WHEN YOU NEED TO BE SURE

CyFlex® Variables, Units, and Computed Expressions

Version 14

November 9, 2022

Developed by SGS North America, Inc.

Version History

Version	Date	Revision Description
1	1/19/2016	Initial publication
2	7/25/2017	Format changes
3	7/19/2018	Added `xml_secs` function to Table 5
4	3/12/2019	Added `polyRoot` function to Table 5
5	6/27/2019	Changed `long` to `int` for numerous functions
6	7/3/2019	Added `month_of_year` function to Table 5
7	3/25/2020	Reformat to new template
8	10/20/2020	• Updated *Section 7.7 Bitwise Operators* on page 8 to add the ^ symbol for `EXCLUSIVE OR`. • Updated *Section 7.9 Expressions without Units Conversion* on page 9 to modify the conversion equation example.
9	5/24/2021	Revised and corrected content of *Table 5* on page 10: • Append “as a string with leading zeroes” to description of `int day_of_month()`on page 11 • Correct `omt day_of_week()`to `int day_of_week()`and append “as a string with leading zeroes” to its description on page 11 • Append “as a string with leading zeroes” to description of `int day_of_year()`on page 12
10	8/4/2021	Added hypertext cross-references to usage help in *Section 7 Computed Expressions* on page 7
11	5/12/2022	Updated all hypertext cross-references to cyflex.com usage help descriptions
12	9/6/2022	Added note in *Section 7.9 Expressions without Units Conversion* on page 9 to clarify allowed use of {} braces in an expression. Added the following function descriptions to *Table 5* on page 10: • `@hex_2` • `@dec_2_hex` • `@parse_line`
13	9/21/2022	Added function description for `@vp_to_dpt` to *Table 5* on page 10.

Version	Date	Revision Description
14	11/9/2022	In *Table 5* on page 10: • Revised `@dpt_to_vp` function and description • Revised `@vp_to_dpt` description • Revised function `double rh_visc_rat( double temp, double rh )` to `double rh_to_vp( rh, dry_bulb_temp )`for `@rh_to_vap` and revised description

Document Conventions

This document uses the following typographic and syntax conventions.

- Commands, command options, file names or any user-entered input appear in Courier type. Variables appear in Courier italic type.

  Example: Select the `cmdapp-`*`relVersion-buildVersion`*`.zip` file….

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable dropdown lists, appear in Arial bold type.

  Example: **Type**: Click **Select Type** to display drop-down menu options.

- Cross-references are designated in Arial italics.

  Example: Refer to *Figure 1*…

- Click intra-document cross-references and page references to display the stated destination.

  Example: Refer to *Section.1* Overview on page 1

  The clickable cross-references in the preceding example are *1*, *Overview*, and on page 1.

CyFlex Documentation

CyFlex manuals are available at https://cyflex.com/. View **Help & Docs** topics or use the **Search** facility to find topics of interest.

Table of Contents

1 OVERVIEW 1
2 REAL VARIABLES 2
3 INTEGER VARIABLES 3
4 LOGICAL VARIABLES 4
5 STRING VARIABLES 5
6 UNITS 6
7 COMPUTED EXPRESSIONS 7
7.1 VARIABLES 7
7.2 CONSTANTS 7
7.3 ARITHMETIC OPERATORS 7
7.4 LOGICAL EXPRESSIONS 8
7.5 LOGICAL OPERATORS 8
7.6 COMPARISON OPERATORS 8
7.7 BITWISE OPERATORS 8
7.8 EXAMPLES OF EXPRESSIONS 9
7.9 EXPRESSIONS WITHOUT UNITS CONVERSION 9
7.10 FUNCTIONS 10

LIST OF TABLES

TABLE 1: REAL VARIABLE CHARACTERISTICS 2
TABLE 2: INTEGER VARIABLE CHARACTERISTICS 3
TABLE 3: LOGICAL VARIABLE CHARACTERISTICS 4
TABLE 4: STRING VARIABLE CHARACTERISTICS 5
TABLE 5: FUNCTIONS SUPPORTED IN EXPRESSIONS 10

1 Overview

A variable is the basic unit of information in CyFlex and is a reference to a piece of information in CyFlex that is often changing in value. It is the ***values*** of variables which are acquired, displayed, stored, used in safety limits, used in closed loop controls, calculated, etc.

There are twelve types of variables. These are:

`REAL`	`REAL_ARRAY`	`INTEGER`
`INTEGER_ARRAY`	`LOGICAL`	`LOGICAL_ARRAY`
`STRING`	`STRING_ARRAY`	`STATISTICAL`
fluid `COMPOSITION`	fluid `PROPERTY`	`EMISSION`

This document discusses only the four most common types which users deal with: `REAL`, `INTEGER`, `LOGICAL`, and `STRING`. Note that the term `REAL` is synonymous with “double precision floating point”.

The values associated with variables are set using various methods according to the use of the variable in the system.

- System inputs are automatically updated by input transfer layers.
- System outputs are typically set by the Controls task, the Test Manager, or the user. Performance variables are automatically updated by applications which compute the values based on current inputs.
- The statistical, composition, and property types are only modified by special processes which are designed for that purpose.

Refer to *Section 7 Computed Expressions* on page 7 for information on the mechanism to specify the value of a variable as a function of other variables in the system.

2 Real Variables

Real variables are variables that contain a single floating-point value. These variables are associated with the following functions:

- Performance and input variables
- Control variables
- User created variables

Table 1: Real Variable Characteristics

Name	Description
Display Format	Specify the number of characters after the decimal points to display using the `format` field available in specifications for real variables. The format does not affect the value, only how the value is displayed.
Label	A unique alphanumeric string of up to 79 characters representing how a variable is to be displayed. Two variables within a CyFlex system cannot have the same label. They may contain any alpha character (a-z and A-Z), numerals and underscores, but must not begin with a numeral. Characters not allowed in any part of the label are `+ - / * & @ , . ! [ ] ? " | ( ) < > ;`
Save Active	A flag indicating whether to save the real variable using the history system. Refer to *Compressed History Data* and *History Plot User Guide*.
Save Tolerance	The maximum deviation between the original information and information reconstructed from the history file. The tolerance should be set low enough to ensure adequate resolution of the stored variable, but not so low that the history system is overloaded. Tolerances are specified in the units specified by "Units" and must be greater than 0.
Units	A string, from a constrained list, representing the physical units of measure which are associated with a variable's value. The user may specify the units associated with a real variable in specifications using the "units" field
Transition Event	The name of event which is set when the value of the variable changes by a value greater than the "save tolerance" (mentioned above)
Value	The information associated with a variable that represents its current condition or state. A real variable's value is a double precision floating point number.

3 Integer Variables

Integer variables contain a single integer value. These variables are associated with the following CyFlex functions:

- Counter input channels
- Performance variables
- General usage variables

Table 2: Integer Variable Characteristics

Name	Description
Label	A unique alphanumeric string of up to 79 characters representing how a variable is to be displayed. Two variables within a CyFlex system cannot have the same label. They may contain any alpha character (a-z and A-Z), numerals, and underscores, but must not begin with a numeral. Characters not allowed in any part of the label are `+ - / * & @ , . ! [ ] ? " | ( ) < > ;`
Save Active	A flag indicating whether to save the variable using the history system. The Save Active flag may be either `ON`, indicating a variable should be saved, or `OFF`, indicating it should not be saved.
Save Tolerance	A value indicating how much of a change in the value of a variable must occur before a value is saved. The tolerance should be set low enough to ensure adequate resolution of the stored variable is achieved, but not so low that the history system is overloaded. Tolerances are specified in the units specified by `Units` and must be greater than 0.
Transition Event	The name of event which is set when the value of the variable changes.
Units	A string representing the physical units of measure which are associated with a variable's value. The user may specify the units associated with an integer variable in specifications using the `Units` field.
Value	The information associated with a variable which represents its current condition or state. An integer variable's value is a long integer number (values up to 2147483648 plus or minus).

4 Logical Variables

Logical variables are variables that contain a single logical value (ON/OFF). These variables are associated with the following CyFlex functions:

- Digital input channels
- Digital output channels
- User defined variables
- Limit state variables
- Limit latch variables

Table 3: Logical Variable Characteristics

Name	Description
Label	A unique alphanumeric string of up to 79 characters representing how a variable is to be displayed. Two variables within a CyFlex system cannot have the same label. They may contain any alpha character (a-z and A-Z), numerals, and underscores, but must not begin with a numeral. Characters not allowed in any part of the label are + - / * & @ , . ! [ ] ? “ | ( ) < > ;
Off Transition Event	A CyFlex event which is set when the value of the variable changes from the ON state to the OFF state.
On Transition Event	A CyFlex event which is set when the value of the variable changes from the OFF state to the ON state.
Save Active	A flag indicating whether to save the real variable using the CyFlex history system. The Save Active flag may be either ON, indicating a variable should be saved, or OFF, indicating it should not be saved.
Value	The information associated with a variable which represents its current condition or state. A logical variable’s value is one of two states. The states may be referred to with specific names (e.g., VALVE_OPEN, VALVE_CLOSED) or by the state names ON and OFF.
Label	A unique alphanumeric string of up to 79 characters representing how a variable is to be displayed. Two variables within a CyFlex system cannot have the same label. They may contain any alpha character (a-z and A-Z), numerals, and underscores, but must not begin with a numeral. Characters not allowed in any part of the label are + - / * & @ , . ! [ ] ? “ | ( ) < > ;

5 String Variables

String variables are variables that contain a sequence of displayable characters. These variables are associated with the following CyFlex functions:

- Performance
- Display
- User defined variables

Table 4: String Variable Characteristics

Name	Description
Label	A unique alphanumeric string of up to 79 characters representing how a variable is to be displayed. Two variables within a CyFlex system cannot have the same label. They may contain any alpha character (a-z and A-Z), numerals, and underscores, but must not begin with a numeral. Characters not allowed in any part of the label are + - / * & @ , . ! [ ] ? “ | ( ) < > ;
Transition Event	The name of event which is set when the value of the variable changes.
Type	The type indicates whether the string is to be displayed in a 15-character field or an 80-character field on the monitor display. When the variable is created, if it is defined with single quotes it is to be a SHORT_STRING (15 characters). If it is defined with double quotes, it is a LONG_STRING (up to 80 characters).
Value	The information associated with a variable that represents its current condition or state. The value may be a string of up to 80 characters. If the string is 15 characters or less, this can be classified as a "short string" and can be displayed on the monitor as if it were a normal variable. If it is more the 15 characters long, an entire line on the display screen may be required for the display.

6 Units

CyFlex is “units aware”. CyFlex uses a system of units conversion for internal calculations that relies on a pre-defined table of units conversion factors. The units that are available in this pre-defined table can be browsed by typing the command “units”. The output will arrange the available units descriptions by dimension For each dimension such as pressure, temperature, length, mass, etc. there is a specified “base” unit which is in SI (metric) units. A conversion factor is specified for all other units in that dimension for converting to the “base” SI units. The internal calculation method uses the following steps when evaluating an expression

- Fetch the value and units of each variable in the expression
- Convert the value to the “base” SI units of the dimension
- Perform the arithmetic operations specified in the expression
- The result of the arithmetic operations will be in “base” SI units
- If the result of the computations is to be stored into an output variable, fetch the units of that output variable
- Convert the result from the “base” SI units to the units of the output variable
- Store the converted value in the output variable

A simple example can illustrate calculation of the power of a rotating shaft for which the speed and torque are being measured. The speed variable has units of [rpm] and the torque has units of [lb_ft]. The power has units of [bhp]. The calculation is simply:

speed * torque

The method uses the following steps:

- Convert speed from [rpm] to [radians/sec] -- [radians/sec] is the base SI units
- Convert torque from [lb-ft] to [Newton-meters] -- [Newton-meters] is base SI units
- Multiply the value of speed by the value of torque in base SI units and the result will be in base SI units of power, [watts]
- Convert the power value in [watts] to [bhp]
- Store the result in the desired power variable

Note:

There are no constants such as 5252 in the expression because this method is “units-consistent” as a result of the conversion to base SI units.

7 Computed Expressions

Users may create variables of the types mentioned above and assign a computed expression for that variable. The variable value is then dynamically computed by CyFlex based on the expression that the user supplies which arithmetically combines other variable values. Use the following CyFlex applications to perform user computations (not a complete list).

- `compvar` for user computations; refer to `compvar` usage help on cyflex.com.
- `gp_test`, the Test Manager; refer to `gp_test` usage help on cyflex.com.
- `evnt_rsp` for event response; refer to `evnt_rsp` usage help on cyflex.com.
- `limit` to examine limits specifications; refer to `limit` usage help on cyflex.com.
- `floger` to use the Data Logger; refer to `floger` usage help on cyflex.com.
- `get_comp` to obtain a computed expression value; refer to `get_comp` usage help on cyflex.com.

The mechanism which allows the user to define a computed variable's value as a function of other variables is called an expression. An expression is a string the user specifies which is equated to the computed variable's value.

Example expressions:

```
"counter + 1[none]"
"torque * speed"
```

7.1 Variables

Variables are referred to in expressions using the variable's label. The units associated with a variable are automatically managed in the evaluation of the expression, i.e., units conversions are not required in expressions.

7.2 Constants

A constant in an expression is a combination of a value and a units specification. Note that CyFlex requires that units be specified for ALL constants used in an expression. Forgetting to include units is one of the most common errors in writing expressions. The units are associated with the value using square brackets ([...]) as follows:

```
100[deg_f]
1200[rpm]
```

7.3 Arithmetic Operators

The following arithmetic operators are supported in expressions:

```
+ addition
- subtraction
/ division
* multiplication
```

Notes:

The + operator, when applied to strings denotes concatenation. The other 3 arithmetic operators cannot be used with strings.

Example – squaring a value found in variable "xyz"

```
@pow( xyz, 2[none] )
```

7.4 Logical Expressions

if (logical_variable_or_expression) **then** (value) **else** (value)

Complex if/then/else logical constructs may be used where the *value* above may itself be an if/then/else expression.

if a **then** (**if**(x) **then** (y) **else** (z)) **else** b

If the target variable of an expression is a logical variable, the if/then/else terms are unnecessary for the case where the result is either TRUE or FALSE.

if (a > b) **then** TRUE **else** FALSE

can be replaced by the expression alone

a > b

since the result of evaluating (a > b) will be either TRUE or FALSE.

7.5 Logical Operators

Logical operators may be used to combine logical values or expressions. Supported logical operators are:

&&	AND
||	OR
!	NOT

7.6 Comparison Operators

>	greater than
<	less than
>=	greater than or equal to
==	equal to
<=	less than or equal to

7.7 Bitwise Operators

|	bitwise OR
&	bitwise AND
>>	right shift
<<	left shift
^	EXCLUSIVE OR

7.8 Examples of Expressions

A calculation of a speed 100 RPM over the engine's idle speed:

```
"IdleSpd + 100[rpm]"
```

A calculation of a control loop target temperature as a function of whether an engine is running:

```
"if( Engine_Run ) then ( 100[deg_f] ) else ( 130[deg_f] )"
```

```
The following is an example of a computed string, where the +
(plus) symbol is used to represent string concatenation.
" 'test_device_serial_number='   +    serial_number "
In the example above, serial_number is a string variable and
might contain a string such as "100321".  The sinqle-quoted
string 'test_device_serial_number' is a literal string, so the
result would be:
test_device_serial_number=100321
```

7.9 Expressions without Units Conversion

Calculations within an expression can be forced to occur without units conversion or to be done with specified units by enclosing the expression in { } braces. In this form, the constants are entered without units and adding units to a variable label is optional. This can be particularly useful when a computation is needed and the computed expression implies units that are not in the list of supported units for PAM. For example, a manufacturer of an instrument may provide a conversion equation such as:

```
"x = magic_num * a * b / @pow( c , 3 )"
```

Where x has units of ft3/min, a is in rpm, b is in psi and c is the diameter in mm2. The expression can be enclosed in the curly braces {} to perform the computation the same as a calculator would.

```
{ ( a * b) / ( c * c * c )  }
```

Note:

> The {} braces can only be used to contain the complete expression and cannot be used for sections within the expression.

The result of the above expression will have the value expressed in kg/min.

A simpler example: Assume that `IdleSpd` has units of RPM and a value of 1000[rpm]:

`" IdleSpd + 100[rpm] "` will yield the equivalent of 1100[rpm] in default units of [rad/sec]

`{ IdleSpd + 100 }` will yield a value of 1100

As another example, using temperatures, assume `my_temp` has a value of `68[deg_f]` and units of `deg_f`:

`my_temp` will have a result of 293 in base SI units of Kelvin

`{my_temp}` will result in 68 (`deg_f`)

7.10 Functions

Functions may be called in expressions using an "at" sign (@) followed by the name of the function, for example:

```
@sin( crank_angle )
```

Table 5: Functions Supported in Expressions

Function	Description
`abs(X)`	integer absolute value of X
`ceil(X)`	smallest integer not less than X. Short for ceiling
`fabs(X)`	floating point absolute value of X
`floor(X)`	largest integer not greater than X
`fmod(dividend, divisor)`	remainder of dividend divided by divisor
`labs(X)`	long absolute value of X
`round(X)`	rounds to nearest integer
`exp(X)`	Napier's number e (2.71828...) raised to the X power
`log(X)`	natural log of X
`log10(X)`	common log of X
`log2(X)`	log, base 2, of X
`pow(base,exp)`	base raised to the exp power
`sqrt(x)`	square root of X
`acos(X)`	arc cosine of X (-1<=X<=1)
`acosh(X)`	inverse hyberbolic cosine of X (X>1)
`asin(X)`	arc sine of X (-1<=X<=1)
`asinh(X)`	inverse hyperbolic sine of X
`atan(X)`	arc tangent of X (-□ /2 < X < □ /2) (2 quadrants)
`atanh(X)`	inverse hyperbolic arctangent of X
`atan2(sin, cos)`	arc tangent of sin/cos (4 quadrants)
`cos(X)`	cosine of X

Function	Description
`cosh(X)`	hyperbolic cosine of X
`sin(X)`	sine of X
`sinh(X)`	hyperbolic sine of X
`tan(X)`	tangent of X
`tanh(X)`	hyperbolic tangent of X
`j0(X)`	Bessel function first kind, zeroth order
`j1(X)`	Bessel function first kind, first order
`y0(X)`	Bessel function second kind, zeroth order
`y1(X)`	Bessel function second kind, first order
`long_3d_comp(X_VAR, TABLE_NUM)`	3D long interpolation Example: `@long_3d_comp( 1000[rpm], 5[none] )` this would use table file `/cell/tables/loc_two_d_5.tbl`
`long_3d_comp_name( double value, char *table_name )`	3d long interpolation test, using the table name Example: `@long_3d_comp_name( 1000[rpm], 'loc_two_d_5' )` this would use table file `/cell/tables/loc_two_d_5.tbl`
`shrt_3d_comp(X_VAR, TABLE_NUM)`	3D short interpolation
`int second_of_minuteL()`	returns current "seconds" of time as an integer
`char* second_of_minute ()`	returns current "seconds" of time as a string with leading zeroes
`int minute_of_hourL()`	returns current "minutes" of time as an integer
`char* minute_of_hour()`	returns current "minutes" of time as a string with leading zeroes
`int day_of_month()`	returns current "day" of month as a string with leading zeroes
`int day_of_week()`	returns current "day" of week as a string with leading zeroes

Function	Description
`int day_of_year()`	returns current "day" of year as a string with leading zeroes
`int hour_of_dayL()`	returns current "hour" of day (24-hour clock) as an integer
`char* hour_of_day()`	returns current "hour" of day (24-hour clock) as a string with leading zeroes
`int year_month_dayL()`	returns an integer of format YYMMDD, e.g. 110621
`int day_month_yearL()`	returns an integer of format DDMMYY, e.g. 210611
`char* year_month_day()`	returns string of format YYMMDD, e.g. “110621”
`char* day_month_year()`	returns string of format DDMMYY with leading zeroes, e.g. “210611”
`int year4_month_dayL()`	returns an integer of format YYYYMMDD, e.g. 20110621
`char* year4_month_day()`	returns string of format YYYYMMDD with leading zeroes, e.g. “20110621”
`char* week_of_year()`	returns string of format NN, where NN is the current week of the year numbered from 1 to 52, e.g. “47”
`int week_of_yearL()`	returns an integer of format NN, where NN is the current week of the year numbered from 1 to 52, e.g. 9
`char* month_of_year()`	Returns a 2-character string representing the month (01 to 12)
`int month_of_yearL()`	Returns an integer value from 1 to 12 representing the month

Function	Description
`double rh_to_vp( rh, dry_bulb_temp )`	Given the relative humidity and dry bulb temperature, calculate the vapor pressure per the combination of equations 40CFR1065.645(a), (b) and (c). WARNING: The proper units for relative humidity are pct, not %. Examples: `@rh_to_vap( vaisala_rh, vaisala_t )` `@rh_to_vp( 50[pct], 59.4[deg_f] )` returns 0.255970[in_hg]
`double kvisc( long code , double t )`	compute the kinematic viscosity (stokes)of lube or fuel oil at a given temperature (t) code and type: 1. #1 fuel oil 2. #2 fuel oil 3. 15W40 lube oil 4. 10W lube oil 5. 30W lube oil
`double dpt_to_vp( dew_point )`	given the dewpoint temperature, calculate the vapor pressure per 40CFR1065.645(a)(1) or (2) depending on the temperature range. Example: `@dpt_to_vp( outside_temp )` `@dpt_to_vp( 68[deg_f] )` returns 2336 pascals

`double vp_to_dpt( vap_pa )`	Given the vapor pressure, calculate the dewpoint temperature per 40CFR1065.645(d)
	Examples:
	`@vp_to_dpt( vap_pa )`
	`@vp_to_dpt( 0.5931[in_hg] )` returns `63.53145[deg_f]`

`double cal_table( x-value, table_name )`	returns interpolated value from a calibration table
	Examples:
	`@cal_table( 100[mv], 'cmp_in_p' )`
	`@cal_table( x_val, 'cmp_in_p' )`

Function	Description

`char ascii_time( time )`	returns MM/DD/YY HH:MM:SS format from a double value of ANSI time – time may be a constant with units, the label of a variable with units of time, or a computed expression
	Examples:
	`@ascii_time( 1150897824.87[sec] )`
	`@ascii_time( time )`
	`@ascii_time( "time – 1[hr]" )`

`char ascii_time4( time )`	returns MM/DD/YYYY HH:MM:SS format from a double value of ANSI time – time may be a constant with units, the label of a variable with units of time, or a computed expression
	Examples:
	`@ascii_time4( 1150897824.87[sec] )`
	`@ascii_time4( time )`
	`@ascii_time4( "time – 1[hr]" )`

`char xml_time( time )`	returns CCYY-MM-DDTHH:MM:SS format from a double value of ANSI time – time may be a constant with units, the label of a variable with units of time, or a computed expression
	Examples:
	`@xml_time( 1150897824.87[sec] )`
	`@xml_time( time )`
	`@xlm_time( "time – 1[hr]" )`

`unsigned int strlen( char *string )`	Find the length of a string
	Examples:
	`var1 = "abcde"`
	`@strlen( var1 )`
	The above returns 5
	Or
	`@strlen( 'abcde' )`
	The above returns 5

Function	Description

`int strcmp( char *s1, char *s2 )`	Compare two strings – functions the same as the standard C-library call
	Example:
	`Var1 = "abc"`
	`Var2 = "def"`
	`@strcmp( Var1, Var2 )`
	The above returns -1
	`Var1 = "ghi"`
	`Var2 = "ghi"`
	`@strcmp( Var1, Var2 )`
	The above returns 0 because they match
	`Var1 = "jkl"`
	`Var2 = "abc"`
	`@strcmp( Var1, Var2 )`
	The above returns 1

`strcpy( char *s1, char *s2 )`	string copy
	Examples:
	`Var1 = "111"`
	`Var2 = "789"`
	`@strcpy( Var1, Var2 )`
	The above results: Var1 = "789"
	`Var1 = "abc"`
	`@strcpy( Var1, '8888' )`
	The above results: Var1 = "8888"

`char *strstr( char *s1, char *s2 )`	Find the start of one string in another string
	Examples:
	`Var1 = "0"`
	`Var2 = "456"`
	`@strstr( Var1, Var2 )`
	The above results: = ""
	`Var1 = "789"`
	`Var2 = "8"`
	`@strstr( Var1, Var2 )`
	The above results: = "89"

Function	Description

`strncmp( char *s1, char *s2, int n )`	string compare – number of characters
	Examples:
	`Var1 = "ABCDEFGH"`
	`Var2 = "ABCDEFHI"`
	`@strncmp( Var1, Var2, 4[none])`
	The above returns: 0
	`@strncmp( Var1, Var2, 8[none])`
	The above returns: -1
	`@strncmp( Var1, 'ab', 2[none] )`
	The above returns: 0

`strncpy( char *s1, char *s2, int n )`	Copy n characters from one string to another
	Examples:
	`Var1 = "55555555"`
	`Var2 = "789"`
	`@strncpy( Var1, Var2, 3[none] )`
	The above results: Var1 = "7895555"
	`Var1 = "TRUE"`
	`@strncpy( Var1, 'FALSE', 2[none] )`
	The above results: Var1 = "FAUE"

`char* strupr( char *s1 )`	convert a string to upper case
	Example:
	`Var1 = "abcdefgh"`
	`@strupr( Var1 )`
	The above results: Var1 = "ABCDEFGH"

`LOGICAL strcmp_lbl_lbl( char *label1, char *label2 )`	compare the values of 2 string variables Example: `@strcmp_lbl_lbl( my_s1, my_s2 )` returns TRUE if the contents of string variable s1 is the same as the contents of string variable s2
`LOGICAL strcmp_lbl_lit( char *label, char *s1 )`	compare the value of a string variable with a string literal Example: `@strcmp_lbl_lit( my_s, 'ctl_spd' ) returns TRUE if my_s contains "ctl_spd"`

Function	Description
`double str_lbl_value( char *label )`	get the value of a variable whose label is contained in a string variable Example: `@str_lbl_value( my_s )` where my_s contains the label 'ctl_spd', returns 1000.5 *See str_lbl_def_value() below to return the value in default units.*
`double str_lbl_def_value( char *label )`	return default (SI units) value of a variable whose label is contained in a string variable Example: `@str_lbl_def_value( my_s )` where my_s contains the label of a variable
`char *str_var_string( char *label, int format )`	get the value of a variable whose label is contained in a string variable - return the value as string with "format" places to the right of decimal. NOTE: for this function, the label must be contained in single quotes as shown below. Example: `@str_var_string( 'my_s', 2[none] )` where my_s contains the label 'ctl_spd', returns "1000.49" #type label -> value contained string my_s -> 'ctl_spd' (label of target variable) real 'ctl_spd' -> 1000.49 (this can be any type)
`short str_lbl_in_str_lbl_out( char *lable1, char *label2)`	where label1 and label2 are string variables which contain the labels i_lbl_1 and i_lbl_2, move the value of i_lbl_1 to i_lbl_2 using appropriate units conversion – return TRUE if successful Example: `@str_lbl_in_str_lbl_out( my_pres1, my_pres2 )`
`int strstrx( char *s1, char *s2 )`	search s1 for the substring s2 and return the position in s1 where s2 starts Example: `@strstrx( 'a valuable lesson', 'val' )` returns a 2

Function	Description
`int strstrci( char *s1, char *s2 )`	search s1 for the substring s2 and return the position in s1 where s2 starts. Do the search in a case-insensitive manner Example: `@strstrx( 'a VALUABLE lesson', 'val' )` returns a 2
`double cal_table_min( char *file )`	return the minimum engineering units of a calibration table file Example: `@cal_table_min( 'air_mtr0_p' )` returns the minimum value of file `/cell/tables/air_mtr0_p.tbl`
`double cal_table_max( char *file )`	return the maximum engineering units of a calibration table file Example: `@cal_table_max( 'air_mtr0_p' )` returns the maximum value of file `/cell/tables/air_mtr0_p.tbl`
`double variable_age( char *label )`	return the time (in seconds) since a variable was updated Example: `@variable_age( ctl_spd )` returns 1.00 if ctl_spd variable was update 1 sec ago
`double variable_time( char *label )`	return the time (time_t) when a variable was last updated Example: `@variable_time( ctl_spd )` returns 1150897824.87
`char* units_tag( char *label, char *tag )`	return a tag with value and units for a variable Example: `@units_tag( 'ctl_spd', 'speed' )` returns "speed=1000.5[rpm]"
`char* value_units( char *label )`	return a string which contains the value[units] of a variable Example: `@value_units( 'ctl_spd' )` returns "1000.5[rpm]"

Function	Description
`short ctrl_loop_mode( char *loop )`	return the open/closed mode of a control loop – where 0 =closed loop, 1=open loop and loop is the label of the feedback variable Example: `@ctrl_loop_mode( fuel_t )`
`short file_info( char *filename, char *type)`	This function duplicates the C access() function. It determines the access permissions of a file. type can be either `W_OK` (for writing), `R_OK` (for reading), or `X_OK` (for execute). The return will be `TRUE` if the specified access mode is permitted. Example: `@file_info( '/specs/PAM_datapoint', 'W_OK' )` returns `TRUE` if the file is writeable
`int array_to_stat( char *arr_label, char*stat_label, int n )`	This function will perform the standard statistical analysis on the first 'n' values found in the specified 1-dimensional array variable. The array variable must be of the `REAL` type. Example: `@array_to_stat( 'myarr', 'mystat', 10[none] )` -- analyzes the first 10 values in *myarr*
`short set_array( char *label, char *value )`	This function will initialize all of the values of an array to the specified value. Example: `@set_array( 'my_array', '0' )` this fills the entire array with zeroes

Function	Description
`double var_to_double( char *label )`	This function searches memory for the specified variable label and returns a double floating-point value equivalent to the contents of the variable, regardless of the data type. The '`label`' argument should not be included in quotes. Example: `@var_to_double( Speed )` -999999 is returned if the label is invalid or a string variable cannot be converted to a floating-point value
`LOGICAL is_name_ready( char *reg_name, char *wait_time )`	This function returns a `1` (TRUE) if *`reg_name`* is a valid registered name of an application. The function will repeatedly test for the name for *`wait_time`* (in seconds) and return a `0` (FALSE) if the name is not present after *`wait_time`* seconds. Example" `@is_name_ready( 'asam3_1', '10' )` The name `asam3_1` is typically used for the instance of `asam3cli` that manages communications with a CUTY device. If `asam3cli` is running, then this function will return a TRUE value.
`int get_bits( char *label,` `int start_bit, int length )`	This function returns an integer value from the variable from the starting bit for the next length bits. Example `@get_bits( Speed, 4, 8 )` This reads the 'Speed' variable and starting from the 4th bit, reads and returns the value of the next 8 combined bits as a value.
`LOGICAL get_logi_bit( char * label, int bit_location )`	This function returns a `LOGICAL` value for the specified bit from the variable. Example `@get_logi_bit( ctrl_spd, 5 )` This reads the *`ctrl_spd`*' variable and returns 1 or 0 for the 5th bit.

Function	Description
`int check_finite( char *label )`	This function returns `1` or `0` depending on whether the variable given is a `NAN` or other undefined or special value. If it is a `NAN`, `IFINITY`, or other special value, it will return a `0`. If it is a finite and readable value, it returns a `1`.

`int xml_secs( char *xml_date_time )`	This function returns the number of seconds since Jan. 1, 1970 for the specified *`xml_date_time`* string. Examples:
	`@xml_secs('2018/06/19T12:00:00Z')`
	`@xml_secs('2018-06-19T12:00:00Z')`

`double polyRoot( char *target, char *range, char *table_pathname )`	This function will return the root of a polynomial calibration table of `POLYNOMIAL_RANGE` type. All 3 arguments are literal strings enclosed by single quotes.
	`target` -- the y-parameter target with units, for example '100[ppm]'
	`range` -- the range index – for example, '1'
	`table_pathname` -- the full pathname of the calibration table file, for example, `'/cell/tables/`*`mytable.tbl`*`'`
	`@polyRoot( '100[ppm]', '2', '/data/CO2Cal.tbl' )`
	Note that the table does not have to be an active calibration table and can be located in any directory.

`int hex_2_dec( char *hexstring )`	Convert a hexadecimal string to a decimal integer value, for example: `@hex_2_dec( '20' ) returns 32`
`Char *dec_2_hex( char *label )`	Convert the value of an integer variable to a hex string, for example: if variable count is 32 `@dec_2_hex( count ) returns the string "20"`
`Char *parse_line( char *label, char *optional_delim, int N )`	Parse a string variable value and return the contents of the Nth token in the string. (N starts at 0). The label may be either a `STRING` or `STRING_ARRAY` variable. Example:

Function	Description
	If the contents of variable ‘my_list” contains a string “ list0 list1 list2 list3” `@parse_line( ‘my_list’, ‘ ‘, 2 )` Will return the string “list2”. Note that the label and the optional_delim must be single-quoted. This function will be available in 6.3.33 and later versions.