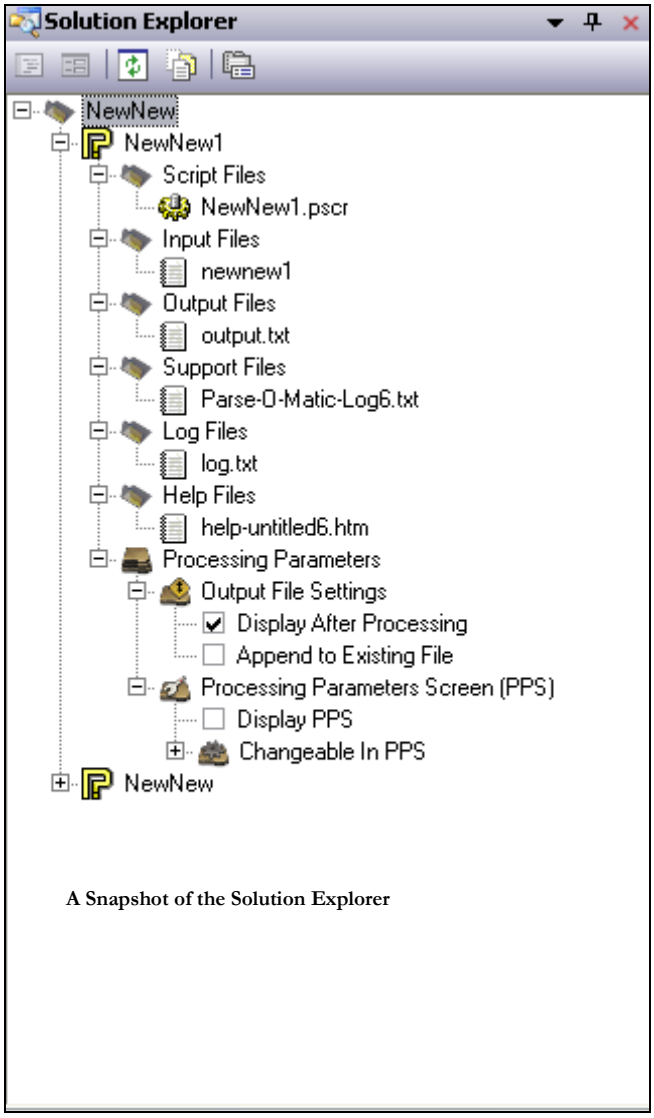
Scripts have a file extension of .pscr Scripts, .ppro Projects, and Solutions, psol

Hierarchy

Script hierarchy is a new concept introduced in version 5. In Parse-O-Matic version 4, there was just one type of file structure, and that was the single script file. If you wanted to execute a number of different scripts, in a particular order, then a batch file had to be used to call Parse-O-Matic those number of times that were needed to run a script.

In version Data Parse Version 5, the concept of the Solution, Project and Scripts was introduced.

A Project can contain one or more script files. A Solution can contain one or more project files.



A Snapshot of the Solution Explorer

Another feature of the script hierarchy is the Parameter Processing Screen (PPS). The PPS screen allows a Solution author to be prompted to enter in various values, before a script is executed.

This can be useful if the script being run needs to be run against input files whose name might not be known at design time. Another situation where the PPS can be useful is where the Solution author is not the person who is running the Solution. If the end-user is not the author of the script, then this feature comes in handy. This is also where the IDE's opening screen, the Quick Links option can be useful to run a selected solution.

Scripting Fundamentals

Values, Literals and Variables

A value is a parameter for a scripting command. It can be specified in the following ways:

Example	Explanation
'Text'	A text string (note the quotes)
15	A number
'15'	Another way to represent a number (i.e. as text)
VarName	The name of a variable
VarName[10 20]	Substring of a variable (columns 10 to 20 in this case)
VarName[19]	Substring of a variable (a single character)
VarName+	A numeric variable, plus 1 (e.g. MyVar = MyVar+)
VarName-	A numeric variable, minus 1 (e.g. MyVar = MyVar-)
VarName(10)	An array variable

A "literal" is a parameter in a script command that does not get changed when the script is running. The first three examples in the table above are literals. Literals are enclosed in 'quotes' — unless they are numbers, in which case the quotes are optional.

A "variable" is a named spot in your computer's memory that holds some data. Variables must start with an unaccented letter (A to Z). Case is ignored, so variables named MyVar, myvar and MYVAR are considered the same.

Substring ranges in square brackets such as MyVar[1 10] must refer to *fixed* range of column positions. If the script needs to vary the substring range, you should use the Cols command.

Array Variables

Array variables are recognized as such because the variable name is immediately followed by the "open parenthesis" character.

Array indices are all treated as strings. Variable indices are only supported in one-dimensional arrays. For example, the following are valid:

Variable indices:

```
Index=1
Begin Index #< 10
  MyArray(Index) = Index * 10
  Inc Index
End
```

Literal indices:

```
MyArray(1,1)=1
MyArray(1,2)=10
MyArray(1,3)=100
MyArray(2,1)=2
MyArray(2,2)=20
MyArray(2,3)=200
```

Uninitialized array elements are assigned the value contained in the special variable $NotDefined. By default this contains the value '[ND]', but you can assign a different value to $NotDefined if you wish.

Special Variables

Data Parse makes available certain internal variables. You can recognize these as "special" variables because — unlike user-defined variables — these start with a dollar-sign ($) character.

Because these variables are used by Data Parse itself, you should avoid altering them. Your script can either make a copy of a special variable (e.g. MyData = $OutData), or use commands such as Cols to extract the part you want (e.g. MyData = Cols $OutData 10 20).

Frequently-Used

Here are the special variables that are used most often.

Special Variable	Explanation
$OutData	Data that the application is sending to the script
$Data	The line of input data (see explanation below)
$PrevData	The previous line of input data read by the application
$ReadLines	The number of lines (or records) read from the input file

The $OutData and $Data variables refer to the same thing. In older Data Parse applications, such as TextHarvest, the input data (i.e. $Data) is preprocessed by the application itself before being passed to the script as $OutData. (The variable name $OutData literally means "preprocessed data sent as output to the script"). In such cases, your script should use $OutData rather than $Data, as it may not contain the *actual* input data from the file.

The $OutData variable can usually be altered without causing problems for the underlying application.

Input/Output

Here are the special variables related to input and output:

Special Variable	Explanation
$ActualIFN	Name of the current input file (including path)
$ActualOFN	Name of the current output file (including path)
$AppendingOutput	Set to 'Y' if output is being added to pre-existing file
$BytesOutCount	Number of bytes sent, so far, to the output file
$ClipboardOutput	Set to 'Y' if output will go to the Windows clipboard
$InputFileBytes	Number of bytes (including buffered) read from input
$OutCSVRec	The accumulator string used by the OutCSV command
$Wildcarding	Y = Multiple input files; N = Processing only one file
$CfgODBCConnection	Set ODBC database connection string

User Interface

Here are the special variables related to the user interface:

Special Variable	Explanation
$CaptionX	Caption for the first option box (usually 'Option &X')
$CaptionY	Caption for the second option box (usually 'Option &Y')
$CaptionZ	Caption for the third option box (usually 'Option &Z')
$IFNMask	What actually appears in the Input File box
$OptionX	First options box These variables contain the values
$OptionY	Second options box in the input boxes near the bottom
$OptionZ	Third options box of the Parsing Parameters window
$CfgShowPPSNote	Displays custom text on the PPS window

Miscellaneous

Here are various special variables that do not fit into the previously mentioned categories:

Special Variable	Explanation
$AppParms(n)	Array of parameters (see application's documentation)
$Compare	Dynamic comparator (e.g. If X $Compare Y Done)
$EndOfData	See "Manual Read Commands"
$Ignore	See explanation below
$NotDefined	Contains the value for uninitialized array variables
$Scrambled	'Y' = script has been scrambled (user cannot view source)
$StepName	Processing step (see application's documentation)
$Success	See explanation below
$TestMode	Set to 'Y' if the application is running in Test Mode

The $Ignore Variable

The $Ignore special variable is used when a function returns a value but you are not interested in what that value is. For example:

```
$Ignore = Parse MyData '2*/' '3*/' 'Cut'
```

This removes everything between the second and third slashes in the variable named MyData. Using $Ignore helps make a script self-documenting. That is to say, if you place a result in $Ignore, it serves as a reminder that you are not using the information elsewhere in the script.

You may sometimes get an error message that looks something like this:

```
Warning: The following variables are referenced only once in ScrMyScript
MyVariable
```

While this error message is usually caused by a mistyped variable name, it can also happen if you use a "throw-away" variable to get rid of a value — and only use it that one time. To avoid getting this message, use the $Ignore variable.

The $Success Variable

Certain commands (such as Overlay and SetFromFile) set a special variable named $Success. This is set to 'Y' (meaning, "Yes, it succeeded") if the command succeeded and 'N' (for "No") if it failed.

Consider this script sample:

```
MyVar = SetFromFile 'MyText.txt'
If $Success = 'N' MyVar = 'No data'
```

If the SetFromFile command fails — which would happen if the file was not found — then $Success is set to 'N'. If it succeeds, though, $Success is set to 'Y'.

When a script first runs, $Success is initially set to 'N'. Once a command sets the value of $Success, it retains its value until set by another command. Because of this, you should test $Success immediately after the command that sets it. Consider this situation:

```
MyVar = SetFromFile 'MyText.txt'
Overlay MyVar 'CUSTOMER' 'Customer'
If $Success = 'N' then MyVar = 'No data'
```

The programmer has apparently forgotten that Overlay also sets $Success. A better approach would be as follows:

```
MyVar = SetFromFile 'MyText.txt'
If $Success = 'N' MyVar = 'No data'
If $Success = 'Y' Overlay MyVar 'CUSTOMER' 'Customer'
```

This example performs the tests and operations in a more logical order.

Special Syntax

Continuation of Long Lines

If a script line is too long for convenient viewing in your text editor, you can continue it on the next line by using the >> symbol. For example:

```
CustomerInfo = CustSalutation FirstCustName MiddleCustName >>
  LastCustName '(' CustomerPhoneNumber ')'
```

You can put comments (i.e. a semicolon followed by some text) after the continuation symbol, though if you put the continuation symbol *after* the start of a comment, the following line of script is considered to be part of the comment.

In the example above, the continuation line was indented by two spaces. This is not mandatory, but it does serve as a reminder that the line is a continuation.

Embedding Quotes in Text Literals

Since text literals begin and end with 'quotes', you cannot simply put a quote inside a text literal. To represent a quote within a text literal, put two quotes in a row. For example:

```
MyVar = 'Isn''t ''scripting'' fun?'
```

This will set MyVar to:

```
Isn't 'scripting' fun?
```

Note that each instance of a doubled-up quote has been replaced by a single instance.

Untypeable Characters

You can specify either hexadecimal or decimal representation of bytes when coding a literal:

```
MyVar = $0A$0D
MyVar = #10#13
```

The first example uses hexadecimal notation to define the Carriage Return and Linefeed characters. The second example uses decimal notation to do the same thing.

You can also mix text and untypeable characters, as in these examples:

```
MyVar = 'Hello'$0A$0D
MyVar = 'Hello' $0A $0D
MyVar = 'Hello'#010#013
```

Any of the examples above will set the variable MyVar to 'Hello' followed by the Carriage Return and Linefeed characters.

Free and Advanced Scripting

Data Parse Free Edition lets you use the majority of the scripting language features at no extra charge. Some of the more powerful language capabilities, however, require the purchase of a license. These editions are Data Parse Basic, Data Parse Business and the Data Parse Enterprise.

If you use an Advanced Scripting command or other higher-edition feature and do not have the License, the program will display a pop-up window. You can skip over this window, so you can make sure that the Advancing Scripting command is appropriate for your requirements. You may try out the Advanced Scripting commands at no charge for up to 30 days.

You can visit:

http://www.DataParse.com/

to learn more about obtaining the Data Parse Basic Edition, as well as the Business and Enterprise Editions.

Sample Scripts

Data Parse is delivered with sample Solutions (which typically have the word `Sample` in their names).

Here is a list of the sample solutions included with **Data Parse.**

Script File Name	Input File to Use	Adv	What is Demonstrated
SampleSolution01.psol	ThingsToDo.txt	-	Basic techniques
SampleSolution02.psol	ThingsToDo.txt	-	Basic techniques
SampleSolution03.psol	InputSample01.txt	-	Basic techniques
SampleSolution04.psol	ToDoListFixed.dat	-	Fixed-record-length input
SampleSolution05.psol	ToDoListDelim.dat	-	Character-delimited input
SampleAdvSolution01.psol	ThingsToDo.txt	Y	Advanced techniques
SampleAdvSolution02.psol	Scr*.txt	Y	Advanced techniques
PSTMain.psol	ThingsToDo.txt	Y	Main scripting commands
PSTOutCSV.psol	ThingsToDo.txt	Y	OUTCSV command
PSTMR.psol	InputSample02.dat	Y	RecLenZero script

Adv = Uses Advanced Scripting commands

It is best to study these scripts in the order they are listed above.

All of the sample scripts have default input and output file names defined.

In addition to the sample scripts included with Data Parse, you can find additional sample scripts in the National Data Parsing Corporation Knowledge Base, available at www.Data Parse.com.

About Older Data Parse Applications

Parse-O-Matic, now Data Parse, was originally created in 1985. We have learned a lot about parsing since that time, and the design of Data Parse Scripting reflects this.

As our long-time customers have probably noticed by now, Data Parse Scripts are similar to the POM files used by our old DOS-based program, but the POM files are not compatible. For example, the old $FLINE variable is now represented by $Data. This does not mean that the old DOS-based program is no longer useful. Certain kinds of operations (such as those performed on binary files) are currently impractical with Scripting, and some arcane capabilities (such as bit-wise operations and date arithmetic) are not implemented.

With the release of Version 5, the concept of Solutions and Projects have been introduced. Also, filename extensions have changed, to better reflect Windows standards.

Script files created with version 4 can still be used in version 5. Simply copy and paste in your script into a blank Script file contained within a Solution and Project. It is best not to simply add your existing version 4.x script file to a project, as file character encoding has changed.

Those running Data Parse with batch files should note all command-line parameters have changed. Also, if your batch files gathered data from your relational database, you may wish to start using the built-in ODBC connectivity. Similarly, those processing HTML or FTP'd files, may wish to switch to the internally available commands that support those transports and/or data formats.

Chapter 4

Data Assignment Commands

Equals (Set Variable)

```
Format # 1      v1 = v2 [v3 v4 v5]
EXAMPLES        MyVar1 = 'Hello'  ; Set var to a literal
                MyVar2 = MyVar3   ; Set one var to another
                MyVar4 = OtherVar[10 20]; Columns 10 to 20
                MyVar5 = 'How ' 'are ' 'you?'  ; Append three literals
PURPOSE         Sets v1 to v2 (and any other values listed thereafter)

PARAMETERS      v1 - Variable being set
                v2 - Value
                v3 - Value (any number of values can be appended)
```

-OR-

```
Format # 2      v1 = f2
Example         MyVar6 = Cols xyz 5 8   ; Set var from a function
Purpose         Sets v1 from a function
Parameters      v1 - Variable being set
                f2 - Function (with any parameters it may use)
```

A "function" is a command that returns a value. The Cols command is an example of a function, while the OutEnd command is *not* a function because it does not return a value.

Len

```
Format          v1 = Len v2 [v3 v4 v5...]
Examples        MyVar1 = Len MyVar2  ; If MyVar2 is 'ABC', MyVar1 will
                be '3'
                MyVar3 = Len X1 X2   ; Measure total length of appended
                values
Purpose         Sets v1 to the length (number of characters) in v2
Parameters      v1 - Variable being set
                v2 - Value being measured
                v3 - Value (any number of values can be appended)
```

ParseName

```
ParseName v1 v2 v3 v4 v5 v6 v7

Example         ParseName 'John Smith' 'No' addform first middle last
                suffix
Purpose         Breaks up a name into its component parts
Parameters      v1 - The unparsed name
                v2 - Control setting: detect company names?
                v3 - Variable to receive address form (e.g. 'Mister')
                v4 - Variable to receive first name (e.g. 'John')
                v5 - Variable to receive middle name (e.g. 'J.')
                v6 - Variable to receive last name (e.g. 'Smith')
                v7 - Variable to receive suffix (e.g. 'the third')
Controls        v2 = Yes/No
```

ParseName provides some basic capability for breaking up a proper name. The results cannot be completely accurate because there are so many possible variations. Thus, if you use ParseName (typically to create a CSV record), you should review the results afterwards and modify your script to handle exceptions.

In addition, you should not assume that ParseName will return the same results when using different versions of Data Parse. The ParseName command is occasionally updated to improve its "intelligence". ParseName is a handy time-saver, but there are no definitive rules for this kind of operation.

If the control setting (v2) is set to 'Yes', ParseName can detect many company names, placing the entire value in v4. This, too, is not entirely reliable. For example, 'John Jones Enterprises' will be recognized as a company, but 'Les Entreprises John Jones' (i.e. the company name in French) is not.

Despite its limitations, ParseName is a helpful command: it can greatly reduce the effort required if you are converting a large list of names.

Plural

```
Format          v1 = Plural v2 v3 [v4]
Example         Word = Plural 'cat' NumBeasts  ; If NumBeasts = 3
                returns 'cats'
Purpose         Provisionally adds the letter 's' to a word if it is
                appropriate
Parameters      v1 - Variable being set
                v2 - The word being counted, which might have an 's'
                added
                v3 - The number of v2 items being considered
                v4 - Control setting (Preserve length? Yes/No)
Controls        If v4 = 'Yes', we append a space to v1 if the 's' is
                omitted.
                This maintains the alignment of columnar output.
Defaults        v4 = 'No'
```

This simple command makes it easy to avoid unattractive "tentative plurals" such as "item(s)". For example:

```
Items = Plural 'item' ItemCount
OutEnd 'We have ' ItemCount ' Items ' in stock'
```

If ItemCount is 1, the output reads 'We have 1 item in stock'. For any other number, an 's' is added. For example: 'We have 3 items in stock'. If ItemCount is a real number — even 1.0 — an 's' is added, since that is the way it would normally be spoken in context (e.g. "The score is one point zero points")

SetFromFile

```
Format          v1 = SetFromFile v2 [v3]
Examples        MyVar1 = SetFromFile 'MyFile.txt'
                MyVar2 = SetFromFile 'C:\Stock\Greeting.txt'
Purpose         Reads data from a file into a variable
Parameters      v1 - Variable being set
                v2 - File name
                v3 - Control settings
Controls        'Text'   = The file is a text file (may end with Ctrl-
                Z)
                'Binary' = The file is a binary file
Defaults        v3 = 'Text'
Similar Cmds    LookUp
```

If v3 is not specified, the file is considered to be text, and any end-of-line (CRLF) characters are stripped from the start and end of v1.

SetFromFile sets the $Success variable to 'Y' if the file was successfully read, 'N' otherwise.

If the filename (v2) does not specify a path, SetFromFile will use the Search Path to look for it.

In theory, SetFromFile can read in a file that is several billion characters long. In practice, however, the size of the file you can read in is limited by your computer's memory.

SplitCSV

```
Format          v1 = SplitCSV v2 [v3 [v4]]
Example         ParsedCSV = SplitCSV FileData
Purpose         Converts data in CSV (Comma Separated Value) format
                into a format that is much easier to take apart with
                the Parse command (using 'Cut Relaxed', for example)
Parameters      v1 - Variable being set
                v2 - The CSV data
                v3 - The string with which to replace the old delimiter
                v4 - The old delimiter (usually a comma or a semicolon)
Defaults        v3 = Carriage-return character (ASCII #13)
                v4 = The comma character
Similar Cmds    Parse
```

SplitCSV parses a line of comma-delimited text, replacing the commas with the new delimiter (v3). Any double-quotes (") around fields are removed, while doubled-up quotes ("") are replaced with single " quotes. For example:

```
MyVar     = '"Mary ""The Parser"" Jones";123.45;"416-555-1212"'
ParsedCSV = SplitCSV A ' / ' ';'
```

This would set the ParsedCSV variable to the following value:

```
Mary "The Parser" Jones / 123.45 / 416-555-1212
```

When processing CSV data, bear in mind that in some countries the standard delimiter is the semicolon (`;`) because they use a comma as the decimal point.

Chapter 5

Data Alteration Commands

Change

```
Format          Change v1 v2 v3 [v4]
Examples        Change MyVar 'Cat' 'Dog'  ; Change 'Cat' to 'Dog'
                Change MyVar 'Dog' ''     ; Remove all 'Dog' strings
Purpose         Changes v1 such that every occurrence of v2 is changed
                to v3
Parameters      v1 - Variable to be changed
                v2 - Value to look for
                v3 - Value to replace it with
                v4 - Control setting
Controls        MultiPass/OnePass
Defaults        v4 = 'MultiPass'
Similar Cmds    ChangeCase, KeepChar, MassChange, Padded, TrimChar
Notes           The comparison is case-sensitive. 'Cat' does not match
                'CAT'.
```

In the default MultiPass mode, the Change command repeats the process until the value being sought (v2) is no longer found. However, consider this situation:

```
X = 'ABCD'
Change X 'A' 'AA' 'MultiPass'
```

The Change command notices that repeating the process would never end (because v3 contains v2), so it only scans v1 once.

ChangeCase

```
Format          v1 = ChangeCase v2 [v3]
Example         ChangeCase MyVar 'HardCaps'
Purpose         Changes text case (e.g. 'Cat' to 'CAT')
Parameters      v1 - Variable being set
                v2 - Original value
                v3 - Control setting
Controls
```

Original (v2)	Control (v3)	Result (v1)
'Fred Jones'	'Uppercase'	'FRED JONES'
'FRED Jones'	'Lowercase'	'fred jones'
'fred jones'	'Capitalize'	'Fred Jones'
'FRED jones'	'Capitalize'	'FRED Jones'
'FRED jones'	'HardCaps'	'Fred Jones'
'WX-XY123'	'HardCaps'	'Wx-Xy123'
'FRED jones'	'NoChange'	'FRED jones'

```
Defaults        v3 = 'Uppercase'
Similar Cmds    Change
```

KeepChar

```
Format          KeepChar v1 v2
Examples        KeepChar MyVar1 '/AZ'      ; Retain A to Z only
                KeepChar MyVar2 '/$/09/.'  ; Retain $, 0 to 9, and
                period
                KeepChar MyVar3 '/AZ/az/'  ; Retain only letters
                KeepChar MyVar4 '*AZ*az'   ; Same as previous example
Purpose         Filters out everything but the characters and
                character-ranges specified.
Parameters      v1 - Variable being modified
                v2 - Control setting
Similar Cmds    Change, TrimChar
```

The first character of the control setting (v2) is the delimiter that will separate the characters or pairs of characters. Paired characters represent a range, while single characters represent precisely that character.

Padded

```
Format          v1 = Padded v2 v3 [v4 [v5]]
Examples        MyVar1 = Padded 'AB'  4             ; 'AB  '
                MyVar2 = Padded 'CD'  5 'Left'      ; '   CD'
                MyVar3 = Padded 'EF'  6 'Center'    ; '  EF  '
                MyVar5 = Padded 'XYZ' 7 'Left' 'x'  ; 'xxxxXYZ'
Purpose         Pads a value to a specific length (number of
                characters)
Parameters      v1 - Variable being set
                v2 - Original value
                v3 - Length of result (number of characters)
                v4 - Edge to pad:  'Left' 'Right' 'Center'
                v5 - Character with which to pad
Defaults        v4 = 'Right'
                v5 = ' ' (i.e. a space)
Similar Cmds    Change, Insert
```

TrimChar

```
Format          TrimChar v1 [v2]
Examples        TrimChar MyVar1
                TrimChar MyVar2 'B M,L R$'
Purpose         Removes unwanted characters from a variable
Parameters      v1 - Variable to be changed
                v2 - Trimming specifications
Defaults        v2 = 'B '
Similar Cmds    KeepChar
```

The "trimming specifications" comprises pairs of characters describing how you want the variable trimmed. Each pair of characters is treated as follows:

- The first character is the instruction (e.g. B = Both edges)
- The second character is the actual character you want trimmed away

Here is an explanation of the various trimming instructions:

Instruction	Meaning
A	Trim all instances of the character
B	Trim both sides of the variable (left and right)
L	Trim the left side of the variable
R	Trim the right side of the variable
M	Replace multiple instances of the character with just one

Consider the following variable:

```
MyVar = ' xxx///yyy zzz/// '   ; Note the spaces on both ends
```

Here is how various trimming specifications would affect the xyz variable:

Trim Spec	Result	Trim Spec	Result
'L '	'xxx///yyy zzz/// '	'R '	' xxx///yyy zzz///'
'B '	'xxx///yyy zzz///'	'A '	'xxx///yyyzzz///'
'B Ay'	'xxx/// zzz///'	'A Az'	'xxx///yyy///'
'M/'	' xxx/yyy zzz/ '	'B M/'	'xxx/yyy zzz/'
'MxMyMzM/'	' x/y z/ '	'B Lx'	'xxx///yyy zzz///'

As you can see from the 'B Lx' example, the trimming instructions are executed simultaneously. If you want to trim both spaces and then trim off the leading x's, you need to do two TrimChar commands in a row.

Chapter 6

Output Commands

Odb

```
Format          Odb v1 [v2 v3 v4...]
Purpose         Same as OutEnd, but separates the fields with vertical
                bars
Parameters      Same as OutEnd
Similar Cmds    OutRuler
```

You can use the Odb ("Output Debug") command while developing or fixing a script. The vertical bars let you see if the variables have spaces on either side. Once your script is working properly, you can do a quick search for "Odb" to see if you left behind any debug lines.

OutCSV

```
Format          OutCSV v1 [v2 [v3 v4 v5...]]
Examples        OutCSV '' 'Init'
                OutCSV CustName
                OutCSV ItemPrice 'Unquoted'
                OutCSV '' 'Done'
Purpose         Generates CSV (Comma Separated Value) output; can also
                be used to generate columnar reports with columns that
                can be turned on and off
Parameters      v1 - Value to send to output (or control information)
                v2 - Control setting
                v3 - If present, v3 and subsequent values are
                concatenated to v1
Controls        The format of v2 is:
                [+/-][Init/Done/Stop/Quoted[...]/Unquoted[...]/Control]
                'Init' starts the accumulation of a new line of CSV
                output.
                'Done' sends the accumulated output to the output file.
                'Stop' terminates accumulation without sending output.
                'Quoted' puts quotes around the field.
                'Unquoted' adds the field without quotes.
                '+' and '-' turn fields on and off.
                '...' changes the default quoting state
                'Control' adjusts OutCSV settings.
Defaults        v2 = 'Quoted' (unless default quoting state has been
                changed)
Similar Cmds    OutEnd, Odb
Notes           Nothing is actually sent to the output file until the
                'Done' step (i.e. v2 = 'Done').
```

The various controls are explained in more detail below.

OutCSV Init

When v2 is 'Init', v1 can be used to specify an alternative separator (other than the usual comma). Typical alternatives include the semicolon (;) and the Tab (ASCII decimal 9). To save you having to look up ASCII values, OutCSV recognizes certain codes for the separator. Here is an overview of the v1 settings...

v1	Explanation
''	Use default field separator — this is usually a comma
','	You can also specify a comma explicitly
'TAB'	The tab character
'CR'	The carriage-return character
'CRLF'	The carriage-return and linefeed characters
'LFCR'	Linefeed then carriage-return (non-standard — rarely used)
'NONE'	No separator (remember: if you use '' it means "default")

You can, in fact, set the separator to any string. Used with padded text (or OutCSV's Control setting with the MaxWidth and MinWidth options), you can use OutCSV to generate columnar reports. Your script can then turn entire columns on and off using the '+' and '-' feature.

Outputting a Field

When v2 is 'Quoted' or 'Unquoted' or null, OutCSV accumulates the field for the current output line. The line is not actually sent to output until the 'Done' step is reached. Here is a brief example:

```
OutCSV '' 'Init'
OutCSV 'Fred Jones'
OutCSV 1234.56 'Unquoted'
OutCSV '' 'Done'
```

This will output a two-field CSV line, with quotes around the first field but not the second one. If the field is quoted, any occurrence of the quote character (") is replaced by double-quotes, as per standard CSV conventions.

OutCSV Nulls

If you have several null fields to insert, you can use the Nulls option:

```
OutCSV 5 'Nulls'
```

This would accumulate 5 null fields for the current output line.

Nothing is done if the parameter is 0 (zero) or a null ("). If the value is more than 1000, OutCSV stops with an error message.

OutCSV Done and Stop

When v2 is 'Done', OutCSV sends the accumulated line to the output file. The v1 value is not used.

An infrequently used alternative to 'Done' is 'Stop'. In this case, the output is *not* sent to the output file but is saved in the special variable $OutCSVRec. You can use this method if you do not wish to send the output immediately. In such case, you should copy the result from $OutCSVRec to another variable before doing another set of OutCSV commands.

OutCSV Control

When v2 is 'Control', OutCSV consults v1 for a command that configures how it will operate. Control settings remain in effect within the script until changed.

The following options are available:

Command	Example	Explanation
MinWidth	OutCSV 'MinWidth 25'	Pad fields (with spaces) to specified width
MaxWidth	OutCSV 'MaxWidth 25'	Truncate fields that exceed specified width
SetWidth	OutCSV 'SetWidth 15'	Set MinWidth and MaxWidth to the same value
QuoteChar	OutCSV 'QuoteChar @'	Specify new character for quoting fields
Separator	OutCSV 'Separator ;'	Change default separator (originally comma)

To set the quoting character to a space, use 'QuoteChar Space'. When the QuoteChar is a space, it is *not* doubled-up when it is found in a field, since the only reason one would set the QuoteChar to a space is to create columnar reports.

You can also use 'QuoteChar None' to mean "don't put any quoting characters around purportedly quoted fields". This feature is useful if you are using OutCSV to produce columnar reports.

The MaxWidth and MinWidth settings take into account the presence or absence of quotes when calculating width. Also, unquoted fields are assumed to be numeric and (if necessary) are padded on the left, while quoted fields are padded on the right.

Turning Fields On and Off

Whenever the first character of v2 is '-' (the minus character), all subsequent fields are "turned off". To turn them back on, set the first character of v2 to '+' (the plus character). Here is an example:

```
OutCSV '' 'Init'
OutCSV 'Fred Jones'           ; Customer name field
OutCSV 1234.56 '-Unquoted'    ; Current balance
OutCSV '416-555-1212' '+'     ; Customer phone number
OutCSV '' 'Done'
```

In this example, the "Current balance" field will not appear in the output.

The ability to turn fields on and off can greatly simplify the testing of scripts that generate CSV output. You can also use this feature to create reports with columns that can be turned on and off.

Changing the Default Quoting State

The default state for OutCSV field accumulation is 'Quoted'. However, sometimes you have a lot of 'Unquoted' fields in a row and it is a chore to have to type 'Unquoted' repeatedly. You can redefine the default state by putting an ellipsis (three periods) after 'Quoted' or 'Unquoted'. Here is an example:

```
OutCSV '' 'Init'
```

```
 OutCSV 1
 OutCSV 2 'Unquoted...'
 OutCSV 3
 OutCSV 'A' 'Quoted...'
 OutCSV 'B'
 OutCSV '' 'Done'
```

This would output the following line:

```
"1",2,3,"A","B"
```

This alteration to the default only lasts until the 'Done' step; OutCSV always starts with the default state of 'Quoted'.

Switchable CSV/Columnar Reports

Here is an example of some code that can be easily switched between CSV output and columnar output, simply by changing one variable (called MyVar here):

```
CSVDelim = ''                       ; Normal setting (i.e. "use a comma")
Begin MyVar = 'Y'                   ; Did we turn on columnar mode?
  CSVDelim = ' '                    ; Separate fields with space, not comma
  OutCSV 'MinWidth 15' 'Control'    ; Pad fields out to 15 characters
  OutCSV 'MaxWidth 15' 'Control'    ; Truncate any fields wider than 15
  OutCSV 'QuoteChar None' 'Control' ; Ignore the quotes around quoted fields
End
OutCSV CSVDelim 'Init'              ; Start of OutCSV accumulation
OutCSV FirstName                    ; A quoted field
OutCSV LastName                     ; A quoted field
OutCSV Balance 'Unquoted'           ; Unquoted field (typical for numbers)
OutCSV '' 'Done'                    ; Send fields to output file
```

Simply by setting the variable MyVar to 'Y', a CSV (Comma Separated Value) file becomes a columnar report. The result may not be elegant, but if you are looking for fast results without having to load the output into a spreadsheet, this can be a real time-saver.

OutCSV Examples

Data Parse includes a sample script named `ScrPSTOutCSV.txt`. It provides examples of the techniques described above. You can also find CSV-oriented sample scripts in the National Data Parsing Corporation Knowledge Base, available at www.Data Parse.com.

OutEnd

```
Format          OutEnd v1 [v2 v3 v4...]
Examples         OutEnd 'Customer List'            ; One value to
                output
                OutEnd 'Customer Name: ' CustName  ; Two values to
                output
Purpose         Sends data to the output file, followed by a Carriage-
                Return and a Linefeed (the standard end-of-line
                characters for text files)
Parameters      v1 - Value to send to output file
                v2 - Value (any number of values can be appended)
Similar Cmds    OutNull, Output, OutRuler
```

OutFile

```
Format          OutFile v1 [v2]
Example         OutFile 'C:\MyFiles\Output.txt' 'Append'
Purpose         Changes the current output file
Parameters      v1 - Name of the output file
                v2 - Control setting
Controls        'New'    = Start with an empty file
                'Append' = Add to the end of the file (if it exists)
Defaults        v2 = 'New'
```

If the file name is not fully qualified (i.e. does not contain a path) the file will be placed in the default output folder, as set by the Path button.

If a file is opened as New and a file already exists with that name, the old file is renamed with a `.bak` extension. For this reason, you should not use OutFile to switch to a file with a `.bak` extension.

The fully-qualified name of the current output file is found in the $ActualOFN variable. If you copy this value into a variable, you can return to the original output file later on by using OutFile with 'Append'.

OutNull

```
Format          OutNull
Purpose         Sends a blank line to the output file (i.e. just a
                Carriage-Return and a Linefeed).
Similar Cmds    OutEnd, Output, OutRuler
```

Output

```
Format          Output v1 [v2 v3 v4...]
Purpose         Same as OutEnd, but does not send "end-of-line"
                characters
Parameters      Same as OutEnd
Similar Cmds    OutEnd, OutNull, OutRuler
```

OutRuler

```
Format          OutRuler v1 [v2 v3 v4...]
Purpose         Same as OutEnd, but includes a measuring scale
Parameters      Same as OutEnd
Similar Cmds    Odb
```

You can use OutRuler while developing a script to help you measure where columns start and end. It outputs the line as OutEnd does, but includes a measuring scale above it.

Chapter 7

Comparators

Overview

A "comparator" is a parameter used in scripting commands which compares one value to another. For example:

```
If AreaCode = '416' Output 'Toronto'
```

In this example, a comparison is being made between the variable named AreaCode and the literal '416'. The equals sign is the "comparator".

Now consider this command:

```
If AreaCode = '514' Region = 'Montreal'
```

In this case, the first equals sign is a comparator because it is comparing two values. The second equal sign is *not* a comparator; it is actually the Equals command, which assigns a value to a variable.

Types of Comparators

Data Parse Scripting supports several types of comparators:

Type	What It Does
Literal	Compares values character by character
Numerical	Compares the arithmetic values of real or integer numbers
Length	Compares the length of one value with a number
Pattern	Compares a value against a pattern

These are explained below in more detail.

Literal Comparators

Here is a list of the literal comparators:

Comparator	Meaning	Comments
=	Identical	
<>	Not identical	
>	Higher	See Note # 1
>=	Higher, or identical	See Note # 1
<	Lower	See Note # 1
<=	Lower, or identical	See Note # 1
^	Contains	
~	Does not contain	
Is	Basically the same	See Note # 2
Longer	Length is longer	
Shorter	Length is shorter	
SameLen	Length is the same	

Note # 1: Depends on sort order. For a discussion of what this means, refer to the section "Literal Comparisons and Sort Order".

Note # 2: The two values are considered basically the same if they contain the same text, regardless of upper or lower case, and any surrounding whitespace. Thus ' CHESHIRE CAT ' is the considered the same as 'Cheshire Cat'.

Examples

With some restrictions (discussed later), literal comparators work on both numeric and alphabetic data. Here are some examples of literal comparisons that are true:

```
'ABC' <>       'ABCD'        '333' <>       '444'
'ABC' <=       'ABCD'        '333' <=       '444'
'ABC' <        'ABCD'        '333' <        '444'
'ABC' Shorter 'ABCD'         '333' SameLen '444'
'ABC' >=       'ABC'         'ABC' <>       'CDE'
'ABC' <=       'ABC'         'ABC' <=       'CDE'
'ABC' =        'ABC'         'ABC' <        'CDE'
'ABC' SameLen 'ABC'          'ABC' SameLen 'CDE'
'ABC' ^        'AB'          'ABC' ~        'CD'
'ABC' ^        'ABC'         'ABC' ~        'CC'
```

Note especially the ^ (contains) and ~ (does not contain) comparators. These are extremely useful when analyzing data.

Literal Comparisons and Sort Order

Some of the literal comparators compare text according to 'PC-ASCII sort order'. For plain English text, this works fine. However, if your text contains diacritical (accented) characters, you should be aware that some comparisons will not work correctly. For example, the 'o-circumflex' character (ô) appears in the PC-ASCII character set *after* the PC-ASCII value for 'Z'.

Numerical Comparators

Here is a list of the numerical comparators:

Comparator	Meaning
#=	Equal
#<>	Not equal
#>	Greater
#>=	Greater, or equal
#<	Less than
#<=	Less than, or equal

Numerical comparators avoid the problem of sort order. For a discussion of this, see Numeric Comparisons and Sort Order.

Examples

Here are some examples of numeric comparisons (encoded variously with and without surrounding quotes) that are true:

```
345  #<> 567       '1.23'  #<>  '9.87'
345  #<= 567       '1.23'  #<=  '9.87'
567  #>  345       9.87    #>   '1.23'
'3'  #<  '6.2'
```

The last example compares an integer ('3') with a real number ('6.2'). The numeric comparators automatically check if one of the numbers contains a decimal point.

In such case, the comparison is performed in 'real number' mode, which imposes the same accuracy restrictions as those imposed by the CalcReal command. This might create a problem if you are comparing a decimal number with a large integer, but this is rarely a cause for worry, since most data analysis tends to compare similar types of numbers.

Numeric Comparisons and Sort Order

You can get unintended results when you use literal comparators on numbers. For example, this does not work as you might expect at first glance:

```
count = count+
If count >= 2 OutEnd count
```

You might expect this to output any number greater than or equal to '2', but in fact you will get a different result, because the comparison is a literal (text) comparison. In the example above, '2' to '9' are greater or equal to '2', but '10' (which starts with '1') is considered *less*, as is evident when you sort several numbers alphabetically:

```
1  10  11  15  100  2  20  200  3  30
```

As you can see, the values 1, 10, 11 and 15 come before '2' when sorted alphabetically.

To compare numbers, you should use the numerical comparators. The correct way to code the previous example is as follows:

```
count = count+
If count #>= 2 OutEnd count
```

Written in this way, numbers greater than or equal to 2 will be sent to the output file.

Length Comparators

Here is a list of the length comparators:

Comparator	Meaning
Len=	Equal
Len<>	Not equal
Len>	Greater
Len>=	Greater, or equal
Len<	Less than
Len<=	Less than, or equal

The length of the value on the left side of the comparator is compared with a *number* on the right side of the comparator. For example:

```
If $OutData Len= 0 NullLine = 'Yes'
```

Of course, you could accomplish the same thing with this command:

```
If $OutData = '' NullLine = 'Yes'
```

However, in most cases the length comparisons will save you some coding because you will not have to use the Len command to obtain a variable for comparison.

Comparing Patterns

The Matches comparator compares a value against a pattern that uses "regular expression" syntax (explained later). For example:

```
If MyVar Matches 'c[aou]t' GotMatch = 'Yes'
```

This will set the variable GotMatch to 'Yes' if MyVar contains 'cat', 'cot' or 'cut' (case is ignored).

The pattern uses "regular expression" syntax (described in the next section) and must be the second item in the comparison.

In order for the comparison to be "true", the item being compared to the pattern must match the pattern precisely — the Matches comparator does not look for substrings.

If you want to allow a substring to match, use the Comprises comparator. For example:

```
If MyVar Comprises 'c[ao]t' GotMatch = 'Yes'
```

This will set GotMatch to 'Yes' if MyVar includes either the word 'cat' or 'cot'. Thus, the strings 'He had a cat' and 'He had a cot' both Comprise the pattern, as do the strings 'cat', 'cot', 'Cat', 'scatter' and so on.

Regular Expressions

A "Regular Expression" is a sequence of characters where certain characters have a special meaning and are not matched literally. For example, a period will match any character (including the period), while the dollar-sign ($) matches the end of the line of text.

In the following list, the letters x, y and z stand in for any character.

```
^xxx     Match a sequence of characters at the start of a line
xxx$     Match a sequence of characters at the end of line
x.y      Match a single character (between 'x' and 'y' in this example)
[xz]     Match a set of characters ('x' and 'z' in this example)
[x-z]    Match a range of characters (this example covers 'x' to 'z')
x*       Match zero or more occurrences of the preceding character
[xyz]*   Match zero or more occurrences from the preceding set
[x-z]*   Match zero or more occurrences from the preceding range
[^xyz]   Match any character but the ones specified
[^x-z]   Match any character but the ones in the specified range
```

The backslash (\) character has a special meaning in regular expressions:

```
\x       Means "take the next character literally"
         For example:  \[  means the actual  [  character
         rather than the start of a set or range
\t       Means "a tab character" (ASCII character 9)
```

Basic Regular Expressions

Here are some examples of matches:

```
C.t      Match Cat, Cot, Cut, Cxt, C3t etc.
C[aou]t  Match Cat, Cot, Cut only
B..d     Match Bird, Bred, Bead etc.
^Dog     Match Dog only if it is at the beginning of a line
Moose$   Match Moose only if it is at the end of a line
Pa*d     Match Pd, Pad, Paad, Paaad etc.
```

Using the Asterisk

The last example given above uses the * character to indicate zero, one or more occurrences of a particular character — in this case, the letter 'a'. Incidentally, this is

different from the way the Windows operating system uses the * wildcard character. In Windows, the * wildcard matches "any single character".

In regular expressions, however, the asterisk is specific about what you are looking for. That is why 'Pa*d' would not match 'Parsed'; the asterisk means "match zero or more of the preceding character specification".

If you actually want to search for 'Pa' followed by one or more letters and then 'd', the correct syntax is:

```
Pa[a-z][a-z]*d
```

This means that we want to match 'Pa', then a letter in the range from 'a' to 'z', then some number (including zero) of characters in the 'a' to 'z' range, and finally the letter 'd'. The character string 'Parsed' would meet these criteria, as would 'Pad', 'Paid' and 'Packed'.

Advanced Regular Expressions

Here are some more complicated examples of regular expressions:

C[^ou]t	Matches Cat, Cxt and so on, but not Cot or Cut
C[ao]*t	Matches Ct, Cat, Caat, Cot, Coot, Cooot, Coat, Coaoat etc.
[0-9][0-9]*	Matches numbers such as 0, 1, 01, 10, 25, 0990, 9999 etc.
-[0-9][0-9]*	Matches negative numbers such as -0, -1, -19, -12345 etc.

In the last example, [0-9] is specified twice to ensure that at least one digit is found. Bear in mind that the * character means "zero or more occurrences". If you had only specified '-[0-9]*' you would get a spurious match within the string 'Hello - there' since the '-' character is indeed found, followed by *zero* occurrences of the digits 0 through 9.

You can create fairly complex patterns using regular expressions. Consider this example:

```
\$[0-9][0-9]*\.[0-9][0-9]
```

This would match dollar amounts with two decimal places, such as $0.00, $03.23, $3.14, $9.99, $1234.56 and so on.

Chapter 8

Comparison Commands

Overview

For a broader overview of comparisons in scripting, consult one of the following sections of this user manual:

- Comparators
- Flow Control Commands

The commands described below deal with special cases involving comparison.

AlphaNumPatt

```
Format          v1 = AlphaNumPatt v2 [v3]
Example         X = AlphaNumPatt '416-287-8892'  ; Set X to 'NNN-NNN-
                NNNN'
Purpose         Creates a pattern of characters representing the format
                of variable v2 in terms of alphabetic, numeric and
                special characters
Parameters      v1 - Variable being set
                v2 - Value being analyzed
                v3 - Control setting
Controls        v3 is a TrimChar specification
Defaults        v3 = '' (no trimming)
Similar Cmds    Numeric
                See also the Matches or Comprises comparators
```

AlphaNumPatt returns an 'A' for every letter (uppercase or lowercase) in v2, and an 'N' for every digit. All other characters (spaces, dashes etc.) are left as-is. Here are some sample results:

Value of v2	Value of v3	Result (v1)	Value of v2	Value of v3	Result (v1)
'12-34-56'	(Not set)	'NN-NN-NN'	' $12.34 '	(Not set)	' $NN.NN '
'AB 1234'	(Not set)	'AA NNNN'	' XY 999 '	'B '	'AA NNN'

AlphaNumPatt is handy for detecting the presence or conformity of a phone number, serial number, part number etc., and is sometimes more convenient than the Matches and Comprises comparators.

CompareCtrl

```
Format          CompareCtrl v1
Example         CompareCtrl 'MatchCase'
Purpose         Changes the default case sensitivity of comparisons
Parameters      v1 = Control setting
Controls        IgnoreCase/MatchCase
Similar Cmds    Que
```

Unless otherwise instructed by CompareCtrl, comparisons ignore text case, so that (for example) 'Cat' is considered the same as 'CAT' or 'cat'. You can use CompareCtrl to change this behaviour.

CompareCtrl affects comparisons *only*; it does not affect commands that search for text, such as Change, FindPosn, Lookup, Parse, Insert and so on.

Numeric

```
Format          v1 = Numeric v2 [v3]
Example         X = Numeric '3.14159' 'Yes'  ; Set X to 'Y'
Purpose         Evaluates whether or not a value is numeric
Parameters      v1 - Variable being set to 'Y' or 'N' (for Yes and No)
                v2 - Value being assessed
                v3 - Control setting: allow decimal point?
Controls        No/Yes
Defaults        v3 = 'No' (do not allow a decimal point — accept only
                integers)
Similar Cmds    The Matches and Comprises comparators
```

This function returns 'Y' if v2 is numeric (i.e. a number). Otherwise, it returns 'N'.

A leading - or + character is considered an acceptable part of a numeric value. Multiple decimal points (e.g. '12.34.56') are not accepted as numeric. Scientific notation (e.g. '1E32') is not accepted as numeric.

Que

```
Format          v1 = Que v2 k3 v4 [v5]
Example         MyVar = Que 'Cat' = 'Dog'  ; Compare two strings
Purpose         Saves the result of a comparison
Parameters      v1 - Variable being set to 'Y' or 'N' (for True or
                False)
                v2 - Value to be compared
                k3 - Comparator
                v4 - Value to compare to v2
                v5 - Control setting
Controls        IgnoreCase/MatchCase
Defaults        v5 = 'IgnoreCase' (unless overridden by CompareCtrl)
Similar Cmds    If, Begin
```

Que (short for "Question") is useful when you need to save the result of a comparison, or if you need a single instance of case sensitivity. For most comparisons, however, you will use If or Begin.

Chapter 9

Positional Commands

Cols

```
Format          v1 = Cols v2 v3 [v4]
Example         MyVar = Cols OtherVar 10 20  ; Columns 10 to 20
Purpose         Copies a range of columns (i.e. character positions)
Parameters      v1 - Variable being set
                v2 - Value (usually a variable) being copied
                v3 - Starting column
                v4 - Ending column
Defaults        v4 = v3 (i.e. copy one character)
Similar Cmds    Equals (Set Variable) with a range specified
Notes           If v3 is less than or equal to 0, it is treated as 1.
                If v3 points to a position beyond the end of v2, v1
                will be null.
                If v4 points to a position beyond the end of v2, it is
                treated as if it was the same as the length of v2.
```

FindPosn

```
Format          v1 = FindPosn v2 d3 [v4]
Examples        MyVar1 = FindPosn 'ABC'  'BC'   ; Set MyVar1 to '2'
                MyVar2 = FindPosn 'ABCC' '>*C'  ; Set MyVar2 to '4'
Purpose         Find the character position of text
Parameters      v1 - Variable being set
                v2 - Value being searched
                d3 - Decapsulator
                v4 - Decapsulator control settings
Controls        Exclude/Include; IgnoreCase/MatchCase
Defaults        v4 = 'Include MatchCase'
Similar Cmds    ScanPosn
Notes           If nothing is found, v1 is set to '0' (zero).
                If the "Exclude" decapsulator setting is used, FindPosn
                willpoint to the character position after the string it
                finds.
```

ScanPosn

```
Format          ScanPosn v1 v2 v3 v4 [v5]
Examples        See below
Purpose         Searches v3 for the start and end columns (character
```

```
                positions) for one of the strings or patterns listed in
                v4.
Parameters      v1 - Variable being set: "From" column
                v2 - Variable being set:  "To"  column
                v3 - The value being searched
                v4 - The list of strings or patterns for which to
                search
                v5 - Control settings
Controls        Any/First/Last; IgnoreCase/MatchCase; RegExp
Defaults        v5 = 'Any IgnoreCase'
Similar Cmds    FindPosn, Parse
Notes           Sets $Success ('Y' = something was found).
                If nothing is found, v1 and v2 are both set to '0'
                (zero).
                If RegExp is included in the control settings, each
                string is treated as a regular expression pattern
                rather than an actual string.
```

When you are analyzing data, a common requirement is to find out if one of several strings can be found in another string. For example, you might want to find out if a name starts with a salutation (Mr., Mrs., Ms.). ScanPosn lets you perform such a search with a single command.

For example, to search for a salutation in a string:

```
ScanPosn from to MyVar '/Mr./Mrs./Miss/Ms.'
```

If MyVar contains one of the scanterms (e.g. 'Mrs.') in the scanlist, ScanPosn will set the appropriate "From" and "To" variables. Thus, if MyVar contains 'Ms. Mary Jones', the "From" variable is set to '1' and the "To" variable is set to '3' (since 'Ms.' goes from positions 1 to 3 in MyVar).

If none of the scanterms is found, the "From" variable is set to '0' and the special variable $Success is set to 'N'. Thus, if MyVar contains 'John Smith', no salutation is found, and the ScanPosn command shown above will set the "From" variable to '0'.

The Scanlist

The scanlist can contain one or more scanterms. The *first* character in the scanlist is interpreted as the delimiter (separator) for the scanterms. Thus, the following scanlists are all valid:

```
'/Mr./Mrs./Miss/Ms.'                          ; Delimiter is: /
'xMr.xMrs.xMissxMs.'                          ; Delimiter is: x
'@Library@School@Gymnasium@Clinic/Hospital'  ; Delimiter is: @
'/Cow.'                                       ; Delimiter is: /
```

The first example ('/Mr./Mrs./Miss/Ms.') has already been demonstrated. The second example uses the letter 'x' as a delimiter. This might be a bad choice for a delimiter; it would cause a problem if one of the scanterms contained an 'x', since it would be treated as *two* scanterms. For example:

```
'xJohnxTrixiexFred'
```

The name 'Trixie' contains an 'x', so it would be broken down into two scanterms ('Tri' and 'ie'). You should always choose a scanlist delimiter that does not appear in the list of scanterms.

Accommodating Variation

When you design a scanlist, you should take into account the possibility that the input might contain strange variations. Consider this command:

```
ScanPosn x y 'Mr John Smith' '/Mr./Mrs./Ms.'
```

This search will fail because the 'Mr' is followed by a space, not a period. A more forgiving command would be:

```
ScanPosn x y 'Mr John Smith' '/Mr./Mrs./Ms./Mr /Mrs /Ms '
```

This would successfully locate the 'Mr ' string, and set x to '1' and y to '3'. (The '3' points to the space.)

HANDLING PREFIXES AND SUFFIXES

When designing a scanlist, you should consider that a scanterm might be part of a word. For example:

```
ScanPosn x y 'Mississippi Sue' '/Mr./Mrs./Miss/Ms.'
```

This will find the 'Miss' in Mississippi, even though this is not part of a salutation. A more appropriate command would be:

```
ScanPosn x y 'Mississippi Sue' '/Mr./Mrs./Miss /Ms.'
```

The space after 'Miss' in the scanlist ensures that if it is found, it will be separate from any word following it.

The trailing space is not necessary for the scanterm 'Mr.', since no word contains a period. However, if you do include spaces after the periods (as in '/Mr. /Mrs. /Miss /Ms. ') the consistency of rationale may simplify your subsequent script code.

You must also take suffixes into account. For example:

```
ScanPosn x y 'Zinc Enterprises' '/Inc/Co/Enterprises'
```

This will find the 'inc' in 'Zinc'. You can add a space in front of each scanterm to ensure that it is separated from any other word:

```
ScanPosn x y 'Zinc Enterprises' '/ Inc/ Co/ Enterprises'
```

You may be tempted to always put spaces on both sides of a word, to handle both prefixes and suffixes. However, consider this example:

```
ScanPosn x y 'Wazoo Inc' '/ Inc / Co / Enterprises '
```

None of the scanterms is found, because the 'Inc' in the source string does not end in a space. The control settings (described next) can help you address this kind of problem.

Control Settings

Unless otherwise instructed, ScanPosn will find the first scanterm that appears anywhere in the source string, and return its start and end positions. It will also ignore text case (e.g. 'CAT' = 'Cat'). You can modify this behaviour by using the optional control setting.

LAST, FIRST AND ANY

The 'Last' (i.e. rightmost) control setting tells ScanPosn to find the scanterm that has the highest "To" value with the lowest "From" value. This means that *all* of the scanterms are evaluated. Consider this command:

```
ScanPosn x y 'SHREWxxxCATxxxMOUSExxx' '/CAT/DOGGY/MOUSE/ELK' 'Last'
```

ScanPosn finds 'CAT', but continues looking to see if there are any better matches to the right. Eventually it finds MOUSE and sets x to '15' and y to '19' (pointing at 'MOUSE').

If you use the 'First' (i.e. leftmost) parameter, ScanPosn will check all the scanterms to find out which one has the lowest "From" position with the highest "To" value. For example:

```
ScanPosn x y 'SHREWxxxCATxxxMOUSExxx' '/CAT/DOGGY/MOUSE/ELK' 'First'
```

This will set x to '9' and y to '11' (pointing at 'CAT').

If you do not specify 'First' or 'Last', ScanPosn assumes you mean to use the 'Any' control setting. It finds the first scanterm it can, and ignores the rest. Here is an example.

```
ScanPosn x y 'SHREWxxxCATxxxMOUSExxx' '/CAT/DOGGY/MOUSE/ELK'
```

The first scanterm is 'CAT', and this can be found at positions 9 to 11. ScanPosn will return those values, and ignore the rest of the scanterms.

The 'Any' technique is useful if you want to know if one of the scanterms appears in the source string, but you are not interested in finding out which one. (You can specify 'Any' explicitly, but since it is the default control setting, this is not necessary.)

THE "BEST MATCH" PRINCIPLE

Note: The "Best Match" principle does not apply to the 'Any' control setting. It applies only to 'First' and 'Last' searches.

To use the ScanPosn command effectively, you must understand the concept of 'the best match'. This can be illustrated with an example:

```
ScanPosn x y 'MegaWhizco International' '/CO/WHIZCO/MEGAWHIZ' 'Last'
```

The ScanPosn command finds the scanterm 'CO' at positions 5 to 6. However, it continues looking for an even better match.

It finds that 'WHIZCO' is just as far to the right (i.e. it ends at position 6), but has a lower starting position. This makes it a better match.

The next scanterm ('MEGAWHIZ') has a lower starting position, but its ending position is not as good for a 'Last' search because it is not as far to the right.

As a result of all this, ScanPosn will set x to '5' and y to '10' — pointing to the "From" and "To" columns for 'WHIZCO'.

In other words, when ScanPosn is looking for the 'Last' scanterm, it will first identify the found scanterms which have the highest ending position, and then choose the longest one.

Here is an example using a 'First' search:

```
ScanPosn x y 'Our catalog is enclosed' '/CAT/MOOSE/CATALOG/DOG' 'First'
```

ScanPosn finds 'CAT' at positions 5 to 7, but as it continues checking the scanterms, it finds that 'CATALOG' is just as far to the left (i.e. it starts at position 5), but it is a better match since it has a higher ending position.

As a result, ScanPosn will set x to '5' and y to '11'.

The "Best Match" principle does not affect 'Any' searches. For example:

```
ScanPosn x y 'Our catalog is enclosed' '/CAT/MOOSE/CATALOG/DOG'
```

This sets x to '5' and y to '7'. Since this is a 'Any' search, ScanPosn stops looking as soon as it has found a match.

When doing an 'Any' search, you cannot be sure if any of the other scan terms appear in the source string. For example:

```
ScanPosn x y 'Our cat and dog are upstairs' '/CAT/DOG'
```

This will find 'CAT' and stop looking for additional matches. If you change the order of the scanlist, you will get *different* values:

```
ScanPosn x y 'Our cat and dog are upstairs' '/DOG/CAT'
```

This would give different values for the "From" and "To" variables. This is normal behaviour; an 'Any' search is useful only for detecting if one of the scanterms appears in the source string. After doing an 'Any' search, you will typically check the special variable $Success to see if a string was found.

Finding Patterns with ScanPosn

You can include the control setting "RegExp" (meaning "Regular Expression") to indicate that ScanPosn should look for a *pattern* of characters rather than specific characters. For example:

```
; Scale   ----+----1----+----
Source = 'Kitty Cats Are Cool'
ScanList = '/c.t/co*l'
ScanPosn p1 p2 Source ScanList 'First RegExp'
ScanPosn p3 p4 Source ScanList 'Last RegExp'
```

This would set the following values:

```
p1 =  7
p2 =  9
p3 = 16
p4 = 19
```

Regular Expressions are explained in the "Comparators" section of the user manual.

Chapter 10

Decapsulators

Overview

A "decapsulator" is a command parameter that defines a search for where a string of characters either begins or ends.

If that definition was not particularly helpful, it is because decapsulators cannot be fully described by a single sentence. But we encourage you to read through this section, because decapsulators are very important in Data Parse Scripting. Here is the reason why:

Decapsulators let a *single* Data Parse Scripting command accomplish what might take *dozens* of commands in a standard programming language.

The underlying concept is this: when analyzing data, the part you are interested in (the "field") is typically surrounded ("encapsulated") by some kind of distinctive text. A decapsulator looks for the distinctive text on either side of the data you want and thus helps you extract the field.

Sometimes the "distinctive text" appears more than once in the data you are examining. Decapsulators can handle this situation.

Sometimes one edge of the field is the beginning or end of the data you are examining, so there is no "distinctive text" to look for. Decapsulators can handle this situation, too.

Quick Reference

Here are some sample decapsulators:

Sample	"From" Decapsulator Meaning	"To" Decapsulator Meaning
'23'	From column 23 onwards	Up to column 23
'AB'	After first occurrence of 'AB'	Before first 'AB'
'1*CD'	After first occurrence of 'CD'	Before first 'CD'
'5*EF'	After fifth occurrence of 'EF'	Before fifth 'EF'

```
'<*GH'  After first occurrence of 'GH'  Before first 'GH'
'>*IJ'  After last  occurrence of 'IJ'  Before last  'IJ'
''      From left edge of data          From right edge of data
'-2'    Two columns in from the right   Same
```

Each of these techniques is explained below in more detail.

A Simple Example

Here is an example of how decapsulators work. Consider the following commands.

```
SourceVar = 'AAABBBCCC'
ResultVar = Parse SourceVar '3*A' '1*C'
```

The second command means "Set ResultVar to everything between the third occurrence of 'A' and the first occurrence of 'C'." In other words, ResultVar will end up containing 'BBB'.

Why Decapsulators are Necessary

When analyzing data, the fields you are interested in are sometimes arranged in tidy columns — but not always. Quite frequently, a field will start after some kind of delimiter, as in the following example.

```
SourceVar = 'Mouse,Gazelle,Mouse,Elephant'
```

Here the fields are separated by commas — a commonly-used format for data known as CSV (Comma Separated Values).

Extracting, say, the second item from free-form data is rather awkward if you are using a standard programming language. Fortunately, Data Parse Scripting has been developed with precisely this kind of situation in mind.

Using decapsulators, the Parse command lets you extract the "Nth" item. For example, to extract the third item in the free-form example above, you could use this command:

```
ResultVar  =  Parse  SourceVar  '2*,'  '3*,'
```

This means "Set the variable ResultVar by looking in SourceVar and taking everything between the second comma and the third comma". ResultVar would thus be set to 'Mouse'.

Introduction to Occurrence Numbers

Let's have another look at that last command.

```
ResultVar  =  Parse  SourceVar  '2*,'  '3*,'
```

The first decapsulator (i.e. the '2*,' part) is the "From" specification. The second decapsulator (i.e. the '3*,' part) is the "To" specification. It is interpreted as follows:

```
3   means "the third occurrence"
*   marks the end of the occurrence number
,   is the text you are looking for
```

Decapsulators can be used to find more than a single character. Let's say that (for some odd reason) a variable named xyz has been set such that each field is separated with a pair of X's, as in the following example (with the XX strings highlighted for clarity).

```
xyz = 'mouseXXgazelleXXmouseXXelephant'
```

You can extract the third item with this command:

```
 abc   =   Parse   xyz   '2*XX'   '3*XX'
 ___               ___    _ __     _ __
  |                 |     |  |     |  |
  Variable to set   |     |  |     |  |
   Variable to search     |  |     |  "To" text being sought
   "From" occurrence number  |     "To" occurrence number
      "From" text being sought
```

This command sets the variable abc to 'mouse', since it is found between the second and third occurrences of **XX**.

Sample Application

The Parse command is particularly useful for extracting information from CSV (Comma Separated Value) files. Here is an example of a CSV file:

```
"Mouse","Gazelle","Mouse","Elephant"
"Dog","Giraffe","Elk","Mongoose"
"Monkey","Snake","Caribou","Trout"
```

These fields could be extracted with this series of commands:

```
field1 = Parse $OutData '1*"'  '2*"'
field2 = Parse $OutData '3*"'  '4*"'
field3 = Parse $OutData '5*"'  '6*"'
field4 = Parse $OutData '7*"'  '8*"'
```

For the first line of the input file, field1 is set to 'Mouse', field2 is set to 'Gazelle', and so on.

Occurrence Number Syntax

Occurrence numbers must be larger than zero. The following lines are *not* valid Parse commands:

```
field1 = Parse $OutData '0*"'  '2*"'   ; "From" decapsulator is zero
field2 = Parse $OutData '-1*"' '2*"'   ; "From" decapsulator is
negative
```

The occurrence number must always be followed by an asterisk (the * character) so you can search for a number. Consider the following example (the meaning of which would be unclear without the asterisk):

```
MyVar  =  Parse  'xxx2yyy2zzz2'  '1*2'  '2*2'
```

This sets MyVar to the text occurring between the first '2' and the second '2'. In other words, MyVar is set to 'yyy'.

Finding the First and Last Occurrence

A decapsulator can refer to "the *last* occurrence":

```
xyz  =  Parse  'AaaBAbbBAccB'  '>*A'  '>*B'
```

In both decapsulators, the > symbol means "the last occurrence". Thus, the command means, "Set the xyz variable to everything between the last A and the last B". Thus, the xyz variable is set to "cc".

You can also use the < character to mean "the *first* occurrence", though this is somewhat redundant, since the following commands are equivalent:

```
abc  =  Parse  'AaaBAbbBAccB'  '<*A'  '<*B'
abc  =  Parse  'AaaBAbbBAccB'  '1*A'  '1*B'
abc  =  Parse  'AaaBAbbBAccB'  'A'    'B'
```

All three commands would set the abc variable to 'aa'.

Finding the Next Occurrence

When using occurrence numbers for certain kinds of data, you will often find that the "To" occurrence number is 1 (one) more than the "From" occurrence number. Consider this example:

```
xyz = 'AB,CD,EF,GH'
Field1 = Parse xyz '' '1*,'
Field2 = Parse xyz '1*,' '2*,'
Field3 = Parse xyz '2*,' '3*,'
```

For Field3 you are extracting everything between the second and third comma. It can become tiresome to write code like this — always adding one to the "From" occurrence number. Fortunately, you can use the "next occurrence" symbol '@*' in the "To" decapsulator:

```
xyz  =  'AB,CD,EF,GH'
abc  =  Parse  xyz  '2*,' '@*,'
```

This will set the "From" position to the second comma, and the "To" position to the comma after that (i.e. the third one). The '@*' symbol means "Look for the To text starting immediately after the From text".

Note: The "next occurrence" symbol (@*) can only be used in the "To" decapsulator.

Positional Decapsulators

Note: Positional decapsulators imply that operations proceed from or to the exact character position indicated, regardless of the control settings.

You can specify a number to indicate the "From" or "To" character position. In this mode, the Parse command behaves exactly like the Cols command. Thus, the following two commands accomplish the same thing:

```
xyz  =  Parse  MyVar  '10'  '20'
xyz  =  Cols   MyVar  '10'  '20'
```

As such, this is not particularly helpful. However, you can combine positional decapsulators with other types of decapsulators, as in this example:

```
MyVar  =  'ABCD/abcd/'
abc    =  Parse  MyVar  '3'  '1*/'
```

This will set the variable abc to 'CD'.

Negative Positional Decapsulators

You can also count backwards from the right edge of the data. Consider this example:

```
MyVar  =  'ABCDEFG'
xyz    =  Parse  MyVar  '-3'  '-2'
```

This will set the variable xyz to 'EF'. (The last character in a variable is represented by position '-1'.)

Using Positional Decapsulators Safely

You need to be careful when you use positional decapsulators. If, for example, you use a negative positional decapsulator, and you end up referring to a character before the beginning of the string, it isn't clear to the Data Parse engine what you "meant" by that. (In all likelihood, you didn't mean anything; these situations sometimes arise if you have not considered all possible variations in format of the input data.)

For the reason just noted, and others that will become evident as you write scripts: if there is a chance that a positional decapsulator will refer to a character position of zero or less, or if it might refer to a position beyond the end of the data, your script should check the length of the data before trying the command.

The Plain Decapsulator

The occurrence number is not always needed. Either the "From" or "To" decapsulator can be represented as a plain (non-numeric) string, as in the following example.

```
OldVar  =  'zzzABChelloXYZzzz'
NewVar  =  Parse  OldVar  'ABC'  'XYZ'
```

This would set the variable named NewVar to 'hello' since it means:

1. Copy from the character following the first 'ABC'
2. Copy up to the character preceding the first 'XYZ'

This is, of course, equivalent to the following command, which uses occurrence numbers:

```
NewVar  =  Parse  OldVar  '1*ABC'  '1*XYZ'
```

In general, it is best to explicitly give occurrence numbers, unless you know that the format of the data is not going to change.

Unsuccessful Searches

When a command that uses decapsulators does not find the search text, it does as little as possible. For example, if a Parse command does not find the encapsulating text, it sets the variable to a null ('). Here are two examples:

```
abc  =  Parse  'ABCDEFGHIJ'  '1*K'  '1*J'   ; There is no 'K'
abc  =  Parse  'ABCDEFGHIJ'  '1*A'  '1*X'   ; There is no 'X'
```

To illustrate this principle further: if the Overlay command does not find the search text, it does nothing at all, as in the following example.

```
abc = 'ABCDEFGHIJ'        ; Set a variable
Overlay abc 'K' 'LMNOP'   ; There is no 'K', so nothing is done
```

If the "From" value is less than the "To" value, the Data Parse engine will display an error message, then terminate further processing. For example:

```
abc  =  Parse  abc  'ABCDEFGHIJ'  '1*J'  '1*A'   ; 'J' comes after 'A'
```

This kind of failure typically happens if the data contains an odd arrangement of text that you had not foreseen. In such case, it would not be reasonable for processing to continue; you need to be warned about departures from what your script implies you expected.

The Control Setting

Commands that use decapsulators typically have a "control setting" that allows you to adjust the way the command is performed. A few examples follow.

The Parse command's control setting tells Parse whether to include or exclude the surrounding (i.e. searched-for) text. By default, the surrounding text is excluded (unless the decapsulator is positional). However, if you want to include it, you can add 'Include' at the end of the Parse command, as in this example:

```
xyz  =  Parse  'aXcaYcaZc'  '2*a'  '2*c'  'Include'
```

This tells the command to give you everything between the second 'a' and the second 'c' — *including* the 'a' and 'c'. In other words, this sets the variable xyz to 'aYc'.

You can also set the Control specification to 'Exclude', though since this is the default setting for Parse, it isn't necessary. Here is an example:

```
xyz  =  Parse  'a1ca2ca3c'  '2*a'  '2*c'  'Exclude'
```

This sets the variable xyz to '2'.

You can specify several control settings at once, separated by spaces. By default, the Parse command's control setting is 'Exclude MatchCase' but you could set this to (for example) 'Include IgnoreCase'.

The Null Decapsulator

Here is a helpful variation of the "From" decapsulator:

```
''   means "Start from the first character in the value being analyzed"
```

A similar variation can be used with the "To" decapsulator:

```
''   means "End with the last character in the value being analyzed"
```

If you use the null ('') decapsulator for "From" or "To", the "found" value (the first character for "From", or the last character for "To") will always be included (see the section "Overlapping Decapsulators" for an exception to this rule). Here is an example:

```
xyz  =  Parse  'ABCABCABC'  ''  '2*C"
```

This sets the variable xyz to 'ABCAB'. The "From" value (i.e. the first character) is *not* excluded. However, when Parse finds the "To" value (i.e. the second occurrence of the letter C) it *is* excluded. If you want to include the second 'C', you should write the command this way:

```
xyz  =  Parse  'ABCABCABC'  ''  '2*C'  'Include'
```

Incidentally, the following two commands accomplish the same thing:

```
xyz  =  Parse  'ABCD'  ''  ''
xyz  = 'ABCD'
```

They are equivalent because the Parse command means "Set the variable xyz with everything between (and including) the first character and the last character".

Why Null Decapsulators Work Differently

It may not be immediately obvious why decapsulator-enabled commands treat the null ('') decapsulator differently. The examples given here are very simple, and not representative of real-world applications.

In day-to-day usage, though, you will frequently find it helpful to be able to specify a command that says, "Give me everything from the beginning of the line to just before such-and-such" or "Give me everything from such-and-such a point until the end of the line."

For example, here is a command that means "Give me everything from just after the dollar sign, to the end of the line":

```
xyz  =  Parse  'Please give me $199.00'  '1*$'  ''
```

This sets xyz to "199.00". If you want to include the dollar sign, write the command this way:

```
xyz  =  Parse  'Please give me $199.00'  '1*$'  ''  'Include'
```

In this example, the 'Include' control setting affects the way the "From" decapsulator works, since it is using an occurrence number. The null decapsulator is not affected.

Overlapping Decapsulators

Earlier, it was mentioned that the text found by the null decapsulator is "always included" and is not affected by the 'Exclude' control setting. There is an exception to this: if the null decapsulator's "found text" is contained in the text found by the other decapsulator, it *can* be affected. For example:

```
xyz  =  Parse  'ABCDEFABCDEF'  ''  '1*AB'  'Exclude'
```

This command means "Give me everything between the first character and the first occurrence of AB". Since the two items overlap (i.e. the first 'AB' includes the first character), the first character does indeed get excluded. As a result, the xyz variable is set to an empty string ('').

Here is another example.

```
xyz  =  Parse  'ABCDEFABCDEF'  '>*F'  ''  'Exclude'
```

This command means "give me everything between the last occurrence of F and the last character". Both decapsulators refer to the same character (i.e. the final 'F'), so it is excluded. As a result, the xyz variable is set to an empty string ('').

Note: In some circumstances, the FindPosn command is *not* affected by this exception. It will do its best to make sense of your request if the decapsulators overlap and one of them is a null decapsulator.

Parsing Empty Fields

Consider the following command, which is operating on CSV (Comma Separated Value) data.

```
xyz  =  Parse  ',,,JOHN,SMITH'  '2*,'  '3*,'
```

There is nothing between the second and third comma, so the xyz variable is set to '' (an empty string).

Now consider this command:

```
xyz  =  Parse  ',,,JOHN,SMITH'  ''  ','
```

You are asking for everything from the first character to the first comma (which also happens to be the first character). Obviously, there is nothing "between" the two characters, so the xyz variable would be set to '' (an empty string). This may be what you wanted, but whenever you are dealing with a field at the beginning or end of data, and there is a chance the field might be empty, it is a good idea to test your script to make sure that it does what you expect.

Decapsulator Commands

Overview

This section documents the specific decapsulator commands. For a broader overview of decapsulators, please see the Decapsulators section of this user manual.

Insert

```
Format          Insert v1 d2 v3 [v4]
Examples        Insert Var '10' 'Cat'    ; Insert 'Cat' at column 10
                Insert Var '-1' 'X'      ; Insert 'X' before last char
                Insert Var '>*A' 'Y'     ; Insert 'Y' before last 'A'
                Insert Var 'B' 'Z' 'Exclude'  ; Insert 'Z' after first
                'B'
Purpose         Inserts v3 into v1 at the position determined by d2
Parameters      v1 - Variable being modified
                d2 - Decapsulator
                v3 - Value to insert at the position found by v2
                v4 - Decapsulator control settings
Controls        Exclude/Include; IgnoreCase/MatchCase
Defaults        v4 = 'Include MatchCase' ("Include" means "insert
                before")
Similar Cmds    Change, Overlay
Notes           If decapsulator d2 is not found, nothing is done.
                Sets $Success ('Y' = decapsulator value was found).
```

Overlay

```
Format          Overlay v1 d2 v3 [v4]
Examples        Overlay MyVar '10' 'Cat'  ; Overlay 'Cat' at column 10
                Overlay MyVar '<*A' 'X'   ; Overlay first 'A' with 'X'
                Overlay MyVar '3*B' 'Y'   ; Overlay third 'B' with 'Y'
                Overlay MyVar '>*C' 'Z'   ; Overlay last  'C' with 'Z'
Purpose         Overwrites v1 with v3 at the position determined by d2
Parameters      v1 - Variable being modified
                d2 - Decapsulator
                v3 - Value to overwrite at the position found by v2
                v4 - Decapsulator control settings
Controls        Exclude/Include; IgnoreCase/MatchCase
Defaults        v4 = 'Include MatchCase'
Similar Cmds    Change, Insert
Notes           If decapsulator d2 is not found, nothing is done.
                If necessary, v1 will be lengthened to make room for
                v3.
                Sets $Success ('Y' = decapsulator was found).
```

Parse

```
Format          v1 = Parse v2 d3 [d4 [v5]]
Examples        See below
Purpose         Parses free-form data
Parameters      v1 - Variable being set
                v2 - Value being searched
                d3 - "From" decapsulator
                d4 - "To" decapsulator
                v5 - Decapsulator control settings
Controls        Exclude/Include; IgnoreCase/MatchCase; Cut; Relaxed
Defaults        d4 = '' (Null decapsulator, meaning "to the end of the
                line")
                v5 = 'Exclude MatchCase'
Similar Cmds    FindPosn, ScanPosn
```

Parse is one of the most powerful commands in the Data Parse Scripting repertoire. For an introduction to working with decapsulators (along with many examples of the Parse command), please see the Decapsulators section of this user manual.

The "Cut" Control Setting

The Cut control setting removes the text that is found in the variable being examined, along with the encapsulating text. This technique is particularly useful when using a technique called "Left-Peeling". Consider the following script:

```
MyVar      =  'John,Aloysius,Smith'
FirstName  =  Parse  MyVar  ''  ','  'Cut'  ; Cut out first name
MidName    =  Parse  MyVar  ''  ','  'Cut'  ; Cut out middle name
LastName   =  MyVar                         ; Save what's left
```

This "peels" off fields from the left side of the variable MyVar. It will set the variable FirstName to 'John', the MidName variable to 'Aloysius', and LastName to 'Smith'.

The "Relaxed" Control Setting

The "Relaxed" control setting lets the "To" decapsulator look for text that may not be there. If it is *not* there, the "To" decapsulator is treated like a null ('') decapsulator.

Let us say you are extracting information from the $OutData special variable and some of the lines you have to parse look like this:

```
Bob
Fred Smith
Mary Anastasia Jones
John Quincy Publique Sr.
```

This data is inconsistent, so you cannot predict how many parsing cuts to make. With the "Relaxed" control setting, this is not a problem.

Consider the following example.

```
Name1   =  Parse  $OutData  ''  ' '  'Cut Relaxed'
Name2   =  Parse  $OutData  ''  ' '  'Cut Relaxed'
Name3   =  Parse  $OutData  ''  ' '  'Cut Relaxed'
Name4   =  Parse  $OutData  ''  ' '  'Cut Relaxed'
Name    =  Name1  '/'  Name2  '/'  Name3  '/'  Name4  '/'
TrimChar Name 'R/'
```

This would set the Name variable to the following values:

```
Bob
Fred/Smith
Mary/Anastasia/Jones
John/Quincy/Publique/Sr.
```

The preceding example could, of course, have been accomplished more easily with the Change command, but it is included here as a demonstration, not a real-world application.

Chapter 11

Lookup and Database Commands

Overview

The LookupFile and Lookup commands give Data Parse Scripting simple database capabilities: you can use a "key" to look up an item of data. For example, a database of country abbreviations could look up 'US' (the "key") to find 'United States of America' (the "data").

The MassChange command can be used to apply search-and-replace edits to a line of data, based on the information contained in a Lookup file.

Lookup files can be prepared in a text editor program. You can name them anything you want, though by convention the file names start with `Luf` and have a `.txt` extension (example: `LufCustomers.txt`).

The ScanFollow command provides a simple form of lookup capability that does not involve an external file.

Lookup

```
Format          v1 = Lookup v2 t3 [v4]
Example         MyVar = Lookup 'Car' 'MyTable'  ; Find 'Car' in
                'MyTable' table
Purpose         Looks up a value in a table read in from an external
                file
Parameters      v1 - Variable being set (this is the "data")
                v2 - Value being sought (this is the "key")
                t3 - Table name (as defined by LookupFile)
                v4 - Control setting
Controls        IgnoreCase/MatchCase
Defaults        v4 = 'MatchCase' (v2 must match the table's key field
                exactly)
Similar Cmds    SetFromFile
Notes           Sets $Success ('Y' = v2 was found).
```

LookupFile

Format	LookupFile t1 v2 [v3 [v4 [v5]]]
Example	LookupFile 'MyTable' 'C:\MyData\LufMyDatabase.txt' 3 2
Purpose	Reads in a table for use with the Lookup command
Parameters	t1 - A name for this table (used by the Lookup command) v2 - Name of the file being read in v3 - Key field number (what you are looking for) v4 - Data field number (what you find) v5 – Control setting
Controls	Decode/NoDecode
Defaults	v3 = 1 v4 = 2 v5 = Decode
Restrictions	LookupFile reads the entire table into memory. Thus, multi-megabyte lookup files may cause problems on some machines. (Comments are ignored, so you can use as many as you want without affecting performance.)
Notes	If the filename (v2) does not specify a path, LookupFile will use the Search Path to look for it.

The sample lookup file `LufSample01.txt` contains comments that explain the fundamental techniques you will need to define a lookup file.

Here is an example of a lookup file, named `ScrSuppliers.txt`:

```
; Lookup file for my suppliers, giving supplier number, name, and phone
number
1,"Pinnacle Software","416-287-8892"
2,"Fred's Computers","514-555-1234"
3,"DigiRamaTech","212-555-4321"
```

This particular lookup file starts with a comment line. The data lines have three fields. You could look up the first field (the supplier number) to determine the supplier name or phone number.

The NoDecode control setting turns off the conversion of encoded text (e.g. $0D and #13). This is occasionally necessary when using a CSV (Comma Separated Value) file that does not put quotes around text fields. The default setting (Decode) will decode the string (see "Untypeable Characters").

MassChange

```
Format          MassChange v1 t2 [v3]
Example         MassChange MyVar 'MyTable' 'IgnoreCase'
Purpose         Applies every possible change listed in a Lookup file
Parameters      v1 - The variable being changed
                t2 - Table name (as defined by LookupFile)
                v3 - Control setting
Controls        IgnoreCase/MatchCase
Defaults        v3 = 'MatchCase'
Similar Cmds    Change
```

MassChange is typically used for applying corrections to common typographical errors, rationalizing address data (e.g. changing 'app.' to 'Apt.') or for remapping one character set to another one.

The sample lookup file `LufSample01.txt` contains comments that explain the fundamental techniques you will need to perform any of these tasks.

ScanFollow

```
Format          v1 = ScanFollow v2 v3 [v4 [v5]]
Example         X = ScanFollow 'C' '/A/B/C/D/E'  ; Set variable X to
                'D'
Purpose         Returns the next item in a character-delimited list
Parameters      v1 - Variable being set
                v2 - The value being sought in the list
                v3 - The list (first character defines the list
                delimiter)
                v4 - Value to return if v2 is not found or is last in
                the list
                v5 - Control setting
Controls        IgnoreCase/MatchCase
Defaults        v4 = Null (empty) string
                v5 = IgnoreCase
Similar Cmds    Lookup
```

ScanFollow looks up a string in a list then returns the *next* string in the list. It can be used as a simple lookup tool, or to step through a series of strings.

If using ScanFollow as a lookup tool, remember that (unlike the Lookup command), ScanFollow does not distinguish between "key" and "data" — it simply finds the first occurrence of the value being sought and returns the next item in the list.

Advanced Database Connectivity

Data Parse allows the reading and writing to supported ODBC sources.

This allows you to connect to your existing Microsoft SQL Server or Oracle, or almost any ODBC compliant data server.

You use the SendToDB, in conjunction with $CfgODBCConnection to send and receive data to your ODBC configured source.

The website http://www.connectionstrings.com currently offers a number of tips on specifying connection strings.

SendToDB

```
Format:         SendToDB v1 [v2] v3 v4
Example:        SendToDB 'select * from customers' 'c:\holdingfile.csv'
                dataholder resultcode
                SendToDB 'update customers set donotcontact=1' ''
                dataholder resultcode
Purpose:
Parameters:     v1 - Command or variable containing command to send to
                database
                v2 - Filename where results returned from the database
                should be saved must not exist)
                v3 - Variable to store result set(s)
                v4 - Handled Exception Code. Any unhandled exceptions
                will stop the script from running
                100 - Connection string is empty ($CfgODBCConnection is
                not defined)
                101 - Invalid connection string
                200 - Incorrect file name
                201 - File already exists
                If v2 is omitted, result data is not written to the
                disk.
```

Data is exported in separated value format, with the delimiter being used as the one defined in $CfgDelimiter. That default value is \0

v3 will use up to about 80% of available memory to store any result set. Please clear out your variables if you are going to be processing very large or millions of records, that do not need to be reused.

Chapter 12

Calculation Commands

Calc

```
Format          v1 = Calc v2 o3 v4
Example         MyVar = Calc 3 + 4  ; Set MyVar to 7
Purpose         Perform an integer calculation
Parameters      v1 - Variable being set
                v2 - First integer number
                o3 - Operation
                v4 - Second integer number
Similar Cmds    CalcReal
Notes           All extraneous text (i.e. everything but 0 to 9 and the
                minus sign) is removed from the values v2 and v4.
                If either v2 or v4 are null, they are interpreted as 0.
```

The operations used by Calc (and also CalcReal) are as follows:

Operation	Meaning	Operation	Meaning	Operation	Meaning
+	Add	*	Multiply	Highest	Pick biggest number
-	Subtract	/	Divide	Lowest	Pick smallest number

The Calc command uses *integer* division. This means that any remainder is discarded. Thus, the calculation 10 / 3 will return a value of 3, since 3 goes into 10 three times, with a remainder of 1 (which is ignored).

The Calc command can handle very large numbers, but if your calculations take you beyond 18 digits, you are getting very close to the edge of Data Parse's integer range.

CalcReal

```
Format          v1 = CalcReal v2 o3 v4 [v5]
Examples        MyVar = CalcReal 3.1 * 4.3       ; Set MyVar to 13.33
                MyVar = CalcReal 10.0 / 3.0  5   ; Set MyVar to 3.33333
Purpose         Perform a real-number calculation
Parameters      v1 - Variable being set
                v2 - First real number
                o3 - Operation
                v4 - Second real number
                v5 - Number of decimal places
Defaults        v5 = 2
Similar Cmds    Calc
Notes           All extraneous text (i.e. everything but 0 to 9, the
                minus sign and the decimal point) is removed from v2
                and v4.
                If either v2 or v4 are null, they are interpreted as
                0.0.
                By default, operations with fixed decimal places are
                subject to rounding. See the Rounding command for
                details.
```

For a list of operations, see the Calc command.

Real number operations have 18 valid digits across the range (expressed in scientific notation) of

```
3.6 x 10^-4951 to 1.1 x 10^4932
```

If you are working with very large numbers, it is a good idea to write some experimental scripts to determine if the accuracy you require can be obtained.

If v5 is set to "Float", CalcReal will calculate as many decimal places as it possibly can. Before you do this, however, you should be aware that when computer calculations are taken to the limit of the software's precision, it can result in inaccuracy.

Dec

```
Format          Dec v1 [v2]
Example         Dec MyVar 3  ; Subtract 3 from variable MyVar
Purpose         Decrements (decreases) an integer number
Parameters      v1 - Variable being set
                v2 - The amount by which to decrement v1
Defaults        v2 = 1
Similar Cmds    Inc
Notes           Decrementing with a negative value increases v1
```

The Dec command can handle very large numbers, but if your calculations take you beyond 18 digits, you are getting very close to the edge of Data Parse's integer range.

Inc

```
Format          Inc v1 [v2]
Example         Inc MyVar 3  ; Add 3 to variable MyVar
                Inc MyVar    ; Add 1 to variable MyVar
Purpose         Increments (increases) an integer number
Parameters      v1 - Variable being set
                v2 - The amount by which to increment v1
Defaults        v2 = 1
Similar Cmds    Dec
Notes           Incrementing with a negative value decreases v1
```

The Inc command can handle very large numbers, but if your calculations take you beyond 18 digits, you are getting very close to the edge of Data Parse's integer range.

Rounding

```
Format          Rounding c1
Example         Rounding 'Yes'
Purpose         Turns rounding-up on or off for fixed-place answers
                calculated by the CalcReal command
Parameters      c1 - 'Yes' or 'No' ('Yes' = Round-up the answers)
Notes           Turning off rounding is not recommended. By default,
                rounding-up is on. If you turn it off, the answers are
                simply truncated according to the number of fixed
                decimal places. If you do this, you should be aware of
                the problems inherent in computer calculation. For
                details, see CalcReal.
```

Fixed-place numbers are rounded-up by adding 5 to the next-lowest position. So 4.56 with one fixed-decimal place is rounded by adding 0.05, yielding 4.61, which truncates to '4.6'. If the answer is negative, the adjustment is subtracted rather than added, so -4.56 with one fixed decimal becomes '-4.6'.

Chapter 13

Date and Time Commands

Overview

All date-oriented commands that involve calculations (e.g. AddDays and AddWeekDays) are limited to the years 1900 to 2999. These commands normally expect to see the year expressed with four digits (e.g. 2009), but if you pass them a two-digit year they will try to guess the appropriate millennium. That is to say, if the two digits are in the range 80 to 99, the year will be taken to mean 1980 to 1999.

When using commands that handle date and time, you should be careful that you are specifying valid values. For example, if you set the hour to 999 the program will terminate with an explanatory error message.

DateTimeFormat

```
Format          v1 = DateTimeFormat v2 v3 v4 v5 v6 v7 v8
Examples        DateTime = DateTimeFormat 2008 12 25 17 29 30 'Y-?N-?D
                H:?I?S'
                DateOnly = DateTimeFormat 2009 12 25 '' '' '' 'Y-?N-?D'
                TimeOnly = DateTimeFormat '' '' ''  17 29 '' '?h:?I a'
Purpose         Formats a date or time, or both, into a text string
Parameters      v1 = Variable being set
                v2 to v4 = Year, Month, Day (all may be set to null if
                not used)
                v5 to v6 = Hour (24-hour), Minute, Second (all may be
                set to null)
                v8 = Date and time format codes (explained below)
Controls        See "Date and Time Format Codes"
```

Date and Time Format Codes

Codes	Explanations
?	Padding position to prefix a zero to a single-digit value
a	Ante Meridiem or Post Meridiem, in lowercase: am or pm
A	Ante Meridiem or Post Meridiem, in uppercase: AM or PM
D	Day of the month

h	Hour of the day (12-hour clock)
H	Hour of the day (24-hour clock)
I	Minute of the hour
m	Month of the year (three letters, capitalized)
M	Month of the year (three letters, uppercase)
N	Month of the year (numeric)
S	Second of the minute (numeric)
t	Month of the year (full name, capitalized)
T	Month of the year (full name, uppercase)
Y	Four-digit year (if input is two digits, 80 to 99 yield 1980 to 1999)
y	Two-digit year (if input is four digits, first two digits are dropped)

Examples

Sample Format Settings	Sample Results	Comments
'M ?D ?y'	JAN 12 09	
'm ?D ''?y ?H:?I:?S a'	Feb 22 '09 04:01:23 am	
't D, Y, H:?I A'	July 4, 1981, 2:01 PM	
't D, Y, ?H:?I:?S'	May 4, 1981, 14:01:02	
'?D/?N/?y'	01/02/03	European date format
'?N/?D/?y'	02/01/03	Date format in USA
'Y-?N-?D'	2003-02-01	ISO 8601 international date

AddDays

Format	AddDays v1 v2 v3 v4
Example	AddDays MyYear MyMonth MyDay 14
Purpose	Adds the specified number of days to the specified date
Parameters	v1 to v3 = Year, Month and Day (these must be variables) v4 = Number of days to add (if negative, days are subtracted)
Similar Cmds	AddWeekDays
Notes	Please see the "Overview" section for more information about working with date data. If v4 = 0 then the date is not changed.

AddWeekDays

Format	AddWeekDays v1 v2 v3 v4 [t5]
Example	AddWeekDays MyYYYY MyMM MyDD 23 'MyHolidays'
Purpose	Adds the specified number of weekdays to the specified date, optionally skipping holidays as well (if t5 is specified)
Parameters	v1 to v3 = Year, Month and Day (these must be variables) v4 = Number of days to add (if negative, weekdays are subtracted) t5 = Table name defined by the LookupFile command
Defaults	If t5 is not specified, AddWeekDays will skip only Saturdays and Sundays.
Restrictions	If a holiday is not listed in the table specified by

```
                t5, AddWeekDays does not know about it.
Similar Cmds    AddDays
Notes           Please see the "Overview" section for more information
                about working with date data.
                If v4 = 0 then the date is moved forward to the next
                day that is considered a weekday (i.e. holidays are
                also skipped).
```

Two sample lookup files for holidays are available from National Data Parsing Corporation The files are:

```
LufHolidaysCanada.txt
LufHolidaysUSA.txt
```

These list the holidays for Canada and the USA. The Canadian file contains extensive notes on calculating and adding new holidays, and also explains how you can create a custom holiday file.

We strongly recommend reviewing a holiday lookup file before using it. Some holidays that are included in the files mentioned above are "commented out" because they are not celebrated nationally. You can edit a copy of the file (and give it a different name) by using a text editor such as Windows Notepad.

Note: If you create a lookup file for holidays in a country other than the ones we have included, we would be most appreciative if you would send us a copy.

DayOfTheWeek

```
Format          v1 = DayOfTheWeek v2 v3 v4 [v5]
Example         DayName = DayOfTheWeek 2010 12 25
                '/Sun/Mon/Tue/Wed/Thu/Fri/Sat'
Purpose         Sets v1 to the name of the day of the week
Parameters      v1 = Variable being set
                v2 to v4 = Year, Month, Day
                v5 = List of day names
Defaults        v5 = '/1/2/3/4/5/6/7' (1 = Sunday)
Notes           Please see the "Overview" section for more information
                about working with date data.
```

If you specify the names of the days of the week (v5), you must list all 7 days (starting with Sunday). The first character in the list is taken as the delimiter. The usual choice is the slash character, but a different character could be used, as long as it does not appear in any of the day names.

Now

```
Format          v1 = Now [v2]
Example         MyDateTime = Now 'Y-?N-?D H:?I?S'
Purpose         Sets v1 to the current date, or time, or both
Parameters      v1 = Variable being set
                v2 = Date and time format codes (see "DateTimeFormat")
Defaults        v2 = 'Y/?N/?D' (e.g. 2010/12/25)
Similar Cmds    DateTimeFormat
Notes           Please see the "Overview" section for more information
```

about working with date data.

Chapter 14

Binary Conversion Commands

Overview

The binary conversion commands deal with transformation of data between a computer's representation (e.g. 10110111) and human-readable format (e.g. plain text).

A computer program that uses the ASCII character set will internally represent the letter A with the number 65 (or, more accurately, the binary value 01000001). This is not normally an issue, since a program designed to work with ASCII characters will show you the letter A on the screen. However, if the data is stored in the EBCDIC character set then the letter A will be represented by a different number. In such case you may need to convert the EBCDIC representation to the ASCII representation. Fortunately, this is quite easy to do, and a sample script to perform this conversion is available in the National Data Parsing Corporation Knowledge Base (available via our web site, at www.Data Parse.com).

A more difficult problem arises when an input file contains numbers in "raw binary". That is to say, numbers in the file do not appear in plain text (e.g. '123'). Rather, they are represented in a form that is familiar to the computer, so the number 123 might be represented as 01111011 (hexadecimal $7B).

Further complicating the issue is the fact that computers can represent numbers in various ways. 123 can also be represented by 0111101100000000. This looks very similar – after all, it is the same 8 bits as shown previously, followed by 8 zero bits – but in this case the number is being represented as a 2-byte value instead of a 1-byte value. The specific representation used by a number can be very important. If you translate a number using the wrong technique you could end up showing incorrect values, such as misinterpreting 255 as -1.

A final twist to this problem is that the various representations for numbers do not always have the same names. The word "byte" always means "8 bits", but even here we can run into trouble. A "byte" is sometimes known as an "octet", and sometimes it

is assumed that one of the bits (the high bit) is not used, or is used for a purpose other than representing data (i.e. it is a "parity bit"). The term "word" can refer to one byte, two bytes, four, eight bytes or more, depending on the context.

For this reason, the binary conversion commands do not refer to data representations using traditional terminology such as "byte", "word" and "integer". Rather, they use "Data Parse Conversion Codes" to avoid confusion. For example, "I1U" means "Integer, 1 Byte, Unsigned". This can only refer to an 8-bit value that holds a value from 0 to 255. A complete list of the Data Parse Conversion Codes is shown below.

Data Parse Conversion Codes

For the reasons given in the Overview (above), Data Parse refers to data representations using "Conversion Codes" rather than standard terms such as "byte", "word", "integer", "long integer" and so on.

Here is a list of the conversion codes:

Code	Definition	Some Conventional Names (see Note)
I1U	Integer, 1 Byte, Unsigned	Byte, Octet
I1S	Integer, 1 Byte, Signed	ShortInt, Byte
I2U	Integer, 2 Bytes, Unsigned	HalfWord, Word
I2S	Integer, 2 Bytes, Signed	Integer, HalfWord
I4U	Integer, 4 Bytes, Unsigned	DoubleWord, LongWord, Word
I4S	Integer, 4 Bytes, Signed	Integer, LongInt, Cardinal
I8S	Integer, 8 Bytes, Signed	DoubleWord, Int64, QuadWord
R4S	Real, 4 Bytes, Signed	Real, Single
R6S	Real, 6 Bytes, Signed	Real, Real48
R8S	Real, 8 Bytes, Signed	Double, Real
R10S	Real, 10 Bytes, Signed	Comp, Extended
R8$	Real (4 places), 8 Bytes, Currency	Currency
HEX	Hexadecimal text (e.g. 'F0')	Hex string
BIN	Binary text ('1111_0000')	(Used only in Data Parse)
BIC	Binary text compressed ('11110000')	Binary string

Note: The conventional names should not be taken too seriously. A "word", for example, might refer to 1, 2, 4, 8 or more bytes, depending on the context. Different computers and different computer languages may use the same term to refer to completely different things.

These codes are not supported by all conversion commands. For example, you cannot convert from BIC format to I1U format. (In actual conversion applications, that particular transformation would almost never be required.)

You may occasionally encounter data representations that are not yet supported by Data Parse. For example, at the moment we do not translate the COMP data types used by COBOL programs. If you encounter an unsupported data type you can inquire about our schedule for adding the feature, and in the meantime you can use the CalcBinary command to transform the data into a form that *is* supported.

BinaryToText

```
Format          v1 = BinaryToText v2 v3 [v4]
Example         MyByte = BinaryToText $Data[20] 'I1U'
Purpose         Returns the text representation of raw binary data
Parameters      v1 = Variable being set
                v2 = Value being converted
                v3 = Data Parse Conversion Code (see "Data Parse
                Conversion Codes", in the "Overview" section)
                v4 = Control setting (decimal places for real number
                conversions)
Defaults        v4 = 2
Similar Cmds    TextToBinary
Notes           Please see the "Overview" section for background
                details
```

All computer data is, of course, binary data at some level. The BinaryToText command is therefore a data format converter. For example, you can transform the value $FF into the string '255' or '-1', depending on the conversion code you use. $FF would produce '255' if you used the conversion code 'I1U' (Integer, 1 byte, Unsigned) and '-1' if you used 'I1S' (Integer, 1 Byte, Signed).

When we speak of conversion to 'text', we are referring to the fact that all variables in Data Parse Scripting are expressed as human-readable text. To provide the ability to develop scripts quickly, there are no "data types" such as Integer, Real and so on, and no need to "declare" the variables you are using. So the Data Parse Engine decides that '1234' is an integer number if it used in a context where that matters, such as the Calc command. Similarly, it decides that '1234.56' is a real number if it is fed into the the CalcReal command.

The BinaryToText command provides you with the ability to translate from "typed" data that you find in a raw binary input file into the generalized "text" format used by Data Parse Scripting. This means that the resulting value can be fed into Data Parse commands, or send to an input file.

The sample script ScrPSTMain provides many examples (with explanatory comments) of data format conversion using the BinaryToText command.

CalcBinary

```
Format          v1 = CalcBinary v2 v3 v4
Example         ShiftedByte = CalcBinary $Data[20] 'SHL' 1
Purpose         Returns the result of a binary operation (e.g. XOR,
                SHL)
Parameters      v1 = Variable being set
                v2 = A value upon which the operation is being
                performed
                v3 = The name of the operation
                v4 = The second value for the operation
Notes           Unlike Calc and CalcReal, the operation name (v3) must
                be in quotes
```

The CalcBinary command lets you manipulate data at the bit level. This can be useful for data format conversions that are not currently supported by the BinaryToText command. It is also useful for data decryption, CRC generation and so on.

In keeping with Data Parse's avoidance of data types (i.e. everything looks like text), you can perform the CalcBinary operations on data of any length. Thus, you could perform the ROR operation on a single byte, or hundreds of bytes.

Here is a summary of the operations supported by the CalcBinary command:

Name	Description	Notes
AND	Logical And	v2 and v4 must be the same length
NAND	Logical Not-And	v2 and v4 must be the same length
OR	Logical Or	v2 and v4 must be the same length
SHL	Shift Bits Left	v4 specifies number of bits to shift
SHR	Shift Bits Right	v4 specifies number of bits to shift
XOR	Exclusive Or	v2 and v4 must be the same length
ROL	Rotate Bits Left	v4 specifies number of bits to rotate
ROR	Rotate Bits Right	v4 specifies number of bits to rotate

If you want to perform a simple "NOT" operation (i.e. flipping bits from 0 to 1 and vice-versa), use the NAND operation, pairing $FF with every byte you want flipped and $00 with every byte you do *not* want flipped.

The SHL and SHR commands are similar to the ROL and ROR commands, except that the latter commands "recycle" the shifted bits to the other end of the data. In the case of SHL and SHR, on the other hand, bits shifted left or right are lost, with the "new" bits being set to 0.

The sample script ScrPSTMain provides many examples (with explanatory comments) of the CalcBinary command in action.

TextToBinary

```
Format          v1 = TextToBinary v2 v3
Example         RawIntegerSigned = TextToBinary 'I2S' -1234
Purpose         Returns the value encoded as the specified data type
Parameters      v1 = Variable being set
                v2 = Data Parse Conversion Code (see "Data Parse
```

```
                Conversion Codes", in the "Overview" section)
                v3 = The value being converted
Restrictions    Conversion to the BIN, BIC, HEX and R8$ data types is
                not supported
Similar Cmds    BinaryToText
Notes           Please see the "Overview" section for background
                details, and  the "BinaryToText" command for a
                discussion of how Data Parse manages to avoid requiring
                data types in scripts.
```

The TextToBinary command is the flip side of the BinaryToText command. You will typically use TextToBinary command if you are creating a raw binary output file which must contain "typed" data such as Signed Integer.

The sample script ScrPSTMain provides many examples (with explanatory comments) of the TextToBinary command.

Chapter 15

Reporting Commands

Overview

The log commands (such as LogMsg) send text to the log file, which is typically used to record non-critical information. If you have a critical message (such as a serious error), you should use the Stop command.

LogDb

```
Format          LogDb v1 [v2 v3 v4...]
Purpose         Same as LogMsg, but separates the fields with vertical
                bars
Parameters      Same as LogMsg
Similar Cmds    OutRuler
```

You can use the LogDb ("Log Debug") command while developing or fixing a script. The vertical bars let you see if the variables have spaces on either side. Once your script is working properly, you can do a quick search for "LogDb" to see if you left behind any debug lines.

LogMsg

```
Format          LogMsg v1 [v2 v3 ...]
Example         LogMsg 'Invalid value ' CustNum ' in customer number
                field'
Purpose         Sends a message to the log file
Parameters      v1 - Value to send to the log file
                v2 - Value (any number of values can be appended)
```

LogMsgLF

```
Format          LogMsgLF
Purpose         Sends a blank line to the log file. The blank line is
                ignored if there is already a blank line at that
                position. By using LogMsgLF instead of LogMsg(''), you
                can ensure that the log file does not contain multiple
                blank lines in a row.
```

ShowNote

Format	ShowNote v1 [v2 v3 v4...]
Example	ShowNote 'Processing database'
Purpose	Displays an informational message on the user interface window
Parameters	v1 - The informational message v2 - Value (any number of values can be appended)
Notes	To remove the message, set it to null: ShowNote ''

PlaySound

Format	PlaySound v1
Example	ShowNote 'c:\windows\media\ding.wav'
Purpose	Plays a sound file asynchronously
Parameters	v1 – Path and name of wav file
Notes	Wav file must use PCM encoding

Chapter 16

Flow Control Commands

Overview

Data Parse's flow control commands (such as If, Begin, End, Again, Stop) let you control the order in which the lines of your script are executed. You can, for example, execute a block of commands only under certain circumstances, or cause a group of commands to be executed repeatedly ("looping"). You can also define generalized procedures to save you having to duplicate code.

Again

```
Format          Again [v1 k2 v3]
Examples        See the Begin command
Purpose         Causes a Begin block to repeat if the comparison is
                true (or if no comparison is specified)
Parameters      v1 - Value to be compared
                k2 - Comparator
                v3 - Value to compare to v1
Restrictions    You cannot combine an Again command with an If command.
```

Begin

```
Format          Begin [v1 k2 v3]
Example         Begin MyVar = 'XYZ'  ; Execute block if MyVar equals
                'XYZ'
Purpose         Marks the start of a conditional block of script code
Parameters      v1 - Value to be compared
                k2 - Comparator
                v3 - Value to compare to v1
Defaults        If no comparison is specified, the block always begins.
                In such case, it makes no sense to have an Else
                command, and it almost invariably means that the block
                will end with an Again command.
Restrictions    You cannot combine a Begin command with an If command.
Similar Cmds    If
Notes           Comparisons are not case-sensitive, so 'CAT' = 'Cat'
                (unless you have altered the CompareCtrl setting).
                The Begin command does not set the $Success variable!
                Begin blocks can be nested up to 25 levels deep.
```

Here is an example of the Begin command, used with Else and End:

```
Begin MyVar = 'Cat'
  OutEnd 'The animal is feline'      ; Executed if MyVar = 'Cat'
  OutEnd 'In fact, it is a cat'      ; Executed if MyVar = 'Cat'
Else
  OutEnd 'The animal is not feline'  ; Executed if MyVar is not 'Cat'
End
```

Note the use of indentation. Indentation of the conditional code blocks is not mandatory, but it does make a complicated script much easier to understand. This is particularly important if a Begin block contains other Begin blocks:

```
Begin CustCode[1 3] = 'USA'
  OutEnd 'The customer is in the USA'
  Begin CustCode[4 5] = 'NY'
    OutEnd 'The customer is in New York'
  End
  Begin CustCode[4 5] = 'TX'
    OutEnd 'The customer is in Texas'
  End
End
```

Without the indentation, the logic of the code above would be hard to follow.

Here is an example of the Begin command used in a loop:

```
Counter = 0
Begin
  Counter = Counter+
  OutEnd 'The counter equals ' Counter
Again Counter #< 10
```

This would output the numbers from 1 to 10. You could also do it this way:

```
Counter = 0
Begin Counter #< 10
  Counter = Counter+
  OutEnd 'The counter equals ' Counter
Again
```

This would output the numbers from 1 to 10.

If you wish, you can put comparisons on both the Begin and Again. Both tests are repeated on every iteration of the loop.

Break

```
Format          Break
Example         If CustNum = MaxCustNum Break
Purpose         Breaks out of the current Begin/Again block, carrying
                on execution at the line following the next Again
                command
Similar Cmds    Continue
```

Call

```
Format          Call v1 [v2 v3 v4...]
Example         Call MyProcedure 'Hello!'  ; Pass 'Hello!' to
                MyProcedure
Purpose         Invoke a generalized section of script code passing
                information to and receiving results back from the
                Procedure
                v2 - Value (any number of values can be appended)
Defaults        If v2 is not specified, the procedure variable v1 is
                assigned a null value.
Restrictions    Calls from procedures into other procedures, which in
                turn call other procedures (and so on), can nest up to
                50 levels deep.
```

When you Call a procedure, execution of the script jumps to the first line of the procedure and continues until the corresponding End statement. The name of the procedure is also the variable name containing any parameters passed in v2, v3 and so on (the values are concatenated). Here is a sample script:

```
Call OutWithExclaim 'Hello, ' 'world' ; Call the procedure
OutEnd 'Glad you could join us!'      ; This line is run after the Call
Stop                                  ; Stop running script lines
Procedure OutWithExclaim              ; Start of the procedure
  OutWithExclaim = OutWithExclaim '!' ; Add an exclamation point
  OutEnd OutWithExclaim               ; Output
End                                   ; Return to the line after the
Call
```

This would output the string 'Hello, world!' then return to the line following the Call command.

Continue

```
Format          Continue
Example         If Status = 'Ignore' Continue
Purpose         Jumps ahead to the Again of the current Begin/Again
                block
Similar Cmds    Break
```

Done

```
Format          Done
Purpose         Skips the rest of the script (for the current record)
Similar Cmds    Stop, NextFile, NextStep
Notes           The Done command is usually used with the If command,
                or at the end of a Begin/End block.
```

Here is an example of the Done command:

```
If EmployeeNum <> 1234 Done
```

In this case, we are checking to see if the variable EmployeeNum is equal to 1234. If it is not, we skip the remainder of the current processing step.

Else

```
Format          Else
Example         See the Begin command
Purpose         Defines the start of the conditional code block that is
                executed if the Begin comparison is false.
Restrictions    You cannot combine an Else command with an If command.
```

End

```
Format          End
Examples        See the Begin command
Purpose         Marks the end of a Begin block
Restrictions    You cannot combine an End command with an If command.
```

Exit

```
Format          Exit
Purpose         Immediately returns from a Procedure
Restrictions    The Exit command can only be used inside a Procedure.
```

The Exit command is typically used in conjunction with a comparison. You do *not* need to include an Exit command in every Procedure; it is used to skip the rest of the procedure if some condition is met. For example:

```
Procedure AdjustPhoneNumber
  TrimChar PhoneNumber 'A '    ; Remove spaces
  Change PhoneNumber '/' '-'   ; Tidy up format
  Change PhoneNumber '.' '-'   ; Tidy up format
  AreaCode = PhoneNumber[1 3]
  If AreaCode = '416' Exit
  If AreaCode = '905' Exit
  PhoneNumber =  '1-' PhoneNumber
End
```

In this example, the procedure puts '1-' in front of a phone number unless it starts with 416 or 905.

If

```
Format          If v1 k2 v3 c4
Examples        If CustCode = 'AB12' OutEnd 'Mary Smith'
                If CustCode = 'CD34' CustAddr = '1234 Happy Lane'
Purpose         Conditionally performs a command
Parameters      v1 - Value to be compared
                k2 - Comparator
                v3 - Value to compare to v1
                c4 - Command
Restrictions    The If command may not be combined with a command that
                defines the start of a code block, such as Begin or
                FileInit.
Similar Cmds    Begin, Again
Notes           The comparison is case-insensitive, so 'CAT' = 'cat'
                unless you have altered the CompareCtrl setting.
                The If command does not set the $Success variable!
```

In deference to the ingrained training of seasoned programmers, you may use the word "then" after the comparison. Thus, the following command will be accepted:

```
If x > y then z = 'Hello'
```

This usage is non-standard, however, and is not recommended. The scripting engine treats the "then" as a variable, but ignores it in this context. Thus, you should never use a variable named "Then".

The If command does not have an "Else" option as in most programming languages. To execute a command when the If condition is false, use the Otherwise command. Alternatively, you can use the Begin command with an Else section.

Otherwise

```
Format          Otherwise c1
Example         If Animal = 'Cat' Type = 'Feline'  ; The initial If
                command
                Otherwise Type = 'Non-feline'      ; Action taken if
                false
Purpose         Executes an alternative command when the If comparison
                is false
Parameters      c1 - Command
Restrictions    The Otherwise command must follow immediately after an
                If.
                The Otherwise command may not be combined with a
                command that defines the start of a code block, such as
                Begin or FileInit.
Similar Cmds    Else
```

Procedure

```
Format          Procedure v1
Example         Procedure MyCode
Purpose         Defines the start of a generalized section of script
                code, which is terminated with the End command
Parameters      v1 - The name of the Procedure (must be a simple
                variable)
Restrictions    Recursive procedures (i.e. procedures that call
```

```
                themselves) are not formally supported and their use is
                not recommended.
Notes           See the Call command for additional details about
                procedures.
```

As the script is being run, any Procedure sections are ignored when encountered; they are only executed when explicitly invoked by Call. Procedures can go anywhere except within conditional blocks such as Begin/End, FileInit/End and so on. Procedures are usually placed together at the end of the script.

Stop

```
Format          Stop [v1]
Example         If CustNum[1] = 'X' Stop 'Invalid customer number'
Purpose         Terminates further processing
Parameters      v1 - Optional pop-up message
Similar Cmds    Done, NextStep
Notes           If v1 is included, a pop-up message is displayed. In
                such case, the Stop is considered an "abnormal" end of
                processing and the script-enabled application should
                proceed accordingly.
```

Step Control

Overview

A simple script runs from top to bottom each time a record is sent to it. But how can you initialize variables before processing starts? How can you output a grand total after all the records have been processed?

These issues and others are addressed by the step control commands.

When processing files, Data Parse performs a series of steps:

```
TaskInit   Executes before data is read from the  first  input file
FileInit   Executes before data is read from the current input file
Main       Executes once for each record sent to the script
FileDone   Executes after the last data is read from the current input
file
TaskDone   Executes after the last data is read from the  last   input
file
```

If you are only processing a single file (i.e. you are not using wildcards to process multiple input files), there is little to distinguish TaskInit and TaskDone from FileInit and FileDone.

Using Step Control

Except for the Main step, each step appears inside a conditional block, as in this example:

```
TaskInit                                          ; Start of the TaskInit step
  OutEnd 'Customer Count Report'                  ; Report header
  OutEnd '----------------------'                 ; Report header
End                                               ; End of the TaskInit step
FileInit                                          ; Start of the FileInit step
  OutEnd 'Input file: ' $ActualIFN                ; Output the file name
  NumInpFiles = NumInpFiles+                      ; Count this input file
End                                               ; End of the FileInit step
CustCount = CustCount+                            ; Main step: count record
TaskDone                                          ; Start of the TaskDone step
  OutNull                                         ; Output a blank line
  OutEnd 'Number of input files: ' NumInpFiles    ; Output statistics
  OutEnd 'Number of customers:   ' CustCount      ; Output statistics
End                                               ; End of the TaskDone step
```

In the example given above, the conditional code for the report header was placed in TaskInit so that the script will output it only once, even if you are processing multiple input files.

The conditional steps are optional. For example, you do not have to include FileInit in your script.

The conditional steps can appear almost anywhere in your script (though not within another conditional block).

FileInit and FileDone

The FileInit section is executed before each input file is processed. The FileDone section is executed after each input file is processed.

You cannot combine the FileInit or FileDone commands with the If command.

TaskInit and TaskDone

The TaskInit section is executed before data is read from the *first* input file. The TaskDone section is executed after the *last* record is read from the *last* input file and has been processed by the Main step.

You cannot combine the TaskInit or TaskDone commands with the If command.

NextStep

The NextStep command can be used to jump out of a step (such as FileInit or Main) and proceed to the next step.

For example, if your Main step has already located the information you are seeking, there is no reason to continue reading the input file. In such case, you can execute a NextStep command to ignore the rest of the input file and proceed immediately to FileDone, as in the following example.

```
CustNum  = $OutData[1 6]              ; Main step: Get customer number
PhoneNum = $OutData[60 70]            ; Main step: Get the phone number
If CustNum = '314159' NextStep        ; Main step: Found the customer?
FileDone                              ; Start of the FileDone step
  OutEnd 'Phone Number = ' PhoneNum   ; Output the information we sought
End                                   ; End of the FileDone step
```

NextStep should not be confused with the Stop command, which causes processing to cease entirely.

NextStep is also different from Done, which skips the rest of the script and then (if used in the Main step) proceeds to process the next record from the input file. The Done command can, however, be used within a conditional step block (such as FileInit) to skip the rest of that step; in such case it will behave the same way as NextStep.

NextFile

The NextFile command jumps out of the FileInit, Main or FileDone step without processing any of the remaining file-oriented steps. For example, if you execute NextFile in the FileInit step you will skip the Main and FileDone steps. (NextFile cannot be used in the TaskInit or TaskDone steps, since these steps are not dealing with a particular file.)

NextFile is used when an input file is rejected for some reason. It may have a serious formatting error, or (if you are using wildcards) it might not precisely match the kind of file name you are looking for.

If you are indeed using wildcards, NextFile will proceed to the FileInit step for the next input file. If your script is working on the last input file, NextFile will cause the script to move to the TaskDone step.

Here is an example of NextFile, as it might be used in the Main step:

```
Begin $Data[1 10] <> 'EMPLOYEE #'
  LogMsg $ActualIFN ' is not formatted correctly'
  HadError = 'Y'
  NextFile
End
```

In this case, the file did not contain the data we expected, so we log the error and move on to the next input file. In such case, it is a good idea to set a flag (HadError in this case) so that the TaskDone step can issue a warning:

```
TaskDone
  If HadError = 'Y' Stop 'One or more errors were detected. ' >>
    'Please consult the log file.'
End
```

Simply logging errors is no guarantee that the user will be aware that there was a problem, so we point out that the log does indeed contain some important information.

Chapter 17

Manual Read Commands

Overview

Data Parse reads a file from top to bottom and feed the input file data to the script one record at a time. In most cases there is no need for Data Parse to behave differently. However, occasionally a parsing challenge arises in which the script writer needs to go backwards and forwards in a file, or needs to read in new data according to varying criteria. The Manual Read commands address these requirements.

RecLenZero Scripts

Manual Read commands are essential is when your script is figuring out for itself how many characters to get for each record. In such case, your script must configure the input file as binary and specify a record length of zero. This is known as a RecLenZero script. Here is a sample script.

```
Config
  $CfgInpFileType = 'Binary'
  $CfgRecLen      = 0
End
$Data = ReadFor 100 'Relaxed'
OutEnd $Data
```

With a record length of zero, the Data Parse application will never read a single byte from the input file. Thus, the first line of the Main step in a RecLenZero script is typically a ReadFor or ReadUntil command. These commands and others are described below.

Using Manual Read for Standard Input File Types

Most Manual Read commands work in the standard input modes (such as TextCR) and one of them (ReadNext) does not do anything in a RecLenZero script (i.e. when `$CfgRecLen` is set to zero).

Bookmark

```
Format          Bookmark v1 v2
Example         Bookmark 'Save' 'MyBookmark'
Purpose         Remembers or returns to the current position in the
                input file
Parameters      v1 - 'Save' or 'Goto'
                v2 - The name of the bookmark
Similar Cmds    Rewind
Notes           The number of bookmarks you can save is limited only by
                your computer's memory.
```

ReadEOF

```
Format          ReadEOF
Example         TestEOF = ReadEOF
Purpose         Tests if the file pointer is positioned at the end of
                the  input file
Similar Cmds    The $EndOfData variable
Notes           Returns 'Y' if at end of file, 'N' otherwise.
```

Since ReadEOF is a function, it cannot be used in a comparison command such as If or Begin. You can use the special variable $EndOfData for that purpose, or you can save the value of ReadEOF in a variable for later use. Both methods are useful for determining if the input file contains any more data.

ReadFor

```
Format          v1 = ReadFor v2 [v3]
Example         MyVar = ReadFor 1000 'Relaxed'
Purpose         Reads the specified number of bytes from the input file
Parameters      v1 - Variable being set
                v2 - Number of bytes to read
                v3 - Control setting
Controls        Strict/Relaxed
Defaults        v3 = 'Strict'
Similar Cmds    ReadUntil, Rewind
Notes           ReadFor does not update Data or PrevData.
                If v2 is zero or negative, v1 is set to null.
                If v3 is 'Relaxed', no error message is generated if
                you attempt to read past the end of the file.
```

ReadNext

```
Format          ReadNext
Purpose         Moves to the next record in the input file
Similar Cmds    ReadUntil, ReadFor
```

The ReadNext command updates $Data with the next record from the input file. This is helpful if you know for certain what kind of data will be in the next record and wish to process it at the current point in the script.

ReadNext cannot be used in RecLenZero scripts, since when `$CfgRecLen` is set to zero Data Parse does not know how you are defining a "record". In such case you should use a command such as ReadUntil or ReadFor.

ReadUntil

```
Format          v1 = ReadUntil v2 [v3]
Example         MyData = ReadUntil #13#10 'Relaxed'
Purpose         Reads from the input file until the specified string is
                found
Parameters      v1 - Variable being set
                v2 - String to search for
                v3 = Control settings
Controls        Include/Exclude; Strict/Relaxed
Defaults        v3 = 'Exclude Strict'
Similar Cmds    ReadFor
Notes           In Include mode, the string being sought is included in
                v1.
                If v2 is null, the program will terminate with an error
                message.
                If v3 is 'Relaxed', no error message is generated if
                you attempt to read past the end of the file.
```

Rewind

```
Format          Rewind v1
Example         Rewind 100
Purpose         Moves the input file's pointer back by the specified
                number of bytes
Parameters      v1 - Number of bytes to move backwards (0 = start of
                file)
Similar Cmds    Bookmark, ReadFor
```

Rewind ignores the sign of v1, so 123 and -123 are treated the same way. If you wish to move *forward* in the file, use the ReadFor command.

Rewind resets the $EndOfData condition, but this needs to be done before the script ends or else you will move on to the FileDone step.

Chapter 18

The Config Section

Overview

The Config (short for "Configuration") section lets your script adjust how the underlying Data Parse application looks and behaves. You can, for example, alter the captions and hints on the optional input boxes.

Sample Script

By convention, the Config section appears at the beginning of your script. Here is a sample script:

```
Config
  $CfgEnableOptionX = 'N'
  $CfgEnableOptionY = 'N'
  $CfgEnableOptionZ = 'Y'
  $CfgCaptionZ      = '&CustNum'
  $CfgHintZ         = 'Enter the 5-digit customer number here'
End
If $OutData[1 5] <> $OptionZ Done
OutEnd $OutData
```

For the standard Data Parse user interface, this would disable the first two optional input boxes, leaving only the third one (known generically as OptionZ). It would be given the caption "CustNum", with a hotkey of Alt-C (as indicated by the ampersand preceding the C in '&CustNum').

Execution of the Config Section

The Config section is run when a script is loaded, and when you press F5. It is also run if the application notices that the script has been changed.

The Config section is run again when the script is run, just before the TaskInit step.

Whenever the Config section is run, the entire script is checked for syntax errors.

Commands Available in Config

Since the Config section deals with overall processing parameters, you should *not* use it to initialize variables — that should be done in the TaskInit step.

In most cases, you will simply assign values to $Cfg variables. In addition to this, though, you can use the following commands:

```
Begin, Else, End, If, Otherwise, Stop, NextStep
```

These let your Config section make certain decisions based on other factors (for example: whether or not $TestMode = 'Y'). You cannot read input (because there is none within the Config section), nor can you generate output.

The $Cfg Variables

The settings you make in the Config section are performed by assigning a value to one of the special variables starting with the characters $Cfg. These are described below.

Optional Input Boxes

The standard Data Parse interface has three combo boxes known generically as OptionX, OptionY and OptionZ.

You can alter the characteristics of these input boxes with the following $Cfg variables:

$CfgCaptionX, $CfgCaptionY and $CfgCaptionZ set the caption. You can include an ampersand in the value to define a hotkey. For example:

$CfgCaptionY = '**&**PhoneNum'

This will alter the caption for the OptionY input box to "PhoneNum", with a hotkey of Alt-P. You should test your script to ensure that the hotkey is not already used by another control, and that the caption fits in the space provided.

$CfgEnableOptionX, $CfgEnableOptionY and $CfgEnableOptionZ turn on or off the optional input boxes. If an input box is turned off, it will be "greyed-out" and will contain the string "(Not used by this application)". For example:

$CfgEnableOptionX = 'N'

$CfgEnableOptionY = 'Y'

$CfgEnableOptionZ = 'N'

This would turn off all optional input boxes except OptionY.

$CfgHintX, $CfgHintY and $CfgHintZ provide a "hover hint". This is a short phrase that appears when the user pauses over the input box with the mouse cursor.

File Names

The standard Data Parse interface has an input box for the Input File name and one for the Output File name. Both of these have default values, which are set by the following variables:

```
$CfgDefaultIFN   Default input file name
$CfgDefaultOFN   Default output file name
```

If you clear (i.e. leave empty) the Input File input box and then exit it (e.g. by pressing Tab), the program fills in the input file name `ThingsToDo.txt` — one of the sample files included in the **Data Parse** package.

You can change these defaults with $CfgDefaultIFN and $CfgDefaultOFN.

Note, however, that when a script is loaded these default names do not automatically override the file names already in the input boxes. These $Cfg variables simply provide the end user with a quick way to enter a commonly-used file name. If the default file name is quite long (for example, if it is located in a sub-sub-sub-directory), this can save the end user a lot of typing.

Two special file names are recognized by Data Parse: `Clipboard` and `None`. `Clipboard` takes input from (or sends output to) the Windows text clipboard. `None` means precisely what its name implies: if you take input from `None`, you'll get no data (except the word "None"); if you send output to `None`, it disappears.

Filename may also be a URL, such 'http://yourdomain/index.html' or 'ftp://yourdomain/file.zip'

File Formats

The format of the input and output files can be altered from the default setting (plain text) with the following $Cfg variables:

```
$CfgInpFileType   Input file format (examples: 'Text', 'Binary',
                    'Delimited', 'HTMLDelimited')
$CfgOutFileType   Output file format
$CfgRecLen        Record length for Binary files
$CfgDelimiter     Record-ending delimiter character for Delimited files
```

These settings are described below.

INPUT FILE FORMAT

If you do not specify a setting for $CfgInpFileType, it is generally assumed to be 'Text' (unless the underlying Data Parse application has a different default).

The Text type can read standard Windows-style text files (i.e. each line ends with the carriage return and linefeed characters: decimal #13#10; hex $0D$0A) or Unix-style text files (where each line ends with the linefeed character).

Here are the supported values for $CfgInputFileType:

```
'Text'          Windows-style or Unix-style text files
'TextLF'        Unix-style text files only
'TextCR'        Macintosh-style text files
'Delimited'     Records terminated with a specific character
'Binary'        Fixed-record-length file or RecLenZero script
'HTMLDelimited' HTML and/or XML Files
```

These file types are described below.

Text Files

The three text file formats (Text, TextLF and TextCR) try to deal gracefully with a certain amount of variation. For example, TextCR will ignore any linefeed characters, while TextLF will ignore any carriage return characters. If for some reason you wish to retain these characters, you can use the Delimited file format (described below).

Delimited Files

If you set $CfgInpFileType to 'Delimited', you must also specify the delimiter character that ends each record (with the possible exception of the last one). For example, you could process Macintosh-style text files by using the following technique instead of the TextCR format:

```
Config
  $CfgInpFileType = 'Delimited'
  $CfgDelimiter   = $0D
End
```

This will read records that end with a carriage return character. The delimiter character is not included in the result.

Multi-character delimiters are not supported, but in most cases you can simply parse out the excess characters. For example, if you read a standard Windows-style text file as a Delimited type, looking only for the linefeed ($0A), each record would have a spurious carriage return ($0D) at the end which is easily removed with the TrimChar command.

HTML/HTTPS/FTP for Input files

In HTMLDelimited mode, each record is delimited as an html or xml element.

$CfgInpFileType = 'HTMLDelimited'

This delimited feature allows you to more easily step through an HTML file.

HTMLDelimited iterates through the HTML file, but rather than defining a line as one ending in CRLF, it would consider each HTML/XML element as a line. So if the page contained:

```
<pre><b>this is

simple</b> text that I have written.</pre>
```

Each record would be:

```
<pre>

<b>

this is simple

</b>

text that I have written.

</pre>
```

In the solution explorer, you may add Add Url as Input File, by right-mouse clicking on the Input Files node. This also allows you to add website URLs as input files. HTTP, HTTPS and FTP, amongst others, are support protocols.

Binary Files

If you set $CfgInpFileType to 'Binary', you must also specify a record length via the $CfgRecLen variable.

A value of 0 (zero) denotes a RecLenZero script: your script will handle all reading with commands such as Bookmark, ReadFor, ReadNext, ReadUntil and so on.

A positive integer value means that you are reading records of fixed length. In a fixed-record-length file, all records (with the possible exception of the last one) are exactly as many bytes as you specify in $CfgRecLen. For example:

```
Config
  $CfgInpFileType = 'Binary'
  $CfgRecLen      = 80
End
```

This will read records that are 80 characters long. In principle you can read records that are several billion characters long, though in practise this might create memory issues.

You should never set $CfgRecLen to a negative number as this currently has no meaning to Data Parse.

OUTPUT FILE FORMAT

Since scripts can control output precisely (using the Output command), your output file can adopt any format you wish. Thus, the $CfgOutFileType variable is used for documentation purposes only. For example, it is displayed when you view a Help file for a script.

For the sake of consistency the value of $CfgOutFileType is checked against a list of permissible file types (Text, TextLF, Delimited and Binary). If you are outputting a proprietary format (such as might be natively supported by a database or spreadsheet), it is best to set $CfgOutFileType to 'Binary'.

Documentation

When you create a script, it is a good idea to also create a Help file to go with it. Data Parse recognizes that a Help file is present when a file exists with the same name as the script, but with the string "`Help-`" in front of the name. Thus, if you created a script named:

```
FixData.pscr
```

then the corresponding Help file would be named:

```
Help-ScrFixData.txt
```

Once you've prepared the Help file, you can then set the following values in your script's Config section:

Variable Name	Explanation
$CfgCopyright	Copyright notice (e.g. 'Copyright (C) 2008 by WhizzCo')
$CfgVersion	The version of the script (e.g. '1.00.00');
$CfgProgrammer	The name of the primary programmer of the script
$CfgEmail	Email address to contact the people who wrote the script
$CfgLicense	Terms of use — you can append several strings with the continuation convention (the >> characters) to create a multi-line explanation.

When the Help file is displayed by the application, these items will be added to the end (provided you assigned them a value).

ODBC Support (Read/Write)

You can read and write from a database that you have access to, as long as it supports simple ODBC connectivity.

Use the $CfgODBCConnection variable to set your connection.

Remember that you will need to match the connection you set in your script file, with the connection you created with your ODBC Connection Manager found in your Windows Administration folder, off of your Control Panel.

All connections to your database, via use of the SendToDB script command will use the information you supplied in the $CfgODBCConnection variable.

Dealing with Web-based Data

Sometimes it's necessary to not only retrieve data from a website, but also to perform a POST back to it. Adding to the complication can be the requirement to use cookies. Data Parse has added the PostToURL command to help deal with these situations.

PostToURL v1 [v2] v3 v4 v5 v6 v7 v8

```
Example PostToURL 'http://www.parseomatic.com' 'c:\returneddata.htm'
                ReturnedData 'value=1' '' '' ReturnedHeader Indicator
                Exception

Purpose Posts request to URL by POST method

Parameters
v1 - URL to initiate POST to
v2 - Filename where data returned from the remote site will be saved
                (optional)
v3 - Variable to store up to 32,000 bytes from the returned data
v4 - Variable that contains the data to send as a POST to the URL at v1
v5 - Cookie variable container
v6 - Returned Header
                (http://en.wikipedia.org/wiki/List_of_HTTP_headers)
v7 - Indicates all data received.  Always has a value of 2 until future
                feature is implemented (future versions to support
                async requests, where variable assigned to v7 will stay
                = 1 until all data is received)
v8 - Handled Exception Code
                (http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
                ) and in description. Any of unhandled exception will
                stop script running

Description of v8 codes:
600 - Url is empty
601 - Url has invalid format
602 - Request type is not http/https
700 - Incorrect file name
701 - File already exists
```

Chapter 19

Command Prompt & Unattended Operation

Command Line Parameters

Data Parse Business and Enterprise Editions support launching Data Parse with command-line parameters.

This can be useful if you wish to more easily launch a solution for a user via a shortcut or allowing for unattended operation.

To call Data Parse from the command line (e.g. in a batch file, a Windows shortcut, or a task scheduler), the following format is used to specify the input and output files:

```
DP.EXE /IFN="Input.txt" /OFN="Output.txt"
```

You can also specify the contents of the three option boxes:

```
/OPX="OptionX data goes here"
/OPY="OptionY data goes here"
/OPZ="OptionZ data goes here"
```

To specify a script file, use /SFN= as in this example:

```
/SFN="Sample01.pscr"
```

For a general overview of command line parameters, start up Data Parse as follows:

```
DP /?
```

This displays a window which summarizes the command-line options, including the parameters required to start parsing automatically (/RUN) and control program termination (e.g. /CLS). The window is also displayed if your command line contains an option that Data Parse does not recognize.

Full List of Command-Line Switches:

```
/SOL="Solution File Name"          Ignored by deployables
/CMD="Command Line Filename"
/SFN="Script File Name"
/IFN="Input File Name"
/OFN="Output File Name"
/SFN="Support File Name"
/LFN="Log File Name"
/HFN="Help File Name"
/RUN="y|N"                         Click Start button?
/DAP="y|N"                         Display after processing?
/APP="y|N"                         Append to output file?
/TST="y|N"                         Test mode?
/CLS="y|N|a"                              Close after processing?
/OPX="value"                       Option X
/OPY="value"                       Option Y
/OPZ="value"                       Option Z
```

Format of a Command Line File

A command-line file allows the specification of parameters for every Project in a Solution. The format is as follows:

```
; This is a comment
PROJECT=<Project Name>
Parameter String
Parameter String
Parameter String
PROJECT=<Project Name>
Parameter String
Parameter String
Parameter String
```

If a Project is not found in the command-line file, the values from the ppro file are used. If a project is found, but one of the settings is missing, we use the setting from the ppro file.

The /SOL parameter is ignored by deployables, even if it is found in a command-line file.

Any command-line switch can also be used on the command line directly. If that is done, it applies to the first Project only. If the command-line contains /? then a Help window is displayed, and all other switches are ignored.

/CLS="A" means "Close after processing always, even if there was an error". In this mode, pop-up error messages are suppressed.

/TST="Y" sets the $TestMode special variable to 'Y'.

Batch Files

Introduction

When calling Data Parse from a batch file, you must use the Windows START command with the /WAIT option so that Data Parse can complete processing before execution moves to the next line in the batch file.

If the batch file is running unattended, you should also feed Data Parse the following parameters:

```
/RUN      Run (i.e. start) processing immediately
/CLS      Close the program after execution, even if there is an error
```

Thus, a batch file line that calls Data Parse would contain the items exemplified below (line breaks and comments inserted for clarity only):

```
START                                                 The Windows START command
  ""                                                  Blank title for the window
  /WAIT                                               Await completion
  "C:\Program Files\Data Parse\DP.exe"                Invoke the program
  /IFN="C:\My Input\Inputfile.dat"                    Input file or wildcard mask
  /OFN="C:\My Output\Output.txt"                      Output file
  /SOL="C:\Program Files\Data Parse\Solutions\MySolution.psol"
  /RUN="Y"                                            Start processing
  /CLS="Y"                                            End afterwards
```

Note the use of quotes — these are mandatory if a parameter contains a space.

Please note that the above example is broken up onto different lines. Below is how it would actually would look like if you opened your batch file in Notepad with WordWrap set to True.

```
START "" /WAIT "C:\Program Files\Data Parse\DP.exe" /IFN="C:\My
Input\Inputfile.dat" /OFN="C:\My Output\Output.txt" /SOL="C:\Program
Files\Data Parse\Solutions\MySolution.psol" /RUN="Y" /CLS="Y"
```

The Error Reporting File

If a serious error occurs during processing, Data Parse creates a file named `POMPT-Error.txt` in the same directory as the Solution file. The file is plain text and contains information about the error. You can view the Error Reporting File using the Support Files input box of the Parsing Parameters window; it will be listed in the drop-down list.

If no error occurs, the file is *not* present after processing is complete.

If you are using Data Parse in a batch file, you can check to see if processing worked by using the IF EXIST test, as in this example:

```
@ECHO OFF
C:
CD "\Program Files\Data Parse\"
START "" /WAIT "C:\Program Files (x86)\Data Parse\DP.exe"
/SOL="ProcessData.psol" /OPX="" /OPY="" /OPZ="" /RUN="Y" /CLS="Y"
IF EXIST POMPT-Error.txt GOTO ERROR
GOTO OKAY
:ERROR
ECHO An error occurred!
GOTO DONE
:OKAY
ECHO Everything was fine!
:DONE
ECHO Processing completed
```

Note that the /CA parameter suppresses pop-up error messages, so if you use it in your batch file, it is up to your batch file to watch for the error file and then determine what to do if an error (such as "File not found") occurs.

The Log File

In addition to the Error Reporting File, Data Parse also creates a log file (named POMPT-Log.txt). Data Parse uses the log file to record the date and time when processing started and ended. It also uses the log file to report anything that is slightly unusual but not a serious problem.

You can view the Log File using the Support Files input box of the Parsing Parameters window; it will be listed in the drop-down list.

Unattended Operation

If you require processing without human intervention, you can set up the Windows Task Scheduler to run an appropriate batch file periodically.

The batch file can check to see if a particular input file (or a particular file wildcard) exists in a particular folder. If so, the batch file would then invoke the parsing application. After a successful run, the batch file would either move or rename the input file. (Deleting the input file is not recommended, unless you have another copy elsewhere.)

Here is an example of an appropriate batch file, which invokes Data Parse.

```
  @ECHO OFF
  IF NOT EXIST "C:\MyInput\*.dat" GOTO QUIT
  ECHO Start of processing
  C:
  CD "\Program Files\Data Parse"
  START "" /WAIT "C:\Program Files (x86)\Data Parse\DP.exe"
/SOL="ProcessData.psol" /OPX="" /OPY="" /OPZ="" /RUN="Y" /CLS="Y"
  IF EXIST POMPT-Error.txt GOTO ERROR
  CD "\MyInput"
  RENAME "*.dat" "*.old"
  GOTO DONE
:ERROR
  ECHO An error occurred!
  PAUSE
  GOTO QUIT
:DONE
  ECHO Processing completed
:QUIT
```

Please note that you may need to replace " C:\Program Files (x86)\Data Parse\" with the path to your installed version of Data Parse.

In order for this technique to work reliably, the batch file must be called with a greater frequency than an input file is likely to appear. For example, if a new input file can show up in as little as 20 minutes, it would be a good idea to call the batch file every 15 minutes. If you do not take this precaution, it is possible that an input file will show up just as you finish parsing, which means it would get renamed and would not be processed.

For this reason, it is not feasible to process input files that arrive every few seconds, unless you have an exceptionally fast machine that does not experience unexpected delays (such as automatic updates of the operating system, people accessing its hard disk from the network, and so on).

If the batch file is running in a *very* unattended fashion (i.e. it handles countless arrivals of new files, but people rarely check the machine), you should not include the `PAUSE` command in the batch file, as this could cause the screen to fill up with open windows.

Multi-User Operation

Technical Issues

Data Parse is designed primarily for use in a *single-user* environment. Problems can arise if multiple users attempt to use the same copy. Data Parse script applications do not explicitly detect multi-user "collisions".

When using Data Parse in a multi-user environment, each user should have their own copy. Ideally, each copy should be located on the user's local machine.

Chapter 20

License & Legal Issues

Free and Basic Editions

National Data Parsing Corporation licenses the Data Parse Free Edition and Data Parse Basic according to "concurrent usage" rather than by machine or by person. Thus, if you have a "single concurrent user license" (sometimes referred to simply as a "single user license") you can install a copy of the product on your machine at work, and yet another on your laptop that you use at home (depending on your own company's internal policies, of course). You can use the same registration code on both copies.

You must be able to ensure that only one installed copy can be in use at any one time. If this cannot be guaranteed, you must purchase additional licenses.

Business and Enterprise

The licensing of Data Parse Business and Enterprise are on a single-user basis. That is to say that a separate license is required for each Data Parse that is installed on a PC instance. For example, if you need to install Data Parse on six PCs, or six Virtual Machines, then you need to purchase six licenses. Site licenses and company-wide licenses are available. Please contact a sales representative for more information, or our website at http://www.DataParse.com

Scripts

Any scripts and accompanying files you write belong to you (or, in some cases, your company). You do not need our permission to distribute them.

You cannot, however, distribute the supporting Data Parse application unless you have purchased a distributor license from us. **Data Parse Free Edition** is available in a freeware version, but some others (such as custom-designed parsing applications) may not be distributed without prior written permission from National Data Parsing Corporation

Deployables™

Deployables created as part of Data Parse Enterprise Edition may only be distributed internally to the license holder. If you need to deploy stand-alone Data Parse solutions to other companies or to customers, you must purchase additional licenses from National Data Parsing Corporation

Deployables can be distributed to multiple PC's within your organization. This can greatly reduce the number of licenses required since a single Data Parse Enterprise Edition can create stand-alone deployables for hundreds of your internal users.

Chapter 21

Security

Encryption

Overview

Scripts can be protected from alteration and execution by "encrypting" them. In this form they cannot be viewed from within any Data Parse application, unless the proper password is entered.

Also, scripts that have been encrypted will only run on the installed instance of Data Parse that encrypted them.

An encrypted script can be loaded into a text editor, but it will look like random characters. Alteration of even one of the characters will typically result in a script that either does not compile, or does not function correctly.

Limitations

Encrypting is not designed to protect confidential data. The scrambling algorithm is sufficiently complex that most people will not be able to decode the file. However, one person is 10,000 certainly has the skills to do this. Such wizards can usually solve this kind of puzzle in under an hour.

Encrypting cannot prevent the duplication of the essential functions of a script. By deliberately introducing errors, the end-user could gradually gain knowledge of the contents of the script. This approach is, of course, quite labor-intensive; it would probably be easier to rewrite the script from scratch.

Only scripts can be protected by encryption. Encrypting is not implemented for files accessed via the LookupFile or SetFromFile commands.

Encrypting a Script

To scramble a script, right-mouse click the script in the Solution Explorer and select the Encrypt option. You will be prompted for an encryption code, which must be at least 6 characters long, and is case sensitive.

After encrypting, a copy of the original, unscrambled script can be found in a file with the same root name, but with a `.bak` extension. Thus, if you scramble a script named `MyScript.pscr` the backup copy will be available in the `MyScript.bak` file. If the end-user is using your machine, it may be appropriate to delete the `.bak` file.

Turning off Encryption

To no longer have your script file encrypted, Right-Mouse Click the script in your Solution Explorer. You will be prompted for the encryption code.

If the encryption code is correct, the script is no longer encrypted. If you type the encryption code incorrectly, you can try again – up to 50 times. If, after 50 attempts, you still have not entered the correct code, you must close down the program and start it up again.

Security Analysis

A relatively unsophisticated keyboard-and-mouse macro routine could try out about 150 encryption codes per minute. Thus, if your encryption code is 6 characters long and contains only lowercase letters, then the average time to obtain the scrambling code can be calculated as follows:

26 ^ 6 / 150 / 525600 / 2 = 1.96 years

That is to say:

NumberOfPossibleCharacters ^ CharactersInCode / CodesPerMinute / MinutesInAYear / 2

This assumes that the person knows the number of characters in the code and the number of possible characters that it uses. But even with these advantages this is not a feasible technique for obtaining the encryption code.

There are, however, more sophisticated approaches. A highly skilled computer expert could probably obtain the scrambling code within an hour or so. Of course, somebody with that kind of ability would be able to write their own script with much less effort!

Index