Working With Dates and Times

Working with dates and times can be a complicated task. Time is pretty simple if all you are measuring is seconds and fractions of a second, but it gets crazy when that starts rolling over into hours, days, months, years, etc. And don't forget that you have to take into account different length months plus leap year and daylight saving time.

Date and time are just two forms of the same thing – only the units change. In QuickCalc BASIC, we define a new type of variable called the **DATETIME variable**, which can contain time values ranging from microseconds up to billions of years. (The earth is around 5 billion years old, so I chose that for the maximum.)

DATETIME Variables

A DATETIME variable is just a DOUBLE (floating-point) variable with a particular meaning. The number represents the time (in seconds) since January 1, 1970 (this is for compatibility with Microsoft's routines).

Since this is a floating-point value, you can have fractions of a second to the right of the decimal point (see "precision", below).

Years may range from 5,000,000,000 BC to 5,000,000,000 AD.

Time is internally stored in Greenwich Mean Time and converted to local time and adjusted for daylight saving time when it is printed out.

Leap year is calculated. There is no year 0 (Gregorian calendar) so negative years are compensated (increased by 1) to take this into account and make leap-year calculations correct.

DATETIME variables may be elements in a DOUBLE array, and may be used like any other number, but be careful – performing arithmetic operations on DATETIME variables may cause them to lose their meaning (see "Operating on DATETIME variables", below).

Creating DATETIME Variables

You can create a DATETIME variable with the function **MAKEDATETIME** ( ), using values or numbers for month, day, year, hours, minutes, and seconds as follows:

mydatetime = **MAKEDATETIME** (*month, day, year, hour, minutes, seconds, dst*)

(**Note** the order of the parameters – *month* is first)

month	1 to 12

day	1 to 31 (or 30 or 28 or 29, depending on the month)
year	-5000000000 to 5000000000, excluding year 0. Negative numbers are B.C.
hour	00 (midnight) to 23 (11 PM)
minutes	00 to 59
seconds	00 to 59.999999…
dst	daylight saving time flag*:* 0 = time is in standard time 1 = time is in daylight saving time -1 (default) = system determines if DST is in effect. (normally, this parameter is not used)

All values reflect local time as set on your computer, taking into account leap years and daylight saving time, if appropriate.

Any parameters not specified are assumed to be zero, except *dst* which defaults to -1. If the year is not specified or is zero, it will be assumed to be the current year. Be careful, as Jan. 1 will not follow Dec. 31 if both assume the current year.

Values should be valid for the given date/time. Parameters which are out of range may have undefined results. Dec 32 will yield Jan 1 of the following year, but not all errors are that well behaved.

The DATETIME is computed in seconds since Jan 1, 1970 and adjusted to Greenwich Mean Time. The result is stored in the destination variable, which must be a type DOUBLE.

Note: The only time you need to use the *dst* parameter is to distinguish between 1: AM DST and 1: AM standard time on the day daylight saving time ends (see Daylight Saving Time notes, below).

Using the Current Date and/or Time

You can create a DATETIME variable using the current date and/or time, or elements of the current date and/or time.

If you set any one of the parameters (except *year*) to -1, that parameter will be replaced with the current date or time as stored in the **DATE$** and **TIME$** variables. To set the current year, use 0 for the *year* value.

Examples:

MAKEDATETIME ( )	sets today at current time,
MAKEDATETIME (-1, -1, 0)	sets current date at midnight (00:00:00),
MAKEDATETIME (-1, 1, 0)	sets the first day of the current month

MAKEDATETIME (1, 1, 0)	sets the first day of the current year,
MAKEDATETIME (-1,-1, 0, 12, 0, 0)	sets today at noon,
MAKEDATETIME (-1,-1,0, -1, 0, 0)	sets today at start of current hour,
MAKEDATETIME (-1,-1,0,-1,-1,-1)	sets today at current time,

<u>**Note**</u>: The current second is an integer.

<u>**Note**</u>: The current date and time (**DATE$** and **TIME$**) are set when QuickCalc starts. If you want a more accurate current time, do **UPDATEDATETIME** before using those values.

<u>**Note**</u>: The current date and time are obtained using the computer's system clock, and are accurate only <u>**to the second**</u>, and are no more accurate then your computers clock setting.

<u>Printing a DATETIME Variable</u>

If you simply print the DATETIME variable, it will simply appear as a large number which makes no sense. There are two ways to print the variable so that you can understand it,

1. Print the formatted DATETIME variable as follows:

PRINT FORMATDATETIME$ (DATETIME *variable*)

This prints a string like this:

```
Wed  Apr  08  12:35:36.123445  DST          1942
```

(DST indicates Daylight Saving Time. For standard time, this field is replaced with blanks.)

This format allows for the entire range of years and also shows microseconds.

You can also assign **FORMATDATETIME$** ( ) to a string variable and perform any string operations you want to change the format.

<u>**Note:**</u> **FORMATDATETIME$** ( ) [*no parameter*] returns a formatted string containing the current date and time. The current date and time (**DATE$** and **TIME$**) are set when QuickCalc starts. If you want a more accurate current time, do **UPDATEDATETIME** before using those values.

2. You can convert the DATETIME variable to an array (see below), and format the individual elements any way you want.

The DATETIME Array

QuickCalc can represent a Date and Time by its component parts in the form of an array, referred to as the **DATETIME array**.

The array is one-dimensional and contains exactly 9 elements. The array is type DOUBLE.

The items in the array are as follows:

Index	*Meaning*	*Range*
0	Year	-5000000000 to +5000000000
1	Month	1-12 (1 = January)
2	Day of Month	1-31 (depending on the month)
3	Hour	00-23
4	Minute	00-59
5	Seconds	00. to 59.999999…
6	Day of Week	1-7 (1 = Sunday)
7	Day of Year	1-365 (or 366) "Julian" Day
8	DST flag	0 = not using daylight-saving time >0 = using DST <0 = DST is unknown

Note: All the array values are integers, except for the seconds, which may contain information to the right of the decimal. The precision of this value will vary, depending on the size of the year (see "Precision", below.)

Note: Years are based on the Gregorian calendar, which does not contain a "year 0" (1 B.C. is followed immediately by 1 A.D.). This is taken into account when calculating leap-years, etc.

Expanding the DATETIME Variable

You can **expand** the DATETIME variable into its component parts, and then use or examine each part separately. The parts are stored in elements in an array. To expand a DATETIME variable, you must first dimension a one-dimensional array with exactly 9 elements, as follows:

DIM *mydatetimearray* (9) [give the array any name you like]

Now, assume you have a DATETIME variable called *mydatetime*, which contains a date and time you want to expand. Use the statement:

mydatetimearray = **EXPANDDATETIME** (*mydatetime*)

This calculates the values for all the elements in the DATETIME array, adjusting for leap-years and compensating for the time zone set on your computer and for daylight-saving time, if appropriate.

You can now reference the array values individually. For example, you could print them all out with the statements:

```
FOR i = 0 TO 8
PRINT mydatetimearray (i)
NEXT
```

Note: **EXPANDDATETIME** ( ) returns an array containing the current date and time. The current date and time (**DATE$** and **TIME$**) are set when QuickCalc starts. If you want a more accurate current time, do **UPDATEDATETIME** before using those values.

Note: **EXPANDDATETIME** is not valid in an expression, since it returns an array. It is only valid as the single element in an assignment statement where it is assigned to an array.

Creating a DATETIME Variable From A DATETIME Array

You can use the function **MAKEDATETIME** ( ) to convert the DATETIME array into a DATETIME Variable, as follows:

mydatetime = **MAKEDATETIME** (*mydatetimearray*)

The array *mydatetimearray* must be one-dimensional with 9 elements, and **all of** the first 6 elements must be correctly filled-in, either by assigning them values or by converting a DATETIME with **EXPANDDATETIME** ( ). The *day-of-week* and *day-of-year* are ignored. The DST-flag tells the system whether the DATETIME array refers to a standard or daylight saving time value (see **MAKEDATETIME**, above). If you don't know, set the DST-flag to -1.

Operating on DATETIME Variables

Besides using them to create more meaningful graphs, you can do ***some*** arithmetic on DATETIME variables. For example:

dt + 10	gives 10 seconds later
dt + 10*60*60	gives 10 hours later
dt + 90*24*60*60	gives 90 days later
dt – 30*24*60*60	gives 30 days earlier
CINT (*dt*)	rounds to the nearest second

dt1 – dt2	calculates the difference (in seconds) between two DATETIME variables.

These **<u>do</u>** take into account leap-years, etc, however if you cross a daylight-saving time change, the hour may be different since we are adjusting the time by exactly the increment specified, not according to the calendar.

Another way of changing dates is to **<u>work on the elements of the DATETIME array</u>**, and then convert it back. Assume you have converted a DATETIME variable to a DATETIME array *dtarray*,

dtarray (0) = *dtarray* (0) + 1	Increases the year by 1.
dtarray (1) = *dtarray* (1) + 3	Advances to the same day/time 3 months later. If the day is beyond the end of the month, it will be fixed when converting back (e.g., June 31 will become July 1).
dtarray (1) = *dtarray* (1) - 3	Goes back to the same day/time 3 months earlier. If the day is beyond the end of the month, it will be fixed when converting back (e.g., June 31 will become July 1) .

<u>Note</u> that Dec 31 + 4 months = Apr 31 = May 1, however May 1 – 4 months = Jan. 1.

dtarray (2) = *dtarray* (2) + 7	Advances one week.

Changing the date this way does not affect the local time when crossing DST changes. The new time will be the same time as before, but it may be wrong (i.e., off by one hour) when viewed in Greenwich Mean Time. (Don't you just *love* daylight saving time?)

After making this kind of adjustments, you will want to convert the array back to a DATETIME variable, which will correct for any range errors.

<u>"Julian" Days</u>

Often you may want to work with "Julian" days, which are the absolute day of the year. January 1 is Julian day 001; February 1 is Julian day 032, etc.

To specify a Julian day, simply use January for the month and the Julian day for the day. For example, specify January 35 for the Julian day corresponding to Feb. 4. (You have to be aware of leap-year if you are doing this.) The date will be converted correctly. When it is printed out, it will appear in the normal format (e,g, Feb 4). You can obtain the Julian day for a DATETIME variable by expanding it into an array (see "The DATETIME Array", above).

Using DATETIME Variables with the Timer

If you want to set DATETIME variables using "real-time" timer values, you can combine a DATETIME variable with the results of **STARTTIMER / ENDTIMER**. The **ENDTIMER** function returns it interval in microseconds, so you will have to convert it into seconds before adding it to (or subtracting it from) the DATETIME variable.

Example:

```
UPDATEDATETIME
starttime = MAKEDATETIME ( )        // current date & time
rc = STARTTIMER
(action or subroutine to be timed)
difference = ENDTIMER
endtime = starttime + difference / 1000000
PRINT "Start Time", FORMATDATETIME$ (starttime)
PRINT "End  Time", FORMATDATETIME$ (endtime)
```

Keep in mind that timer functions include the time to process and interpret the BASIC statements, and therefore accuracy is limited by the complexity of the program and the speed of your computer (See "Debugging"). Avoid placing **PRINT**, **INPUT**, or **STOP** statements inside the timing interval unless you want to time them as well.

If you are timing a sequence of events such as the times for different portions of a program, you can use one **STARTTIMER** at the beginning and an **ENDTIMER** at each desired checkpoint. The times will all be relative to the **STARTTIMER**. Then each could be added to a DATETIME variable which was initialized just before the **STARTTIMER**, creating a set (or array) of DATETIME values.

Writing DATETIME Variables To A File.

Remember, DATETIME variables are just floating-point (DOUBLE) variables. You can **WRITE** them to a file just like any other variable, and use **INPUT #** to read them back in later.

Of course, the number written to the file will not make sense to you if you examine the file, but when read it back, it will be correctly interpreted as a DATETIME variable.

Note: There may be a slight round-off error when the value is formatted for **WRITE** and re-converted for **INPUT**. This is the same as when writing and reading regular DOUBLE variables.

You can also write out the contents of a DATETIME **array** as a group of DOUBLE values:

```
FOR i = 0 TO 8
WRITE # 2; mydatetimearray ( i )
NEXT
```

The string produced from **FORMATDATETIME$** can also be written to a file, although decoding it after it is read back is more difficult:

```
WRITE # 2; FORMATDATETIME$ (mydatetimevariable)
```

Precision

DATETIME variables are signed DOUBLE floating-point numbers, representing the date and time as seconds (based on Jan 1, 1970). As floating-point numbers, the precision of the number is approximately 16 digits. The larger the magnitude of the year, the less precision you have in the seconds.

For "modern" dates (1900 – 2100), the precision will be better than 1 microsecond. As the dates get larger (farther from 1970 in either direction), the number will have less precision.

An approximate calculation for the precision is:

correct digits to the right of the decimal = 9 – digits in year.

For example, if the year is 1,000,000 there are 7 digits in the year, so you can expect about 9 – 7 = 2 correct digits to the right of the decimal, *i.e.* the precision is < 0.01 seconds. The maximum DATETIME variable value is 5 billion years. The precision at that value is approximately 32 seconds.

Note: If you are working with huge year values, you are probable not going to be concerned with microseconds!

If you are working with very short time differences (like microseconds), then the greatest precision will be achieved when the dates are closest to the current date.

Graphing with DATETIME Variables

If you want to plot a time-line chart or graph a function over time (such as stock prices), then you should use DATETIME variables for the values along the "time" axis (usually the *x*-axis). In order to correctly interpret the time values and properly scale the graph, specify **HSCALETYPE** (or **VSCALETYPE**) **= DATETIME**.

Just as with logarithmic scales, there is no zero, so the position of the y-axis (if the *x*-axis is the time scale) is arbitrary. By default, the axis will appear at Jan 1, 1970, local time. You can force the axis to appear anywhere along the time scale by setting the DATETIME variable ***zerodatetime*** to the date and time desired. For example,

> ***zerodatetime*** = **MAKEDATETIME** (1, 1, 2011, 12, 00, 00)

positions the axis at Jan 1, 2011 at noon. (See **MAKEDATETIME** examples, above). Also, the **GRAPH INCLUDEYAXIS=0** may be used if the axis position is too far from the area of interest. The "fake" red axis and the tick values will still be displayed.

> **<u>Note</u>**: If you change the time zone or the daylight saving time rules (see below), this may affect the ***zerodatetime***, so you should always set time zone and daylight saving time rules first.

Auto-scaling works in this mode. You can still specify the offset and scale factors manually, however keep in mind that the offset must also be a DATETIME value. For example,

> **HOFFSET** = **- MAKEDATETIME** (1, 1, 2000) [*note the minus sign*]

will start the graph at Jan 1, 2000.

The scale factor, if set manually, must be given in <u>seconds per inch</u>. For example, to set a scale factor of 1 inch = 10 years, set **HSCALE** = 10 * 365.2425 * 86400, which is 10 years expressed in seconds.

> If you are setting the offset manually, be sure to set the scale factor first.

Automatic scaling is still the easiest way, and you can always adjust the scaling using the "Zoom" menu or the arrow keys.

If you are drawing a **<u>bar chart</u>**, the width of the boxes (or any other measurement on the *x* (time) axis) must be specified in **<u>seconds</u>**. The height of the box is given in the units of the *y*-axis.

If you are doing a **<u>time-line graph</u>** using long rectangles to show periods in time, specify the left and right sides of the rectangle as DATETIME values.

> **<u>Note</u>**: When zooming a time-line graph, it is usually more effective to zoom only the "time" axis. If the time axis is the x-axis, you can zoom horizontally using the left and right arrow keys.

Note: Daylight Saving Time transitions cause an hour to be dropped or an extra hour to be added on certain days. These are shown correctly on the graph scales, however the appearance may seem strange (hours missing or extra hours inserted).

When you zoom in or out on a DATETIME scale, the values displayed along the axis will change depending on the scale factor. As you zoom in to a given year, you will begin to see months displayed. Zooming into the month you will see days, etc. Try it and see.

You may zoom in farther on a DATETIME scale than you can on LINEAR or LOG scales. The zoom limit is determined by the precision of the DATETIME value, so you can zoom in to microseconds on years closer to 1970, but at 5 billion years the limit is around 15 minutes.

Note: When you zoom in on a DATETIME scale, you may not be able to tell which day, month and year you are in. If you want that information displayed, place a title on the DATETIME axis which includes the string "**&date**". When the graph is displayed, this string will be replaced by the portion of the date which does not change over the range of the graph, *i.e.,* the part you can't see.

See the document "*Intermediate Graphics*" for more information. Also check out some of the sample programs.

Notes regarding Daylight Saving Time.

The rules for when (and if) daylight saving time (DST) happens are anything but consistent. Before 1966 there was no official regulation in the U.S. governing daylight saving time. Since then it has changed four times (see below), and it is not the same in all states.

Windows allows you to set the time zone, which affects when DST changes happen. Make sure your Windows is up-to-date regarding DST fixes. (In 2011, DST began on March 13 and ended on Nov. 6.) If Windows is not up-to-date or the time zone and clock are not set correctly, the current time and DST changes may not be correct.

QuickCalc BASIC assumes the following "rules" for DST:

2007 and up	2nd Sun in March / 1st Sun in Nov.
1986 - 2006	1st Sun in April / Last Sun in Oct.
1966 - 1985	Last Sun in April / Last Sun in Oct.
Prior to 1966	(no DST)

These rules will be reflected in all calculations involving DATETIME variables, and will appear in graphs using DATETIME scale types.

If you are in a time zone (as set in Windows) that does not observe DST, then DST will not be used and all DATETIME calculations will be in Standard time.

Changing the Rules (for advanced programmers)

It is recognized that the above rules do not cover all the local variations in all the cities and states across the ages, and may not cover changes that happen in the future. (Future releases of QuickCalc BASIC will attempt to keep up with DST changes).

- You may be in a locality which did not observe the conventional rules,
- You may be in a different time zone not supported by Windows,
- You may be interested in historical dates when you knew DST was in effect,
- You may have an older version of QuickCalc BASIC and the rules have changed,
- You may be doing calculations or preparing a graph for a different time zone,
- You may be in the Southern Hemisphere,
- You may have better information than I did when preparing this.

If you want to change the rules for calculating DST, you may do it in two ways:

To change the rules for **all programs running under QuickCalc**, you may create a small text file which must be named DSTRULES.TXT. The file must reside in the same folder as QUICKCALC.EXE. Each line contains 12 values, separated by commas, which specify a range of years and the rules for that range (see below). If the rules overlap, the last one will take precedence. These rules will override the standard rules shown above.

To change the rules **for a single program**, create a 2-dimensional array (n, 12) where n is the number of new rules you wish to set up. The 12 values in each row of the array are the same as those in the file, and specify a range of years and the rules for that range (see below). These rules override those specified in the override file.

The meaning of each set of 12 numbers is as follows:

first-year	Starting year of the range (-6 billion to +6 billion)
last-year	Ending year of the range (must be >= first-year)
DST start-month	Month in which DST starts (1 = January), zero, if no DST in this range,

	-1 if DST is in effect at start of year,
DST start-day	Day of month when DST starts, **OR** which week to start (1=first, 5=last) See examples, below.
DST start-day-of-week	1 = Sunday, 2 = Monday, etc. 0 = use day of month.

(examples:

start month = 3, *start day* = 2, *start-day-of-week* = 1 means 2nd Sunday in March,

start month = 4, *start day* = 5, *start-day-of-week* = 1 means last Sunday in April,

start-month = 3, *start-day* = 15, *start-day-of-week* = 0 means March 15,

DST *start-bias*	Minutes to adjust the clock ***<u>forward</u>*** for DST (usually 60). **<u>Note:</u>** This must be a multiple of 60 minutes – if not it will be rounded down (toward zero) to a multiple of 60. Maximum is 4 hours in either direction (+/- 480 minutes).
DST *start-hour*	Hour when the change happens (usually 2). **<u>Note</u>**: Should be larger than |*start-bias*|/ 60.
DST end-month	Month in which DST ends (1 = January), zero, if no DST in this range, 13 if DST is in effect at end of year. **<u>Note:</u>** This must not be the same as *DST start-month.*
DST end-day	Day of month when DST ends, **OR** which week to end (1=first, 5=last) See examples, above.
DST end-day-of-week	1 = Sunday, 2 = Monday, etc. 0 = use day of month.
DST *end-bias*	Minutes to adjust the clock ***<u>forward</u>*** for Standard time (usually 0). **<u>Note</u>**: This is an adjustment from the normal time zone to be applied during periods of “Standard” time. It is not a number to be subtracted from DST. **<u>Note:</u>** This must be a multiple of 60 minutes – if not it will be rounded down (toward zero) to a multiple of 60. Maximum is 4 hours in either direction (+/- 480 minutes).

DST *end-hour*	Hour when the change happens (usually 2) **Note**: Should be larger than |*start-bias*|/ 60.

The following example shows how to change the rules for the years 1950 to 1970 to start on the third Sunday on April and end of the last Sunday in October:

For the DST override file, insert the following line:

```
1950, 1970, 4, 3, 1, 60, 2, 10, 5, 1, 0, 2
```

Or, inside the program, insert the following lines:

```
DIM  dstrules (1, 12)
DATA  1950, 1970, 4, 3, 1, 60, 2, 10, 5, 1, 0, 2
FOR i = 0 TO 11: READ dstrules (0, i): NEXT
SETDSTRULES  dstrules
```

You can have multiple **SETDSTRULES** statements, however the total number of rules in the override file and the program is limited to 26, not counting the original four.

Rules set inside a BASIC program will be reset when the next BASIC program starts. Rules set in the override file will take effect then next time QuickCalc starts. Setting overrides in the file is a good way to "fix" QuickCalc if the DST rules change.

Note: When a **SETDSTRULES** statement is executed, the "current" date/time used in **MAKEDATETIME**, **EXPANDDATETIME** and **FORMATDATETIME$** functions will be adjusted, if necessary, to the new rules. This could mean that the DATETIME current date/time is no longer the same as the values in **DATE$** and **TIME$.** Also, graphs using DATETIME scales will be affected.

Note: If you are going to change the daylight saving time rules, you should do it at the beginning of the program to avoid some DATETIME values being calculated or converted using different rules than others.

Note: The daylight saving time rules **may not be changed while a graph is active**. Graphs are plotted using the time zone and daylight saving time rules in effect when the graph is started.

Southern Hemisphere, Double Daylight Saving Time, etc.

In the **southern hemisphere**, summer and winter are reversed, therefore Daylight Saving Time begins in the fall and ends in the spring. To implement this in a “rule”, simply set the *DST-start-month* later then the *DST-end-month*.

Note: In this mode, DST will be in effect in the winter, when the year changes. The year change will be in DST. If the “rule” also changes at the beginning of the year, this could result in some confusion as to exactly when the year changes. To avoid this confusion, the rule for the previous year will remain in effect until a few hours after midnight before the change takes effect, similar to what happens during DST shifts.

“**Double Daylight Saving Time**” is sometimes implemented as 2 hours ahead in the summer and 1 hour in the winter. To implement this, set *DST-start-bias* = 120 and *DST-end-bias* = 60.

This will also result in an offset or bias being in effect in the winter when the year changes (see note, above).

“Permanent” Daylight Saving Time. If (God forbid) Congress sets Daylight Saving time on permanently (for the foreseeable future), you can set a rule effective from the current year until the “end of time” (+6 billion). However, there will be a “transition year” in which DST begins but does not end. In that case, set *DST-end-mon* = 13 in the rule for the transition year. Similarly, when Congress comes to their senses and cancels it, you will have another transition year. Set *DST-start-mon* = -1 for that year.

Turning DST completely on or off.

If you want to run the program with DST off for all years, then issue the statement **SETDSTRULES "OFF".** All calculations will then be done using standard time, and all of the DST rules will be ignored.

If you are in a time zone that doesn’t observe DST, but you want your calculations and graphs to use it, then issue the statement **SETDSTRULES "ON".** The standard rules, the rules in the override file, and the rules specified in your program will now take effect.

Changing the Time Zone

You can change the time zone used to calculate and format DATETIME values with the statement **SETTIMEZONE** *time-zone-bias*. The program will now assume you are in a different time zone than the one specified in Windows. This will not affect the DST rules. The current time used for DATETIME calculations will be adjusted to reflect the current time in the new time zone. This change will last until the next BASIC program is run.

The time zone is specified with the *time-zone-bias* value (in **minutes**), which is negative for West of GMT, *e.g.,* Pacific Standard time is -8 * 60 = -480 minutes. Eastern time is -5 * 60 = -300 minutes.

Note: If you are going to change the time zone, you should do it at the beginning of the program to avoid some DATETIME values being calculated or converted using a different time zone than others.

Note: The time zone may not be changed while a graph is active. Graphs are plotted using the time zone and daylight saving time rules in effect when the graph is started.

Note: The time zone you set here **only affects DATETIME variables** and their calculations and graphs, and remains in effect until another program is started. It does **not** affect any Windows system settings, the system date and time, nor the values in **TIME$** and **DATE$**.

Note: If you want to set Greenwich Mean Time (GMT) or Coordinated Universal Time (UTC), then use **SETTIMEZONE 0** and **SETDSTRULES "OFF".**