MMPz

FOR
MINIMUM MISCIBILITY
PRESSURES AND COMPOSITIONS

VERSION 1.3
NOVEMBER, 2002

ZICK TECHNOLOGIES
PETROLEUM ENGINEERING
CONSULTING & SOFTWARE

5816 SW HUDDLESON STREET • PORTLAND, OR 97219-6627
PHONE: 503-768-9386 • FAX: 503-768-4824 • E-MAIL: ZICK@ZICKTECH.COM

Contents

Chapter 1. Introduction

MMPz is a program for calculating the minimum miscibility pressure (MMP) or minimum miscibility enrichment (MME) of any petroleum fluid system described by a cubic equation of state (EOS). It is designed to predict a good approximation of the true, thermodynamic MMP or MME regardless of whether the displacement mechanism is that of a condensing gas drive (CGD), a vaporizing gas drive (VGD),[1] or a condensing/vaporizing (C/V) gas drive.[2] It is assumed here that the reader is already familiar with the concepts of these displacement mechanisms, developed miscibility in general, and minimum miscibility conditions in particular.[3]

The universally accepted method for predicting an MMP or MME with an EOS is to simulate a series of one-dimensional slim tube displacements, looking for the pressure or solvent enrichment level where the recovery (after 1 to 1.2 pore volumes of injection) first reaches nearly 100%. Unfortunately, such simulations are very time-consuming and subject to numerous interpretation errors.

MMPz utilizes proprietary techniques to predict an MMP or MME by capturing the compositional mechanisms of a true displacement process, but with far greater efficiency, accuracy, and consistency than can be achieved through slim tube simulations. Essentially, through a simulated multistage, multicontact experiment, **MMPz** estimates the compositional path taken during an actual displacement. It looks for conditions of pressure or solvent enrichment under which the path will develop miscibility. Acceleration techniques are used to speed convergence toward the ultimate path, and extrapolation techniques are used to approximate that path through a finite number of stages and contacts. The end result is an accurate (within 1 or 2%, typically), consistent (i.e., differentiable) estimate of the thermodynamic MMP or MME with a minimal amount of computation.

This manual describes how to use **MMPz**. The next three chapters—"Concepts," "Conventions," and "Building an Input File"—should be studied carefully. The rest of the manual is more for reference and example. It should initially be skimmed for an overview of **MMPz**'s capabilities and later referred to as needed.

---

[1] Stalkup, F.I., Jr.: *Miscible Displacement*, Monograph Series, SPE, Richardson, Texas (1984).

[2] Zick, A.A.: "A Combined Condensing/Vaporizing Mechanism in the Displacement of Oil by Enriched Gases," paper SPE 15493 presented at the 1986 SPE Annual Technical Conference and Exhibition, New Orleans, October 5–8.

[3] Whitson, C.H., and Brulé, M.R.: *Phase Behavior,* Monograph Series, SPE, Richardson, Texas (2000), 121-141.

Chapter 2. Concepts

Think of **MMPz** as a virtual phase behavior laboratory. At one end of the lab is a storage area for fluid tanks. Some of these tanks are initially filled with single-component fluids. The rest are initially empty, but can be filled at any time with mixtures of fluids sampled from non-empty tanks. At the other end of the lab is an apparatus that can be used as a simple PVT (pressure-volume-temperature) cell for equilibrating fluids, or as a virtual, multistage mixing device for multicontact miscibility experiments. This apparatus can be set to operate at any temperature and pressure. It is also connected to a few special fluid tanks. Some of these tanks feed fluids into the apparatus before or during each experiment and can be filled with any desired mixture of fluids from the other tanks within the lab. Other tanks capture fluids from the apparatus during or at the end of each experiment. Samples from these tanks can be transferred to other tanks for later use. Finally, **MMPz** acts as the capable lab technician, performing any sequence of instructions to mix fluids, run experiments, and record results.

MMPz's user is the director of this virtual laboratory. He or she creates a set of instructions for the virtual technician (**MMPz**) to follow. This usually begins with a list of the fluid components that are available, then proceeds with instructions for creating mixtures of those components and storing them in labeled tanks, which, in turn, can be used to create other mixtures, including those in the experimental apparatus's input tanks. The temperature and/or pressure of the experimental apparatus can be changed at any time and sequences of PVT and/or miscibility experiments can be performed in any desired order. The fluids generated by any experiment can be saved, mixed with other fluids, and/or used as input for subsequent experiments. In this manner, **MMPz**'s user can be very creative about the results that are generated and reported. The more familiar the user is with the instructions understood by **MMPz**, the more creative he or she can be.

The user enters the list of instructions for **MMPz** into a text-only input file. This file can be created, edited, or viewed in a text editor, a word processor, or even a spreadsheet program. It consists of a sequence of free-format, keyword-activated commands and their associated data. This sequence of commands can be of practically any length. The commands can be issued in practically any order, and are always executed in that order. Thus, the input file acts as a script for **MMPz** to follow.

In turn, **MMPz** generates text-only output files. Again, these can be opened with a text editor, a word processor, or even a spreadsheet program (which can make it particularly convenient to manipulate or plot the results).

The next chapter discusses two important types of conventions—those followed by **MMPz** when it reads an input file, and those followed by this manual. The easiest way to learn how to use **MMPz** is to learn these conventions first.

Chapter 3. Conventions

When **MMPz** reads an input file, it distinguishes between several different types of input: *comments*, *tokens*, *numbers*, *character strings*, *delimiters*, *keywords*, *commands*, and *tables*. All must follow certain conventions so **MMPz** can recognize them. This manual will also follow certain conventions to help the reader recognize them. These conventions will be described in the following sections.

3.1. Comments

When building an input file, it is always a good idea to include plenty of *comments*, so the file can be understood more easily at a later date and by others. **MMPz** allows two different types of comments: the *comment line* and the *trailing comment*.

3.1.1. Comment Lines

Comment lines are comments that occupy an entire line within an input file. No matter where a comment line is located (even if it's in the middle of a table or some other collection of data), **MMPz** will behave as though that line was not in the file at all. The following section of an input file lists all of the ways to insert a comment line.

```
# A pound sign in column 1 indicates a comment line.
! An exclamation point in column 1 indicates a comment line.
: A colon in column 1 indicates a comment line.
; A semicolon in column 1 indicates a comment line.
 ; Column 1 contains a blank, so this is not a comment line.

* The previous line was blank, but not a comment line.
-- Hyphens in columns 1 and 2 indicate a comment line.
== Equal signs in columns 1 and 2 indicate a comment line.
```

This also illustrates one of the manual's conventions: examples that show portions of an input file will be displayed in a mono-spaced typeface, surrounded by a box.

3.1.2. Trailing Comments

Any line (except a comment line) can be ended with a trailing comment, which consists of a semicolon (";") and anything else on the rest of the line. Here are some examples:

```
1 2 3 ; This line has three numbers and a trailing comment.
; This is a comment line, which was described earlier.
; Comment lines are treated as though they didn't exist.
 ; This is different: a blank line with a trailing comment.
4 5 6 ; Another three numbers and another trailing comment.
```

Since the comments are ignored, the five-line sequence above is treated exactly the same as the following three-line sequence:

```
1 2 3

4 5 6
```

3.2. Tokens

MMPz will spend most of its time, on input, looking for *tokens*. Tokens can be keywords, user-defined names (for components, mixtures, etc.), or numbers. It would not be a bad idea to know what constitutes a token, as far as **MMPz** is concerned. Table 3–1 lists all of the possible token characters, some of which can only be used after the first character. No other character would ever be recognized as part of a token.

Table 3–1. Token Characters

Allowable Token Characters	Allowed for First Character?
Any Alphanumeric Character	Yes
+-_~%.?	Yes
*#/	No

Note: tokens are always **case-insensitive**. The user may mix uppercase and lowercase characters at will. **MMPz** will keep each token just as the user input it, but whenever tokens are compared, case will be ignored. For example, the token "Åbç" would be considered identical to the token "åBÇ". Also note that a token's alphanumeric characters are not restricted to English language characters. From now on, tokens will be shown in a mono-spaced typeface.

3.3. Numbers

Numbers, when required, can be entered in integer format (`1, 2, -3`), floating point format (`1.0, 2., -3.0`), or in any of the common scientific formats (`0.1e1, 20E-01, -30.0d-001`). Note that the scientific exponent can be preceded by `"e"`, `"E"`, `"d"`, or `"D"` and possibly a plus or minus sign. A number cannot contain any extraneous characters and cannot be immediately followed by any extraneous token character except a slash (`"/"`). Note that all legal numbers are also tokens, but not all tokens are numbers (for example, the token `"3E"` is not a number, but `"3E0"` is).

If the same number is to be entered numerous times, a repeat factor can be used. The proper syntax is a positive integer (for the number of repetitions), immediately followed by an asterisk (`"*"`), immediately followed by the number to be repeated

(with no intervening blanks). A repeat factor can also be used to enter default values of a number. In that case, the syntax is a positive integer (for the number of repetitions), immediately followed by an asterisk (`"*"`), immediately followed by a blank, the end of the line, or any non-alphanumeric character other than `"+"`, `"-"`, `"."`, or `"*"`. Repeat factor examples are shown in the following:

```
3*4.2 ; equivalent to 4.2 4.2 4.2
1*-2  ; equivalent to -2
2*.0  ; equivalent to .0 .0
1*    ; equivalent to one default entry
3*  1 ; equivalent to three default entries followed by a 1
4*, 2 ; equivalent to four default entries followed by a 2
```

3.4. Character Strings

In some cases, **MMPz** will allow the input of *character strings*. Examples include titles, notes, and file names. Character strings may contain any characters. They should usually be quoted with a matching pair of single or double quotes.

Character strings can also be tab-delimited. If tab-delimited, the tabs take precedence over any quotes, but the resulting string will also be stripped of any outer matching double quotes and then any outer matching single quotes, in that order. The reason for these strange rules is to allow input files to be created and edited with the spreadsheet program, Microsoft Excel, which follows even stranger rules. Excel will always tab-delimit its cells, but it may also (sometimes) add double quotes, even if the cell was initially single-quoted.

If a character string is simple (i.e., if it begins like a token and contains no blanks), then it can be entered without quotes or delimiting tabs.

If a string is quoted, but it needs the same type of quote within the string, then the embedded quote should be entered twice in a row.

In the following examples, the character `"»"` represents the tab character (which would otherwise be invisible) and the actual, recognized string is shown in a trailing comment (from the first non-blank character after the semicolon to the last, including any quote marks).

```
Title "A through Z"       ;  A through Z
Title 'A through Z'       ;  A through Z
Title »A through Z»       ;  A through Z
Title »"A through Z"»     ;  A through Z
Title »'A through Z'»     ;  A through Z
Title »"'A through Z'"»   ;  A through Z
Title »'"A through Z"'»   ;  "A through Z"
Title »"A»through»Z"»     ;  "A
Title "'A through Z'"     ;  'A through Z'
Title '"A through Z"'     ;  "A through Z"
Title """A through Z"""   ;  "A through Z"
Title '''A through Z'''   ;  'A through Z'
Title  A through Z        ;  A
Title  A-through-Z        ;  A-through-Z
```

Note that if **MMPz** is looking for a token rather than a character string, then quotes will make no difference (i.e., any quotes will be treated just like blanks). It is impossible to define a token that contains anything but the allowable token characters.

Also note that character strings are always **case-insensitive**. The user may mix uppercase and lowercase characters at will. **MMPz** will keep each character string just as the user input it, but if character strings are ever compared, case will be ignored. For example, “the time is now” would be considered identical to “The Time is NOW”.

3.5. Delimiters

Delimiters are characters that separate tokens, numbers, and character strings from each other. The most common delimiter is the blank character. The end of each line is also a delimiter. Most of the punctuation characters (except those listed as token characters in Table 3–1) also behave as delimiters, usually just like blanks, but often with better readability. Here are just a few examples:

```
Title = "A through Z"  ; same as: Title "A through Z"
Mix C: A & B           ; same as: Mix C A B
Pressure ==> 1 (atm)   ; same as: Pressure 1 atm
```

There are a few special delimiters, however. The slash (`"/"`) is one of those. The slash can be part of a token (e.g., `"G/CC"`). Otherwise, however, a slash ends **MMPz**’s current search for input and causes the rest of the current line to be ignored. This is for compatibility with input that may have originally been designed for FORTRAN programs. This use of the slash, however, is never necessary and therefore discouraged.

The comma (`","`) is another special delimiter. The comma ordinarily behaves just like a blank. However, if **MMPz** is searching for some type of input, and it encounters two commas in a row (even if separated by blanks or other delimiters), then it will use the

default value for that input. This is also the case if **MMPz** encounters a slash following a comma. Again, this type of behavior is for compatibility with the rules for FORTRAN input.

In addition, for aesthetic reasons, the above rule has been extended to apply if **MMPz** encounters a comma following an equal sign (`"="`), a colon (`":"`), an *at* sign (`"@"`), an ampersand (`"&"`), a vertical bar (`"|"`), or the beginning of a new line (with or without other delimiters in between). A comma after any of these characters makes it look as though something has been omitted (e.g., `"A = ,"`), much the same as with two consecutive commas (e.g., `"A,,"`). Therefore, this omission, like one between commas, will also represent a default value.

Here are some examples on the use of slashes and commas:

```
Mix A 1 2  / assigns 2 (of possibly many) numbers
Mix A, , 2 / assigns 2 numbers, the 1st of which is defaulted
Mix A = 1, / assigns 2 numbers, the 2nd of which is defaulted
Mix A = ,  / assigns 2 numbers, both of which are defaulted
Mix A
,          / assigns 2 numbers, both of which are defaulted
Mix A: ,
,          / assigns 3 numbers, all of which are defaulted
```

It is highly recommended, however, that consecutive commas, or commas following other punctuation characters, be avoided unless the user is very certain of the rules that will apply.

3.6. Keywords

Keywords are tokens that have special meaning to **MMPz**. They may trigger commands, set options, or comprise the headers in a table, for example. There may, in fact, be a number of tokens that will be recognized as the same keyword. That's because many keywords allow *abbreviations* and/or *aliases*.

3.6.1. Keyword Abbreviations

Let's consider the keyword that would trigger a saturation pressure calculation. Some people might prefer to remember the token `"SATP"`. Others might prefer `"SatPres"` or maybe `"SatPress"`. Still others might prefer `"SatPressure"`. All of these would be allowed, because **MMPz** recognizes the abbreviation `"SATP"` (remember that all tokens are case-insensitive).

On the other hand, the keyword for triggering a flash calculation must be spelled exactly as `"Flash"`. The tokens `"FlashIt"` or `"FlashCalc"` would not be recognized. That's because `"FLASH"` is *the* keyword, and not an abbreviation.

So how can one know when abbreviations are allowed? If you can think of a sensible abbreviation, then it's probably allowed. You can always try it. But this manual is the final authority.

That brings up another convention that will be followed in this manual. From now on, keywords within the text (but not the examples) will be written in a bold, mono-spaced typeface. Take, for example, the keyword **SATPres**. The mandatory characters will be written in uppercase, with optional characters in lowercase. Whenever optional characters are shown, the mandatory characters will represent the allowable abbreviation. An abbreviation might also be depicted by showing it with a trailing asterisk (to be viewed as an optional, wild-card character): **SATP***, for example. If a keyword is presented in uppercase characters only, however (e.g., **FLASH**), then that keyword is *not* an abbreviation for anything else.

3.6.2. Keyword Aliases

Because of its abbreviation, the keyword **SATPres** can be shortened to four characters, or expanded to even more, as the user prefers. But suppose the user would find it easier to remember **PSATuration** as the keyword for saturation pressure. Well, that's okay, because **PSAT*** is an *alias* for **SATP*** (and vice versa).

Many keywords have aliases. Whenever this manual describes the function of a keyword, it will list all of the possible aliases. The user doesn't need to learn every alias, though. One is enough. The idea is that one alias or another might be easier for each user to remember. Also, if a user can't immediately recall a particular keyword, there's a good chance that a logical guess might hit upon one of the aliases.

3.7. Commands

Many of the keywords recognized by **MMPz** are used to issue *commands* to the program. Each command might allow additional input, often in the form of additional keywords that are used to issue *subcommands* or specify *options*. Much of this manual is devoted to describing these commands and subcommands. When doing so, it will follow some specific conventions, which are described below.

After a heading to introduce a new command, the syntax of the command will be given. Here, the **TEMPerature** command will be used for illustration purposes:

TEMPerature **[(***value* unit**)**|**(**unit **[***value***])]** **[STOre** name**]**…
[REStore saved **[***fraction* **[LINear**|**LOGarithmic]]]**…

The keyword for the command (**TEMPerature**, in this case) will be given first, using the keyword conventions described in the previous section. Any subcommand keywords (e.g., **STOre** and **REStore**) or option keywords (**LINear** or **LOGarithmic**) will also be displayed according to the keyword conventions.

Any expected numerical input (e.g., *value* or *fraction*) will be shown in mono-spaced italics. Any expected non-numerical token input (e.g., unit, name, or saved) will be shown in the regular mono-spaced typeface. Any expected character string

input (not found in this example) will be shown in quoted italics, e.g., *"string"*. The instructions for the command will explain what the numbers, tokens, and strings are supposed to represent.

Input that is optional will be bracketed, such as **[STOre** name**]**. Brackets can be nested if an option can itself take optional input.

If there is a need to group two or more pieces of input into a single input pattern, they will be surrounded by parentheses, as with **(***value* unit**)**. Parentheses can be nested if grouped patterns need to be grouped further.

If there is a choice of two or more pieces of input (or input patterns), the choices will be separated by vertical bars ("|"). If the choice has a default selection, the default will be underlined. In the example, if a *fraction* is specified with the **REStore** subcommand (for making the temperature change only a *fraction* of the full change to the saved temperature), then one has the option of specifying either a **LINear** or a **LOGarithmic** scale for the *fraction*, with **LINear** being the default choice. Similarly, if one wishes to enter a *value* and a unit, they must both be entered immediately after the **TEMPerature** keyword (since they're not part of a subcommand), but they can be entered in either order (due to the choice of the two possible patterns).

If an optional (bracketed) input pattern can be entered more than once, it will be followed immediately by an ellipsis ("…"). This can be seen twice in the example. An ellipsis might also be used after the keyword for a subcommand (for example, **KEYword** …) if the syntax for the subcommand is complicated enough to warrant a separate explanation. This type of usage is not found in this example.

The parentheses, brackets, bars, and ellipsis characters do not need to be entered in the input file. They're only used in the manual to show syntax. Characters like this are normally delimiters, however, so they would just be ignored if they *were* entered.

Normally, the input for a command can be spread out over as many lines as desired (even with blank lines interspersed), and subcommands can normally be entered in any order. Any exceptions to these rules will be noted in the instructions for the command in question.

Instructions for the **TEMPerature** command will be given later in this manual, but the following examples illustrate legal syntax:

```
TEMPERATURE = 60 F
TEMPERATURE (F) = 60
TEMP: STORE LOW
TEMP 150 C, STORE HIGH, RESTORE LOW
TEMP RESTORE HIGH 0.75
TEMP 450 K, RESTORE LOW @ 0.5 LOG-SCALED
```

3.8. Tables

In some cases, data must be input in *tables* (component properties and binary interaction parameters, for example). **MMPz** allows a lot of flexibility in formatting tables. The columns can be placed in almost any desired order. So can the rows. Columns can be separated by spaces, tabs, or a combination of the two. Tables can be built with a text editor, or imported directly from a spreadsheet program. Cells can be left blank (for default values), even in the middle of a table. Entire rows or columns can easily be commented out, even in the middle of a table. Tables can even be assembled in sections. And yet, tables are actually quite easy to build, once the rules are understood.

The following is an example of a component property table, formatted in two sections. The meaning of the table will be explained later. For now, just concentrate on the format:

```
Component       Tc(K)      Pc(atm)      AF         MW       VShift
---------     --------    --------   -------    --------   --------
N2                                                          -0.0079
CO2                                                          0.0833
Methane                                                      0.0234
Ethane                                                       0.0605
Propane                                                      0.0825
Butanes         425.2       37.5      0.193      58.123     0.0902
Pentanes        469.6       33.3      0.251      72.150     0.1115
Hexanes         507.4       29.3      0.296      86.177     0.1467

Component         Vc           ~ZC         Tb         SG
                ft3/lbmol                   R
---------     -----------   --------   -------   --------
Butanes          4.080        0.2736     490.8     0.5844
Pentanes         4.870        0.2623     556.6     0.6301
Hexanes          5.929        0.2643     615.4     0.6604
```

Notice that each column has a *header* keyword and that some columns (for dimensional properties) require *unit* keywords. At least one of the columns in each table will contain independent variables; the rest will contain dependent variables. One of the independent variables must be in the first column. The rest of the columns can be placed in any desired order. In the example above, **`Component`** is the independent variable.

Comment lines have been strategically placed to underline each header above. They are not required, however. They are strictly cosmetic, since comment lines are completely ignored.

A table can be entered in multiple sections (two, in the example). The first section must define every row of the table, but not necessarily all of the columns. Subsequent sections can add or modify columns of data for any previously defined row, but

cannot define new rows. Rows are identified by their independent variables. The second section of the example table contains rows corresponding to the last three rows of the first section (i.e., the last three components), but not to the first five. That is perfectly acceptable.

In the example table, the only property that has been specified for the first five components (`N2` through `Propane`) is **`VShift`**. The rest of their properties have been left blank, for default values (these will normally be filled in from the component library or from various correlations, but such details are unimportant here).

How does **MMPz** know that `-0.0079` is `N2`'s **`VShift`** property, and not its **`TC`** property? The answer is simple. It *lines up* the data in each column with the appropriate header by *character position*. If a column entry looks like it's lined up underneath a particular heading, then it *probably* belongs to that heading. The actual rule is a little more specific, however. Locate the last character position of the column entry and then find that position on the header line. The first header keyword that is encountered *at or to the left of* that position is the one to which that entry belongs. Each header is allowed no more than one entry per row; extraneous entries constitute errors.

Note that **a mono-spaced typeface is required** to guarantee that a properly aligned table will *look* like it's properly aligned. No matter how a table looks, however, the above rules dictate the proper data assignments.

In the example below, where the alignment is not very obvious at first, the associations between the data and their headers are indicated by their indices. Two of the data entries are misplaced however. They are indicated, appropriately, by the word `error`. The first entry is in error because it's not aligned with any header. The third entry is also in error because it's aligned with `Header2`, which has already been assigned the entry `datum2`.

```
     Header1   Header2   Header3   Header4   Header5
error     datum2    error     datum4    datum5
```

Going back to the example of the component property table, note that some of the columns require units, which are specified by their own keywords. Unit keywords can be entered in one of two ways: they can be placed *after* their corresponding header keywords, or they can be placed on the next line, *under* their corresponding header keywords (and lined up according to the rules above).

If a column's header begins with a question mark (`"?"`), an underscore (`"_"`), or a tilde (`"~"`), that column will be read but then ignored. The column for **`ZC`** has been commented out that way in the example.

3.8.1. Tabs Within Tables

One needs to be a little careful about using tab characters to separate the columns within a table. This is especially true if tabs and spaces are combined to align a table (visually) in a text editor. The problem is that there's no standard way for editors to

display tabs, making visual alignment editor-dependent. For many editors, the default behavior is to simulate a *tab stop* (an old typewriter term) every 4 characters. Others use a default tab setting of 8. Still others treat tabs as single spaces. With most editors, this setting can be changed, often from document to document. For **MMPz** to read a table the way it was intended, it has to know (or be able to guess) how the table appeared to the user. When tabs are present, this is not always straightforward.

It's easy to end up with tabs in an input file. Tabs are commonly introduced with spreadsheet data and some text editors insert them at every opportunity. Fortunately, there are four different ways, all of them fairly simple, to get **MMPz** to handle tabs correctly.

The first suggestion is to not use tabs. Most text editors can be instructed to never insert them. If they *are* introduced somehow, most text editors can be instructed to convert them to spaces, or at least to find them so they can be replaced manually. This is not necessarily the most practical suggestion, however.

The second suggestion, and probably the simplest, is to align the tables visually, with a text editor, but then to tell **MMPz** about the editor's tab setting. If **MMPz** knows how many character positions correspond to each tab, then it will see the table exactly as it appears in the editor. **MMPz** has a **`TABS`** command specifically for this purpose. If the editor's tab setting is 8, for example, then one simply inserts the corresponding instruction, **`TABS`** `= 8`, somewhere in the input file, prior to any tables. In fact, it is strongly advised that **whenever a table containing tabs is aligned in a text editor, one should use the `TABS` command to inform MMPz of the editor's tab setting**.

The third suggestion, which is also very simple, is to increase the editor's tab setting to a value larger than the width of any column in any table (10, 15 or even 20, perhaps), guaranteeing that each tab signifies a new column.

The final suggestion is to adopt the strategy of putting exactly one tab (with no extra spaces) between any two entries from adjacent columns, and not worrying about whether the resulting tables line up visually. This strategy is difficult to implement with a text editor, but it becomes the natural choice if tables are built with a spreadsheet program (or even a word processor, if one is careful).

If you build a tab-delimited table with a spreadsheet program, then you probably won't need to use **MMPz**'s **`TABS`** command (as long as you don't realign the table in a text editor afterwards). That's because every entry in a column (including the header) will be separated from its corresponding entry in the next column by a uniform number of tabs. By default, **MMPz** assumes a tab setting of 100, so it will think that the columns start in character positions 1, 101, 201, and so forth. As long as all entries are less than 100 characters long, they're guaranteed to line up under **MMPz**'s rules. If a tab-delimited table is imported to a text editor, it may not *appear* to line up correctly. That can be fixed, however, simply by increasing the editor's tab setting. It should **not** be "fixed" by inserting additional tabs or spaces.

Chapter 4. Building an Input File

The best way to build an input file for **MMPz** is probably to modify an existing input file. This chapter, however, will present a recipe for building a typical input file from scratch. It will do this primarily by example, showing the most important features of the most important commands. A complete description of all the commands will be given in the next chapter.

The goal here will be to calculate a couple of MMPs and a couple of MMEs, with a few other simple phase behavior calculations along the way. In order to calculate an MMP or an MME, you first need:

- An EOS fluid characterization.
- An oil composition.
- A temperature.

In order to calculate an MMP, you also need:

- An injectant composition.

In order to calculate an MME, you also need:

- Compositions of a lean gas and a rich gas.
- A pressure.

Therefore, this chapter will demonstrate the commands necessary to:

- Specify a fluid characterization.
- Specify an oil composition.
- Calculate the oil's saturation pressure at a given temperature.
- Specify compositions for a lean gas and an enriching gas.
- Specify an injection gas mixture of the lean and rich gases.
- Calculate the MMP of the injection gas in displacing the oil, using two different numbers of stages.
- Calculate the MME of the lean gas with the rich gas in displacing the oil.
- Deplete the oil to a lower pressure.
- Calculate the MME of the lean gas with the rich gas in displacing the depleted oil at the original MMP.
- Check **MMPz**'s previous estimate of multicontact miscibility.

4.1. Title

Although not required, a good way to start is by giving this particular problem a title, which will help to document both the input file and the output file. In this case, we'll give it a multiple line title. It is also sometimes helpful to **`ECHO`** portions of the input file to the output file for later reference, so we will **`ECHO`** the entire file:

```
Echo ON

Title: "Simple MMPz Test Case"
Title: "Jessen-Michelsen-Stenby Fluid System"
Title: "SPE 50632"
```

4.2. Fluid Characterization

Now we'll input something essential: the EOS fluid characterization. This consists of a name for the characterization, a specification of the EOS (if not the 1979 Peng-Robinson, **`PR79`**), and a list of components, along with their physical properties and binary interaction parameters, which are specified in separate tables (each of which can be split into sections, if desired). The only physical properties that are mandatory for MMP or MME calculations are molecular weight, critical temperature, and critical pressure. If a fluid characterization includes acentric factors, binary interaction parameters, or any modifications to the Ω_A and Ω_B parameters of the EOS, then these parameters must also be specified (to get the correct phase behavior predictions). One way to input a valid characterization would be:

```
Characterization "JMS (Tc-Tuned)", EOS = SRK

Component      Tc(K)       Pc(atm)        AF          MW
N2           126.200      33.6000      0.0400      28.016
CO2          304.200      72.9000      0.2280      44.010
C1           190.600      45.4000      0.0080      16.043
C2           305.400      48.2000      0.0980      30.069
C3           369.800      41.9000      0.1520      44.096
i-C4         408.100      36.0000      0.1760      58.123
n-C4         425.200      37.5000      0.1930      58.123
i-C5         460.400      33.4000      0.2270      72.150
n-C5         469.600      33.3000      0.2510      72.150
C6           507.400      29.3000      0.2960      86.177
C7           632.800      30.2987      0.1842     109.007
C11          659.605      23.4598      0.4773     175.327
C16          703.646      19.2900      0.8197     256.674
C23          766.497      16.7852      1.2114     370.099
C33          830.500      15.1302      1.3718     590.374

Binaries      N2         CO2
C1           0.02       0.12
C2           0.06       0.15
C3           0.08       0.15
i-C4         0.08       0.15
n-C4         0.08       0.15
i-C5         0.08       0.15
n-C5         0.08       0.15
C6           0.08       0.15
C7           0.08       0.15
C11          0.08       0.15
C16          0.08       0.15
C23          0.08       0.15
C33          0.08       0.15

END; of characterization
```

For accurate density predictions, most fluid characterizations also include volume shift factors (**`VShift`**), but this characterization does not. Volume shift factors do not affect any miscibility or equilibrium calculations—only volumetric predictions.

If there are any blank entries in the component property table, **MMPz** will attempt to apply default values from the component library or from various correlations. Except for the EOS parameters **`A`** and **`B`** (or **`AMOD*`** and **`BMOD*`**), all other component properties (which have not been given a heading in the component property table) will default to zero. The **`A`** and **`B`** parameters will default to the Ω_A and Ω_B of the particular EOS, however, while the multipliers **`AMOD*`** and **`BMOD*`** will default to 1. All binary interaction parameters will default to zero.

Even though Therefore, characterizations must be specified completely ().

4.3. Oil Composition

Compositions are specified with the **MIXture** command, which has a great deal of flexibility. But in the simplest case, it just needs a token for the resulting mixture's name and a list of the mole fractions or percentages:

```
Mix Oil:  0.450   1.640  45.850   7.150   6.740
          0.840   3.110   1.030   1.650   2.520
         12.440   6.320   5.024   3.240   1.996
```

4.4. Temperature and Pressure

The temperature is easily set with the **TEMPerature** command (which must include the desired units). It remains set until another command is used to change it. We might as well specify a **PRESsure**, also. This will be changed shortly by a saturation pressure calculation, so the value we choose here is not very important. But by specifying a pressure, we also specify the units, which will then be maintained by the subsequent saturation pressure calculation.

```
Temp = 387.45 K, Pres = 0 atm
```

Notice that multiple commands can be entered on a single line.

4.5. Saturation Pressure Calculation

It is not necessary to calculate the oil's saturation pressure, but it will be instructive to do so anyway.

Before any phase behavior experiment can be performed, the experiment's **FEED** tank must be filled. Once filled, it will remain filled, although some experiments may alter its contents (none of the experiments performed here will do that, however). The **MIXture** command is again used to fill the **FEED** tank, this time by taking one **MOLE*** from the previously defined OIL mixture.

After filling the FEED tank, we are ready to issue the **SATPressure** command:

```
Mix Feed: 1 mole Oil, SATP
```

4.6. Injection Gases

Now let's specify some injection gas compositions. First, we'll specify a lean gas and a rich gas. Then we'll fill the special **INJectant** tank with a mixture of 35% enrichment: 0.35 moles of rich gas, with lean gas added to make up 1 total mole (**TMOLE***). The

INJectant mixture is always used as the injection gas in an **MMP** experiment (it is also created by an **MME** experiment).

```
Mix Lean:  0.49    1.82   81.39    9.15    4.67    0.50
           1.24    0.20    0.26    0.09    0.19

Mix Rich:  0.67    2.44   68.16   10.32    9.50    1.09
           3.75    0.95    1.31    0.91    0.90

Mix Injectant:  0.35 moles Rich, 1 Tmole Lean
```

4.7. MMP Experiments

Now that we've set the temperature, the **FEED**, and the **INJectant**, we're ready to calculate our first MMPs, using the **MMP** command:

```
MMP

MMP 100, ID "Original Oil, 35% Enriched Solvent"
```

The simplest form of the **MMP** command is just the **MMP** keyword, in which case it will use 50 mixing stages, or cells, in performing its calculation. Then, however, we've chosen to repeat the calculation using 100 stages. This second time, we've also chosen to add an identifying character string with the optional **ID** subcommand.

4.8. MME Experiment

Now let's calculate an MME with the **MME** command. Our current pressure is the 100-stage MMP of the 35% enriched solvent. If we don't change that pressure, and calculate the 100-stage MME of our `Lean` gas with our `Rich` gas, the result should be 35%. Let's also make a **NOTE** of that in both the input and output files:

```
Note: "Calculate the MME of the Lean gas with the Rich gas"
Note: "at the MMP of a 35% enriched injectant. Thus, the"
Note: "MME should turn out to be 35%, if consistent."

MME Lean Rich 100
    ID "Original Oil at MMP of 35% Enriched Solvent"
```

4.9. The Resultant MME Injectant

One effect of the **MME** command is to fill the **INJectant** tank with the resulting mixture of gases having the minimum miscibility enrichment. If we want to view the composition of this resulting mixture, the simplest way to do so is to refill the **INJectant** tank with its own contents:

```
Mix Injectant = Injectant
```

4.10. Depletion

Just for fun, let's try calculating the MME of a depleted oil sample, but at the original MMP. The first step is to **STOre** the current pressure (still the original oil's MMP). Then, we **FLASH** the oil to a new pressure and fill the **FEED** tank with the resulting equilibrium liquid (found in the **EQLiquid** tank). Finally, we **REStore** the pressure to the original MMP:

```
Pressure: Store MMP
Pressure: 150 atm, Flash, Mix Feed: EQL
Pressure: Restore MMP
```

Notice that the MMP token above is simply a user-selected name to be used by the **STOre** and **REStore** subcommands of the **PRESsure** command. It has nothing to do with, nor does it conflict with, the **MMP** command.

4.11. Additional MME Experiment

Now that we have depleted the oil to 150 atm and restored the pressure to the original MMP, let's calculate the new MME, to see if it has been affected by the depletion:

```
MME Lean Rich, Stages 100
    ID "Depleted Oil at Original MMP of 35% Enriched Solvent"
```

The only thing new here is the use of the **STAGES** subcommand to make it clearer that 100 stages are to be used.

4.12. Multicontact Miscibility Tests

When determining the miscibility or immiscibility of a given displacement process, **MMPz** calculates a minimum measure of *immiscibility* at each stage of its multistage simulation. Then, it extrapolates that minimum immiscibility to an infinite number of stages to see if the process would remain immiscible or become miscible. If one wants to check on the trend of the calculated immiscibilities, or on their extrapolation (to see if enough stages are being used, perhaps), then one can use the **MCM** command to perform a multicontact miscibility experiment.

While performing the multicontact miscibility experiment, we might also want to check for any possible three-phase equilibria (which can wreak havoc on a reservoir simulator). We simply use the **TEST2** command to have **MMPz** test the thermodynamic stability of all subsequent 2-phase equilibria. Warnings will be issued if any 2-phase solutions are found to be unstable (i.e., if a third phase is indicated).

```
Test2 All; (check the stability of all 2-phase solutions)

MCM 100 ID "Extrapolation Check"
```

4.13. Final Input File

The final input file, with a few additional comments, is shown below.

```
Echo ON

Title: "Simple MMPz Test Case"
Title: "Jessen-Michelsen-Stenby Fluid System"
Title: "SPE 50632"

Characterization "JMS (Tc-Tuned)", EOS = SRK

Component      Tc(K)       Pc(atm)        AF          MW
---------    ---------    ---------    --------    --------
N2            126.200      33.6000      0.0400      28.016
CO2           304.200      72.9000      0.2280      44.010
C1            190.600      45.4000      0.0080      16.043
C2            305.400      48.2000      0.0980      30.069
C3            369.800      41.9000      0.1520      44.096
i-C4          408.100      36.0000      0.1760      58.123
n-C4          425.200      37.5000      0.1930      58.123
i-C5          460.400      33.4000      0.2270      72.150
n-C5          469.600      33.3000      0.2510      72.150
C6            507.400      29.3000      0.2960      86.177
C7            632.800      30.2987      0.1842     109.007
C11           659.605      23.4598      0.4773     175.327
C16           703.646      19.2900      0.8197     256.674
C23           766.497      16.7852      1.2114     370.099
C33           830.500      15.1302      1.3718     590.374
```

```
Binaries      N2        CO2
--------    ------    ------
C1           0.02      0.12
C2           0.06      0.15
C3           0.08      0.15
i-C4         0.08      0.15
n-C4         0.08      0.15
i-C5         0.08      0.15
n-C5         0.08      0.15
C6           0.08      0.15
C7           0.08      0.15
C11          0.08      0.15
C16          0.08      0.15
C23          0.08      0.15
C33          0.08      0.15

END; of characterization

* Define the original oil composition:

Mix Oil:  0.450   1.640  45.850   7.150   6.740
          0.840   3.110   1.030   1.650   2.520
         12.440   6.320   5.024   3.240   1.996

* Set the temperature and initialize the pressure units:

Temp = 387.45 K, Pres = 0 atm

* Calculate the saturation pressure of the original oil:

Mix Feed: 1 mole Oil, SATP

* Define lean and rich gas compositions:

Mix Lean:  0.49    1.82   81.39    9.15    4.67    0.50
           1.24    0.20    0.26    0.09    0.19

Mix Rich:  0.67    2.44   68.16   10.32    9.50    1.09
           3.75    0.95    1.31    0.91    0.90

* Calculate the MMP of a 35% enriched injectant,
* first with 50 stages, then with 100:

Mix Injectant:  0.35 moles Rich, 1 Tmole Lean

MMP

MMP 100, ID "Original Oil, 35% Enriched Solvent"
```

```
Note: "Calculate the MME of the Lean gas with the Rich gas"
Note: "at the MMP of a 35% enriched injectant. Thus, the"
Note: "MME should turn out to be 35%, if consistent."

MME Lean Rich 100
    ID "Original Oil at MMP of 35% Enriched Solvent"

* View the resulting, minimum enriched injectant:

Mix Injectant = Injectant

* Deplete the oil to 150 atm, but restore the pressure:

Pressure: Store MMP
Pressure: 150 atm, Flash, Mix Feed: EQL
Pressure: Restore MMP

* Calculate the new MME for the depleted oil:

MME Lean Rich, Stages 100
    ID "Depleted Oil at Original MMP of 35% Enriched Solvent"

* Double-check the miscibility results:

Test2 All; (check the stability of all 2-phase solutions)

MCM 100 ID "Extrapolation Check"
```

Chapter 5. Command Reference

This chapter describes all of **MMPz**'s commands in detail. The commands will be ordered by relative importance, with the most important first.

5.1. Characterization Command

CHARacterization *"name"*
[EOS PR79|PR77|SRK|RK] [COMPonent…]… [BINARies…]…

Alias: PROPerties

The **CHARacterization** command initiates the input of a new EOS fluid characterization (the characterization can be changed at any point in the input file).

Argument *"name":* this is the name of the characterization. It should appear immediately after the **CHARacterization** keyword, on the same line (otherwise, the rest of this line should remain blank). The name can be used later to **RESTore** a previously defined characterization.

Option EOS: this defines the equation of state for this characterization. It can be
- **PR79**: the 1979 version of the Peng-Robinson EOS.
- **PR77**: the 1977 version of the Peng-Robinson EOS.
- **SRK**: the Soave-Redlich-Kwong EOS.
- **RK**: the Redlich-Kwong EOS.

5.1.1. Component Subcommand

```
COMPonent [MW*] [TC] [PC] [AF*] [VShift]
          [A] [B] [AMOD*] [BMOD*] [TB*] [SG*]
          [VC] [ZC] [VISVc] [VISZc] [PCHOR*]
          [LMW*] [LTB*] [LSG*] [UMW*] [UTB*] [USG*]
          [FULLname]
```

Alias: NAME*

The **COMPonent** subcommand introduces the component properties table. It also serves as the header keyword for the column of component names. Column header keywords for the rest of the component properties listed in the table should be entered after the **COMPonent** keyword, on the same line, in any order. The properties themselves should be *lined up* underneath their corresponding headers according to the rules given in section 3.8.

The component properties table can be divided into several sections by issuing the **COMPonent** subcommand more than once. The first table must list all of the

characterization's components. Subsequent tables only need to list those components that are to be assigned new properties, but new components cannot be introduced.

Table 5–1 lists the component properties that will affect **MMPz** calculations. Table 5–2 lists a number of other component properties that will be recognized on input and written on output, but that won't affect any of the calculations. They are included for future compatibility with a **Zick Technologies** PVT program currently under development.

Table 5–1. Component Properties That Will Affect MMPz Calculations

Keyword	Property	Type of Units	Aliases
`MW*`	Molecular Weight		
`TC`	Critical Temperature	Temperature	`TCRit`
`PC`	Critical Pressure	Pressure	`PCRit`
`AF*`	Acentric Factor		`ACentric`
`VShift`	Volume Shift Factor		`VTran`
`A`	EOS Constant, Ω_A		
`B`	EOS Constant, Ω_B		
`AMOD*`	EOS Ω_A Modifier		
`BMOD*`	EOS Ω_B Modifier		

Table 5–2. Component Properties That Won't Affect MMPz Calculations

Keyword	Property	Type of Units	Aliases
`TB*`	Boiling Point Temperature	Temperature	
`SG*`	Specific Gravity		
`VC`	Critical Volume	Molar Volume	`VCRit`
`ZC`	Critical Z-Factor		`ZCRit`
`VISVc`	Critical Volume for Viscosity Correlations	Molar Volume	`VCV*, VV*`
`VISZc`	Critical Z-Factor for Viscosity Correlations		`ZCV*, VZ*`
`PCHOR*`	Parachor		`PARAchor`
`LMW*`	Lower Molecular Weight		
`LTB*`	Lower Boiling Point Temperature	Temperature	
`LSG*`	Lower Specific Gravity		

UMW*	Upper Molecular Weight		
UTB*	Upper Boiling Point Temperature	Temperature	
USG*	Upper Specific Gravity		
FULLname	Full Name of Component		

All of the properties require numerical input except **FULLname**, which requires character string input. Some of the properties require physical units, which should be specified according to the rules in section 3.8. The correct type of units (temperature, e.g.) is given in Table 5–1 or Table 5–2 for each property. The keywords for the available units of each type are given in Chapter 6.

COMPonent subcommand (component property table) examples are shown below.

```
Component       Tc(K)      Pc(atm)      AF          MW       VShift
N2                                                          -0.0079
CO2                                                          0.0833
Methane                                                      0.0234
Ethane                                                       0.0605
Propane                                                      0.0825
Butanes          425.2       37.5      0.193      58.123     0.0902
Pentanes         469.6       33.3      0.251      72.150     0.1115
Hexanes          507.4       29.3      0.296      86.177     0.1467

Component          Vc           ~ZC         Tb          SG
                ft3/lbmol                    R
Butanes            4.080       0.2736     490.8      0.5844
Pentanes           4.870       0.2623     556.6      0.6301
Hexanes            5.929       0.2643     615.4      0.6604
```

5.1.2. Binaries Subcommand

BINARies [component-name]…

Alias: BIPS

The **BINARies** subcommand introduces the table of binary interaction parameters (BIPs). It also serves as the header keyword for the first column, which will contain a list of previously defined component names. Component names are also used for the header keywords of the other columns in the table. Hence, component names identify both the rows and the columns of the table, forming a matrix of component pairs. Each entry in that matrix should be the interaction parameter of the corresponding component pair. The table should be constructed according to the rules given in section 3.8. An example is shown here:

Binaries	CO2	C1	C30+
CO2			0.1150
C1	0.1050		0.0928
C2-3	0.1300	0.0030	0.0750
C4-5	0.1150	0.0138	0.0527
C6-7	0.1150	0.0276	0.0352
C8-10	0.1150	0.0362	0.0267
C11-14	0.1150	0.0440	0.0201
C15-19	0.1150	0.0520	0.0143
C20-24	0.1150	0.0641	0.0074
C25-29	0.1150	0.0780	0.0021

The BIPs table can be divided into several sections by issuing the **BINARies** subcommand more than once. **MMPz** will ensure that the final BIPs matrix will be symmetric and that its diagonal will be zero. If the parameter for a component pair is entered more than once, the last non-blank entry will take precedence. If there is no non-blank entry for a component pair, then its parameter will default to zero.

5.2. Mixture Command

MIXture name **[[***amount***] [**unit**] [**tank**]]…**

The **MIXture** command is one of the most important in **MMPz**. It allows fluid compositions to be assembled with great flexibility. It can be entered as frequently as needed and can be spread out over as many lines as desired.

Argument name: this is the name of the *destination tank* in which the new mixture will be stored (refer to the concepts described in Chapter 2). Once defined, this tank can then be used in subsequent MIXture commands as a *source tank*.

The name of the tank can be any token that doesn't conflict with any of the unit keywords shown below, any of the current characterization's component names, or any of the reserved tank names: **EQL***, **EQV***, **NFL***, or **NFV*** (these will be discussed later).

The **FEED** tank is special. If the mixture is stored in the tank named **FEED**, then it becomes the new feed mixture for the next set of phase behavior experiments. The **FEED** tank must be filled at some point before the first experiment.

The **INJectant** tank is also special. An **MMP** experiment will always use the current contents of the **INJectant** tank for its injection gas. This tank can be filled at any time by a **MIXture** command with a name argument that matches the keyword **INJectant**.

Arguments *amount*, unit, tank: one set of these three optional arguments forms an *ingredient*. The **MIXture** command will read as many ingredients as it can find. The three arguments of each ingredient can be input in any order, and each is optional,

although one must be careful if the number or order of the arguments is changed from ingredient to ingredient (more on this below).

Argument `tank:` this argument specifies the name of the source tank for the current ingredient. It can be the name of a user-defined tank, one of the user-defined *component tanks*, or one of **MMPz**'s pre-defined tanks (**`FEED`**, **`INJectant`**, **`EQLiquid`**, **`EQVapor`**, **`NFLiquid`**, or **`NFVapor`**).

A user-defined tank always contains whatever the user last stored in it, in whatever amount. A user-defined *component tank* always contains exactly 1 mole of the corresponding, user-defined component.

The **`FEED`** tank contains whatever the user last stored in it, in whatever amount, until any type of phase behavior calculation is performed, after which, the FEED tank will contain exactly 1 mole of whatever the user last stored in it.

The **`INJectant`** tank normally contains whatever the user last stored in it. When an **`MME`** command is executed, however, it will fill the **`INJectant`** tank with exactly 1 mole of the gas mixture it finds to have the minimum enrichment for miscibility.

The **`EQLiquid`** and **`EQVapor`** tanks will contain 1 mole of the equilibrium liquid and equilibrium vapor, respectively, found during the most recent phase behavior calculation. If the last phase behavior calculation found only one equilibrium phase, then the **`EQLiquid`** and **`EQVapor`** tanks will both contain 1 mole of that overall phase.

The **`NFLiquid`** and **`NFVapor`** tanks will contain 1 mole of the *negative flash*[1] liquid and *negative flash* vapor, respectively, found during the most recent phase behavior calculation. If an equilibrium tie line passes through the overall composition of a fluid, then the intersections of that tie line with the boundaries of the two-phase envelope define the *negative flash* phase compositions, *regardless of whether the overall composition is inside or outside of the two-phase envelope*. Negative flash phases are always in equilibrium with each other. If the overall composition was within the two-phase envelope during the last phase behavior calculation, then the contents of the **`NFLiquid`** and **`NFVapor`** tanks will be exactly the same as the contents of the **`EQLiquid`** and **`EQVapor`** tanks, respectively. If the overall composition was outside of the two-phase envelope, but a tie line passing through it was found, then the **`NFLiquid`** and **`NFVapor`** tanks will end up with the unique, negative flash compositions, while the **`EQLiquid`** and **`EQVapor`** tanks will both end up with the overall composition. If no tie line passing through the overall composition was found, then the **`NFLiquid`**, **`NFVapor`**, **`EQLiquid`**, and **`EQVapor`** tanks will all end up with the overall composition.

The default `tank` for an ingredient is the component tank following the last component tank that went into the mixture (component tanks follow the same ordering as their corresponding components). If no previous ingredient has used a component tank, then the current ingredient's default `tank` will be the first component tank (the one containing the first defined component).

---

[1] Whitson, C.H., and Michelsen, M.L.: "The Negative Flash," *Fluid Phase Equilibria* **53** (1989), 51.

Note that the destination tank can also be used as a source tank. The contents of the destination tank are not changed until after all of the sources are **MIXed**. There are several examples of this given near the end of this section.

Argument `unit:` this can be one of the following:

- **TANKs**: the *amount* of the current ingredient will be measured in *tankfuls* of the ingredient's source tank, regardless of how many moles or how much mass that tank might contain.
- **MOLEs**: the *amount* of the current ingredient will be measured in *mole units*.
- **MASS***: the *amount* of the current ingredient will be measured in *mass units*.
- **TMOLEs**: enough of the current ingredient will be added (or subtracted) to bring the *total moles* of the current mixture to the specified *amount*.
- **TMASS***: enough of the current ingredient will be added (or subtracted) to bring the *total mass* of the current mixture to the specified *amount*.

Note that a mixture can be constructed with some ingredients measured in mole units, others in mass units, and still others in tankfuls, as long as the mole and mass units remain consistent (kilograms are consistent with kg-moles, but not with lb-moles, for example).

The default `unit` for an ingredient is the `unit` of the mixture's previous ingredient, while the mixture's first ingredient has a default `unit` of **TANKs**.

Argument *amount:* This determines how much of the current ingredient, measured in the ingredient's unit, is to be added to the current mixture.

The default *amount* of an ingredient is 1 (regardless of the ingredient's `unit`).

When the **MIXture** command looks for a new ingredient to be added, it will see if the next three arguments include one of each type: *amount*, `unit`, and `tank`. If it encounters a second argument of one type before it encounters one of each, however, it will assume the duplicate belongs to the next ingredient. The current ingredient will then be added, with default values for any missing arguments.

For the examples below, assume that three components have been defined: `C1`, `C2`, and `C3` (in that order). Assume also that their molecular weights are 16, 30, and 44, respectively.

	Resulting Moles			Resulting Masses		
	C1	C2	C3	C1	C2	C3
Mix A:	0.0	0.0	0.0	0.0	0.0	0.0
Mix B: 2 1	2.0	1.0	0.0	32.0	30.0	0.0
Mix C: C2 3 C3 2	0.0	3.0	2.0	0.0	90.0	88.0
Mix C: 3 C2 2 C3	0.0	3.0	2.0	0.0	90.0	88.0
Mix C: C2 3 2	0.0	3.0	2.0	0.0	90.0	88.0
Mix C: 3 C2 2	0.0	3.0	2.0	0.0	90.0	88.0
Mix D: 3 2 C3	3.0	0.0	2.0	48.0	0.0	88.0
Mix C: C D	3.0	3.0	4.0	48.0	90.0	176.0
Mix D: 2 D	6.0	0.0	4.0	96.0	0.0	176.0
Mix D: D 0.5	3.0	0.0	2.0	48.0	0.0	88.0
Mix D: 1 mole D	0.6	0.0	0.4	9.6	0.0	17.6
Mix D: D mass 272	6.0	0.0	4.0	96.0	0.0	176.0
Mix F: moles						
C1 4						
C2 2						
C3	4.0	2.0	1.0	64.0	60.0	44.0
Mix G: masses						
C1 16						
C2 30						
C3 44	1.0	1.0	1.0	16.0	30.0	44.0
Mix H:						
0.2 G						
0.7 moles F	0.6	0.4	0.3	9.6	12.0	13.2
Mix H:						
G 0.2						
F 0.7 moles	0.6	0.4	0.3	9.6	12.0	13.2
Mix H:						
0.7 moles F						
0.2 tanks G	0.6	0.4	0.3	9.6	12.0	13.2
Mix I:						
0.7 moles F						
0.3 tanks G	0.7	0.5	0.4	11.2	15.0	17.6
Mix J:						
0.7 moles F						
0.3 G	0.5	0.3	0.2	8.0	9.0	8.8
Mix K:						
0.7 moles F						
1.0 Tmole G	0.5	0.3	0.2	8.0	9.0	8.8
Mix L:						
1.0 moles C3						
100 Tmass C1						
400 C2	3.5	10.0	1.0	56.0	300.0	44.0

5.3. Temperature Command

TEMPerature [(*value* unit)|(unit [*value*])] [**STOre** name]…
[**REStore** saved [*fraction* [**LINear**|**LOGarithmic**]]]…

The TEMPerature command allows the temperature for subsequent phase behavior calculations to be set, stored, and/or restored. It can be entered as frequently as needed, but it must always fit on a single line.

Argument *value:* the temperature will be set to this *value*. The default is to not change the temperature.

Argument unit: The unit of temperature will be specified by this token, which can by any of the keywords listed in Table 6–1. A unit can be specified without a *value* (to change units without changing the temperature), but if a *value* is specified, then the unit must also be specified.

Option STOre (**Alias: SAVe**)**:** this option allows the current temperature (whether or not it has just been given a new *value*) to be **STOred** as a named temperature (for later **REStoration**).

Argument name: this token will be the name assigned to a **STOred** temperature.

Option REStore: this option allows the temperature to be **REStored** to a previously **STOred** temperature, or at least changed toward that **STOred** temperature by a specified *fraction*, either **LINearly** or **LOGarithmically**.

Argument saved: this token must be a name under which a temperature was previously **STOred** (or **SAVed**).

Argument *fraction:* the temperature will be changed by this *fraction* of the distance from the current temperature (whether or not it has just been given a new *value*) toward the saved temperature, either on a **LINear** scale or a **LOGarithmic** scale. The *fraction* can be any number. By default, it is 1.

Option LINear: if a *fraction* is specified with the **LINear** option (the default), then this *fraction* of the difference between the current and the saved temperatures (i.e., saved minus current) will be added to the current temperature.

Option LOGarithmic: if a *fraction* is specified with the **LOGarithmic** option, then this *fraction* of the difference between the logarithms of the absolute current and the absolute saved temperatures (i.e., log of saved minus log of current) will be added to the logarithm of the absolute current temperature.

Some examples are shown below.

```
Temperature = 0 Celcius                                ; now    0 C
Temperature => Fahrenheit                              ; now   32 F
Temp store Freezing                                    ; now   32 F
Temp 100 K                                             ; now  100 K
Temp R, store Low                                      ; now  180 R
Temp(C): 100, store Boiling, restore Freezing          ; now   32 F
Temp restore Boiling 0.4                               ; now   40 C
Temp 1600 K, restore Low @ 0.75 Log-scaled             ; now  360 R
```

5.4. Pressure Command

PRESsure **[(***value* unit**)**|**(**unit **[***value***])]** **[STOre** name**]…**
[REStore saved **[***fraction* **[LINear**|**LOGarithmic]]]…**

The PRESsure command allows the pressure for subsequent phase behavior calculations to be set, stored, and/or restored. It can be entered as frequently as needed, but it must always fit on a single line.

Argument *value:* the pressure will be set to this *value*. The default is to not change the pressure.

Argument unit: The unit of pressure will be specified by this token, which can by any of the keywords listed in Table 6–2. A unit can be specified without a *value* (to change units without changing the pressure), but if a *value* is specified, then the unit must also be specified.

Option STOre (**Alias: SAVe**)**:** this option allows the current pressure (whether or not it has just been given a new *value*) to be **STOred** as a named pressure (for later **REStoration**). This can be particularly useful after a **SATPressure** or **MMP** command.

Argument name: this token will be the name assigned to a **STOred** pressure.

Option REStore: this option allows the pressure to be **REStored** to a previously **STOred** pressure, or at least changed toward that **STOred** pressure by a specified *fraction*, either **LINearly** or **LOGarithmically**.

Argument saved: this token must be a name under which a pressure was previously **STOred** (or **SAVed**).

Argument *fraction:* the pressure will be changed by this *fraction* of the distance from the current pressure (whether or not it has just been given a new *value*) toward the saved pressure, either on a **LINear** scale or a **LOGarithmic** scale. The *fraction* can be any number. By default, it is 1.

Option `LINear:` if a *`fraction`* is specified with the **`LINear`** option (the default), then this *`fraction`* of the difference between the current and the `saved` pressures (i.e., `saved` minus current) will be added to the current pressure.

Option `LOGarithmic:` if a *`fraction`* is specified with the **`LOGarithmic`** option, then this *`fraction`* of the difference between the logarithms of the absolute current and the absolute `saved` pressures (i.e., log of `saved` minus log of current) will be added to the logarithm of the absolute current pressure.

Some examples are shown below.

```
Pressure = 0 psig                          ; now   0 psig
Pressure => atm                            ; now   1 atm
PRES store Ambient                         ; now   1 atm
PRES(BAR) 0, store Zero                    ; now   0 bar
PRES 40.53 MPa                             ; now  40.53 MPa
PRES restore Zero 0.5                      ; now 202.65 bar
PRES restore Zero 0.5                      ; now 101.325 bar
PRES restore Ambient 0.5 log               ; now  10 atm
```

5.5. MMP Command

`MMP` [stages] **[`ID`** *"identifier"*`]` **[`STAGES`** stages]
[`DATum` **(**(*mmp* unit)|(unit *mmp*)**)** **[`WEIGht`** wt]]

The **`MMP`** command computes the minimum miscibility pressure for the displacement of the **`FEED`** tank's fluid by the **`INJectant`** tank's fluid at the current **`TEMPerature`**. It uses a proprietary, iterative, multistage, multicontact algorithm to determine the minimum miscibility pressure no matter what the displacement mechanism (condensing, vaporizing, or condensing/vaporizing gas drive). The **`MMP`** command will normally use the currently assigned **`PRESsure`** as one of its initial guesses. As it finishes, the **`MMP`** command will set the current **`PRESsure`** to the calculated minimum miscibility pressure.

Argument *`stages:`* this argument, if present, can appear immediately after the **`MMP`** keyword, or after an optional **`STAGES`** (or **`CELLS`**) keyword. It sets the number of *`stages`* to be used during the multistage calculations. Increasing the number of stages can improve the expected accuracy of the solution, but at the expense of increased computation time. The default of 50 stages is usually enough to compute an MMP within about 2% of the true solution, although the displacement mechanism can sometimes be so complicated that it will require more than the usual number of stages to develop fully. It is recommended that important calculations be performed with 50, 100, and perhaps 200 stages, with a careful inspection of the results. For an accuracy within 0.5%, one would rarely require more than 200 stages. The results should not be extrapolated to an infinite number of stages, however. The results will have already been extrapolated and will often approach the true solution non-monotonically. The best one can say is that the likelihood of a given accuracy increases with the number of

stages. For a fixed number of stages, however, the results will be consistent and differentiable with respect to changes in temperature, composition, and the fluid characterization.

Note: If a negative number of *stages* is specified, then the multistage, condensing/vaporizing calculations will not be performed. The minimum pressure for condensing or vaporizing gas drive miscibility will be determined from single-cell calculations (using the absolute value of the specified number of *stages* as the number of contacts), but this pressure may not be the true MMP, so this option should be used only with caution.

Option ID: this option can be used to input a character string to help identify the current **MMP** experiment.

Argument *"identifier":* this can be any character string.

Option STAGES (**Alias: CELLS**)**:** this option can be used as an alternate means of specifying the number of *stages*.

Option DATum (**Alias: MEASurement**)**:** this option can be used to input the experimental MMP, with which the calculated MMP can be compared on output.

Argument *mmp:* the experimental MMP.

Argument unit: The unit of pressure for the experimental MMP is specified by this token, which can by any of the keywords listed in Table 6–2.

Option WEIGht (**Aliases: WT**, **WGT**, **WGHt**)**:** this option can be used as part of the **DATum** option to assign a weight factor to the relative error between the calculated MMP and the experimental MMP. The weighted average of all such errors will be displayed at the end of the output file.

Argument *wt:* the weight factor for the relative error between the calculated MMP and the experimental MMP. By default, this weight factor will be 1.

5.6. MME Command

```
MME gas1 gas2 [stages] [ID "identifier"] [STAGES stages]
    [DATum (mme|(mme %)|(% mme)) [WEIGht wt]]
```

The **MME** command computes the minimum miscibility enrichment of a lean gas with a rich gas for the displacement of the **FEED** tank's fluid at the current **TEMPerature** and **PRESsure**. It uses a proprietary, iterative, multistage, multicontact algorithm to determine the minimum miscibility enrichment no matter what the displacement mechanism (condensing, vaporizing, or condensing/vaporizing gas drive). As it finishes, the **MME** command will fill the **INJectant** tank with one mole of the resulting mixture of two specified gases, **gas1** and **gas2**, having the minimum enrichment for miscibility. **MMPz** will automatically determine the leaner and the richer of these two gases, so the order in which they're specified doesn't matter.

Argument `gas1:` this should be the name of the tank containing the first of the two gases to be mixed into the final injectant.

Argument `gas2:` this should be the name of the tank containing the second of the two gases to be mixed into the final injectant.

Argument *`stages:`* this argument, if present, can appear immediately after the `gas2` token, or after an optional **`STAGES`** (or **`CELLS`**) keyword. It sets the number of *`stages`* to be used during the multistage calculations. Increasing the number of stages can improve the expected accuracy of the solution, but at the expense of increased computation time. The default of 50 stages is usually enough to compute an MME within about 2% of the true solution, although the displacement mechanism can sometimes be so complicated that it will require more than the usual number of stages to develop fully. It is recommended that important calculations be performed with 50, 100, and perhaps 200 stages, with a careful inspection of the results. For an accuracy within 0.5%, one would rarely require more than 200 stages. The results should not be extrapolated to an infinite number of stages, however. The results will have already been extrapolated and will often approach the true solution non-monotonically. The best one can say is that the likelihood of a given accuracy increases with the number of stages. For a fixed number of stages, however, the results will be consistent and differentiable with respect to changes in temperature, composition, and the fluid characterization.

Note: If a negative number of *`stages`* is specified, then the multistage, condensing/vaporizing calculations will not be performed. The minimum enrichment for condensing or vaporizing gas drive miscibility will be determined from single-cell calculations (using the absolute value of the specified number of *`stages`* as the number of contacts), but this enrichment may not be the true MME, so this option should be used only with caution.

Option `ID:` this option can be used to input a character string to help identify the current **`MME`** experiment.

Argument *`"identifier":`* this can be any character string.

Option `STAGES` (**Alias: `CELLS`**)**`:`** this option can be used as an alternate means of specifying the number of *`stages`*.

Option `DATum` (**Alias: `MEASurement`**)**`:`** this option can be used to input the experimental MME, with which the calculated MME can be compared on output.

Argument *`mme:`* the experimental MME, i.e., the minimum fraction (or percentage, if preceded or followed by the `%` argument) of the enriching gas (the richer of `gas1` and `gas2`) in a miscible injectant made up of `gas1` and `gas2`.

Argument `%:` this token, either before or after (and delimited from) the *`mme`* argument, indicates that the experimental *`mme`* is specified as a percentage instead of as a fraction (the default).

Option WEIGht (Aliases: WT, WGT, WGHt): this option can be used as part of the **DATum** option to assign a weight factor to the error between the calculated MME and the experimental MME. The weighted average of all such errors will be displayed at the end of the output file.

Argument *wt:* the weight factor for the error between the calculated MME and the experimental MME. By default, this weight factor will be 1.

5.7. MCM Command

MCM [stages] **[ID** *"identifier"***]** **[STAGES** stages**]**

The **MCM** command computes a measure of *minimum immiscibility* as a function of the displacement stage for a single displacement of the **FEED** tank's fluid by the **INJectant** tank's fluid at the current **TEMPerature** and **PRESsure**. It uses a proprietary, multistage, multicontact algorithm to determine the closest approach to miscibility at each stage for each of the possible displacement mechanisms (condensing, vaporizing, and condensing/vaporizing gas drive). These results can be used to help judge how many stages might be necessary to produce reliable results with the **MMP** or **MME** command.

Argument *stages:* this argument, if present, can appear immediately after the **MCM** keyword, or after an optional **STAGES** (or **CELLS**) keyword. It sets the maximum number of *stages* to be used during the multistage calculations of immiscibility. The default number of *stages* is 50. The recommended use of the MCM command is to see how many stages it takes for the calculated immiscibilities to begin converging predictably. This is the minimum number of stages that should be used in a similar **MMP** or **MME** experiment.

Note: If a negative number of *stages* is specified, then the multistage, condensing/vaporizing calculations will not be performed. Only the condensing and vaporizing gas drive immiscibilities will be determined (from single-cell calculations, using the absolute value of the specified number of *stages* as the number of contacts).

Option ID: this option can be used to input a character string to help identify the current **MCM** experiment.

Argument *"identifier":* this can be any character string.

Option STAGES (Alias: CELLS): this option can be used as an alternate means of specifying the number of *stages*.

5.8. Flash Command

FLASH

The **FLASH** command performs a two-phase equilibrium flash calculation on the contents of the **FEED** tank at the current **TEMPerature** and **PRESsure**. Besides printing its results, it will also fill the **EQLiquid** and **EQVapor** tanks with one mole (each) of the corresponding equilibrium phase and the **NFLiquid** and **NFVapor** tanks with one mole (each) of the corresponding negative flash phase (see the discussion of these tanks and phases in section 5.2). In addition, the **FEED** tank will exit the **FLASH** command containing exactly one mole of its original contents.

5.9. Saturation Pressure Command

SATPressure

Aliases: **PSAT***, **BUBP***, **PBUB***.

The **SATP*** command finds (when possible) the *upper* saturation pressure of the contents of the **FEED** tank at the current **TEMPerature**, using the current **PRESsure** (if positive) as its initial guess. In addition to reporting its results, it will change the current **PRESsure** to the calculated saturation pressure. It will also fill the **EQLiquid** and **EQVapor** tanks (as well as the **NFLiquid** and **NFVapor** tanks) with one mole (each) of the appropriate equilibrium phase (see the discussion of these tanks and phases in section 5.2). In addition, the **FEED** tank will exit the **SATP*** command containing exactly one mole of its original contents.

If the fluid appears to be single phase at all pressures, the **SATP*** command will issue a warning and calculate a pseudo vapor pressure instead of a true saturation pressure (see the **VAPP*** command below). If a saturation pressure does exist in such a case, a better initial guess will be required to find it.

If the fluid appears to have an infinite saturation pressure, the **SATP*** command will issue a warning and report the two-phase equilibrium found at the last (extremely high) pressure tested.

5.10. Dew Point Pressure Command

DEWPressure

Alias: **PDEW***.

The **DEWP*** command finds (when possible) the *lower* saturation pressure (i.e., the lower dew point pressure) of the contents of the **FEED** tank at the current **TEMPerature**, using the current **PRESsure** (if positive) as its initial guess. In addition to reporting its results, it will change the current **PRESsure** to the calculated dew point pressure. It will also fill the **EQLiquid** and **EQVapor** tanks (as well as the **NFLiquid** and **NFVapor** tanks) with one mole (each) of the appropriate equilibrium phase (see the discussion of these tanks and phases in section 5.2). In addition, the **FEED** tank will exit the **DEWP*** command containing exactly one mole of its original contents.

If the fluid appears to be single phase at all pressures, the **DEWP*** command will issue a warning and calculate a pseudo vapor pressure instead of a true saturation pressure (see the **VAPP*** command below). If a dew point pressure does exist in such a case, a better initial guess will be required to find it.

5.11. Vapor Pressure Command

VAPPressure

Alias: VAPR*.

The **VAPP*** command finds the *pseudo vapor pressure* of the contents of the **FEED** tank at the current **TEMPerature**.

For a pure component at a temperature below its critical, the **VAPP*** command will find the component's true vapor pressure. Above the component's critical temperature, however, it will find a pressure that is on a continuous extension of the component's vapor pressure (versus temperature) curve. This will be a pseudo vapor pressure.

The **VAPP*** command treats mixtures as though they were single components (with properties that have been averaged by the equation of state's mixing rules). It finds the pseudo vapor pressure of this pseudo component in the manner described above for true components. The fluid will be treated as a single phase, but at the transition between liquid and vapor. The **VAPP*** command will not attempt to perform a true phase split calculation (that can always be done by a subsequent **FLASH** command).

In addition to reporting its results, the **VAPP*** command will change the current **PRESsure** to the calculated pseudo vapor pressure. It will also fill the **EQLiquid** and **EQVapor** tanks (as well as the **NFLiquid** and **NFVapor** tanks) with one mole (each) of the overall **FEED** mixture (see the discussion of these tanks in section 5.2). In addition, the **FEED** tank will exit the **VAPP*** command containing exactly one mole of its original contents.

5.12. Title Command

TITLe *"title"* **[TITLe** *"subtitle"***]…**

The **TITLe** command causes a single or multiple line title to be printed to the output file, in sequence with the output of the surrounding commands. It is similar to the **NOTE*** command below, but titles and notes are formatted differently.

Argument *"title":* this can be any character string. It will form the title (or the first line of a multiple line title).

Subcommand TITLe (**Alias: SUBTitle**)**:** this optional subcommand allows subtitles to be added to the main title to form a multiple line title. It can be repeated as often as desired.

Argument *"subtitle":* this can be any character string. It will form a subtitle line in a multiple line title.

5.13. Note Command

NOTE* *"note"* **[NOTE*** *"subnote"]…**

Alias: COMMent.

The **NOTE*** command causes a single or multiple line note to be printed to the output file, in sequence with the output of the surrounding commands. It is similar to the **TITLe** command above, but notes and titles are formatted differently.

Argument *"note":* this can be any character string. It will form the note (or the first line of a multiple line note).

Subcommand NOTE* (**Alias: COMMent**)**:** this optional subcommand allows *subnotes* to be added to the main note to form a multiple line note. It can be repeated as often as desired.

Argument *"subnote":* this can be any character string. It will form a *subnote* line in a multiple line note.

5.14. TEST2 Command

TEST2* [SUSPect|ALways|NEVer]

The **TEST2*** command dictates the circumstances under which **MMPz** will test the thermodynamic stability[1] of subsequently calculated 2-phase and negative flash equilibria, looking for the presence of 3-phase equilibria or a more stable 2-phase solution.

MMPz is not designed to calculate 3-phase equilibria, but it does have the ability to detect when a third equilibrium phase is due to appear. It can be very useful to know about such situations (so they can perhaps be avoided), because they will create severe numerical difficulties for most reservoir simulators.

MMPz can detect the imminent appearance of a third equilibrium phase only by testing a calculated 2-phase equilibrium solution for thermodynamic stability. This would normally be a waste of time, since 3-phase equilibria are not usually found at typical reservoir conditions. If the conditions of interest are somewhat atypical, however (low temperatures combined with high concentrations of carbon dioxide, for example), then **MMPz** can be directed to test all of its 2-phase solutions.

Whenever **MMPz** tests its 2-phase solutions for thermodynamic stability, it will also test its negative flash solutions. If any solutions are found to be thermodynamically

[1] Michelsen, M.L.: "The Isothermal Flash Problem. Part I. Stability," *Fluid Phase Equilibria* **9** (1982), 1.

unstable, then **MMPz** will find a new 2-phase solution that is more stable. The testing process will be repeated until the most stable 2-phase solution is found. If the most stable 2-phase solution is still unstable, **MMPz** will report on the fact that true equilibrium would require at least three phases.

Option `SUSPect` (**Alias: `DEFault`**)**:** this **`DEFault`** option causes **MMPz** to test its 2-phase and negative flash solutions for thermodynamic stability only when it has its own reasons to **`SUSPect`** the possibility of multiple 2-phase solutions. This should cover the most obvious circumstances for multiple solutions and 3-phase equilibria, but certainly not all.

Option `ALways` (**Aliases: `ON`**, **`MAXimum`**)**:** this option causes **MMPz** to **`ALways`** test its 2-phase and negative flash solutions for thermodynamic stability. It will detect all of the multiple and 3-phase solutions it can, but at the expense of additional computation time.

Option `NEVer` (**Aliases: `NOne`**, **`OFF`**, **`MINimum`**)**:** this option causes **MMPz** to **`NEVer`** test its 2-phase and negative flash solutions for thermodynamic stability, even when it suspects the possibility of multiple solutions. This option should be used only if multiple solutions or 3-phase equilibria are known to be absent or of no importance.

5.15. TEST1 Command

`TEST1* [SUSPect|ALways|NEVer]`

The **`TEST1*`** command dictates the circumstances under which **MMPz** will test the thermodynamic stability of *trivial* solutions from subsequent phase equilibrium calculations.

A 2-phase flash calculation can result in 2-phase equilibrium, negative flash equilibrium, or a *trivial* solution. A trivial solution is one in which the two resulting phases are identical. A trivial solution can always be found, but the objective is to find a nontrivial solution, if one exists.

If a nontrivial flash solution exists, then **MMPz** will almost always find it, given a reasonable initial guess. If it finds a trivial solution, instead, it almost always means that there is no other solution. Very infrequently, however, it means that the initial guess wasn't good enough. To make absolutely sure that there is no 2-phase solution (at the expense of additional calculations), **MMPz** can be directed to test the thermodynamic stability of the trivial solution.

Option `SUSPect` (**Alias: `DEFault`**)**:** this **`DEFault`** option causes **MMPz** to test its trivial flash solutions for thermodynamic stability only when it has its own reasons to **`SUSPect`** the existence of a nontrivial solutions. This should cover the most obvious circumstances where nontrivial solutions are expected, but not all.

Option `ALways` (**Aliases: `TRIVial`**, **`CRITical`**, **`ON`**, **`MAXimum`**)**:** this option causes **MMPz** to **`ALways`** test its trivial flash solutions for thermodynamic stability. If there is a 2-phase solution, this will ensure its detection. This is the safest option, but it

requires additional computations and it is very seldom necessary. It is most likely to be useful for sequences of flash calculations that are not very closely related, where the initial guesses might be poor.

Option `NEVer` (**Aliases: `NOne`, `OFF`, `MINimum`**)**:** this option causes **MMPz** to **`NEVer`** test its trivial flash solutions for thermodynamic stability, even when it suspects the existence of a nontrivial solution. The only purpose for this option is to force the tracking of a particular solution branch in a system with multiple solutions, which is usually of academic interest only.

5.16. INIT2 Command

`INIT2* [PRevious|WILson|STability|SKIP]`

The **`INIT2*`** command determines how **MMPz** will attempt to initialize its subsequent equilibrium flash calculations.

Option `PRevious` (**Aliases: `Kvalues`, `DEFault`**)**:** this **`DEFault`** option causes **MMPz** to initialize each equilibrium flash calculation with the **`K-values`** from the solution of the **`PRevious`** equilibrium calculation, whenever it's nontrivial. If that's not possible, it automatically reverts to the **`STability`** option. The **`PRevious`** option is usually the fastest. It will also find negative flash solutions as well as 2-phase solutions. Very infrequently, it may converge to an unwanted trivial solution. If that's a concern, the **`ALways`** option of the **`TEST1*`** command can be used as a safeguard.

Option `WILson`: this option causes **MMPz** to initialize each equilibrium flash calculation with the K-values from the commonly used **`WILson`** equation.[1] This option is not generally recommended, however.

Option `STability`: this option causes **MMPz** to initialize each equilibrium flash calculation by testing the thermodynamic **`STABility`** of the overall, single-phase fluid. This option is practically guaranteed to find a 2-phase solution, if one exists, but it is usually slower than the **`PRevious`** option, and it is much less likely to find any negative flash solutions. The **`STability`** option is very safe at finding 2-phase solutions, but the same safety can be achieved by combining the **`PRevious`** option with the **`ALways`** option of the **`TEST1*`** command.

Option `SKIP` (**Aliases: `NEVER`, `NOne`, `OFF`**)**:** this option causes **MMPz** to **`SKIP`** all subsequent phase split calculations until directed otherwise. It will instead assume that all fluids remain single-phase.

5.17. Stability Command

`STABility [ON|OFF]`

[1] Whitson, C.H., and Brulé, M.R.: *Phase Behavior,* Monograph Series, SPE, Richardson, Texas (2000), 42.

The **STABility** command will direct **MMPz** to use thermodynamic stability testing as much as possible, or only as needed. It is actually a shortcut for setting common options of the **INIT2***, **TEST1***, and **TEST2*** commands simultaneously.

Option ON (Aliases: YES, Y, TRUE, T): this option is equivalent to the following sequence of commands:

```
INIT2 Stability, TEST1 Always, TEST2 Always
```

Option OFF (Aliases: NO*, N, FALSE, F): this option is equivalent to the following sequence of commands:

```
INIT2 Default, TEST1 Default, TEST2 Default
```

5.18. Equation of State Command

EOS [PR79|PR77|SRK|RK]

Alias: EQUA*.

The **EOS** command sets the default equation of state for subsequent fluid characterizations (although each characterization is free to override this default with its own **EOS** option).

Option PR79: the 1979 version of the Peng-Robinson EOS will now be the default.

Option PR77: the 1977 version of the Peng-Robinson EOS will now be the default.

Option SRK: the Soave-Redlich-Kwong EOS will now be the default.

Option RK: the Redlich-Kwong EOS will now be the default.

5.19. Tabs Command

TABS *positions*

The **TABS** command tells **MMPz** how many character *positions* are represented by each tab character in the input file. This command allows **MMPz** to correctly read tables that were visually aligned in a text editor with a combination of tabs and spaces (see the complete discussion of this subject in 3.8.1).

Argument *positions:* this should be the number of character *positions* represented by each tab. This should correspond to the tab setting in the text editor that was used to align subsequent tables.

5.20. End-of-File Command

EOF

The **EOF** command indicates that the end of the current input file has been reached. Anything in the file after this command will be ignored.

5.21. End Command

END

This command can be used (instead of a blank line) to signal the **END** of a table. It can also be used to signal the **END** of the previous command, in the rare case of ambiguities. For example, consider the following sequence of commands:

```
Mix temp feed
Mix feed oil
Temp 60 F
Flash
Mix feed temp
```

The intent may have been to save the contents of the **FEED** tank in a temporary tank called `temp`, then to change the contents of the **FEED** tank, perform a **FLASH** calculation at a new **TEMPerature** of 60 F, and then to restore the original contents of the **FEED** tank. Because of the unfortunate choice of `temp` for a tank name, however, **MMPz** will interpret this sequence of commands as:

```
Mix temp: 1 tank feed END
Mix feed: 1 tank oil, 60 tanks temp END
F END; ERROR! Command F not recognized!
Flash END
Mix feed: 1 tank temp END
```

Without changing the name of the temporary tank, the original sequence would have been okay with the insertion of a single **END** command, to make sure the **TEMPerature** command was not misinterpreted as the tank name, `temp`:

```
Mix temp feed
Mix feed oil END
Temp 60 F
Flash
Mix feed temp
```

5.22. Include Command

INCLude *"filename"*

The **INCLude** command causes **MMPz** to pause in its reading of the current input file, to read another input file (as though its input were **INCLuded** in the original file), and then to return to reading the rest of the original file.

The **INCLude** command can be nested to any desired level (**INCLuded** files can **INCLude** other files, which can also **INCLude** files, and so on). It can be issued wherever any command is expected, and in most instances wherever a subcommand is expected. In the latter case, the previous command will continue looking for subcommands within the newly **INCLuded** file.

Argument *"filename":* this can be any character string specifying the name of the file to be **INCLuded**. It can specify either a full path name or a partial path name to the file, as long as it is recognized by the computer's operating system. Partial paths should be specified relative to the *current directory*. This is normally the directory in which the current input file (the one issuing the current **INCLude** command) resides. The current directory can be changed, however, with the **CD** command (see below). Anything on the end of the line (following the *"filename"* argument) will be ignored.

For an example, suppose a computer is running a Windows operating system and contains the following files:

```
D:\MMPz\Data\input.dat
D:\MMPz\Data\expts.dat
D:\MMPz\Data\Fluids\fluid.dat
D:\MMPz\sample.dat
D:\MMPz\Tests\test.dat
E:\MMPz\Examples\example.dat
```

These **INCLude** commands would all work within the `input.dat` file:

```
Include "expts.dat"
Include "Fluids\fluid.dat"
Include "\MMPz\sample.dat"
Include "D:\MMPz\sample.dat"
Include "..\sample.dat"
Include "..\Tests\test.dat"
Include "E:\MMPz\Examples\example.dat"
```

Now suppose a computer is running a Macintosh operating system and contains the following files:

```
D:MMPz:Data:input.dat
D:MMPz:Data:expts.dat
D:MMPz:Data:Fluids:fluid.dat
```

```
D:MMPz:sample.dat
D:MMPz:Tests:test.dat
E:MMPz:Examples:example.dat
```

These **INCLude** commands would all work within the `input.dat` file:

```
Include "expts.dat"
Include ":expts.dat"
Include ":Fluids:fluid.dat"
Include "D:MMPz:sample.dat"
Include "::sample.dat"
Include "::Tests:test.dat"
Include ":::MMPz:Tests:test.dat"
Include "E:MMPz:Examples:example.dat"
```

5.23. Current Directory Command

CD *"directory"*

The **CD** command can make it easier to use the **INCLude** command. When an input file **INCLudes** another file, the path to that file needs to be specified relative to the *current directory*. This is normally the directory in which the current input file (the one issuing the current **INCLude** command) resides. However, the **CD** command can make any other directory the *current directory*. Subsequent **INCLude** and **CD** commands would then search for files and directories within (or relative to) this new current directory.

The effect of the **CD** command continues until the next **CD** command or until the end of the current file, whichever comes first. It will not carry back to any file that may have **INCLuded** the current file. It can be issued wherever any command is expected, and in most instances wherever a subcommand is expected. In the latter case, the previous command will continue looking for subcommands after the **CD** command has been processed.

Argument *"directory":* this can be any character string specifying the full or partial path, relative to the *current directory* (initially, the one in which the current input file resides), to a *new current directory*. The path must be recognized as a legal directory path by the computer's operating system. Subsequent **INCLude** and **CD** commands within the current input file must then specify their file and directory path arguments relative to this new current directory.

For an example, suppose a computer is running a Windows operating system and contains the following files:

```
D:\MMPz\Data\input.dat
D:\MMPz\Data\expts.dat
D:\MMPz\Data\Fluids\fluid.dat
D:\MMPz\sample.dat
```

```
D:\MMPz\Tests\test.dat
E:\MMPz\Examples\example.dat
```

This sequence of commands would work within the `input.dat` file:

```
CD "Fluids"
Include "fluid.dat"
CD ..
Include expts.dat
CD "..\Tests"
Include "test.dat"
CD "\MMPz\"
Include "sample.dat"
CD "E:\MMPz\Examples"
Include "example.dat"
```

Now suppose a computer is running a Macintosh operating system and contains the following files:

```
D:MMPz:Data:input.dat
D:MMPz:Data:expts.dat
D:MMPz:Data:Fluids:fluid.dat
D:MMPz:sample.dat
D:MMPz:Tests:test.dat
E:MMPz:Examples:example.dat
```

This sequence of commands would work within the `input.dat` file:

```
CD ":Fluids"
Include "fluid.dat"
CD "::"
Include expts.dat
CD "::Tests"
Include "test.dat"
CD "D:MMPz:"
Include "sample.dat"
CD "E:MMPz:Examples"
Include "example.dat"
```

5.24. Define Command

DEFine *"macro" "replacement"*

This command **DEFines** a *"macro"* string and its *"replacement"* string. From this point on, any string of characters *?macro?* (the *"macro"* string quoted by question marks, regardless of the surrounding characters) will be replaced by the text of the *"replacement"* string.

Argument *"macro":* any character string.

Argument *"replacement":* any character string.

Examples:

```
Define "std temp" "60 F"

Temp ?std temp? ; equivalent to: Temp 60 F

Define "dir name"  "Test Files"
Define "root name" "test"
Define "extension" ".dat"

Include "?dir name?\?root name??extension?"
Include "?dir name?\?root name?1?extension?"

: The two previous commands were equivalent to:
:   Include "Test Files\test.dat"
:   Include "Test Files\test1.dat"
```

5.25. Echo Command

ECHO [ON|OFF]

The **ECHO** command determines whether subsequent lines of input will be copied, or *echoed*, to the standard output file. **ECHO** is **OFF**, initially.

Option ON (Aliases: YES, Y, TRUE, T): this option activates the echoing.

Option OFF (Aliases: NO*, N, FALSE, F): this option deactivates the echoing.

5.26. Timing Command

TIMing

The **TIMing** command causes the current execution time (at the moment the command is executed) to be reported.

Chapter 6. Units

The tables in this chapter list the keywords for the available units of each physical type understood by **MMPz**.

Table 6–1. Temperature Units

Units	Keyword(s)
Celcius	`C, CEL*`
Fahrenheit	`F, FAH*`
Kelvin	`K, KEL*`
Rankine	`R, RAN*`

Table 6–2. Pressure Units

Units	Keyword(s)
Atmospheres (absolute)	`ATM, ATMA`
Atmospheres (gauge)	`ATMG`
Bar (absolute)	`BAR, BARA`
Bar (gauge)	`BARG`
Pascal (absolute)	`PA, PAA`
Pascal (gauge)	`PAG`
KiloPascal (absolute)	`KPA, KPAA`
KiloPascal (gauge)	`KPAG`
MegaPascal (absolute)	`MPA, MPAA`
MegaPascal (gauge)	`MPAG`
Pounds per square inch (absolute)	`PSI, PSIA`
Pounds per square inch (gauge)	`PSIG`
Torr (absolute)	`TORR, TORRA`
Torr (gauge)	`TORRG`

Table 6–3. Molar Volume Units

Units	Keyword(s)
Cubic centimeters per gram-mole	CM3/MOL, CM3/GMOL, CC/MOL, CC/GMOL
Cubic meters per kilogram-mole	M3/KMOL, M3/KGMOL
Liters per gram-mole	L/MOL, L/GMOL
Liters per kilogram-mole	L/KMOL, L/KGMOL
Cubic decimeters per kilogram-mole	DM3/KMOL, DM3/KGMOL
Cubic feet per pound-mole	FT3/LBMOL, CF/LBMOL
Gallons per pound-mole	GAL/LBMOL
Barrels per pound-mole	BBL/LBMOL
Thousands of cubic feet per pound-mole	MCF/LBMOL
Millions of cubic feet per pound-mole	MMCF/LBMOL
Cubic meters per standard cubic meter	M3/SM3
Liters per standard cubic meter	L/SM3
Cubic decimeters per standard cubic meter	DM3/SM3
Cubic centimeters per standard cubic meter	CM3/SM3, CC/SM3
Cubic feet per standard cubic foot	FT3/SCF, CF/SCF, MCF/MSCF
Cubic feet per thousand standard cubic feet	FT3/MSCF, CF/MSCF, MCF/MMSCF
Gallons per standard cubic foot	GAL/SCF
Gallons per thousand standard cubic feet	GAL/MSCF
Gallons per million standard cubic feet	GAL/MMSCF
Barrels per standard cubic foot	BBL/SCF
Barrels per thousand standard cubic feet	BBL/MSCF
Barrels per million standard cubic feet	BBL/MMSCF

Chapter 7. Running MMPz

MMPz is available as a 32-bit Windows application, or as a Macintosh application. This chapter provides instructions for running **MMPz** on these two platforms. Unix or Linux versions of **MMPz** may be available by special request. Contact **Zick Technologies** for more information.

7.1. Windows Version

MMPz consists of a single, executable file. It can be installed in any directory on a Windows system, although it might be a good idea to install it among other, similar applications. It may also be a good idea to place a Shortcut to **MMPz** in one or more convenient locations (on the Desktop, for example). If **MMPz** is to be launched from a command line prompt, it may prove convenient to add its location to the environment variable, `PATH`, in the `autoexec.bat` file. If these suggestions aren't clear, ask a Systems Administrator for assistance.

MMPz can be launched in several different ways. The easiest is probably to double-click on its icon (or on that of a Shortcut—from now on, this will go without saying). Dialog boxes will then allow the user to specify the input and output files.

Another method is to drag the icon for the input file onto **MMPz**'s icon. A dialog box will then allow the user to specify the output file.

The user can also select a pair of file icons and drag them together onto the **MMPz** icon. The file whose icon is under the cursor during the dragging operation will be opened as the input file. The other file will be overwritten as the output file. Be careful not to confuse the two!

MMPz can also be launched from a command line by entering the name of the program at the prompt. Depending on the location of the program, the current working directory, and the current setting of the `PATH` variable, the name may or may not have to include a path to the program's directory. Again, if this is confusing, ask a Systems Administrator for assistance. Up to two arguments can follow the program name on the command line. The first argument would be the name of the input file. Its path should be specified relative to the current working directory. The second argument would be the name of the output file, again specified relative to the current working directory. If less than two file names are entered, the user will be prompted for the missing information.

Whenever it's necessary for **MMPz** to prompt the user for the name of an output file, it will generate a suggested name from the name of the input file, usually by changing its extension to `".out"`. In order to see the extensions (so as not to confuse output files with input files), it may be necessary to deselect the "Hide file extensions…" check box in the View panel of the "Folder Options…" dialog, which can be opened from the View menu of any window (again, ask a Systems Administrator for assistance, or better yet, buy a Mac).

7.2. Macintosh Version

MMPz consists of a single, application file. It can be installed in any folder on a Macintosh system, although it might be a good idea to install it among other, similar applications. It may also be a good idea to place an Alias to **MMPz** in one or more convenient locations (on the Desktop or in the "Apple Menu Items" folder, for example).

MMPz can be launched by double-clicking on its icon (or that of an Alias). If it (or an Alias) has been placed in the "Apple Menu Items" folder, then it can also be launched from the Apple Menu. Once launched, dialog boxes will allow the user to specify the input and output files.

7.3. Console I/O

Regardless of platform, when **MMPz** is launched, it will open a console window on the screen. As it is running, it will continuously write progress information to that window. When the program is finished, the window can be dismissed.

If the standard output file is named "@" (without the quote marks), then all output will be written only to the console window. This is probably not very practical, however, except perhaps for very small jobs.

If no input file is specified (i.e., if the dialog box for the input file is canceled), then input will be accepted interactively from the console window. This is practical only for entering a small number of commands (perhaps a few `INCLude` commands).

7.4. Input Files

Regardless of platform, **MMPz** can read input files created on any other platform. There is no restriction on file size or line length.

Files can be created in text editors, word processors, or spreadsheet programs. They should always be saved as text-only files, however. Tab-delimited text files are perfectly acceptable, as long as the advice in section 3.8.1 is followed.

For **MMPz** to open an input file, the file must not be open in any other application. This ensures that the file cannot be accidentally overwritten while in use. It also makes it impossible for an input file to `INCLude` itself, which would result in an infinite loop—never a good thing.

7.5. Problems

MMPz has been thoroughly tested, but no program can be guaranteed free of bugs. Please report any problems to **Zick Technologies**. If there are any defects in the program, they will be promptly fixed.