VS Commands

Reference Manual

Mechanical Simulation

Last revision: May 2022.

Table of Contents

1. Introduction

Mechanical Simulation Corporation produces and distributes software tools for simulating and analyzing the dynamic behavior of motor vehicles in response to inputs from steering, braking, throttle, road, and aerodynamics. The simulation packages are organized into families of products named BikeSim®, CarSim®, SuspensionSim®, and TruckSim®. All are based on the simulation architecture named VehicleSim®.

These products construct VS Math Models with variables and equations that model the physics of a specified vehicle and its environment. It calculates values for the variables and writes them into output files for viewing with plots and animations. Results from simulated tests are typically analyzed using the same methods as would be used for physical test measurements.

Besides the built-in features of an existing VS Math Model, advanced users can extend the model using several methods. One option, which is the subject of this manual, is to use a scripting language called VS Commands to add variables and equations and provide advanced control over the simulation process. This manual also covers advanced options for using symbolic formulas for specifying values of existing parameters.

Who Should Use this Manual

The information in this manual is not needed for the basic use of a VehicleSim product.

This manual is intended for users who want to define scripted procedures or to add equations to existing VS Math Models. These capabilities are provided with VS Commands. This manual also describes how symbolic equations can be used in input files to calculate values for parameters based on current values of other parameters in the VS Math Model.

This manual is a continuation of the material presented in the **Help** document **Reference Manuals > VS Math Models**. Besides being familiar with the material covered in the *VS Math Models Manual*, you should have basic knowledge about how your VehicleSim product is used. As a minimum, you should have gone through the Quick Start Guide for your VehicleSim product.

This manual covers general-purpose VS Commands for adding variables, adding equations, configuring functions, and using symbolic formulas. These are applicable to all VS products. There are also VS Commands that are specific for parts of a vehicle model, such as the speed controller, interactions with paths and road surfaces, interactions with ADAS sensors and moving objects, and so on. VS Commands that are specific to a particular part of a model (such as ADAS sensors) are covered in the same document as the other features, in the PDF documents listed in the main part of the Help menu (e.g., *Driver Control Screens*, *ADAS Sensors and Moving Objects*, etc.).

Notational Conventions

VS Math Model is an entity constructed to run a simulation in a VehicleSim product. It contains variables describing the model and equations and algorithms to calculate their values. It reads input files and writes output files for viewing with plots and animation or use by other software.

VS Solver is a dynamic library with functions that are used to construct VS Math Models.

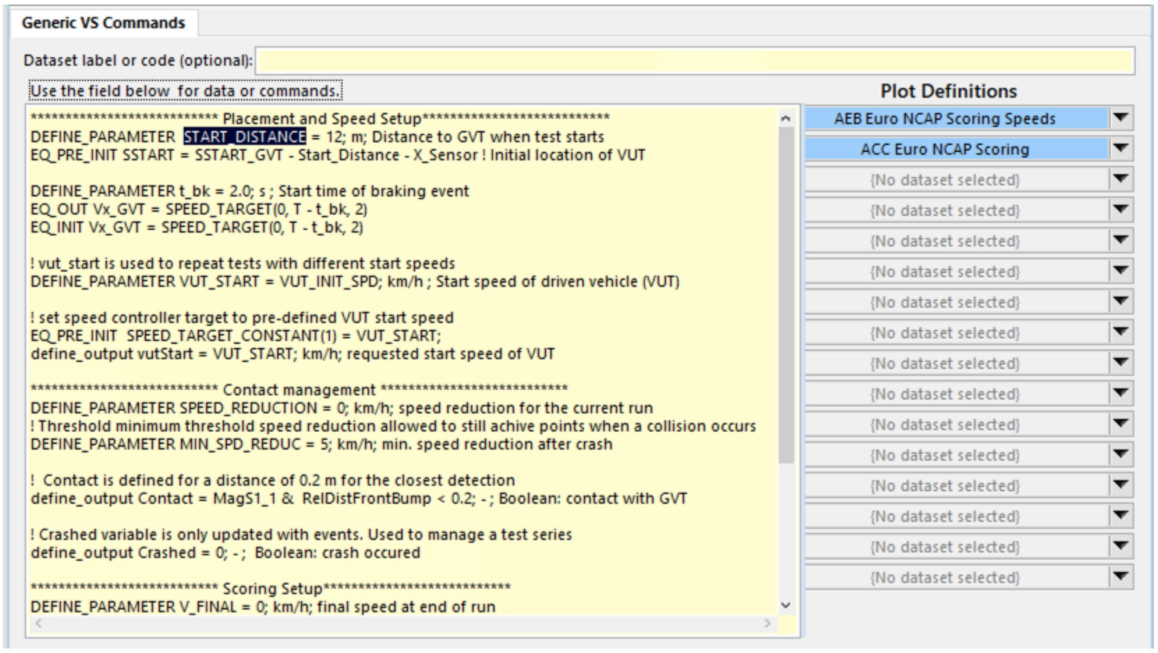
VS Browser is the main program of the Windows product that provides the GUI and manages the database (e.g., `carsim.exe`).

When showing the syntax for data, words shown in the `Courier` font indicate specific literal keywords or file names; words in *italics* indicate the name of the item whose value can vary in different situations, and [square brackets] indicate that content between the brackets is optional. An asterisk before braces *[] indicates that whatever is between the braces may be repeated.

Where to Provide VS Commands

From the point of view of a VS Math Model, all inputs are read from one or more text files that follow the basic Parsfile syntax described in the **Help** document **Reference Manuals > VS Math Models**. When managing simulations from a VS Browser, most parameter values and model settings are made using standard GUI controls such as drop-down lists, checkboxes, yellow fields, spreadsheets, etc. When using the Linux command line tools, VS Commands can be given by modifying the `Run_all.par` file in the results folder of a database for a given run with any text editor.

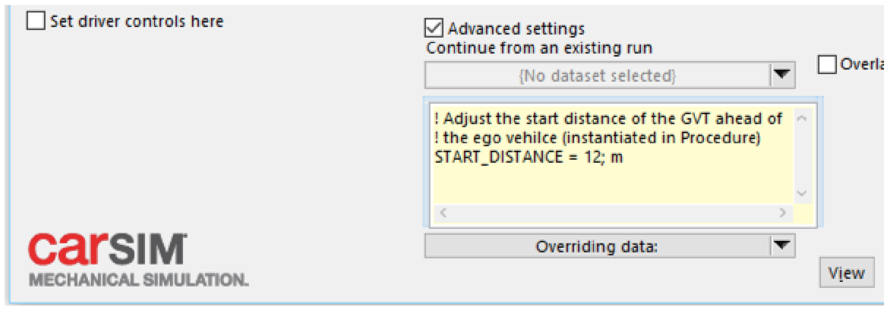
When using the VS Browser, most VS Commands will be provided in miscellaneous yellow fields that are available on many of the library screens. The library **Generic VS Commands** has a large yellow field that is often used to add multiple VS Commands (Figure 1). Depending on the purpose of the commands, the dataset may be linked from datasets in many other libraries that have miscellaneous links, including the **Run Control** screen, the **Procedures** screen, and many of the vehicle parameter screens.



```
*************************** Placement and Speed Setup***************************
DEFINE_PARAMETER START_DISTANCE = 12; m; Distance to GVT when test starts
EQ_PRE_INIT SSTART = SSTART_GVT - Start_Distance - X_Sensor ! Initial location of VUT

DEFINE_PARAMETER t_bk = 2.0; s ; Start time of braking event
EQ_OUT Vx_GVT = SPEED_TARGET(0, T - t_bk, 2)
EQ_INIT Vx_GVT = SPEED_TARGET(0, T - t_bk, 2)

! vut_start is used to repeat tests with different start speeds
DEFINE_PARAMETER VUT_START = VUT_INIT_SPD; km/h ; Start speed of driven vehicle (VUT)

! set speed controller target to pre-defined VUT start speed
EQ_PRE_INIT  SPEED_TARGET_CONSTANT(1) = VUT_START;
define_output vutStart = VUT_START; km/h; requested start speed of VUT

*************************** Contact management ***************************
DEFINE_PARAMETER SPEED_REDUCTION = 0; km/h; speed reduction for the current run
! Threshold minimum threshold speed reduction allowed to still achive points when a collision occurs
DEFINE_PARAMETER MIN_SPD_REDUC = 5; km/h; min. speed reduction after crash

!  Contact is defined for a distance of 0.2 m for the closest detection
define_output Contact = MagS1_1 &  RelDistFrontBump < 0.2; - ; Boolean: contact with GVT

! Crashed variable is only updated with events. Used to manage a test series
define_output Crashed = 0; - ;  Boolean: crash occured

*************************** Scoring Setup***************************
DEFINE_PARAMETER V_FINAL = 0; km/h; final speed at end of run
```

Figure 1. The Generic VS Commands screen is well-suited for adding sets of VS Commands.

Smaller miscellaneous fields are available on many screens. For example, Figure 2 shows how a miscellaneous field on the **Run Control** screen is used to set a value for a parameter `START_DISTANCE` that was defined with a VS Command `DEFINE_PARAMETER` as part of the set of commands shown in Figure 1.



```
! Adjust the start distance of the GVT ahead of
! the ego vehilce (instantiated in Procedure)
START_DISTANCE = 12; m
```

Figure 2. Smaller miscellaneous fields are available on many screens.

Revision History

This manual applies to the solvers in CarSim, TruckSim, BikeSim, and SuspensionSim. The version of the manual is indicated by the date shown at the bottom of page ii. The manual is updated when new features are added to the VehicleSim architecture, when errors are found in the current version, or when improved descriptions have been written.

- The May 2022 revision added information about the `DEFINE_TABLE` and `DEFINE_IMPORT` commands, and more guidelines for coordinating VS Commands with index parameters. Hyperlinks to other PDF documents were removed due to security concerns.
- The December 2021 revision added more information about using indexed parameters in symbolic formulas, and where to locate VS Commands in VS Browser datasets. It added documentation about VS Command functions `PARTIAL` and `PARTIAL2`, which provide partial derivatives of Configurable Functions with respect to independent variables.
- The June 2021 revision added the `vs.statement` command for embedded Python, the variables `T_Real_Last`, `T_Real_Step`, and `T_Real_Elapsed`, and had some minor edits.
- The December 2020 revision noted some VS Commands which are not applicable to SuspensionSim.
- The August 2020 revision had some edits changing “VS Solver” to “VS Math Model.” Descriptions of VS Assignment Commands were extended, a subsection was added for Extended VS Assignment Commands, and more information was provided for indexed parameters.
- The June 2020 revision extended the documentation for user-defined functions and adds information about the differences between commands and functions. It added information about how a VS Math Model determines a model layout. The descriptions

of the variables `T` and `T_Stamp` were updated. A chapter on VS Commands and functions related to roads and paths was removed, with the content now a part of the *Paths and Road Surfaces* document. Content involving Set and Get commands for making new tables, road, and paths with custom IDs was also moved to the *Paths and Road Surfaces* document. Tips were added to some descriptions to highlight recommended best practices.

- The December 2019 revision covers an improvement to `LOCAL_DEFINE` and introduced the command `INIT_CURRENT_PATH_SWEEP.`
- The June 2019 revision documents math operators that can be used in formulas, a new IF function, and mentions symbolic expression feature that also work in VS Visualizer. Functions defined with `BEGIN_FUNCTION` command were added. Commands related to VS Events were revised, with several being discontinued and a new one (`MAKE_EVENT`) added to provide backward compatibility. The descriptions of custom ID number and their use was revised. A typo was fixed involving VS Command function `PATH_YAW_ID`.
- The November 2018 revision reordered the sections in Chapter 4 and described the new location for equations added with the `EQ_INIT2` command. Import variable names must now begin with the prefix `IMP_`. New commands were added for managing Events: `SET_EVENT_ID` and `DELETE_EVENTS_ID`. Names for output variables added with the `DEFINE_OUTPUT` command used to have a length limit of 8 characters; the limit is now 32 characters. The `PATH_YAW_ID` VS Command function was added.
- The May 2018 revision separated Configurable Functions and roads into separate chapters. Information about ADAS sensors and moving objects is covered in separate documentation and was removed. Chapter 7 was modified to describe the timelines for the simulation sequences as revised in version 2018.1 of the products. It added the `EQ_DYN` VS Command.
- The November 2017 revision had minor updates involving ID numbers for tables, paths, and roads. It describes range checking, added to some of the functions that can be used in VS Commands, e.g., `SQRT`.
- The December 2016 revision added the `DEFINE_DZ_TABLES` and `SET_IROAD_DZ` commands for specifying DZ layers on road surfaces, and commands to load tabular data from a simulation output file (VS or ERD format) with commands such as `LOAD_TABLE_FILE` and `FILE_TO_TABLE`.
- The October 2015 revision added the `SET_DESCRIPTION` command and options for setting descriptions for new and existing variables inline. It added details on setting units for Configurable Functions. It documented new VS Commands for paths, roads, and tables, and managing user-defined ID numbers for paths, roads, and tables. The sections with road and path functions were reorganized.
- The October 2014 revision added VS Commands for multiple roads and paths, introduced with CarSim 9.0.

- The August 2013 revision had minor corrections of typos and clarifications about the `EQ_SAVE` and `EQ_FULL_STEP` commands.
- The July 2013 revision expanded the manual with formatting changes and more information about Events and the simulation process in general. It added information about the use of root names of indexed parameters and Configurable Function variables within symbolic expressions. It added more information about the commands used to add equations.
- The January 2013 revision added information about conversion between user unit systems and the internal SI unit system.
- The December 2012 revision added information about setting units for parameters, variables, and tables. It added documentation for the `LINEARIZE` command and described the format for making strings that include values of symbolic expressions (titles, messages for the log file, file names, etc.). The chapter about the VS Browser Symbol Stack was removed (it is now in the VS Browser manual).
- The December 2011 revision added information about the VS Browser Symbol Stack.
- The October 2011 revision updated the descriptions of functions that work with roads and driver reference paths using S and L coordinates. It also added the `RESET_EQS_ALL` command.
- The July 2011 revision was reorganized to cover more options for customizing Configurable Functions, including the `_EQUATION` option and new scaling parameters. Methods were presented for extending models by modifying Configurable Functions dynamically, using VS Commands and possibly variables imported from other software. Descriptions of some VS Commands were updated to match improvements.
- The June 2010 revision added information about equations used to save values of variables between timesteps during a run, and commands for saving and restoring the state of the entire VS Math Model. The `DEFINE_EVENT` command was extended to cover more conditional tests. Editorial changes were made in the descriptions of Configurable Functions (tables).
- The November 2009 revision included a new command for user-defined Configurable Functions (tables) and the use of integer variables in equations.
- The May 2009 revision replaced a single reference manual with two: *VS Solvers* (since renamed *VS Math Models*) and *VS Commands* (this document). This version also described more VS Command options, including options to add equations for controls and forces without affecting import and export.
- The May 2008 revision included new VS Commands for removing equations and resetting conditions, and new functions (Boolean operators and 3D road geometry access) that could be used in symbolic expressions.
- The January 2008 revision added more detail and improved some descriptions of existing features.

- The April 2007 revision described a few more functions that had been added to VS Math Models.
- The first version (February 2007) was prepared for the release of CarSim 7.0.

2. Symbolic Expressions

A VS Math Model reads parameter files (called *Parsfiles*) to obtain values of parameters involving the vehicle, test conditions, and import/export options. The Echo files generated at the initialization and termination of every run list all these settings and serve as reference documents for the many keywords that are recognized.

The *VS Math Models* manual describes the syntax for assigning numerical values to model parameters and variables. In this chapter, the concept is extended to include symbolic expressions.

Comments and Continuation of a Line of Text

When scanning a line of text from a Parsfile, the VS Math Model always checks for two special indicators:

1. The character '!' is always interpreted as a comment indicator. If found, the line of text is terminated at the position where the '!' character was found before any other processing.
2. A sequence of three periods '…' is interpreted as an indicator that a self-contained statement continues on the next line. If found, the line of text stops at the position just before the first of the three periods and continues with the first non-blank character of the next line from the Parsfile.

VS Algebraic Expressions and Math Functions

Almost any line of input from a Parsfile may include algebraic expressions.

VS Assignment Commands

After comments are removed and continued lines are merged, the VS Math Model scans the line of text to see if is begins with a keyword that is recognized. If the purpose of the of text is to assign a value to an internal parameter or variables, then the keyword is interpreted as a *VS Assignment Command*.

In most cases, a line of data that assigns a value to a variable (parameter, state variable, etc.) in the VS Math Model has the syntax:

keyword [*index_list*] [=] *expression* [; [[`UNITS`] [=] *units*] [; [*description*]]]

where *keyword* is the name of a variable (or root name of an indexed variable), *expression* is either a number or an algebraic expression, *units* is an optional keyword used to specify the units associated with the parameter or variable specified by *keyword*, and *description* is a new text description of the parameter or variable.

The optional *index_list* may be used for indexed parameters, and has the form:

(*index* *[, *index*])

where each *index* is evaluated to obtain a number. If there are multiple indices, they are separated by commas. Usually, the indices are numbers, e.g., `A_KPI(1,1)`.

A line of data in this form is processed as a VS Assignment Command if *keyword* (possibly with an index-list) is recognized as the name of a variable. In this case the assignment command potentially performs three actions:

1. Assign a value to the identified variable.
2. Optionally change the units for the variable.
3. Optionally change the text description for the variable that will be printed in Echo files and other documentation. (This option is described in a later subsection *Setting Descriptions of Variables* on page 27.)

Every variable installed in the VS Math Model already has a value, assigned units, and a text description. The second and third parts are optional, because most lines of input from a Parsfile exist only to change the values of existing variable.

Tip View an Echo file to see this form used for all parameters used in the model. In general, the syntax shown in the Echo file is recommended. However, the optional '=' sign is skipped in the Echo file to reduce the amount of text. When entering commands in miscellaneous fields, the '=' sign is recommended to make the command easier to read.

For example, in CarSim the kingpin inclination for the left wheel on axle 1 can be set to 8° with the line of input:

```
A_KPI(1,1) = 8
```

The Echo file shows more information:

```
A_KPI(1,1)     8 ; deg ! Kingpin inclination for axle 1, L wheel [I]
```

Some alternative inputs might be:

```
A_KPI(1,1) = 8 ; deg ! ensure units are degrees
A_KPI(1,1) 0.140 ; - ! specify value using radians '-'
A_KPI(1,1) 8 ; deg ; Reference kingpin inc. angle ! change description
A_KPI(1,1) 8 ;; Reference kingpin inc. angle ! change description
```

All of the above statements set the parameter `A_KPI(1,1)` to 8° (0.140 radian), specifying the numerical value directly.

Read-Only Values

A VS Math Model includes internal variables with associated keywords that are defined as "read only." These include `PI` (π = 3.1415926…), `DR` (degrees per radian = 180/π = 57. 2957…), `G` (9.80665), `T` (current simulation time), and `ZERO`. There are also many properties of the specific math model that are calculated and have keywords, such as total mass and inertia properties, static suspension and tire loads, etc.

Read-only values can be used in formulas but cannot be assigned new values. Trying to assign a value to a read-only variable will generate an error and cause the simulation to stop. Read-only values are shown as comments in the Echo file, to prevent errors in case the Echo file is later used as an input.

Syntax for a VS Formula

The value assigned to a VS parameter or variable from the right-hand side of the '=' sign may be a symbolic formula. The simplest symbolic formula is a number; the next simplest is the name of another variable. For example, the kingpin inclination for the right wheel be set match the kingpin inclination for the left wheel on the same axle with the line:

```
A_KPI(1,2) = A_KPI(1,1) ! they are the same
```

If the VS Math Model recognizes the name of a variable, then that name can be used on either side of the '=' sign. The advantage of using the symbolic version is that if the angle is changed (e.g., `A_KPI(1,1)`), it only needs to be changed in one line of the input text.

Notes When the keyword associated with a parameter or variable appears in an algebraic expression or the right-hand side of the (optional) '=' sign, it is replaced by the current value of the linked internal variable. When it appears on the left-hand side, it is processed as part of a VS Assignment Command to set a new value for the linked internal variable.

If using a symbolic expression on the right-hand side of the '=' sign, be careful that any variables in the expression have been properly updated. For example, in the example above, if `A_KPI(1,1)` is changed after the value is assigned to `A_KPI(1,2)`, the value of `A_KPI(1,2)` is not automatically updated.

Mathematical Operators and Math Functions

VS Expressions can include basic mathematical operators. Table 1 lists the operators in the order of precedence. Operators in the same row (e.g., X*Y, X/Y, and X%Y) have the same precedence, which means they are evaluated left to right. If there is any doubt in operator precedence, use parentheses to ensure the calculations will be made as intended.

Most of the variables in a VS Math Model are defined internally as 64-bit floating-point numbers. The only integer variables that would be used in a symbolic expression are parameters that typically identify a part (e.g., `IAXLE`, `ISIDE`) or specify an option (e.g., `OPT_DM`).

Note When shown in an Echo file, integer parameters can be identified because they do not have associated units.

All results of operators listed in Table 1 are 64-bit floating-point numbers. For example, the results of applying the logical operators (==, <, etc.) are always 0.0 or 1.0.

Table 1. Expression operator precedence in VS Math Models and VS Visualizer.

Operator	**Description**
`()`	Priority: expressions in parentheses are evaluated first
`x^y`	Exponentiation: x^y
`-x`	Unary minus
`~x`	Unary logical not (result is 1 if x is 0, else 0)

`x * y`
`x / y`
`x % y`
	Multiplication, division, and modulo

`x + y`
`x -y`
	Addition and subtraction

`x == y`
`x < y`
`x <= y`
`x ~= y`
`x > y`
`x >= y`
	Comparisons between *x* and *y*; result is 1 if the comparison is true, 0 if it is false

`x & y`	Result is 1 if both x and y are not 0, else 0
`x | y`	Result is 1 if either x or y are not 0, else 0

VS Expressions also support math functions common for scientific computing (Table 2). A few of the function descriptions include the note “with range check.” These functions have range limits, such that a value out of range is automatically corrected. For example, negative numbers are not valid for `SQRT`. In these cases, the function checks the range of the arguments and sets it to the limit if out of range. For example, if given an argument of -0.001, `SQRT` will return 0 rather than crash.

Note Floating point data, whether 32-bit or 64-bit, does not always hold real numbers faithfully due to the nature of its underlying representation. For example, a 32-bit float (IEEE754) can represent 0.25 precisely, but its representation of 0.2 is actually 2.00000002980... Even large integers are somewhat approximated as 123456789 is represented as 1.23456792E8.

The issue is muted somewhat for 64-bit values, but still present. The point is that floating point representation cannot always represent a precise constant value that might be entered in by a user. For more information on floating-point arithmetic can affect computational results, look at "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*" by David Goldberg (ACM, 1991). It is easily found with an internet search.

Table 2. Math functions supported in VS Math Models and VS Visualizer symbolic expressions.

Function	Description	VS Visualizer?
`ABS`(*x*)	Absolute value	•
`ACOS`(*x*)	Arc-cosine (with range check)	•
`ASIN`(*x*)	Arc-sine (with range check)	•
`ATAN`(*x*)	Arc-tan with result $\pm \pi/2$	•
`ATAN2`(*y*, *x*)	Arc-tan(*y/x*) with result $\pm \pi$	•
`CEIL`(*x*)	Smallest integer $\geq x$	•
`COS`(*x*)	Cosine function	•
`COSH`(*x*)	Hyperbolic cosine function	•
`EXP`(*x*)	e^x	•
`FLOOR`(*x*)	Largest integer $\leq x$	•
`FIX`(*x*)	Truncate to integer closer to zero	
`IF`(*x*, *y*, *z*)	If *x* is not 0, return *y;* otherwise return *z*. Between *y* and *z*, only the one used is evaluated.	•
`INT`(*x*)	Nearest integer to *x*	•
`LOG`(*x*)	Natural log (base e)	•
`LOG10`(*x*)	Base 10 log function	•
`MAX`(*x*, *y*)	Maximum of two arguments	•
`MIN`(*x*, *y*)	Minimum of two arguments	•
`SIN`(*x*)	Sine function	•
`SINH`(*x*)	Hyperbolic sine function	•
`SIGN`(*x*, *y*)	If $y > 0$ Then $\|x\|$ Else $-\|x\|$	
`SQRT`(*x*)	Square root (with range check)	•
`TAN`(*x*)	Tangent function	•
`TANH`(*x*)	Hyperbolic tangent function	•

The function `IF` mimics the special IF forms provided in programming languages. It has three arguments (x, y, z) that can be expressions. When this function is evaluated, the first argument is always evaluated first; if the result is anything except zero, then the second expression (y) is evaluated and returned. In this case, the third argument is not evaluated. However, if the first argument evaluates to zero, then the third argument is evaluated and the result is returned; in this case, the second argument is not evaluated. The fact that only two of the three arguments are evaluated is helpful if the unused argument were an expression that would cause an error if evaluated. For example, a "safe" expression for a turn curvature (A_y/v_x^2) might be: `if(vx,ay/vx^2,0)`. If the variable V_x is anything except 0, the function returns A_y/v_x^2; however, if V_x is zero, the function returns 0 and a divide-by-zero error message is avoided.

Symbolic Expressions and VS Visualizer

VS Visualizer is the application used in VS products to visualize results via animation and plotting. VS Visualizer reads data from an output file generated by the VS Math Model. Plots are specified with names of output variables. Animations are set up with moving coordinate systems, defined in 3D space using output variables.

In setting up plots or animation motions, variables used by VS Visualizer are specified with algebraic expressions. In most cases, the variables of interest exist as outputs, such as `Vx`, `Vy`, `Ax`, `Ay`, `Xo`, etc., and are specified by their names. However, VS Visualizer can also handle formulas constructed from output variable names, the math operators from Table 1, and most of the functions from Table 2. (The last column indicates which functions also work in VS Visualizer.)

Special Functions

Each function listed in Table 2 produces a value by making calculations involving the argument(s).

Table 3 lists several "special" VS functions that have the syntax of a function call but use additional resources to produce a result.

Table 3. Special functions supported in VS symbolic expressions.

Special Function	**Description**
`RAND(`*x*`)`	Generate pseudo-random number between 0 and 1.0. The argument *x* is ignored.
`GENSEED(`*x*`)`	Generate truly random seed for use with `SRAND`. The argument *x* is ignored.
`SRAND(`*x*`)`	Seed the random number generator used by `RAND` with *x*, which should be a positive number. It is converted to an integer internally and sent to a C function `srand`. This function always returns a value 1.0.
`PARTIAL(`*func, xcol, x, i*`)`	Partial derivative of a Configurable Function with respect to one of the two possible independent variables
`PARTIAL2(`*func, xcol, x, i*`)`	
`INVERSE(`*func, fg, x, i*`)`	Inverse of a Configurable Function with one (1D) or two (2D) variable functions.
`INVERSE(`*func, y, x, i*`)`	
`PYTHON(`c, f, s, i, o`)`	Interface to Python routine with tables for I/O.

The first three functions in Table 3 exist to generate random numbers using an internal random number generator. The function `RAND` produces a pseudo-random number and is also unusual because the argument x is not used. (VS functions all require an argument.) The `SRAND` function is unusual because the returned value is always the same: 1.0. The function exists because when it is used, it sets up the internal random number generator, so the series of pseudo-random numbers generated are not always the same. The function `GENSEED` creates a random number that can be fed into `SRAND`, for example: `SRAND(GENSEED(0))`. If you use `SRAND(GENSEED(0))` to set up the random number generator, then simulations involving `RAND` will have different sequences of random numbers.

The functions PARTIAL and PARTIAL2 are used to obtain partial derivatives of Configurable Functions. As described later in the Configurable Functions chapter (page 70), a Configurable Function calculates a value as a potentially nonlinear function of one or two independent variables called *X* and *Xcol*. (When the functions use 2D tables, the calculated function values are provided in rows for values of *X*, and columns for values of *Xcol*.) The inputs to the functions PARTIAL and PARTIAL2 are:

- *func* is the name of the Configurable Function,
- *Xcol* is the value of the column independent variable (ignored if *func* has only a row independent variable),
- *X* is the value of the row independent variable, and
- *i* is index for the table dataset.

A typical call might be like this:

```
EQ_IN Kfx = partial(fx_tire,fz_test,kappa,itab)
```

where Kfx is a stiffness defined by ∂Fx/∂K for a given value of Fz.

Note As suggested by the name, Configurable Functions may be configured to support many calculation methods. Not all Configurable Functions support all calculation methods; functions that do not support 2D tables (with two independent variables) will ignore the *Xcol* value and will always cause PARTIAL2 to return a zero value.

Some Configurable Functions support a user-defined symbolic equation to calculate a value. If configured with the _EQUATION option, the PARTIAL and PARTIAL2 functions will always return zero (they do not have enough information to develop a formula for the derivative of a user-defined formula).

The function INVERSE is used to obtain the inverse value of a Configurable Function, if available. The function can be applied to Configurable Functions of either one or two independent variables. The INVERSE function is used as follows:

- *func* is the name of the Configurable Function,
- *Fg* (or *Y*) is function value *Fg* (2D) or the *Y* value of the column input variable (1D).
- *X* is the value of the row independent variable (2D only; ignored for 1D functions).
- *i* is index for the table dataset.

Some calls and a call of INVERSE showing usage might be like this:

itab = 1

In_val = 0.333

```
EQ_IN Tval = fs_ext(0,In_val,itab)
```

```
        EQ_IN Ival = inverse(fs_ext,Tval,0,itab)
```

In this case, Ival would have the value of In_val, or 0.333.

The last function listed in Table 3 is `PYTHON`, an interface used to apply a Python routine. `PYTHON` is executed with the Python interpreter that might have been loaded into the VS Math Model. Numerical input values are provided to the routine with a table. The routine is called as follows:

```
EQ_IN pyret = PYTHON(pycond,"module.func","Signal",in_table,out_table)
```

This function is a bit different than other functions in that:

- it has string inputs,
- is not intended to nest with other functions, and
- has a return value that is secondary to its output table.

In addition to the tables for I/O, the routine accepts a conditional value (0 or 1) to determine whether the routine should execute, a string indicating the Python module and function, and a signal string to provide text input to the routine if needed. The use of the Embedded Python utility is described in a later section (page 84) and explained further in the technical memo *Extending a Model with Embedded Python*.

Indices for Parameters and Configurable Functions

Because a vehicle has many features that are repeated (axles, wheels, etc.), VS Math Models often have parameters that are indexed relative to repeated components in the vehicle, as shown earlier for `A_KPI` indexed parameters.

Indices are sometimes added for more complicated models

Indices are usually not used in a VS Math Model unless they are needed for that specific model. For example, if a VS Math Model has a single vehicle without any trailers, the indexed parameters do not have an index for unit number or vehicle number. However, if there is a trailer, many parameters and Configurable Functions include the unit number as an index.

TruckSim always has an index for dual tires (1 or 2 for inner/outer); BikeSim never has dual tires; and CarSim has a dual tire option, enabled with the keyword command `SET_DUALS = 1`. Depending on the model in CarSim, a tire parameter might have as few as two indices (axle and side), or as many as four (unit, axle, side, inner/outer). For example, the parameter description `RRE(1,2)` will be valid in CarSim for the right-front tire if there are no duals or trailer, while the parameter description `RRE(1,1,2,1)` would be needed for the inner tire on the right-front wheel if the model had duals and a trailer.

In the cases of CarSim or TruckSim simulations with multiple vehicles, parameters in the powertrain and steering that are not indexed for a single vehicle (e.g., `OPT_POWER` for power steering) will be indexed with the vehicle number (e.g., `OPT_POWER(2)` for vehicle number 2).

Use of root keywords for indexed parameters and Configurable Functions

When a root name of a set of indexed parameters is encountered as a keyword in a Parsfile, the current values of integer *index parameters* (`IVEHICLE`, `IUNIT`, `IAXLE`, `ISIDE`, `ITIRE`, etc.) are used to indicate which specific indexed parameter is identified with the root name.

For example, if the index parameter IAXLE is currently 1 and ISIDE is currently 2, then when the keyword A_KPI is encountered, it is interpreted the same as if it were the keyword A_KPI(1,2).

The rules for identifying an indexed parameter are also applied to commands for Configurable Functions that are used with multiple datasets. For example, there is a function for calculating vertical tire force from tire deflection: FZ_TIRE. If the name FZ_TIRE_COEFFICIENT is encountered while IAXLE is 1 and ISIDE is 2, then the effect is the same as if the keyword had been FZ_TIRE_COEFFICIENT(1,2) (assuming the model is in CarSim without dual tires and with just one vehicle unit).

The rules for identifying an indexed parameter or Configurable Function using either a keyword followed by an index list, e.g., A_KPI(1,2), or the root keyword combined with current values of supporting integer parameters, e.g., A_KPI, together with IAXLE and ISIDE, also apply to VS Expressions. For example, the assignment statement:

```
        L_RELAX_Y_CONSTANT(1,2) = 2*RRE(1,2)
```

could also be written with as:

```
        L_RELAX_Y_CONSTANT = 2*RRE
```

if the indexed parameters were set such that IAXLE = 1 and ISIDE = 2.

Note In this example, RRE(1,2) is an indexed parameter for a tire effective rolling radius, for axle 1 and side 2 (right); L_RELAX_Y is a Configurable Function for the lateral relaxation length of the same tire, with L_RELAX_Y_CONSTANT(1,2) being a command that sets the function for the tire to be a constant whose value is calculated using the formula 2*RRE(1,2) using the value of RRE(1,2) at the time the statement was read by the VS Math Model.

Rather than setting the Configurable Function to a constant, this convention can also be used to scale a nonlinear tabular function. For example, if the table is normalized with respect to amplitude, it can be scaled using the gain, specified with an indexed parameter with the root name L_RELAX_Y_GAIN. (More information about transformation parameters in Configurable Functions are described in a later chapter, page 70.) There is no built-in yellow field for a table gain such as this, so an advanced user might use a miscellaneous yellow field on the generic table screen to place an input such as:

```
        L_RELAX_Y_GAIN = 2*RRE
```

As with the other example, this entry will always use the values of IAXLE and ISIDE to identify the instances of RRE and L_RELAX_Y_GAIN and set the value of L_RELAX_Y_GAIN based on the current value of RRE.

Adding VS Commands that involve indexed parameters

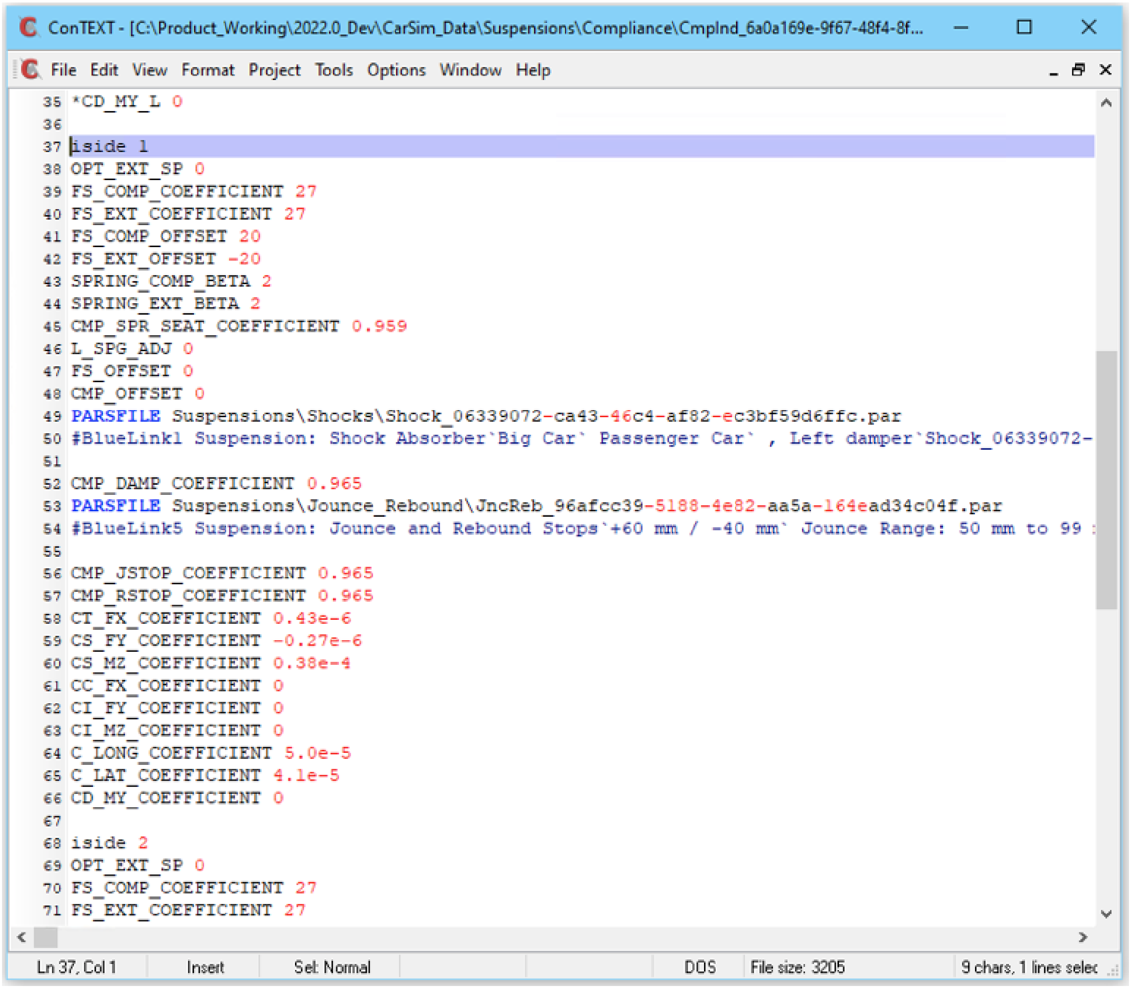
The option to set values using root keywords is much more portable and is almost exclusively used for the Parsfiles generated with a VS Browser such as carsim.exe. When possible, example datasets with VS Commands that are included with VS products use formulas and parameter names with root names, rather than explicit indices. This practice is recommended for most cases.

When a VS Command equation involves existing indexed parameters or Configurable Functions, it is usually easiest (and safest) to specify the equation in a miscellaneous field for a screen where the indices are set by values of index variables (IAXLE, ISIDE, etc.), as in the previous examples for tire parameters. For example, the statement shown above (L_RELAX_Y_GAIN = 2*RRE) could be entered into the optional miscellaneous yellow field on the Tire screen in CarSim or TruckSim. All index parameters that might be needed to identify the tire have been set properly, given the design of the library links in the VS Browser.

However, the practice of using root keywords in VS Command expressions does require some awareness on the part of the user for the current values of index parameters for a given screen or miscellaneous field.

Be aware that some screens involve multiple indices for some parts. Suspension screens (with IUNIT and IAXLE values set outside the Parsfile) set parameters for the two sides (set with distinct ISIDE values). For example, Figure 3 shows part of a Parsfile for a front **Suspension: Springs, Dampers, and Compliance** dataset in CarSim. The index parameters IVEHICLE, IUNIT, and IAXLE were assigned values in Parsfiles that were read before the linked Parsfile in the figure was provided to the VS Math Model, such that indexed parameters in the Parsfile are set for the proper vehicle, unit, and axle. The Parsfile is written with the statement "iside 1" (line 37) such that Configurable Function datasets and indexed parameters specified in the following lines apply for the left side. The side is changed in line 68 (iside 2), such that the following statements apply to Configurable Function datasets and indexed parameters on the right-hand side.

The same approach can be used in VS Commands that are linked to suspension screens. A VS Command dataset that is linked to a suspension dataset will be read by the VS Math Model with the proper settings for IVEHICLE, IUNIT, and IAXLE. The index parameter ISIDE can be set as needed to set the proper side for custom VS Commands that involve indexed parameters or Configurable Functions that can differ on the two sides.


ConTEXT - [C:\Product_Working\2022.0_Dev\CarSim_Data\Suspensions\Compliance\CmpInd_6a0a169e-9f67-48f4-8f...
File Edit View Format Project Tools Options Window Help

```
*CD_MY_L 0

iside 1
OPT_EXT_SP 0
FS_COMP_COEFFICIENT 27
FS_EXT_COEFFICIENT 27
FS_COMP_OFFSET 20
FS_EXT_OFFSET -20
SPRING_COMP_BETA 2
SPRING_EXT_BETA 2
CMP_SPR_SEAT_COEFFICIENT 0.959
L_SPG_ADJ 0
FS_OFFSET 0
CMP_OFFSET 0
PARSFILE Suspensions\Shocks\Shock_06339072-ca43-46c4-af82-ec3bf59d6ffc.par
#BlueLink1 Suspension: Shock Absorber`Big Car` Passenger Car` , Left damper`Shock_06339072-

CMP_DAMP_COEFFICIENT 0.965
PARSFILE Suspensions\Jounce_Rebound\JncReb_96afcc39-5188-4e82-aa5a-164ead34c04f.par
#BlueLink5 Suspension: Jounce and Rebound Stops`+60 mm / -40 mm` Jounce Range: 50 mm to 99

CMP_JSTOP_COEFFICIENT 0.965
CMP_RSTOP_COEFFICIENT 0.965
CT_FX_COEFFICIENT 0.43e-6
CS_FY_COEFFICIENT -0.27e-6
CS_MZ_COEFFICIENT 0.38e-4
CC_FX_COEFFICIENT 0
CI_FY_COEFFICIENT 0
CI_MZ_COEFFICIENT 0
C_LONG_COEFFICIENT 5.0e-5
C_LAT_COEFFICIENT 4.1e-5
CD_MY_COEFFICIENT 0

iside 2
OPT_EXT_SP 0
FS_COMP_COEFFICIENT 27
FS_EXT_COEFFICIENT 27
```

```
Ln 37, Col 1    Insert    Sel: Normal    DOS    File size: 3205    9 chars, 1 lines selec
```

Figure 3. Parsfile for a CarSim dataset for suspension springs, dampers, and compliance.

Alert Given that the VS Browser sets some index parameters to provide the proper context for data on the screen (e.g., Figure 3 shows a Parsfile that resets the parameter `ISIDE`, while leaving other index parameters such as `IAXLE` and `IUNIT` "as is"), care must be taken not to set index parameters in yellow fields on screens that might interfere with the values that are written automatically by the VS Browser.

Typical Settings of Index Parameters from VS Browsers

When choosing locations in the VS Browser for providing VS Commands, you should consider how the VS Browser sets values of index parameters for the simulation. Consider the following sections of a **Run Control** setup:

1. The top of the Parsfile for the **Run Control** dataset sets `IUNIT` and `IVEHICLE` to 0, anticipating that vehicle screens will increment the values as vehicles and trailers are added to the model with linked vehicle dataset(s).

2. Each Lead Unit vehicle screen increments IUNIT and IVEHICLE. The new values apply for aerodynamics, powertrain, and sprung mass properties.
3. Each trailer vehicle screen increments IUNIT, whose value applies for aerodynamic and sprung mass properties.
4. Each vehicle screen (lead units and trailers) sets IAXLE prior to each link to axle-based datasets, such as suspensions, brakes, and tires.
5. The vehicle screens also set ISIDE prior to links to tire data. However, if the vehicle has dual tires, IAXLE, ISIDE, and ITIRE are set in a linked Tire Group dataset.
6. Datasets that add optional parts (payloads, moving objects, ADAS sensors, motion sensors, etc.) use VS Commands to define new parts (e.g., DEFINE_PAYLOADS, DEFINE_MOVING_OBJECTS, etc.). Those commands also increment associated index parameters (e.g., ILOAD, IOBJECT, etc.).
7. Datasets that add paths and roads use VS Commands such as SET_IPATH_FOR_ID and SET_IROAD_FOR_ID, which increment associated index parameters IPATH and IROAD, respectively.

When choosing where to locate VS Commands that use formulas that interact with indexed parameters or Configurable Functions, it is a good idea to consider where the VS Browser screen fits into the sequence of data provided from the VS Browser to the VS Math Model. The miscellaneous field on the **Run Control** screen is sent after almost everything else, so any changes made to index parameters there will not interfere with the automatic management for the vehicle, path, and other screens. The same is also true for Generic datasets that are linked to the **Run Control** screen after the vehicle and Procedure links. On the other hand, if the intent is to add VS Commands involving a vehicle part, then it might be appropriate to link to a Generic dataset (that has VS Commands) from a vehicle screen, to take advantage of index parameters such as IUNIT and IAXLE that have been given suitable values automatically.

Unit Systems: User Display Units and Internal SI Units

A VS Math Model maintains two major sets of units:

1. All internal calculations made during a run assume all parameters and variables are scaled according to the *International System of Units* (SI), which require no scale factors in the internal equations of motion. Length and distance variables use meters, mass variables use kilograms, force variables use Newtons, angular variables use radians, acceleration variables use m/s^2, pressure variables use Pascals, etc.
2. *User display units* are accepted as inputs and shown for outputs. All machine-generated documentation files (Echo files, etc.) show values of variables and parameters using user display units. The VS Math Model automatically scales values as needed to convert to and from internal SI units during input and output.

VS Math Models support user display unit systems to allow inputs to be provided with convenient units such as mm, degrees, g, etc. Also, new display units can be provided at runtime (e.g., feet, inches, etc.) to support inputs with unit systems that are not built in. The handling of units is as follows:

- All calculations are made with internal SI units.
- Any numerical values read from files or imported from other software (e.g., Simulink) are assumed to have user display units; therefore, they are divided by a gain associated with the user display units at the time they are read or imported.
- Any numerical values with internal SI units written to file or exported to other software are multiplied by the gain associated with the user display units at the time they are written or exported.

For example, consider a wheelbase parameter `LX_AXLE` (longitudinal axle location) with user display units of millimeters. The value provided is 3000 (mm). When read by the VS Math Model, it is divided by 1000 such that the value used in all calculations during the simulation is 3.000m. When written in an Echo file for user documentation, the internal value of 3.000 is multiplied by 1000 to write the value 3000mm.

The Echo file can display the complete set of units used for any run. To specify this option, enter the keyword `OPT_ECHO_ALL_UNITS` with a value of 1 in a miscellaneous data field, and run a simulation. The list of units appears at the beginning of the new Echo file, available from the drop-down menu in the lower-right corner of the **Run Control** screen.

When a parameter is given a numerical value, the value should be expressed in the user display units associated with the parameter. The VS Math Model will perform conversions as needed. For example, the 8° kingpin inclination angle in a previous example would be automatically converted by the solver to 0.139626 radian for use in the internal equations.

If you want to provide a numerical value with different units, then include the name of the units associated with the value after the ';' delimiter, e.g.,

```
A_KPI(1,1) 0.139626; - ! dimensionless units have the name '-'
```

In the syntax of a VS Math Model Parsfile, an expression is either numerical or symbolic. If the expression includes the symbolic name of any variable or parameter, it is considered symbolic. On the other hand, if the expression is a number or a mathematical expression involving only numbers, then the expression is considered numerical. This distinction determines whether a conversion is done based on units.

Consider the previous example:

```
A_KPI(1,2) = A_KPI(1,1) ; ! they are the same
```

In this case, both angular parameters have internal SI units of radians, so no conversion is needed when assigning the same value to the parameter `A_KPI(1,2)`, regardless of whether the parameters are written in Echo files with radians or degrees.

Consider another example. If the CarSim aerodynamic reference length were specified with a numerical value, the value would be in millimeters:

```
L_REF_AERO   3048
```

Alternatively, it could be specified as the wheelbase, which in turn is the difference between the front and rear axle locations:

```
L_REF_AERO = LX_AXLE(2) - LX_AXLE(1) ;
```

In the second case, the implied units are meters (SI). This example is typical in that the distinction about the units is not obvious because all the symbols in the equation have the same units. However, if you were to define the aerodynamic reference length to be 500mm longer than the wheelbase, then you would need to convert the numerical value of 500mm to the internal SI units of meters, with a value of 0.5:

```
L_REF_AERO = LX_AXLE(2) - LX_AXLE(1) + 0.5 ;
```

The optional specification of units after the ';' delimiter changes the units associated with the variable and will affect how the value is written in the Echo file. However, if *expression* includes a symbol, then the expression is always interpreted with SI (internal) units, regardless of the units associated with the variable.

Dynamic Strings

Most variables in a VS Math Model have numerical values. However, a few are strings of text, used as titles, messages, file names, etc. For example, the `LINEARIZE` command specifies the name of a file to which MATLAB code is written with a linear description of the current VS Math Model state. There are also some VS Commands that involve a string of text.

Use of Double Quotes in Strings

Lines in Parsfiles that specify strings of text have a different interpretation than other lines. In most cases, the first word is the keyword that identifies the command, and the remainder of the line is the string of text. Any spaces between the keyword and the first word of the rest of the line are removed, and any spaces at the end of the line are removed. There should not be a '=' symbol after the keyword (it would be interpreted as the first character of the text string).

It is also possible to specify titles and messages using a mixture of strings identified by double quote marks (") combined with printed values of symbolic expressions.

The rule used by the VS Math Model for obtaining a string of text is:

- If the line of input after the command does not contain any double quote marks, then the string is defined with the first non-blank character and continues to the last non-blank character on the line.
- Otherwise, the line of input after the command is interpreted as a sequence of strings separated by '+' delimiters.

For example, consider the command:

```
STOP_RUN_NOW The run was stopped because the speed was too low.
```

When this command is encountered, the VS Math Model will stop the run and write a line into the Log file that reads: "`The run was stopped because the speed was too low.`"

Here is another use of the command:

```
    STOP_RUN_NOW "The run was stopped at V = " + int(Vx) + " m/s."
```

Suppose the speed is 4.9999 m/s when the command is encountered. In this case, the VS Math Model will write this line into the log file: "`The run was stopped at V = 5 m/s.`"

The syntax for a command such as this is:

command string1 *[+ *string*]

where at least one of the *string* values must be enclosed in double-quote marks.

Any strings not enclosed in quotes are assumed to be valid symbolic expressions. Accordingly, they are evaluated and printed into a string. In the above example, the expression `int(Vx)` was taken as a symbolic math expression, evaluated using the current value of the output variable `Vx`, printed to a string to yield "5", and that string was combined with the other strings.

The three following examples all have exactly the same result.

```
STOP_RUN_NOW The run was stopped because the speed was too low.
STOP_RUN_NOW "The run was stopped because the speed was too low."
STOP_RUN_NOW "The run was stopped " + "because the speed was too low."
```

If there are no quotes, the effect is the same as if the remainder of the input line is quoted. For example, consider the input:

```
    STOP_RUN_NOW The run was stopped at V =  + int(Vx) +  m/s.
```

In this case, the VS Math Model will write this line into the log file:

```
    The run was stopped at V =  + int(Vx) +  m/s.
```

Because there were no double quote marks, everything after the command was copied as a string and the symbolic expression `int(Vx)` was not evaluated.

Backward Compatibility

Dynamic strings were added to VS Math Models in 2013 (the first products were CarSim 8.2, BikeSim 3.2, TruckSim 8.2). Older datasets with titles and other strings that have quoted words are likely to generate error messages. Because the double quote character defines how the line of input is interpreted, it can no longer be used in its normal role in English. Instead, use the single quote character. For example, instead of a title:

```
    This is a "fishhook test" run
```

Use the title:

```
    This is a 'fishhook test' run
```

Setting Descriptions of Variables

As noted earlier, any variable in a VS Math Model that has a keyword used to identify the variable also has descriptive text that is written in Echo files and other machine-generated documents. Any statement that begins with the keyword for a variable has the syntax presented earlier:

keyword [*index_list*] [=] *expression* [; [[`UNITS`] [=] *units*] [; [*description*]]] where *keyword* is the name of a variable (or root name of an indexed variable), *index_list* may provide indices for an indexed parameter, *expression* is either a number or an algebraic expression, *units* is an optional keyword used to specify the units associated with the parameter or variable specified by *keyword*, and *description* is a new text description of the parameter or variable.

When the VS Math Model processes a statement in this form, it looks for a second semicolon ‘;’ in the line of text. If found, the remainder of the line of text is processed as a dynamic string. That string then replaces the existing description text for the variable.

If no text follows the second semicolon, then the existing description text for the variable is not changed.

Another method for setting the descriptive text for a variable is via the `SET_DESCRIPTION` command, described later (page 46).

3. Extend VS Math Models

Each VS Math Model contains highly optimized equations for the physics of the vehicle, plus predefined controls and environmental interactions (3D road, wind, etc.). By itself, a VS Math Model covers the complete system-level dynamic behavior of the vehicle as controlled by a driver or rider, with closed-loop control options sufficient to mimic driver/rider behavior in basic maneuvers that have traditionally been used by manufacturers to characterize vehicle behavior.

Several options are available for extending a VS Math Model at runtime, using combinations of third-party tools and built-in capabilities of the VS Math Model (Table 4).

Table 4. Options for extending a VS Math Model at runtime.

Method	Built-In Support	External Tool	Description
Built-in options	Math models include common control and configuration options	none	Use built-in options for setting controls, tire models, powertrain options, etc.
Optional modules	Commands for adding optional features with multiple variables	none	Commands such as `DEFINE_PATHS` add features to the model
VS Commands	VS Math Models recognize VS Commands	none	Add variables and equations to extend the model
Embedded Python	Interface to Python	Python	Apply Python functions from within the VS Math Model
Import/Export variables	Import and export arrays activated at runtime	Simulink, LabVIEW, ASCET, FMI, Custom programs	Several hundred built-in variables can be replaced or modified with values imported from other software.
		Real-time systems with hardware in the loop (HIL)	Live measures from HIL are imported to the VS Math Model
VS API	VS Math Models have documented API	Programming languages such as C/C++, Python, VB, MATLAB, etc.	External program can control the simulation, access many variables in the model, and add complex calculations

The first row in the table is a reference to many built-in options that can be activated or disabled based on settings. For example, CarSim supports powertrains with rear-wheel drive, front-wheel drive, and all-wheel drive, with more options for transmission types and other features. Every VS Math Model for BikeSim, CarSim, and TruckSim includes many built-in options that are mutually exclusive. Changing an option can drastically modify the capabilities of the model by eliminating or adding features. Built-in options are described by Echo files written by the VS Math Models, as described in the **Help** document **Reference Manuals > VS Math Models** and other documents available from the **Help** menu (powertrain options are documented in the **Powertrain** help document, tire options are documented in the **Tire Models** help document, etc.)

Optional modules are features built into the VS Math Models that require commands to install or define them. These include Payloads, Moving Objects, Sensors, Paths, Roads, and others. Some of these are very specific to the module and are described in the documentation for the module.

VS Math Models include a scripting language called VS Commands, which is the subject of this manual. VS Commands allow you to add variables and equations to the model, and script changes in the settings of the model as a simulation runs.

In addition to the commands and functions described in this document, other VS Commands that are specific to model options are documented along with the other model options. For example, the command for adding a payload is described in the *Payloads* document. There are quite a few commands related to paths and roads; these are described in the *Paths and Road Surfaces* document.

When running on Windows or non-RT Linux, VS Math Models can load Python and run Python code from within the model, in a similar manner to VS Commands. This document includes the basic commands needed to load Python and apply user-defined functions (page 84). For more information, please see the Technical Memo *Extending a Model with Embedded Python*.

The options for using Import and Export variables to communicate with external software (Simulink, LabVIEW, Functional Mockup Interface (FMI), etc.) involve making a model in the external software, such as an advanced controller, and connecting it to the VS Math Model using appropriate Import and Export variables as described in the *VS Math Models Manual*.

Options for extending a VS Math Model with routines written in programming languages such as MATLAB, Visual Basic, C/C++, etc. are supported by the VS application program interface (API) as described in the *VS API Manual*, which is part of the VS SDK (software development kit).

The options listed in Table 4 are often combined. Even when external programs interact with the VS Math Model using arrays or using the VS API, it is still common for advanced users to use VS Commands to customize the integration between the VS Math Model and the external software.

4. Basic VS Commands

A VS Math Model has an internal database with thousands of keywords that are recognized and linked internally to the parameters and variables within the VS Math Model. In addition, VS Commands can be used to extend the model defining new variables, adding equations, controlling the simulation procedure with scripted VS Events, and applying advanced simulation tools such as linearization or jumping back in time.

Commands and Functions

Be aware that there is a fundamental difference between a function and a command. Functions are used in symbolic expressions as described in Chapter 2 and return a value. Functions are used within expressions that are used to calculate a value that is assigned to a variable. If the optional '=' sign is used, function calls always appear on the right-hand side of '='.

A command is identified by a keyword that is the first thing on a line of text in a Parsfile. For example, consider the line of text:

```
define_units cm 100
```

The word `DEFINE_UNITS` is a VS Command. As with other keywords, the VS Math Model is not sensitive to case when reading them. VS Commands are always shown in ALL CAPS in echo files and in documentation. However, it is common to write them in lower case, especially when using fields in the GUI with limited width (lowercase names take up less room).

Most VS Commands are followed by more information, usually in the form of one or more arguments that are separated by spaces. In the example of the `DEFINE_UNITS` command, there are two arguments.

Alert A VS Command does not return a value and should never be used on the right-hand side of an '=' sign.

Settings and Defining Units

As noted earlier in the section *Unit Systems: User Display Units and Internal SI Units* (page 20), a VS Math Model maintains two major sets of units:

1. All calculations during a run assume all user input parameters and variables are scaled according to the *International System of Units* (SI), which require no scale factors in the internal equations of motion.
2. *User display units* are accepted for inputs and shown for outputs. All machine-generated documentation files (Echo files, etc.) show values of variables and parameters using user display units. The VS Math Model automatically scales values as needed to convert to and from internal SI units during input and output.

Table 5 lists the VS Commands available for adding and managing user display units in a VS Math Model. The complete set of units available in a VS Math Model can be obtained by setting the parameter `OPT_ECHO_ALL_UNITS` to 1 and viewing the Echo file produced for that run.

Table 5. VS Commands for adding and managing user display units.

Command	Action
`DEFINE_UNITS`	Install new units in the VS Math Model.
`REDEFINE_UNITS`	Redefine existing units in the VS Math Model.
`SET_UNITS`	Set the units for a variable or Configurable Function.
`SET_UNITS_TABLE_ROW`	Set the units for the row variable in a Configurable Function.
`SET_UNITS_TABLE_COL`	Set the units for the column variable in a Configurable Function.

DEFINE_UNITS

`DEFINE_UNITS` *name gain*

Use this command to add new units to a VS Math Model. For example, to install units of centimeters (written as cm) put this command into an input Parsfile:

```
define_units cm 100
```

Name is the name for the new units, and *gain* is the scale factor used to multiply a value in internal SI units to the new user display units. Note that *gain* can be a formula (e.g., `180/PI`). In this example, the gain is the number of cm per meter.

The command installs the name as a case-insensitive keyword (e.g., `CM`) associated with the units, and keeps the original case-sensitive name (e.g., `cm`) for printing.

Note VehicleSim products use SI units internally and metric units as defaults for user units. Popular English units are also built in, including inches, feet, mi/h, pounds (force and mass), and others.

As mentioned above, you can view all of the units that are installed by setting the system parameter `OPT_ECHO_ALL_UNITS` to 1 and viewing the Echo file.

REDEFINE_UNITS

`REDEFINE_UNITS` *keyword name gain*

This is a powerful (i.e., potentially dangerous) command to universally redefine a type of units, where *name* is the new printed representation and *gain* is the numerical gain to multiply a value when converting from internal SI units to the user-units. For example, to convert units of MM to units of inches, use the command:

```
redefine_units mm in 1/0.0254
```

This modifies the internal representation of the units with the keyword "`MM`." The keyword is still `MM`, but the new printed description is "`in`" and any internal values (meters are the SI units of length) are multiplied by 1/0.0254 (39.370079) when calculating outputs, or divided by 39.370079 when reading inputs.

SET_UNITS Commands

`SET_UNITS` *keyword units*
`SET_UNITS_TABLE_ROW` *keyword units*
`SET_UNITS_TABLE_COL` *keyword units*

Use the `SET_UNITS` command to set the user units for any variable or Configurable Function identified by *keyword*, as described below. Use the `SET_UNITS_TABLE_ROW` and `SET_UNITS_TABLE_COL` commands to set units for row and column variables, respectively, for tables associated with Configurable Functions.

Note More information about setting units for Configurable Functions is provided in Chapter 5 (page 70).

The keyword *units* is normally the same as the printed representation for the units. For example, the keyword for units of millimeters is `MM`. As with other keywords, it is not case sensitive.

Note An in-line option for setting units was added to VS Math Models in 2012. This makes the `SET_UNITS` command unnecessary for most parameters and variables. The command still works, in support of legacy datasets.

Any line of input that specifies a value for a parameter or variable can also set the units associated with the parameter or variable, with an optional ';' character followed by a *units* keyword. For example, to specify a CarSim dimension in meters (rather than the default millimeters), use an input such as:

```
L_REF_AERO 3.048 ; units = m
```

or, more simply:

```
L_REF_AERO 3.048 ; m
```

Using the `SET_UNITS` command, this would be accomplished with two lines of input:

```
SET_UNITS L_REF_AERO m
L_REF_AERO 3.048
```

VS Events

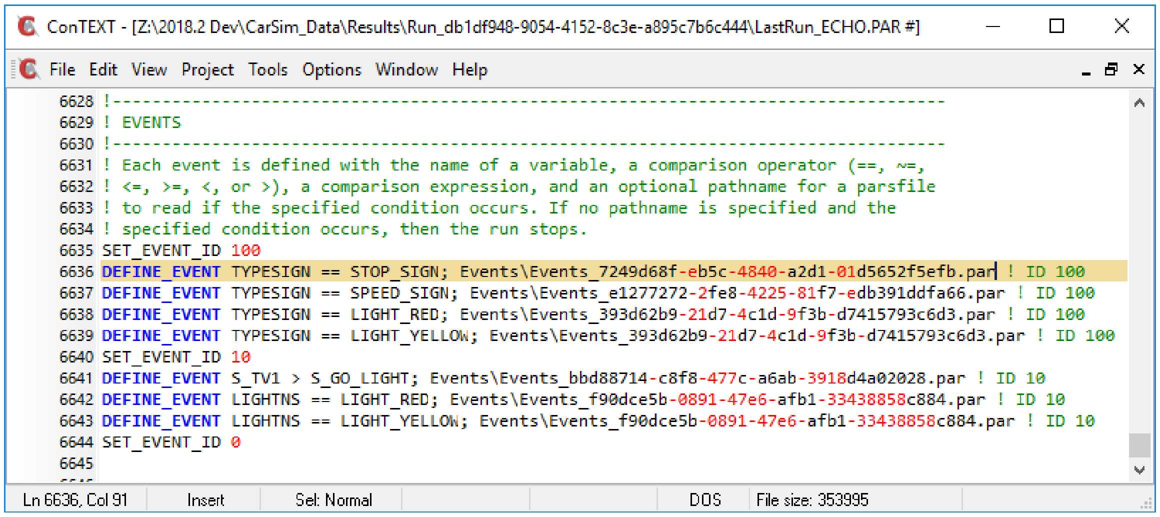
A VS Math Model can monitor a list of conditions that you specify with formulas. When this happens, it is called a *VS Event.* The possible actions associated with an Event are simple: either the run stops, or at least one additional Parsfile is read. The options here are almost unlimited; the new data files can change inputs, control settings, or even vehicle and road properties.

Table 6 lists the VS Commands for handling Events and scripting the simulation.

Table 6. VS Commands for handling Events.

Command	**Action**
`DEFINE_EVENT`	Define new Event.
`DELETE_EVENTS`	Delete all Events created with `MAKE_EVENT` using a specified variable.
`DELETE_EVENTS_ID`	Delete all existing Events that have a specified group ID.
`MAKE_EVENT`	Define new Event with old units convention.
`RESET_EVENTS`	Clear all pending Events.
`SET_EVENT_ID`	Set an integer ID that will be associated with new Events.
`STOP_RUN_NOW`	Stop the simulation run.
`WRITE_LOG`	Write a dynamic string into the log file.

If any Events are pending at the start of a simulation, they are listed in the Echo file after other VS Commands (Figure 4).


ConTEXT - [Z:\2018.2 Dev\CarSim_Data\Results\Run_db1df948-9054-4152-8c3e-a895c7b6c444\LastRun_ECHO.PAR #]
File
Edit
View
Project
Tools
Options
Window
Help
Ln 6636, Col 91
Insert
Sel: Normal
DOS
File size: 353995

```
!--------------------------------------------------------------------------------
! EVENTS
!--------------------------------------------------------------------------------
! Each event is defined with the name of a variable, a comparison operator (==, ~=,
! <=, >=, <, or >), a comparison expression, and an optional pathname for a parsfile
! to read if the specified condition occurs. If no pathname is specified and the
! specified condition occurs, then the run stops.
SET_EVENT_ID 100
DEFINE_EVENT TYPESIGN == STOP_SIGN; Events\Events_7249d68f-eb5c-4840-a2d1-01d5652f5efb.par ! ID 100
DEFINE_EVENT TYPESIGN == SPEED_SIGN; Events\Events_e1277272-2fe8-4225-81f7-edb391ddfa66.par ! ID 100
DEFINE_EVENT TYPESIGN == LIGHT_RED; Events\Events_393d62b9-21d7-4c1d-9f3b-d7415793c6d3.par ! ID 100
DEFINE_EVENT TYPESIGN == LIGHT_YELLOW; Events\Events_393d62b9-21d7-4c1d-9f3b-d7415793c6d3.par ! ID 100
SET_EVENT_ID 10
DEFINE_EVENT S_TV1 > S_GO_LIGHT; Events\Events_bbd88714-c8f8-477c-a6ab-3918d4a02028.par ! ID 10
DEFINE_EVENT LIGHTNS == LIGHT_RED; Events\Events_f90dce5b-0891-47e6-afb1-33438858c884.par ! ID 10
DEFINE_EVENT LIGHTNS == LIGHT_YELLOW; Events\Events_f90dce5b-0891-47e6-afb1-33438858c884.par ! ID 10
SET_EVENT_ID 0
```

Figure 4. Listing of pending Events near the end of an Echo file.

Given that the Echo file might be used as an input for a future simulation, it presents the information in the form of VS Commands needed to create the state of the model that existed with the Echo file was written. The two commands used are `DEFINE_EVENT` and `SET_EVENT_ID`.

Warning Most VS Commands may be written in scrollable miscellaneous yellow fields in a VS Browser. **Do not do this** with the VS Event Commands `DEFINE_EVENT` or `MAKE_EVENT`. If either of these commands are used with the *pathname* argument, then the dataset cannot be exported using the normal **File** menu items of the VS Browser: **Export Consolidated Parsfile** or **Export Expanded Parsfile**. A file will be generated, but it will not import properly.

The VS Browser has an **Events** screen that automatically generates proper `DEFINE_EVENT` and `MAKE_EVENT` commands. Mechanical Simulation recommends that you always use this screen to define Events, rather than attempting to type them in a miscellaneous scrollable field.

DEFINE_EVENT

`DEFINE_EVENT` *condition* [; [*pathname*]]

A VS Event is defined by establishing a pending condition defined by algebraic expression involving any variables recognized by the VS Math Model. (See Chapter 2 for details about VS algebraic expressions.)

Condition is evaluated at the end of each full timestep during the run, with three possible results:

1. If the condition is 0, no action is taken.
2. If the condition not 0 and *pathname* was not provided, then the run is terminated.
3. If the condition not 0 and *pathname* was provided, then the Event is triggered and the following steps are taken by the VS Math Model:
   a. this Event is removed from the list of pending Events,
   b. the specified Parsfile is immediately loaded (along with any Parsfiles referenced with `PARSFILE` links, as is always the case when reading Parsfiles),
   c. the built-in initializations that are needed after reading a new Parsfile are performed, and
   d. the run continues.

The specified Parsfile and others that might be linked to it must not attempt to redefine the information that must remain fixed during the simulation, such as the timestep, the list of output variables, Import variables, etc. However, all Configurable Functions in the VS Math Model and most parameters can be changed.

The initializations (step 3c) involve equations that are built into the VS Math Model, and are performed to ensure consistency. Equations added at runtime using VS Commands as described later are not applied when an Event occurs. The initialization done in this step is often not as thorough as the one done at the start of the simulation, because all state variables have valid values and all derivatives and output variables have already been calculated.

Values of state variables and variables defined with VS Commands can be given new values in the Parsfile(s) read in step 3b. However, the effects on output variables will not be seen until the next timestep.

Chapter 6 (page 88) provides details about the steps taken when an Event is triggered within the context of the many other operations that take place during the simulation process.

DELETE_EVENTS

`DELETE_EVENTS` *variable*

This command removes all pending Events that were created with the command `MAKE_EVENT` that used a named variable. For example, if several Events were created by `MAKE_EVENT` using `Vx` as *variable*, then the command `DELETE_EVENTS Vx` will remove all of those pending Events that used `Vx`. Other Events are left intact.

This command cannot be used to delete Events created with the `DEFINE_EVENT` command.

DELETE_EVENTS_ID

`DELETE_EVENTS_ID` *group_id*

This command removes all pending Events that have the specified *group_id*. Other pending Events are left intact.

MAKE_EVENT

`MAKE_EVENT` *variable operator reference* [; [*pathname*]]

VS Events were introduced in VehicleSim products before any other VS Commands (indeed, even before the name “VehicleSim” was adopted!). In versions 2019.0 and earlier, the text following a `DEFINE_EVENT` command was parsed to obtain four pieces of information: *variable, operator, reference*, and optionally, *pathname*. The intent was that *variable operator reference* in sequence defined a valid formula for a Boolean condition. These items, separated by one or more spaces, are:

- *Variable* is the name (keyword) for any floating-point variable in the model.
- *Operator* is the one of the following six operators: (==, ~=, <, <=, >, >=).
- *Reference* is a number, name of a variable, or formula. If it is a number, it was assumed to have the same user units as *variable*.

In version 2019.1, major improvements were made in the options for specifying formulas with in-line Boolean operators, as described earlier (Table 1, page 16). The text sequence *variable operator reference* was simply replaced in the VS Math Model processing with a single piece of text: *condition*. For example, the text “`T > 10`” can be parsed as *variable* `T`, *operator* >, and *reference* `10`, or, as *condition* `T>10`.

In almost every way, the old convention is a subset of what is available in the simpler syntax for `DEFINE_EVENT`. However, there are two cases where the old and new are not the same. One is when *variable* has user units with an associated scale factor, and *reference* is a number. For example, the text “`Vx > 80`” with the old convention would be the same as the new *condition* “`Vx > 80/3.6`” if the variable `Vx` has user-units km/h (divide km/h speed by 3.6 to obtain m/s speed).

The `MAKE_EVENT` command was introduced in 2019.1 to support legacy datasets prepared with a VS Browser. If an Event is defined with the new format that has a single *condition* formula, then a `DEFINE_EVENT` command is written to the Parsfile by the Browser. However, if an Event is defined with the old format, then a `MAKE_EVENT` command is written.

The `MAKE_EVENT` command acts as a preprocessor for making an event. It performs the following steps:

1. It parses the text after the commend to obtain the text for *variable*, *operator*, and *reference.*
2. It identifies the variables associated with keyword *variable* and obtains the gain of the units associated with the variable. If the gain is unity, then it passes the original text to the `DEFINE_EVENT` command.
3. Otherwise, it evaluates *reference* to see if it is numerical. If *reference* is not numerical, then it passes the original text to the `DEFINE_EVENT` command.
4. Otherwise, it makes a new string of text with *variable, operator, reference*, the '/' symbol, the gain for the units (e.g., 3.6 in the above example for speed), ';', and *pathname*, and passes that text to the `DEFINE_EVENT` command.

A second difference between `MAKE_EVENT` and `DEFINE_EVENT` is that the `DELETE_EVENTS` command was available to delete all pending Events that used the same variable.

The `DELETE_EVENTS` command still exists for this purpose.

Tip The Echo file shows all Events using the `DEFINE_EVENTS` Command. If possible, we recommend using `DEFINE_EVENTS` rather than `MAKE_EVENTS`. For one thing, the Log file is more readable. Another factor is that the Echo file matches the screen display more closely.

RESET_EVENTS

`RESET_EVENTS`

This command removes all pending Events from memory.

SET_EVENTS_ID

`SET_EVENTS_ID` *group_id*

This command specifies that new Events will have the specified *group_id* (an integer). This will remain in effect until the command is used again with a different value of *group_id*.

STOP_RUN_NOW

`STOP_RUN_NOW` *message*

This command causes the run to stop after completing the current timestep. The command also places a string of text into the log file that indicates the run was stopped with this command. The string *message* is required. It can be a dynamic string, created by combining static strings with symbolic expressions that are printed to the form of a string, as described earlier in the section *Dynamic Strings* (page 26).

For example, suppose the following command is encountered when the vehicle forward speed (vx) is 4.9999 m/s:

```
STOP_RUN_NOW "The run was stopped at V = " + int(Vx) + " m/s."
```

In this case, the VS Math Model will stop the run and write this line into the log file: "`The run was stopped at V = 5 m/s.`"

`WRITE_LOG`

`WRITE_LOG` *message*

This command places a string of text into the log file as soon as the command is encountered. The string *message* is required. It can be a dynamic string, created by combining static strings with symbolic expressions that are printed to the form of a string, as described earlier in the section *Dynamic Strings* (page 26), or shown by example for the previously described command, `STOP_RUN_NOW`.

This command has no effect on the running of the simulation. It is useful to add documentation when a Parsfile is loaded, either at the start of the run, or during the processing of an Event.

Discontinued Event Commands

Two commands were removed in 2019.1 (Table 7).

Table 7. Discontinued VS Event Commands.

Command	**Action**
`EVENT_SET_GT`	Define new Event (old version, not recommended).
`EVENT_SET_LT`	Define new Event (old version, not recommended).

Define New Variables

VS Math Models can be extended at runtime to include new variables for uses as parameters, imports, outputs, etc. Values for the variables can be calculated with algebraic or differential equations.

Table 8 lists the main VS Commands for adding variables to the VS Math Model.

The commands that begin with the prefix "`DEFINE_`" listed in Table 8 all act to define new variables. The new variables that can then be used in other VS Commands that extend the VS Math Model.

All new variables are listed in the Echo file, near the end (Figure 5). Because the Echo file may be used as an input for future simulations, the variables are shown in the form of valid VS Commands.

As with most programming languages, variables cannot be used in VS Commands unless they exist, either as built-in VS Math Model variables, or as variables created with a `DEFINE_` command. Therefore, when adding new variables or units with commands from Table 8, be sure the `DEFINE_` commands are provided to the VS Math Model before any VS Commands that make use of those variables, such as new equations or VS Events.

Table 8. VS Commands for defining new variables and setting their properties.

Command	**Action**
`DEFINE_IMPORT`	Define a new potential Import variable.
`DEFINE_OUTPUT`	Define a new output variable for export, plotting, and animation.
`DEFINE_PARAMETER`	Define a new parameter for the VS Math Model.
`DEFINE_VARIABLE`	Define a new state variable available to the VS Math Model.
`DELETE_VARIABLE`	Delete an Import, output, parameter, or variable added earlier.
`RESET_EXPORTS`	Disable all Export variables.
`RESET_IMPORTS`	Disable all Import variables.
`RESET_LIVE_ANI`	Disable all live animator broadcast variables.
`SET_DESCRIPTION`	Set the text description for an existing parameter or variable.
`SET_OUTPUT_COMPONENT`	Set the 32-character component name for a new output variable.
`SET_OUTPUT_GENERIC`	Set the 32-character generic name for a new output variable.
`SET_OUTPUT_LONG_NAME`	Set the 32-character long name for a new output variable.

Here is the syntax for the commands to define a new variable:

command variable [[=] *expression* [; [*units*] [; [*description*]]]]

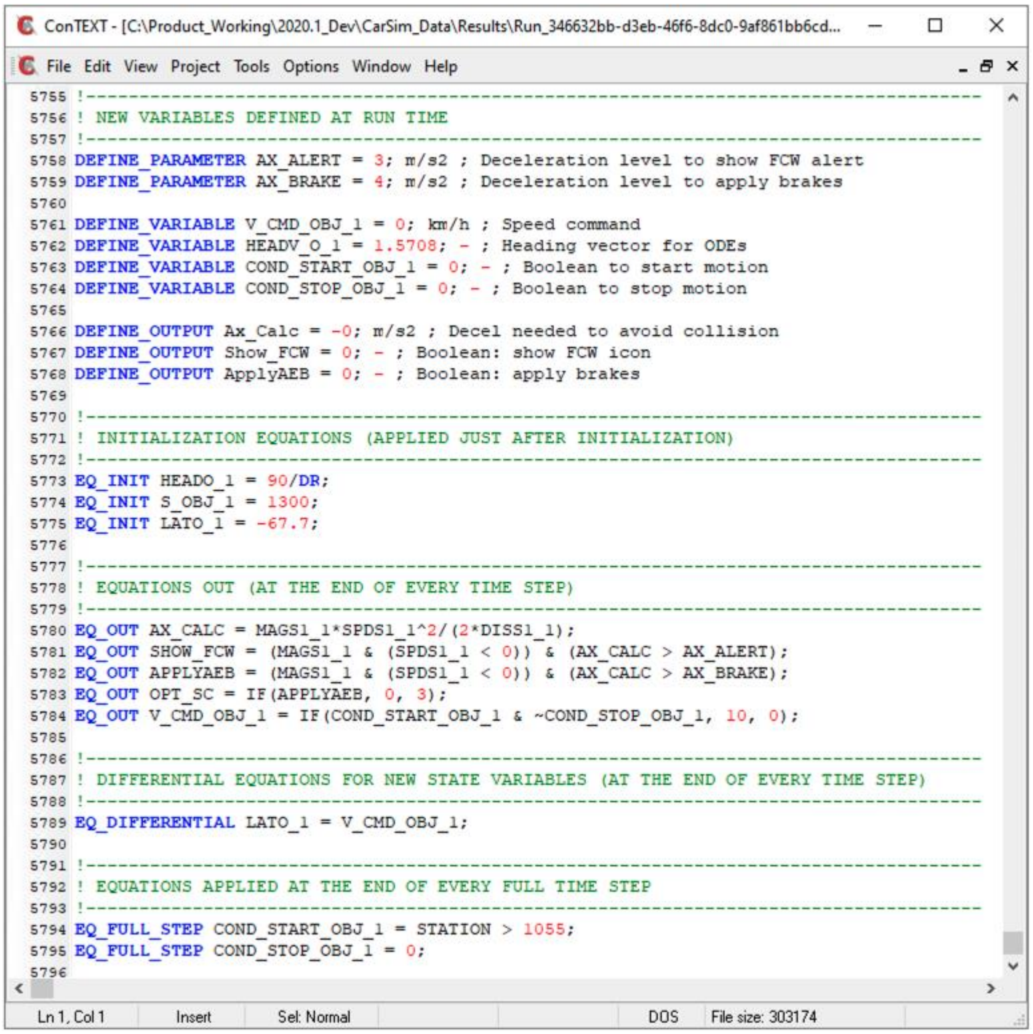
where *command* is a `DEFINE_` command listed in Table 8; *variable* is the name of the new variable; *expression* is either a number or an algebraic expression, *units* is an optional keyword used to specify the units associated with the parameter or variable specified by *keyword*, and *description* is a text description of the parameter or variable.

This syntax is like the syntax used for setting properties for existing variables, which had three parts. In this case, the command has four parts:

1. Create the new variable and define a keyword to identify it.
2. Optionally assign a value to the new variable (using an optional '=' sign).
3. Optionally set the units for the new variable.
4. Optionally assign a text description to the new variable that will be printed in Echo files and other documentation; in the case of an output variables, this is a longer name that appears in lists of outputs and may be used in plots made by VS Visualizer.

The first part puts the variable into the model and puts a keyword (*variable*) into the internal database such that the variable can be identified in other commands or statements. The three optional actions are the parts of a command that are used to modify the value, units, or description for an existing variable.

A new variable name can consist of alphanumeric characters and underscores ('_') but should begin with a letter. Upper- and lower-case letters are allowed, but like all VS Commands and keywords, case is not recognized, so the strings "`New_Variable`" and "`NEW_VARIABLE`" would both refer to the same variable.


ConTEXT - [C:\Product_Working\2020.1_Dev\CarSim_Data\Results\Run_346632bb-d3eb-46f6-8dc0-9af861bb6cd...
File Edit View Project Tools Options Window Help
Ln 1, Col 1
Insert
Sel: Normal
DOS
File size: 303174

```
5755 !----------------------------------------------------------------------------------
5756 ! NEW VARIABLES DEFINED AT RUN TIME
5757 !----------------------------------------------------------------------------------
5758 DEFINE_PARAMETER AX_ALERT = 3; m/s2 ; Deceleration level to show FCW alert
5759 DEFINE_PARAMETER AX_BRAKE = 4; m/s2 ; Deceleration level to apply brakes
5760
5761 DEFINE_VARIABLE V_CMD_OBJ_1 = 0; km/h ; Speed command
5762 DEFINE_VARIABLE HEADV_O_1 = 1.5708; - ; Heading vector for ODEs
5763 DEFINE_VARIABLE COND_START_OBJ_1 = 0; - ; Boolean to start motion
5764 DEFINE_VARIABLE COND_STOP_OBJ_1 = 0; - ; Boolean to stop motion
5765
5766 DEFINE_OUTPUT Ax_Calc = -0; m/s2 ; Decel needed to avoid collision
5767 DEFINE_OUTPUT Show_FCW = 0; - ; Boolean: show FCW icon
5768 DEFINE_OUTPUT ApplyAEB = 0; - ; Boolean: apply brakes
5769
5770 !----------------------------------------------------------------------------------
5771 ! INITIALIZATION EQUATIONS (APPLIED JUST AFTER INITIALIZATION)
5772 !----------------------------------------------------------------------------------
5773 EQ_INIT HEADO_1 = 90/DR;
5774 EQ_INIT S_OBJ_1 = 1300;
5775 EQ_INIT LATO_1 = -67.7;
5776
5777 !----------------------------------------------------------------------------------
5778 ! EQUATIONS OUT (AT THE END OF EVERY TIME STEP)
5779 !----------------------------------------------------------------------------------
5780 EQ_OUT AX_CALC = MAGS1_1*SPDS1_1^2/(2*DISS1_1);
5781 EQ_OUT SHOW_FCW = (MAGS1_1 & (SPDS1_1 < 0)) & (AX_CALC > AX_ALERT);
5782 EQ_OUT APPLYAEB = (MAGS1_1 & (SPDS1_1 < 0)) & (AX_CALC > AX_BRAKE);
5783 EQ_OUT OPT_SC = IF(APPLYAEB, 0, 3);
5784 EQ_OUT V_CMD_OBJ_1 = IF(COND_START_OBJ_1 & ~COND_STOP_OBJ_1, 10, 0);
5785
5786 !----------------------------------------------------------------------------------
5787 ! DIFFERENTIAL EQUATIONS FOR NEW STATE VARIABLES (AT THE END OF EVERY TIME STEP)
5788 !----------------------------------------------------------------------------------
5789 EQ_DIFFERENTIAL LATO_1 = V_CMD_OBJ_1;
5790
5791 !----------------------------------------------------------------------------------
5792 ! EQUATIONS APPLIED AT THE END OF EVERY FULL TIME STEP
5793 !----------------------------------------------------------------------------------
5794 EQ_FULL_STEP COND_START_OBJ_1 = STATION > 1055;
5795 EQ_FULL_STEP COND_STOP_OBJ_1 = 0;
5796
```

Figure 5. VS Commands listed at the end of an Echo file.

Note The optional arguments *units* and *description* cannot be specified unless the *expression* argument is also provided. If the value will be set later, you can still specify a "placeholder" value such as 0 to specify the units with a `DEFINE_` command.

For example, consider the new variable `AX_ALERT` that specifies a deceleration level that is used to show a forward collision warning (FCW) in VS Visualizer (line 5758, Figure 5).

Here are alternate valid commands for defining the parameter called `AX_ALERT`:

```
DEFINE_PARAMETER AX_ALERT  ! value not specified
DEFINE_PARAMETER AX_ALERT = 3 ! units not specified
DEFINE_PARAMETER AX_ALERT = 3; m/s2 ! also specify units
DEFINE_PARAMETER AX_ALERT = 3; m/s2 ; ...
   Deceleration level to show FCW alert ! continuation line with desc.
DEFINE_PARAMETER AX_ALERT 3;; level to show FCW alert ! skip units
```

In the simplest form, the line has only the command and variable name. The rest of the input line is optional. If provided, the next part is interpreted as an algebraic expression that continues to the end of the line or to the delimiter ';'. The third part, if provided, specifies the units associated with the parameter. This is useful mainly when scaling is intended to convert from user units (such as g's) to SI units used internally by the VS Math Model (e.g., m/s^2). The fourth part gives a description that will be listed in the Echo file to describe the parameter.

As indicated in the syntax definition, the '=' sign is optional. If units are specified, the ';' delimiter is required between *expression* and the units.

If text is specified, two semicolon delimiters are required, even if units are not specified.

Tip Always look at the Echo file for a simulation in which you have added new variables via the `DEFINE_` commands, to ensure they were interpreted as you intended. We recommend always using the '=' sign to provide a good match between the input and the appearance in the Echo file, and also to make the input a little easier to read.

DEFINE_IMPORT

`DEFINE_IMPORT` *variable* [[=] *expression* [; [*units*] [; [*description*]]]]

This defines a new Import variable that can be used to pass through variables from an external source (such as Simulink) to VS Visualizer for animation and/or plotting.

The keyword *variable* must begin with the prefix `IMP_`, e.g., `IMP_NEW_IMPORT`.

The new Import variable must be activated with the `IMPORT` command, the same as the original (native) Import variables described in the section *Calculate Import Variables with VS Commands* on page 57. However, the modes for combining the imported variable with a native variable are limited because there is no native calculated value for an import variables defined with this command. (If this were an internally defined Import, the description would not have the "[V]" indicator at the end of the description.)

Notes The Import variable can be activated using the `IMPORT` command. Alternatively, the name of the variable also serves as a command for backward compatibility with older datasets (the `IMPORT` command was introduced in 2012). All Import variables have names that begin with `IMP_` (e.g., `IMP_STEER_SW`).

If this command is used to define a new Import variable, but the variable is not activated, then the `DEFINE_IMPORT` command for the unused Import variable will not appear into the Echo or End files.

The `DEFINE_IMPORT` command exists in SuspensionSim but is of limited utility because SuspensionSim does not interface with external software such as Simulink.

If provided, *expression* is evaluated using current values of any variables in *expression*, and the resulting number is assigned to *variable*.

If semicolon delimiters are included, the command can also set the *units* and text *description* for the new variable, as described earlier.

A variable that is defined with this command cannot directly affect the native equations of the VS Math Model. However, the new Import variable can be used in other equations added at runtime with VS Commands. If no *expression* is provided, a default initial value of 0 is assigned.

If this command is used more than once with the same *variable*, and the system parameter `OPT_ERROR_DUP_DEF` is not zero, then an error is triggered, and the VS Math Model stops. If `OPT_ERROR_DUP_DEF` is zero and `DEFINE_IMPORT` is used multiple times with the same *variable*, then warnings are written into the log file when the repeat commands are processed.

DEFINE_OUTPUT

`DEFINE_OUTPUT` *variable* [[=] *expression* [; [*units*] [; [*long_name*]]]]

Output variables can be written to file, exported to external software such as Simulink, sent live to animators for real-time applications. They can also be used in equations used to calculate other variables.

The `DEFINE_OUTPUT` command installs a new output variable.

A single `DEFINE_OUTPUT` command installs four keywords for the variable:

1. The name provided as *variable* is installed as a keyword recognized by the VS Math Model. As with other keywords, it is not case sensitive. For example, if the *variable* name is `NewVar`, the keyword `NEWVAR` is installed in the keyword database of the VS Math Model.
2. The prefix `WRT_` is used to automatically install a keyword `WRT_`*variable*. This keyword can then be used to indicate that the variable should be written to the VS or ERD file for post-processing. For example, if the *variable* name is `NewVar`, a keyword `WRT_NEWVAR` is automatically installed.
3. The prefix `EXP_` is used to automatically install a keyword `EXP_`*variable*. This keyword can then be used to indicate that the variable should be put on the list of active Exports for Simulink or other external software. For example, if the *variable* name is `NewVar`, a keyword `EXP_NEWVAR` is automatically installed.
4. The prefix `ANI_` is used to automatically install a keyword `ANI_`*variable*. This keyword can then be used to indicate that the variable should be broadcast to active animators during interactive simulation, as with driving simulators. For example, if the *variable* name is `NewVar`, a keyword `ANI_NEWVAR` is automatically installed.

Note The keywords created with the `WRT_`, `EXP_`, and `ANI_` prefixes were required as commands in older versions of VS Math Models (prior to 2012). The commands `WRITE`, `EXPORT`, and `ANIMATE` can also be used to activate output variables using the keyword *variable* (without any prefix). E.g., the two commands `WRT_NEWVAR` and `WRITE NEWVAR` have the same effect if an output variable exists with the keyword `NEWVAR`.

In addition to the four keywords, the original mixed-case name is saved as the short name of the output variable, to be used in plot labels (e.g., `NewVar`).

Any equations that involve a new output variable that are made with the basic name (e.g., `NEWVAR`) will use the current value of the variable in internal SI units with no scale factors (meter, radian, m/s^2, etc.). Equations that use the names with prefixes will use the current value in user units (degrees, g's, etc.).

The command `DEFINE_OUTPUT` typically is associated with an equation applied every timestep to calculate a variable that will be written to the VS or ERD file for plotting and/or animation, or exported to other software. This is a more specific use than exists for variables added with the other three commands (`DEFINE_IMPORT`, `DEFINE_PARAMETER`, or `DEFINE_VARIABLE`). Hence, the `DEFINE_OUTPUT` command is processed a little differently than the other commands with respect to the optional *expression* argument. Depending on the form of *expression*, there are three possible outcomes:

1. If *expression* is a symbolic expression, an `EQ_OUT` command (page 54) is automatically applied using this expression. The result is that *expression* is evaluated and the result is assigned to the new output variable every timestep as the simulation runs. The Echo file will show two separate commands (`DEFINE_OUTPUT` and `EQ_OUT`).
2. If *expression* is a number, it is converted from user units to internal units and assigned to the new variable. If *expression* is a numerical expression (e.g., 1/18), the expression is evaluated, and the numerical value is converted to internal units and assigned to the new variable. No `EQ_OUT` command is applied.
3. If *expression* is not provided, a default value of 0 is assigned to the new variable. No `EQ_OUT` command is applied.

The Echo file shows the full status for all three cases. In the first case (an `EQ_OUT` equation is added), information about the new variable is written in the Echo file as two commands: `DEFINE_OUTPUT` (e.g., line 5766, Figure 5, page 40), and `EQ_OUT` (line 5780, Figure 5).

It is also possible to make the output variable an ODE state variable by using the VS Command `EQ_DIFFERENTIAL`. If this is done, the `DEFINE_OUTPUT` command should use a numerical value, which serves as the initial condition. The new output variable will be listed along with other state variables in the `End.Par` file and any machine-generated lists of state variables.

The optional text provided after the second semicolon (*long_name)* defines the long name for the output variable. If not included, the long name is automatically set to the short name *variable*. In this command, the text *long_name* is not dynamic, and is limited to 32 characters. If the text is too long, it is truncated to 32 characters.

If this command is used more than once with the same *variable*, and the system parameter `OPT_ERROR_DUP_DEF` is nonzero, then an error is triggered, and the VS Math Model stops. If

`OPT_ERROR_DUP_DEF` is zero and `DEFINE_OUTPUT` is used multiple times with the same *variable*, then warnings are written into the log file when the repeat commands are processed.

After a simulation has started, it is still possible to add equations and new variables. However, it is not possible to modify the lists of variables activated for import and export, writing to file, or streaming for live animation. If the `DEFINE_OUTPUT` command is applied after the simulation starts (during an Event), the variable is created as requested, but a warning is written to the Log file.

Variables defined with this command are not saved or restored when using the `SAVE_STATE` and `RESTORE_STATE` commands to back up in time (page 60), even if an ODE is added with the `EQ_DIFFERENTIAL` command. If you want the variable to be saved and restored, use the `DEFINE_VARIABLE` command instead to create the variable.

DEFINE_PARAMETER

`DEFINE_PARAMETER` *variable* [[=] *expression* [; [*units*] [; [*description*]]]]

This command installs a new variable that will probably remain constant during a simulation, as is the case with built-in parameters. The optional *expression* is interpreted the same as for an existing parameter: it is evaluated immediately to provide an initial value. If no *expression* is provided, a default value of 0 is assigned.

If semicolon delimiters are included, the command can also set the *units* and text *description* for the new variable, as described earlier.

If this command is used more than once with the same *variable*, and the system parameter `OPT_ERROR_DUP_DEF` is zero, and the optional *expression* argument is provided, then *expression* is evaluated and used to update the current value of *variable*. If the additional *units* argument is provided and *expression* is a number, then the command is processed with the units for *variable* being temporarily set to *units*.

`DEFINE_VARIABLE` can also be used to define a one- or two-dimensional array of variables. The *units* and text *description* work as with an ordinary variable and the optional *expression* assignment, if present, provides the initial value for all entries in the array.

`DEFINE_VARIABLE` *array*(*expression* [,*expression*]) [[=] *expression* [; [*units*] [; [*description*]]]]

Because arrays in VS Commands have index variables associated with them, a new integer index variable will be created for the array if it does not already exist. The name of the index variable is the same as the array name, but with an extra '`I`' prepended, unless there is an underscore in the array name, in which case, the text up to and including the first underscore are ignored. Thus, both array names '`GADGET`' and '`X_GADGET`' will have the same index variable '`IGADGET`'. If the array has two dimensions, then a second prepended '`I`' is added to the first index variable. In the cited example, the second index variable would be '`IIGADGET`'.

As with built-in parameters, variables defined with this command are not saved or restored when using the `SAVE_STATE` and `RESTORE_STATE` commands to back up in time (page 60). If you want a variable to be affected when a VS Math Model state is saved or restored, use the `DEFINE_VARIABLE` command instead to create the variable.

Tip If *expression* is a symbolic expression involving existing parameters or variables, the *expression* is evaluated immediately using the current values of any variables whose keywords appear in *expression*.

It is possible that the variables whose names appear on the right-hand side of the '=' sign will be given different values before the simulation starts. If it is your intention that the final values be used in *expression*, then an alternative is to set the value to 0 as a placeholder, and add an equation using the `EQ_PRE_INIT` or `EQ_INIT` command (see page 51) to calculate the value later.

DEFINE_VARIABLE

`DEFINE_VARIABLE` *variable* [[=] *expression* [; [*units*] [; [*description*]]]]

This installs a new variable that has an initial value and will not be needed as an output. It can be updated during the run with an algebraic equation (defined with the commands such as `EQ_IN` and `EQ_OUT`) or differential equation (defined with the `EQ_DIFFERENTIAL` command) as described later (page 54). The optional *expression* is interpreted the same as for an existing state variable or parameter: it is evaluated immediately to provide an initial value. If no *expression* is provided, a default value of 0 is assigned.

If semicolon delimiters are included, the command can also set the *units* and text *description* for the new variable, as done with the `DEFINE_PARAMETER` command.

If this command is used more than once with the same *variable*, and the system parameter `OPT_ERROR_DUP_DEF` is zero (duplicate `DEFINE_` statements do not trigger errors), and the optional *expression* argument is provided, then *expression* is evaluated and used to update the current value of *variable*. If the additional *units* argument is provided and *expression* is a number, then the command is processed with the units for *variable* being temporarily set to *units*.

As with built-in state variables, variables defined with this command are saved and restored when using the `SAVE_STATE` and `RESTORE_STATE` commands to back up in time (page 60). If you want a variable to be ignored when a VS Math Model state is saved or restored, use the `DEFINE_PARAMETER` or `DEFINE_OUTPUT` command instead to create the variable.

Note In older versions of the software, output variable names were limited to 8 characters. Some users would use the `DEFINE_VARIABLE` command to create a descriptive name for use in equations and create an output with the `DEFINE_OUTPUT` command with an equation to copy the variable value to the output value.

This is not necessary and is not recommended. If a variable will be of interest as an output, and you are not doing Save/Restore commands, simply use the `DEFINE_OUTPUT` command to create it.

DELETE_VARIABLE

`DELETE_VARIABLE` *variable*

This command deletes a variable added with one of the above `DEFINE_` commands such that it does not appear in the End Parsfile that is written at the end of the run. This command is useful for certain advanced applications where a simulation is continued over two or more runs.

If the End file from one run is used to continue the simulation with another run, then any VS Commands that defined variables or parameters in the first run are normally echoed in the End file. In cases like this, variables may have been created for the first run but may no longer be useful in the continued run. When that file is read as input for a subsequent run, the same variables will be defined again, even if the equations that assign values to them have been deleted. If they are not used in the subsequent runs, they can make future Echo files confusing to read.

If you define variables for a run but do not want the VS Commands echoed, then use the `DELETE_VARIABLE` command to remove them. Be aware, however, that if there are any equations that use the variables, then future run attempts based on the End file might fail due to unrecognized variable names.

RESET_ Commands

```
RESET_EXPORTS
RESET_IMPORT
RESET_LIVE_ANI
```

The `RESET_` commands have no arguments and perform the functions as described in Table 8 (page 39). These are not used in routine applications when running under Windows OS, because the VS Solver library is always unloaded at the end of each simulation. However, in some RT systems or setups under the control of external software, they can be used to reset the arrays of Import, Export, and Live Animation variables.

SET_DESCRIPTION

`SET_DESCRIPTION` *variable description*

This command processes the text *description* as a dynamic string and assigns it to the description of the parameter or variables identified with the keyword *variable*.

If *variable* identifies an output variable, then *description* is not a dynamic string; it is simply the remainder of the line of text with leading spaces removed. In this case, it functions the same as the `SET_OUTPUT_LONG_NAME` command described in the next subsection. When used to set the long name of an output variable, the text should be limited to 32 characters (it will be truncated if longer than this limit).

As noted in the VS Math Models manual, a typical line of input from a Parsfile assigns a new value to a variable in the solver with the form:

variable [=] *expression* [; [*units*] [; [*description*]]]

In this form, *units* and *description* are optional, but *expression* (evaluated to give a numerical value) is mandatory. The `SET_DESCRIPTION` command provides a convenient means for setting the *description* for a *variable* without touching the value or units.

For most VS Commands, an error is generated if *variable* does not exist. The `SET_DESCRIPTION` command differs in this respect. If *variable* does not exist, then the command does nothing, as is the case with the assignment form shown above (*variable* = …).

SET_OUTPUT Commands

`SET_OUTPUT_COMPONENT` *variable name*
`SET_OUTPUT_GENERIC` *variable name*
`SET_OUTPUT_LONG_NAME` *variable name*

In these three commands, *variable* is the name of the output variable (not case sensitive) and *name* is the rest of the line (up to 32 characters). Starting with the first non-blank character after *variable*, the rest of the line is set as the specified *name*. It can include blanks.

If a new output variable (added with the `DEFINE_OUTPUT` command) will be plotted, then the `SET_OUTPUT_LONG_NAME` command can be used to give it a more descriptive name. Alternatively, the long name can be included in the `DEFINE_OUTPUT` command using a second semicolon delimiter.

The commands `SET_OUTPUT_GENERIC` and `SET_OUTPUT_COMPONENT` can be used to enable the plotter to label overlay plots nicely.

On the other hand, plot labels can be set from the VS Browser using the **Plot: Setup** screen for specific plots. Often, it is more convenient to specify labels in the **Plot: Setup** screen rather than using the VS Commands to add properties.

Define New Functions

New functions may be defined to provide a sequence of reusable equations. User-defined functions, along with the IF command, allow for the conditional execution of groups of commands. This allows for simpler, cleaner code when a condition changes during a simulation which requires the updating of several variables.

Using User-Defined Functions

User-defined functions can be used in algebraic expressions, with the syntax:

function_name ([*arg1* [, *arg2*[, *arg3*[, *arg4*… [, *arg8*]]] …])

where *function_name* is the name of the function and *arg1-arg8* are the (up to) eight expressions that can be used as inputs to the function.

User defined functions may be used as functions that are included in formulas on the right-hand side of the '=' sign in an equation. All user-defined functions return a real value, either explicitly, using the `RETURN` command, or implicitly (returning 0.0) if the defined function has no `RETURN`.

They may also be used as commands, in which case the returned value is not assigned to any variable.

For example, both of these command lines are valid:

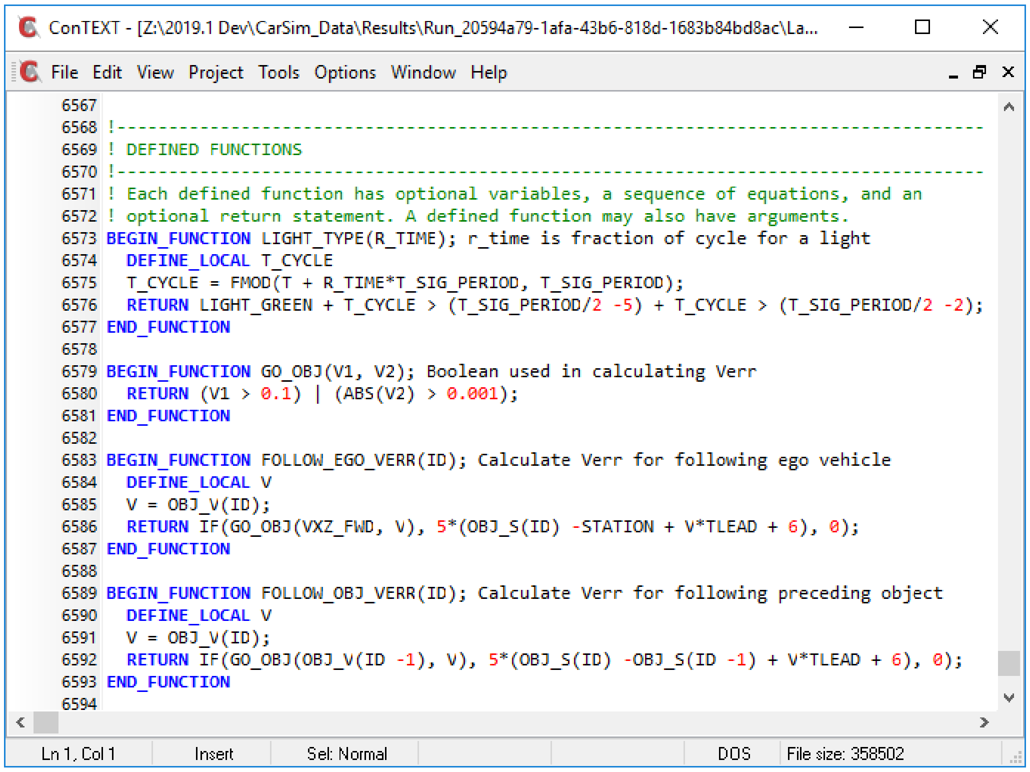
```
EQ_IN VERR_O_CURRENT = FOLLOW_OBJ_VERR(ID)
EQ_IN FOLLOW_OBJ_VERR(ID)
```

Defining a New Function

New functions are made using equations involving variables and functions in the VS Math Model, along with up to four new commands (Table 9). Figure 6 shows how user-defined functions appear in an Echo file.

Table 9. Commands for defining a new function.

Command	Purpose	Comment
`BEGIN_FUNCTION`	Start the definition of a new function and specify the name and names of arguments	Required, first line
`DEFINE_LOCAL`	Define one or more variables that are local to the function	Optional for local variables
`RETURN`	Calculate a value returned by the function	Optional for specifying value returned by function
`END_FUNCTION`	Indicate end of function	Required, last line


ConTEXT - [Z:\2019.1 Dev\CarSim_Data\Results\Run_20594a79-1afa-43b6-818d-1683b84bd8ac\La...
File Edit View Project Tools Options Window Help
Ln 1, Col 1
Insert
Sel: Normal
DOS
File size: 358502

```

!----------------------------------------------------------------------------------
! DEFINED FUNCTIONS
!----------------------------------------------------------------------------------
! Each defined function has optional variables, a sequence of equations, and an
! optional return statement. A defined function may also have arguments.
BEGIN_FUNCTION LIGHT_TYPE(R_TIME); r_time is fraction of cycle for a light
  DEFINE_LOCAL T_CYCLE
  T_CYCLE = FMOD(T + R_TIME*T_SIG_PERIOD, T_SIG_PERIOD);
  RETURN LIGHT_GREEN + T_CYCLE > (T_SIG_PERIOD/2 -5) + T_CYCLE > (T_SIG_PERIOD/2 -2);
END_FUNCTION

BEGIN_FUNCTION GO_OBJ(V1, V2); Boolean used in calculating Verr
  RETURN (V1 > 0.1) | (ABS(V2) > 0.001);
END_FUNCTION

BEGIN_FUNCTION FOLLOW_EGO_VERR(ID); Calculate Verr for following ego vehicle
  DEFINE_LOCAL V
  V = OBJ_V(ID);
  RETURN IF(GO_OBJ(VXZ_FWD, V), 5*(OBJ_S(ID) -STATION + V*TLEAD + 6), 0);
END_FUNCTION

BEGIN_FUNCTION FOLLOW_OBJ_VERR(ID); Calculate Verr for following preceding object
  DEFINE_LOCAL V
  V = OBJ_V(ID);
  RETURN IF(GO_OBJ(OBJ_V(ID -1), V), 5*(OBJ_S(ID) -OBJ_S(ID -1) + V*TLEAD + 6), 0);
END_FUNCTION

```

Figure 6. Four example functions as shown in an Echo file.

The definition of a function goes as follows:

> `BEGIN_FUNCTION` *name*(*arg1*, *arg2*, …) ; *description*
>
> `DEFINE_LOCAL` *var*
>
> <*equations*>
>
> `RETURN` *formula*
>
> `END_FUNCTION`

Any number of equations may be included after the optional `DEFINE_LOCAL` statements and before the optional `RETURN` statement. The function definition ends with the `END_FUNCTION` statement.

BEGIN_FUNCTION
END_FUNCTION

> `BEGIN_FUNCTION` *name* ([*arg1* [, *arg2* [, *arg3* [, *arg4*… [, *arg8*] …]]]]]) [; *description*]
>
> [*content*]
>
> `END_FUNCTION`

The VS Commands `BEGIN_FUNCTION` and `END_FUNCTION` define the body of the new function. The function has a required name (e.g., `LIGHT_TYPE` in line 6573, Figure 6). It has required opening and closing parentheses, which may enclose up to eight arguments that are separated by commas. After the closing parenthesis, an optional colon may be added, followed by a text description of the function. That description is listed in the Echo file along with the rest of the function.

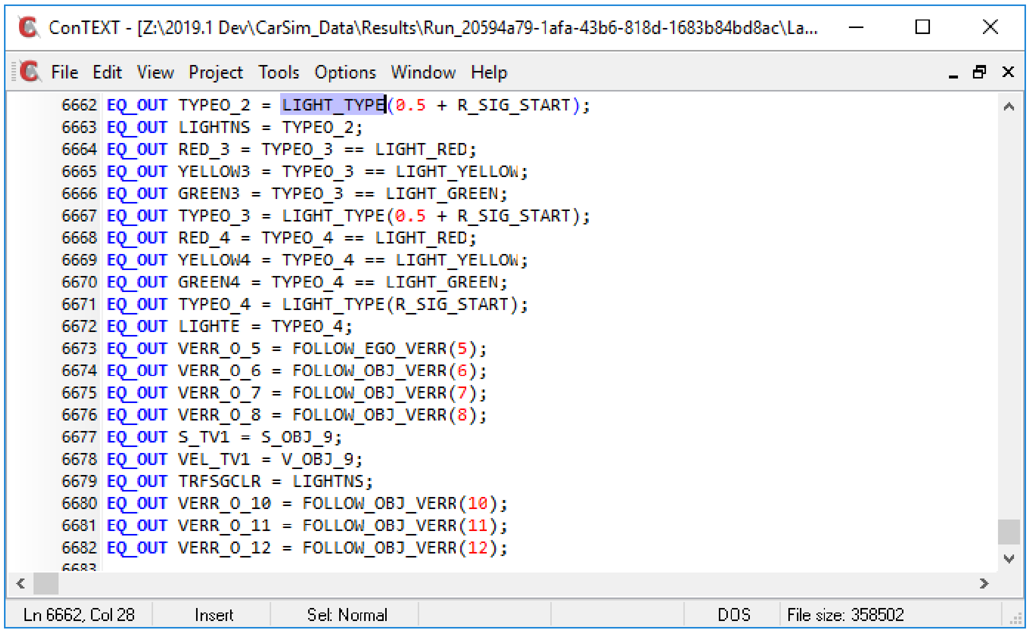
A subset of equations is allowed within the body of the Defined Function, and the definition is finally closed with the VS Command `END_FUNCTION`. After definition, the newly defined function can be used like any other function in an equation or other statement. For example, Figure 7 shows a later portion of the same Echo file in which four functions were defined (Figure 6).

The function `LIGHT_TYPE` is used three times, the function `FOLLOW_EGO_VERR` is used once, and the function `FOLLOW_OBJ_VERR` is used six times. The function `GO_OBJ` was used twice in Figure 6, within the functions `FOLLOW_EGO_VERR` and `FOLLOW_OBJ_VERR`.

There is no limit to the number of equations that can be placed in the function block. The equations can be used to update outside (global) variables or local variables, defined with the command `DEFINE_LOCAL`.

An equation within a function block can make use of another user-defined function. Currently, a user-defined function cannot be called recursively. If this is attempted, even indirectly, then an error will occur and the simulation will not proceed.

Function blocks are intended to provide greater utility of VS Commands to the user. Although still somewhat limited in their scope, they nonetheless provide a powerful programming capability to the VS Command user.


ConTEXT - [Z:\2019.1 Dev\CarSim_Data\Results\Run_20594a79-1afa-43b6-818d-1683b84bd8ac\La...
File Edit View Project Tools Options Window Help
Ln 6662, Col 28
Insert
Sel: Normal
DOS
File size: 358502

```
EQ_OUT TYPEO_2 = LIGHT_TYPE(0.5 + R_SIG_START);
EQ_OUT LIGHTNS = TYPEO_2;
EQ_OUT RED_3 = TYPEO_3 == LIGHT_RED;
EQ_OUT YELLOW3 = TYPEO_3 == LIGHT_YELLOW;
EQ_OUT GREEN3 = TYPEO_3 == LIGHT_GREEN;
EQ_OUT TYPEO_3 = LIGHT_TYPE(0.5 + R_SIG_START);
EQ_OUT RED_4 = TYPEO_4 == LIGHT_RED;
EQ_OUT YELLOW4 = TYPEO_4 == LIGHT_YELLOW;
EQ_OUT GREEN4 = TYPEO_4 == LIGHT_GREEN;
EQ_OUT TYPEO_4 = LIGHT_TYPE(R_SIG_START);
EQ_OUT LIGHTE = TYPEO_4;
EQ_OUT VERR_O_5 = FOLLOW_EGO_VERR(5);
EQ_OUT VERR_O_6 = FOLLOW_OBJ_VERR(6);
EQ_OUT VERR_O_7 = FOLLOW_OBJ_VERR(7);
EQ_OUT VERR_O_8 = FOLLOW_OBJ_VERR(8);
EQ_OUT S_TV1 = S_OBJ_9;
EQ_OUT VEL_TV1 = V_OBJ_9;
EQ_OUT TRFSGCLR = LIGHTNS;
EQ_OUT VERR_O_10 = FOLLOW_OBJ_VERR(10);
EQ_OUT VERR_O_11 = FOLLOW_OBJ_VERR(11);
EQ_OUT VERR_O_12 = FOLLOW_OBJ_VERR(12);
```

Figure 7. Example usage of user-defined functions in EQ_OUT equations.

DEFINE_LOCAL

`DEFINE_LOCAL` *var1, var2, …*

`DEFINE_LOCAL` is used to provide working variables which are only needed within a function block. Multiple `DEFINE_LOCAL` statements may be used within a function block, with each defining one or more local variables.

All variables and parameters defined outside of the function block (but not within another function block) are accessible to the function and can be used in its equations. If a locally-defined variable has the same name as an outside variable, then the local variable will take precedence. This also follows with argument names; they take precedence in the case of an outside variable or parameter with the same name. A local variable cannot have the same name as an argument.

RETURN

`RETURN` *formula*

The return value of the function block is created by the Command `RETURN`. If no `RETURN` command is used in the function block, then the function will return `0.0`. `RETURN` will take any valid formula (or variable or number) and evaluate it. The use of a receiving variable to accept the return is optional when using a function block in an equation.

Add Equations

VS Math Models can be extended at runtime with new equations for updating Import variables, output variables, state variables defined by differential equations, and other variables (parameters and intermediate variables). The commands for adding equations are listed in Table 10, in the order in which they are applied in the simulation. In the table, “EOM” indicates the full equations of motion for the built-in model.

Table 10. Commands for defining new equations at runtime.

Command	**When Applied**
`EQ_PRE_INIT`	Once, after reading input Parsfiles but before the built-in initialization.
`EQ_INIT`	Once, after the built-in initialization, before applying the EOM, before writing the Echo file, and before receiving any Import variables.
`EQ_INIT2`	Once, after writing the Echo file and after receiving Import variables.
`EQ_IN`	Every timestep, just before any EOM are applied.
`EQ_DYN`	Every timestep, after kinematical EOM but before dynamical EOM.
`EQ_OUT`	Every timestep, just after all EOM were applied.
`EQ_DIFFERENTIAL`	Every timestep, just after the `EQ_OUT` equations.
`EQ_SAVE`	Every timestep, just after the `EQ_DIFFERENTIAL` equations.
`EQ_FULL_STEP`	At the end of every full timestep, just before Events are processed.
`EQ_END`	Once, just before writing the End file.

In every case the syntax is:

command variable [=] *expression* [;]

where *variable* is a variable and *expression* is a VS Expression involving other symbols in the VS Math Model (including new variables added with VS Commands such as `DEFINE_VARIABLE`). The delimiters ‘=’ and ‘;’ are optional but recommended (they are always shown in Echo files).

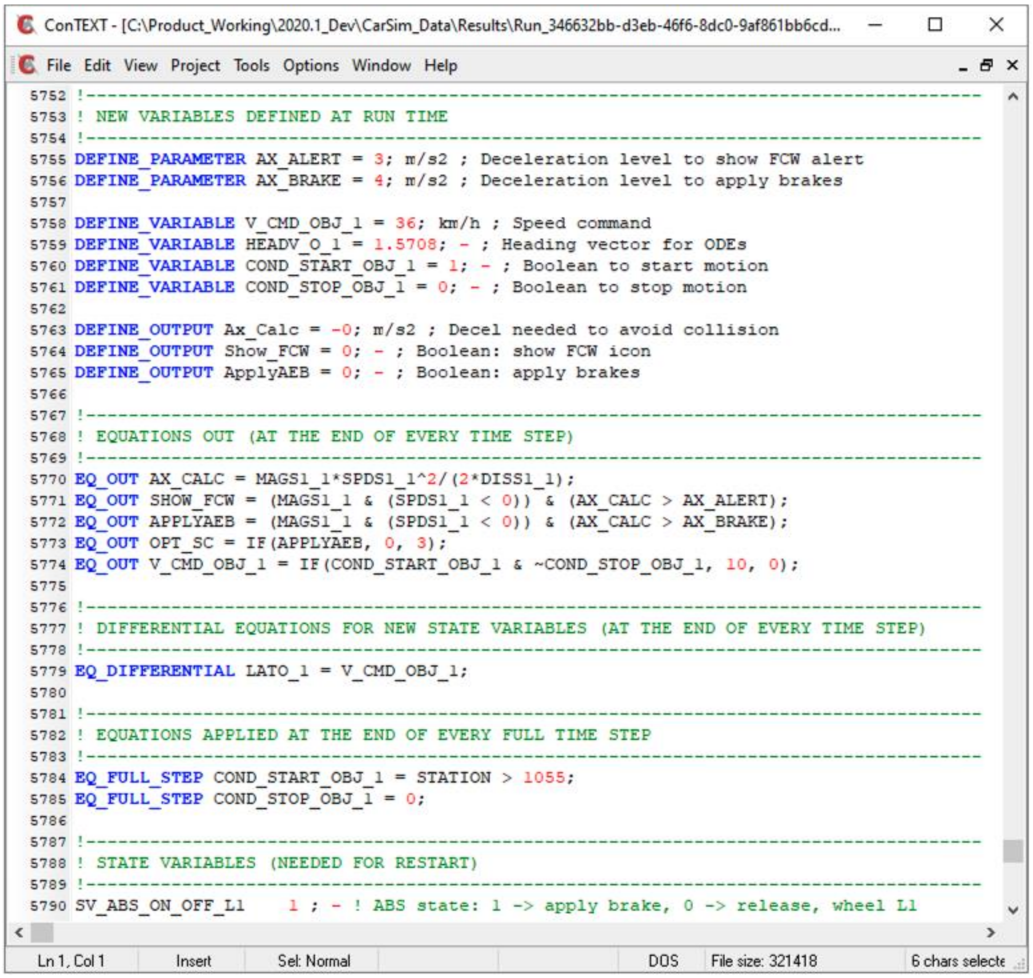
Read-only variables cannot be specified as *variable* (doing so will generate an error and stop the simulation) but may be used in *expression.*

The commands all assume *expression* is expressed in internal units (e.g., an angular variable would be expressed in radians, rather than degrees), even if *expression* is numerical.

All the user-defined variables and equations that exist at the time the simulation starts are listed in the Echo file as a sequence of VS Commands, as shown earlier (Figure 5, page 40). They appear after commands for adding new variables, to ensure that new variables that might appear in the equations have already been defined. Also, the equations in each group (`EQ_INIT`, `EQ_IN`, etc.) appear in the order in which the commands were encountered from input Parsfiles, preserving the intended sequence of equations in each group.

A similar list appears in the Echo file written at the end of the run (Figure 8). In this case, the `EQ_PRE_INIT`, `EQ_INIT`, and `EQ_INIT2` equations are not listed to avoid conflicts if the End Echo file is used to continue the run with a new simulation.

In Table 10, there is a distinction between “every timestep” and “every full timestep.” In some cases, the simulation is run with the full timestep as specified by the parameter `TSTEP` handled in two half steps, as described in detail later in Chapter 6 (page 88). “Every timestep” means the equation is applied at both half steps (typically at intervals of `TSTEP/2` s); “full step” means the equation is only applied at the end of the second half step (at intervals of `TSTEP` s).


ConTEXT - [C:\Product_Working\2020.1_Dev\CarSim_Data\Results\Run_346632bb-d3eb-46f6-8dc0-9af861bb6cd...
File Edit View Project Tools Options Window Help

```
!------------------------------------------------------------------------------
! NEW VARIABLES DEFINED AT RUN TIME
!------------------------------------------------------------------------------
DEFINE_PARAMETER AX_ALERT = 3; m/s2 ; Deceleration level to show FCW alert
DEFINE_PARAMETER AX_BRAKE = 4; m/s2 ; Deceleration level to apply brakes

DEFINE_VARIABLE V_CMD_OBJ_1 = 36; km/h ; Speed command
DEFINE_VARIABLE HEADV_O_1 = 1.5708; - ; Heading vector for ODEs
DEFINE_VARIABLE COND_START_OBJ_1 = 1; - ; Boolean to start motion
DEFINE_VARIABLE COND_STOP_OBJ_1 = 0; - ; Boolean to stop motion

DEFINE_OUTPUT Ax_Calc = -0; m/s2 ; Decel needed to avoid collision
DEFINE_OUTPUT Show_FCW = 0; - ; Boolean: show FCW icon
DEFINE_OUTPUT ApplyAEB = 0; - ; Boolean: apply brakes

!------------------------------------------------------------------------------
! EQUATIONS OUT (AT THE END OF EVERY TIME STEP)
!------------------------------------------------------------------------------
EQ_OUT AX_CALC = MAGS1_1*SPDS1_1^2/(2*DISS1_1);
EQ_OUT SHOW_FCW = (MAGS1_1 & (SPDS1_1 < 0)) & (AX_CALC > AX_ALERT);
EQ_OUT APPLYAEB = (MAGS1_1 & (SPDS1_1 < 0)) & (AX_CALC > AX_BRAKE);
EQ_OUT OPT_SC = IF(APPLYAEB, 0, 3);
EQ_OUT V_CMD_OBJ_1 = IF(COND_START_OBJ_1 & ~COND_STOP_OBJ_1, 10, 0);

!------------------------------------------------------------------------------
! DIFFERENTIAL EQUATIONS FOR NEW STATE VARIABLES (AT THE END OF EVERY TIME STEP)
!------------------------------------------------------------------------------
EQ_DIFFERENTIAL LATO_1 = V_CMD_OBJ_1;

!------------------------------------------------------------------------------
! EQUATIONS APPLIED AT THE END OF EVERY FULL TIME STEP
!------------------------------------------------------------------------------
EQ_FULL_STEP COND_START_OBJ_1 = STATION > 1055;
EQ_FULL_STEP COND_STOP_OBJ_1 = 0;

!------------------------------------------------------------------------------
! STATE VARIABLES (NEEDED FOR RESTART)
!------------------------------------------------------------------------------
SV_ABS_ON_OFF_L1      1 ; - ! ABS state: 1 -> apply brake, 0 -> release, wheel L1
```

Figure 8. Echo file written at the end of a run.

The time increment is available as a system parameter `T_DT`. Depending on the numerical integration method, it is either set to `TSTEP` or `TSTEP/2`.

All equations applied with these commands are applied with no built-in unit conversions.

If *variable* is an integer parameter, the floating-point value from the formula is rounded to the nearest integer value.

An error message is generated if *variable* is not recognized or if it is a locked parameter such as `IPRINT`, `T`, `TSTEP`, or `T_DT`. (Protected parameters are identified in the Echo file).

If a variable is added with `DEFINE_PARAMETER` or `DEFINE_VARIABLE` and will be constant during a run, it is not necessary to update it every timestep with an equation added by an `EQ_` command; the initial value will remain in effect throughout the run.

Tip In general, we recommend using `DEFINE_PARAMETER` to define new variables that will not have associated equations, `DEFINE_OUTPUT` for variables that can be plotted or used to support animation, and `DEFINE_VARIABLE` for variables that are updated every timestep with equations added with `EQ_IN`, `EQ_OUT`, or `EQ_DIFFERENTIAL` but will not be used in plots or animations.

The exact sequence in which equations are applied can be important because the results depend on whether a variable appearing in an equation has been updated for the current time, or whether the variable still has a value from an earlier time. Please review Chapter 6 *The Simulation Process* (page 88) for details of the full sequence of operations.

EQ_PRE_INIT

`EQ_PRE_INIT` *variable* [=] *expression* [;]

Add an equation that is applied once at the start of the run. Equations in this group are applied after reading input Parsfiles, but before performing the built-in initialization.

It is unusual to put equations here; this command is intended for advanced users who need to ensure that these equations are applied before the main initialization calculations are made.

EQ_INIT

`EQ_INIT` *variable* [=] *expression* [;]

Add an equation that is applied once at the start of the run. Equations in this group are applied after the built-in initialization, but before calculating initial values of the output variables.

Equations added with this command are typically used to calculate values for new variables (added with VS Commands) that are based on other parameters and variables as they exist after all Parsfiles have been read, after the built-in initializations have been performed, but before the EOM are applied.

EQ_INIT2

`EQ_INIT2` *variable* [=] *expression* [;]

Add an equation that is applied as late as possible before the run starts.

Unless the parameter `OPT_SKIP_INIT_DYN` has been given a nonzero value, the EOM were applied in the initialization, along with all `EQ_IN`, `EQ_DYN`, and `EQ_OUT` equations. At the first timestep, the `EQ_INIT2` equations are applied after possibly receiving imports from external software such as Simulink.

Equations added with this command are typically used to calculate values for new variables (added with VS Commands) that are based on output variables at the start of the run, or to make use of variables imported from external software that were not available during at the time the `EQ_INIT` equations were applied.

EQ_IN

`EQ_IN` *variable* [=] *expression* [;]

Add an equation that is applied every timestep just before the EOM are applied.

Equations added with this command are typically used to apply controls that will affect the EOM as they are applied at every timestep. Derivatives and output variables might be affected by these equations.

It sometimes happens that you want to add an equation with `EQ_IN` to define an equation for a control at the start of the timestep that involves output variables that are not updated until the end of the timestep. In these cases, add the equations with `EQ_IN`, and be aware that any output variables used in the equation will have values left over from the previous timestep. However, if the output variables are kinematical (with units of m, deg, m/s, deg/s), consider using the `EQ_DYN` command instead; it might eliminate the lag.

EQ_DYN

`EQ_DYN` *variable* [=] *expression* [;]

Add an equation that is applied every timestep just after the built-in kinematical equations of motion are applied, but before the built-in dynamical equations are applied.

Equations added with this command are typically used to apply forces or moments that will affect the VS Math Model equations as they are calculated at every timestep, and which are calculated using kinematical variables such as deflections and velocities. Derivatives and output variables might be affected by these equations.

SuspensionSim does not apply dynamical equations; `EQ_DYN` can be specified but will have the same effect as `EQ_OUT`.

EQ_OUT

`EQ_OUT` *variable* [=] *expression* [;]

Add an equation that is applied every timestep, just after all of the built-in EOM are applied.

Equations added with this command are typically used to calculate values for output variables whose values depend on other output variables that are built-in to the VS Math Model.

EQ_DIFFERENTIAL

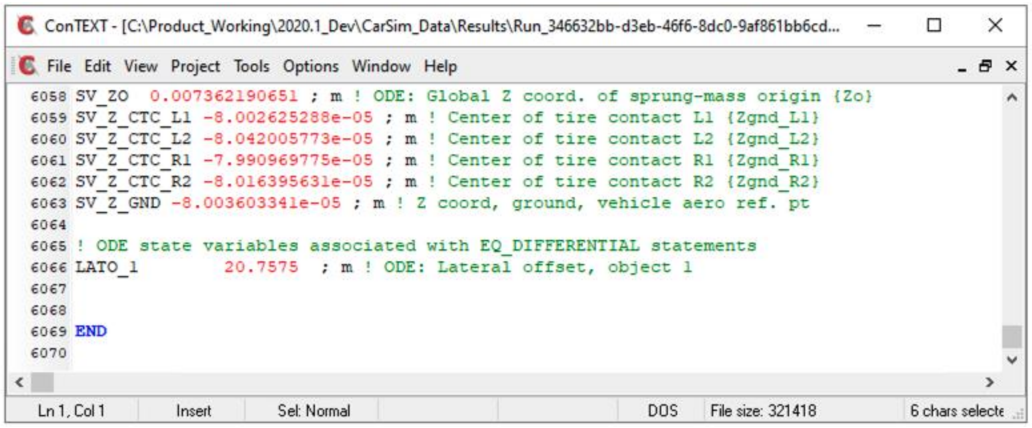
`EQ_DIFFERENTIAL` *variable* [=] *expression* [;]

Add an equation that is applied every timestep when the model variables are calculated. Equations in this group are applied just after the equations added with `EQ_OUT` are applied.

Unlike all of the other `EQ_` commands, *expression* is not used to calculate a value for *variable* directly. Instead, *expression* is used to calculate the derivative of *variable*. The VS Math Model then integrates this derivative at each timestep to calculate new values for *variable.*

As shown Figure 5 and Figure 8, the `EQ_DIFFERENTIAL` commands are written in the Echo files written at the start and end of each run. Besides providing the equation to calculate the derivative, this command changes the status of *variable* from a normal variable to an ODE state variable. The End Echo file ends with a list of all state variables and their values, in support of using the file to continue the run (Figure 9). In this list, the descriptions begin with the text “ODE” for ODE state variables. For example, `SV_ZO` (line 6058) is an ODE state variable. (State variables without

"ODE" in the description are needed to define the state of model but are not calculated by numerically integrating their derivatives.)


ConTEXT - [C:\Product_Working\2020.1_Dev\CarSim_Data\Results\Run_346632bb-d3eb-46f6-8dc0-9af861bb6cd...
File Edit View Project Tools Options Window Help

```
6058 SV_ZO  0.007362190651 ; m ! ODE: Global Z coord. of sprung-mass origin {Zo}
6059 SV_Z_CTC_L1 -8.002625288e-05 ; m ! Center of tire contact L1 {Zgnd_L1}
6060 SV_Z_CTC_L2 -8.042005773e-05 ; m ! Center of tire contact L2 {Zgnd_L2}
6061 SV_Z_CTC_R1 -7.990969775e-05 ; m ! Center of tire contact R1 {Zgnd_R1}
6062 SV_Z_CTC_R2 -8.016395631e-05 ; m ! Center of tire contact R2 {Zgnd_R2}
6063 SV_Z_GND -8.003603341e-05 ; m ! Z coord, ground, vehicle aero ref. pt
6064
6065 ! ODE state variables associated with EQ_DIFFERENTIAL statements
6066 LATO_1          20.7575   ; m ! ODE: Lateral offset, object 1
6067
6068
6069 END
6070
```

Figure 9. End of an Echo file listing state variables.

If the `EQ_DIFFERENTIAL` command was used to add variables to the set of ODE state variables, the Echo file includes a section listing those ODE state variables (e.g., line 6066).

Once the simulation has started, the number of ODEs is fixed. An attempt to use the `EQ_DIFFERENTIAL` command after the simulation has started (via an Event) will cause an error message to be generated, stopping the run.

EQ_SAVE

`EQ_SAVE` *variable* [=] *expression* [;]

Add an equation that is applied every timestep just after the equations added with `EQ_DIFFERENTIAL` are applied.

Equations in this group are applied after all the other groups at every timestep except the `EQ_FULL_STEP` group. They are typically used to save a value for use at the next timestep, e.g., to calculate a derivative using finite difference.

Equations in this group are not applied at the last timestep of the simulation. This way, conditions recorded in the End Parsfile written at the end of the simulation have values based on the next-to-last timestep, as is typically intended.

EQ_FULL_STEP

`EQ_FULL_STEP` *variable* [=] *expression* [;]

Add an equation that is applied at the full timestep just after the equations added with `EQ_SAVE` are applied.

This option is provided because VS Math Models support numerical integration methods that calculate ODE state variables at the half-step in order to calculate derivative at the half-step, and then obtain more accurate values of the ODE state variables at the full step. With these methods, values of ODE state variables calculated at the half step are discarded, as are all variables that are calculated from the ODE state variables.

If an equation is added that involves a comparison of an ODE with some reference (e.g., reaching a limit, or comparing with the value from the previous timestep to see if a change of sign occurred), it is better to add the equation here.

In some advanced applications, VS Commands are used to reset ODE state variables, such as moving the X and Y coordinates of the vehicle to a new location. Any changes made to ODE state variables should be made either with an Event or with an `EQ_FULL_STEP` equation.

Equations in this group are applied every full timestep, including the last one in the simulation.

`EQ_END`

`EQ_END` *variable* [=] *expression* [;]

Add an equation that is applied at the end of the simulation.

Delete Equations

Most of the equations added with the commands listed in Table 10 (page 51) can be removed during a run (while processing an Event) with the `DELETE_EQS_`*x* commands listed in Table 11.

Table 11. VS Commands for removing equations from the VS Math Model.

Command	Action
`DELETE_EQS_DYN`	Delete equations added with `EQ_DYN` for one variable.
`DELETE_EQS_END`	Delete equations added with `EQ_END` for one variable.
`DELETE_EQS_FULL_STEP`	Delete equations added with `EQ_FULL_STEP` for one variable.
`DELETE_EQS_IN`	Delete equations added with `EQ_IN` for one variable.
`DELETE_EQS_INIT`	Delete equations added with `EQ_INIT` for one variable.
`DELETE_EQS_INIT2`	Delete equations added with `EQ_INIT2` for one variable.
`DELETE_EQS_OUT`	Delete equations added with `EQ_OUT` for one variable.
`DELETE_EQS_PRE_INIT`	Delete equations added with `EQ_PRE_INIT` for one variable.
`DELETE_EQS_SAVE`	Delete equations added with `EQ_SAVE` for one variable.
`RESET_EQS_ALL`	Delete all equations added with VS Commands.
`RESET_EQS_END`	Delete all equations added with `EQ_END`.
`RESET_EQS_FULL_STEP`	Delete all equations added with `EQ_FULL_STEP`.
`RESET_EQS_IN`	Delete all equations added with `EQ_IN`.
`RESET_EQS_INIT`	Delete all equations added with `EQ_INIT`.
`RESET_EQS_INIT2`	Delete all equations added with `EQ_INIT2`.
`RESET_EQS_OUT`	Delete all equations added with `EQ_OUT`.
`RESET_EQS_PRE_INIT`	Delete all equations added with `EQ_PRE_INIT`.
`RESET_EQS_SAVE`	Delete all equations added with `EQ_SAVE`.

Delete Equations for One Variable

The syntax is:

delete_eqs_command variable

where *variable* is the variable whose value is being updated by the equation. This can be useful when assembling complicated test sequences with Events, in which different equations might be used at different stages. For example, suppose you were defining steering wheel angle with a command such as

```
EQ_IN IMP_STEER_SW = SIN(T) ;
```

Later, you might want to apply a Configurable Function with an equation such as

```
EQ_IN IMP_STEER_SW = GENERIC(0, T, 1) ;
```

Adding the second equation might be OK, but the first equation would still be present. For very long runs, the number of equations would grow and might lead to complications. To clear the first equation before defining a replacement, you could put two commands in:

```
DELETE_EQS_IN IMP_STEER_SW
EQ_IN IMP_STEER_SW = GENERIC(0, T, 1) ;
```

There is no `DELETE_EQS` command to remove differential equations added with the `EQ_DIFFENTIAL` command. If you want to change differential equations, you can assign the derivative to a variable that you have introduced with the `DEFINE_VARIABLE` command, and then change the algebraic equation used to calculate that variable.

A related command is `DELETE_VARIABLE` (see Table 8, page 39), which can be used to delete parameters or variables defined with VS Commands from the Echo file. This command can be used to remove variables that are no longer useful after relevant equations have been deleted.

Delete Equations for All Variables

To clear all equations added with one of the VS Commands, use the associated `RESET_EQS_`*x* command. For example, use `RESET_EQS_IN` to delete all equations added with the `EQ_IN` command. The command `RESET_EQS_ALL` clears all equations added with VS Commands.

Calculate Import Variables with VS Commands

The VS Math Models are formulated to include variables that can be given values with equations that did not exist when the VS Solver libraries were created. These include forces, moments, controls, road properties, and other information. Values for these variables can be imported from other simulation environments, such as Simulink, LabVIEW, ASCET, or custom software. Alternatively, these variables can also be updated using equations added with the `EQ_` VS Commands described in the preceding sections.

Note SuspensionSim models do not have built-in import variables in the sense described in this section. Once a joint is added to the model, its motions and loads can be prescribed with configurable functions as described in the *SuspensionSim Commands and Parameters* reference manual.

The list of potential Import variables can be viewed from the **Run Control** screen of a VS Browser using the **View** button located in the lower-right part of the window, or by using the interactive browser for the VS Browser library screen used for specifying Import variable interactively.

By default, all potential Import variables are ignored unless they are activated from the Parsfile. Within the Parsfile, the syntax for activating an Import variable is:

[`IMPORT`] *keyword* [*mode* [*initial_value* [; [*units*]]] [;]

where *keyword* is the name of the Import variable, *mode* is a keyword that specifies a mode for using the Import variable, *initial_value* is an expression that is evaluated to obtain the numerical value of the variable to be used before imports are available, and *units* are the user display units to be used for the variable. Note that all keywords except the name of the variable (*keyword*) are optional. Also, *initial_value* cannot be specified unless *mode* is set, and *units* cannot be specified unless both *mode* and *initial_value* are set.

The *mode* determines how the imported variable is combined with a native variable built into the VS model in case the Import variable is associated with a built-in variable. There are seven valid *mode* names, listed in Table 12. Existing built-in Import variables all have a default *mode* of `IGNORE`. This is also the default mode for any new Import variables created with the `DEFINE_IMPORT` command.

Table 12. Modes for using Import variables.

Mode Symbol	**Increment Array of Imports?**	**Description**
`IGNORE`	No	Default mode: ignore externally provided value; use only the built-in variable.
`ADD`	Yes	Add the Import variable to the built-in variable.
`VS_ADD`	No	
`MULTIPLY`	Yes	Multiply the Import variable with the built-in variable. The Import variable will be dimensionless in this case.
`VS_MULTIPLY`	No	
`REPLACE`	Yes	Replace the value of the built-in variable with the value of the Import variable.
`VS_REPLACE`	No	

For example, if the imported variable is the brake control and the mode is `REPLACE`, then the brake control within the VS Math Model is replaced with the value of the Import variable. If the VS Math Model is running as part of a Simulink model, the brake control is included in the array of variables that are transferred each timestep from Simulink to the VS Math Model. If the mode is `ADD`, then the brake control is also included in the array of variables transferred from Simulink; however, in this case, the imported value is added to the internal value. If the mode is `MULTIPLY`, then the internal value is multiplied by the imported value. If not specified, the default is that the mode is `ADD`.

Note The `REPLACE` and `MULTIPLY` modes are only useful for built-in Import variables that are linked to internal variables in the VS Math Model. For all Import variables created with the `DEFINE_IMPORT` command, and for some built-in Import variables, there are no linked internal variables.

If the mode keyword has the prefix `VS_` (`VS_ADD`, `VS_MULTIPLY`, or `VS_REPLACE`), and the VS Math Model is running with Simulink or LabVIEW, the Import variable is **not** added to the array of variables transferred from the external software to the VS Math Model. These modes allow built-in features of the VS Math Model to be extended with equations added with VS Commands, without affecting the interface to any external models defined in environments such as Simulink, LabVIEW, or ASCET. If the solver is running by itself, then there is no difference between the modes with and without the `VS_` prefix.

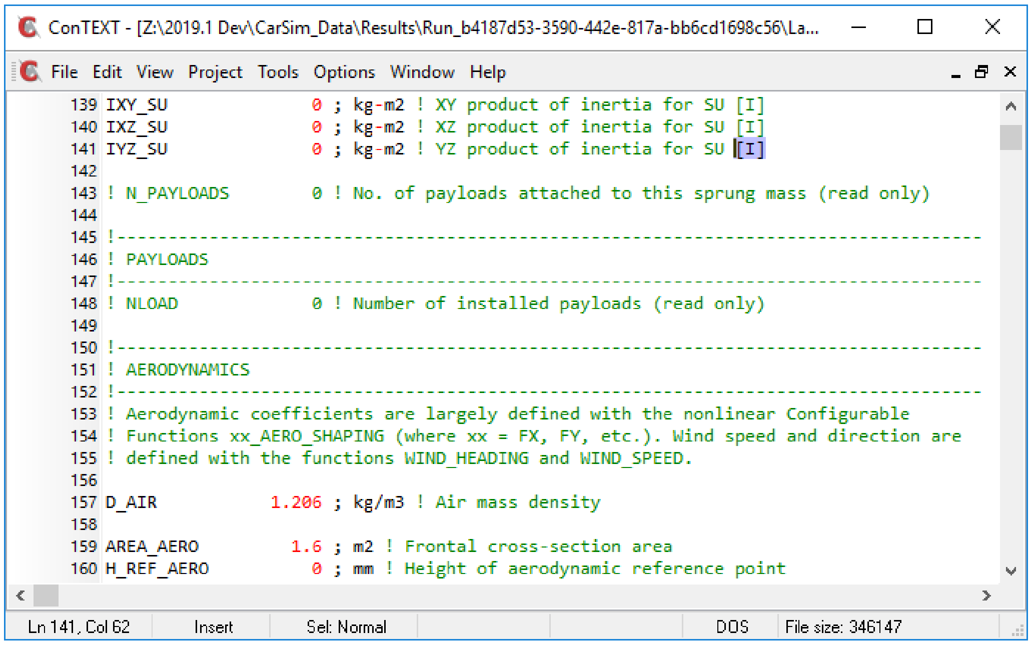
If the Import variables are imported from other software (e.g., a Simulink model), there is a potential problem during a initialization, because the values have not yet been imported. If the variable affects calculations made during the initialization, the *initial_value* expression is evaluated to obtain the numerical value of the variable during initialization. If an initial value is not specified, a value of 0.0 is used if the mode is `ADD` or `REPLACE`; an initial value of 1.0 is used if the mode is `MULTIPLY`. With normal operation, the initialization is done twice: the first time to calculate output variables that might be exported, and the second time using import variables obtained at the start time.

If the optional `IMPORT` command is used and *keyword* is not recognized as a valid potential Import variable, then an error message is generated. On the other hand, if the line in the Parsfile begins with *keyword* and it is not recognized, no error message is generated. In most applications, failure to recognize *keyword* will cause problems with the external model (e.g., Simulink), so aggressive error handling is usually preferred. For this reason, the `IMPORT` command option is recommended.

Add Dynamic Effects to Parameters

VS Commands can be used to add equations for some parameters that are nominally constant, in order to add sensitivity to the model (Table 10, page 51). However, this is not safe to do with all parameters in the model, because some are used to calculate quantities during the initialization step in the simulation. In these cases, the initialization should be repeated whenever a parameter is changed; use a *VS Event* to change one or more parameters and invoke an initialization.

The Echo files list all of the parameters used in the run, and indicates where a change in the parameter value requires some initialization. Parameters that affect initialization are indicated with `[I]` (see the first few lines in Figure 10).


ConTEXT - [Z:\2019.1 Dev\CarSim_Data\Results\Run_b4187d53-3590-442e-817a-bb6cd1698c56\La...
File Edit View Project Tools Options Window Help
Ln 141, Col 62
Insert
Sel: Normal
DOS
File size: 346147

```
IXY_SU              0 ; kg-m2 ! XY product of inertia for SU [I]
IXZ_SU              0 ; kg-m2 ! XZ product of inertia for SU [I]
IYZ_SU              0 ; kg-m2 ! YZ product of inertia for SU [I]

! N_PAYLOADS        0 ! No. of payloads attached to this sprung mass (read only)

!------------------------------------------------------------------------------------
! PAYLOADS
!------------------------------------------------------------------------------------
! NLOAD             0 ! Number of installed payloads (read only)

!------------------------------------------------------------------------------------
! AERODYNAMICS
!------------------------------------------------------------------------------------
! Aerodynamic coefficients are largely defined with the nonlinear Configurable
! Functions xx_AERO_SHAPING (where xx = FX, FY, etc.). Wind speed and direction are
! defined with the functions WIND_HEADING and WIND_SPEED.

D_AIR           1.206 ; kg/m3 ! Air mass density

AREA_AERO         1.6 ; m2 ! Frontal cross-section area
H_REF_AERO          0 ; mm ! Height of aerodynamic reference point
```

Figure 10. An Echo file indicates whether parameters affect initialization.

Restore a Saved Time and State

Note This feature is not available in SuspensionSim.

A VS Math Model performs calculations at closely spaced intervals of time, using equations from the internal VS Math Model. The current state of the model is defined by the simulation time, plus a set of independent variables called *state variables*. Some of the state variables are calculated from ordinary differential equations (ODE's) using numerical integration as described in the VS Tech Memo: *Numerical Integration Methods in VS Math Models*. Others are calculated using conditional equations to provide additional information. Regardless of how they are calculated, the state of the VS Math Model is completely determined if values are known for all state variables (and all parameters and Configurable Function datasets).

VS Math Models have the built-in capability to continue a previous run. Values of all state variables are written in a Parsfile at the end of the run, along with all of the parameters, tables (Configurable Functions), and VS Commands in use by the model at the end of the run. The Parsfile — typically named with a suffix `_END.PAR` (e.g., `Run_a03…_END.PAR`) — can be used to start a new run that continues from where the previous one was stopped. Rather than using initialization options to set up the state variables, the variables are simply assigned the values that existed at the end of the setup run.

A similar capability can be applied during a run, using the VS Commands listed in Table 13 to save the state of the VS Math Model at various times, and to go back to a saved state if some condition occurs. The capability of jumping back in time is fully supported when running VS Math Models with no external software, or when running under the control of custom code that communicates with the VS Math Model using the VS API.

Table 13. Commands (and one function) used to save and restore the model state.

Command	**Description**
`RESET_STATES`	Free memory used to save states.
`RESTORE_STATE`	Back up to a specified time and restore the state of the model, then continue from that point in time.
`SAVE_STATE`	Save the current state of the model in memory.
`SAVED_STATE_TIME(`*t*`)`	Function: time for the last state saved before *t*.
`START_SAVE_TIMER`	Start saving the state of the model automatically at a specified time interval.
`STOP_SAVE_TIMER`	Stop automatic saving of the model state.

Three of the commands involve saving the state of the model for future use (these are `SAVE_STATE`, `START_SAVE_TIMER`, and `STOP_SAVE_TIMER`). The command `RESTORE_STATE` causes the simulation to back up to a saved state (if possible) and continue.

RESET_STATES

```
RESET_STATES
```

This command removes all saved states from memory.

RESTORE_STATE

`RESTORE_STATE` *time*

This command causes the VS Math Model to jump back in time to the most recent saved state made at or before *time*. All state variables are reset to saved values, and the simulation time is reset to the time at which the state was saved.

Any saved states that are more recent than the one restored are erased. For example, suppose that states are being saved at intervals of 0.1s as the result of having used the `START_SAVE_TIME` command. If, at t = 5s, the command `RESTORE_STATE 2.03` is applied, the solver will restore the state of the model to the most recent state that was saved before t = 2.03. This might be a state saved at t = 2.000. This has the effect of jumping back in time, from t = 5s to t = 2s. All states saved after t = 2.000 are erased. As the run continues, states will continue to be saved at 0.1s intervals.

If the system parameter `OPT_BUFFER_WRITE` is set to zero (the default value), then output variable are written to the VS or ERD file as the run proceeds. The file is not overwritten when the `RESTORE_STATE` command is applied, which means that the file contains the original time histories of the outputs followed by new values that are calculated after the restore operation. On the other hand, if the parameter `OPT_BUFFER_WRITE` is set to specify the use of a memory buffer, then the buffer is managed when a `RESTORE_STATE` command is activated. The effect is that any results that occurred after the time of the saved state are erased, to be replaced with new results as the run continues.

Keeping track of time and the various options for viewing results from a simulation can be tricky when the `RESTORE_STATE` command is used. The options are discussed in more detail in the next section (page 63).

Parameters and Configurable Functions (tables) in the model are not affected by this command. Indeed, the main reason for restoring a previous state is to see the effect of a change in a parameter or Configurable Function.

Variables added using the `DEFINE_VARIABLE` command are saved and restored; those added with `DEFINE_PARAMETER` are not.

SAVED_STATE_TIME (function)

Table 13 also lists a function `SAVED_STATE_TIME(`*`t`*`)` that returns the time associated with the most recent state that was saved at or before the specified time *t*. For example, if a timer was started with the command `START_SAVE_TIMER` to save at intervals of 0.1, then the function `SAVED_STATE_TIME(1.234)` would return a value of 1.2.

SAVE_STATE

This command causes the VS Math Model to take a snapshot of the current state of the model and save it in memory for possible future use. The snapshot consists of the current simulation time, plus the current values of all state variables (including the built-in variables, those added with VS Commands, and those added with VS API functions), and the current values of all active Import variables (if there are any).

In addition, the snapshot contains the indices into all VS Reference Paths, used to determine path station and lateral position (S and L) based on global X and Y values. This is done because these particular actions can have multiple valid solutions in some cases; the save information ensures that the correct positions are restored.

Parameters and Configurable Functions in the model are considered independent of the state, and are not saved with this command. (Except for advanced applications, they are constant during a run.) Variables added using the `DEFINE_VARIABLE` command are saved and restored; those added with `DEFINE_PARAMETER` and `DEFINE_OUTPUT` are not.

This command is applied during a run by putting the command into a Parsfile that is read when an Event is triggered.

START_SAVE_TIMER

`START_SAVE_TIMER` *time_interval*

This command starts an automated process in which the solver starts saving the model state automatically at *time_interval*. The effect is the same as if a chain of Events were defined to apply the `SAVE_STATE` command repeatedly at the specified interval.

If the command is applied at the start of the run (where simulation time is equal to the start time), then the first state is saved after *time_interval* seconds have elapsed. However, if the command is applied after the start of the run via an Event, the first state is saved at the same timestep in which the command is read.

States are saved automatically until the run ends, or until the timer is stopped with the `STOP_SAVE_TIMER` command.

STOP_SAVE_TIMER
This command stops the automated process initiated with `START_SAVE_TIMER`. It does not affect the status of any saved states (they remain in memory, available for use).

Limits to Restoring Saved States

The ability to restore states can be negatively affected by user-defined VS Commands equations or by external software. The critical rule is that all equations must solely depend on state variables, or else Restore States can produce non-deterministic results. For example, see the following VS Commands equation which depends on the output variable, `Z_RP1`.

```
EQ_IN IMP_FZ_RP_1 = 0.5 * Z_RP1;
```

Since `Z_RP1` is not a state variable, it will contain an unpredictable value after the Restore operation, and it will ultimately affect vehicle motion after Restore. This equation can be rewritten using a state variable (see `DEFINE_VARIABLE`), and its behavior through the Restore process will be deterministic.

```
DEFINE_VARIABLE SV_Z_RP1 0;
EQ_IN IMP_FZ_RP_1 = 0.05 * SV_Z_RP1;
EQ_OUT SV_Z_RP1 = Z_RP1;
```

Restoring states is not possible on real-time systems with hardware-in-the-loop, because jumping back in time clearly violates the connection between simulation time and physical real time.

Restore capability might also be limited when running under the control of external simulation workspace software such as Simulink and LabVIEW. Variables internal to the VS Math Models are saved and restored properly, but simulation time is not because it is calculated in the external workspace. Also, model variables in the external workspace are not directly affected by the `RESTORE_STATE` command. Overall, the capability of restoring a saved state when working with external models depends on the nature of the external model; if the external model has its own internal dynamics, then the capability to restore a saved state will probably not work as intended.

The capability to restore a past state requires that the number of state variables be unchanged between the time a state was saved and the time it is restored. If the `DEFINE_VARIABLE` or `DEFINE_OUTPUT` commands are used after a state is saved, then that state cannot be restored because the array sizes are not consistent. On the other hand, it is OK to use `DEFINE_PARAMETER` after a state is saved, because parameters are not affected by a restore operation.

Simulation Variables for Time

By definition, time is the one truly independent variable in a time-domain VS Math Model; it is not calculated by the model, but is instead provided by the simulation environment. The simulation time in a VS Math Model starts with a specified value `TSTART` and increases incrementally as the simulation proceeds.

Time in the Math Model and the Output File

The VS Math Model for a vehicle or other dynamic system integrates internal ODEs with respect to simulation time T. When viewing results by video or plotting, the main independent variable is also time. In this case, the concept of time is based on the recording. The VS Math Models allow output to be enabled and disabled, so the duration of the recording can be less than the duration of the simulation.

Table 14 lists four names for time. Two (`T` and `T_Stamp`) apply to the simulation time. Both are associated with the same internal variable, but only `T_Stamp` is used to identify a channel in the output file. The other two (`Time` and `T_Record`) apply to the recorded time. The variable `Time` is calculated by post-processing software such as VS Visualizer using the start time `T_START`, the (calculated) time interval for output, `TSTEP_WRITE`, and the sample number from the file (1, 2, 3…). The variable `T_Record` is calculated internally the same way within the VS Math Model. This is done to support advanced user in VS Commands, and to support plotting and analyses using CSV output files that do not include timestep information.

Table 14. Variables for simulation time and recording time.

Variable	**Location**	**Definition**
`T`	VS Math Model	Simulation time calculated by numerical integrator or provided by external software (e.g., Simulink)
`Time`	VS/ERD File	Time from sample number in an output file
`T_Record`	Model and File	Alternate name for `Time` that is also written to output file
`T_Stamp`		Alternate name for `T` that is also written to output file

Three of the variables are recognized within the VS Math Model: `T`, `T_Record`, and `T_Stamp`. They may be referenced in VS Commands. However, they are protected and cannot be modified by VS Commands. Three are available for setting up plots with VS Visualizer: `Time`, `T_Record`, and `T_Stamp`.

Note Multiple names are used for the two variables to support legacy results when different variables were used internally and written to file.

Variables Representing Real Time

There are times when it is useful to track the "real time," sometimes called clock time or wall-clock. For example, if a simulation covers 60 simulated seconds, it might run in 4 seconds of real time. Clock precision can vary by platform, but the solver will attempt to use highest precision available. Two variables are provided for the real time needed each timestep. One — `T_Real_Step` — is used only by the VS Math Model, from the start of the calculations to the finish. The other — `T_Real_Last` — goes from the finish of the previous timestep to the finish of the last timestep. This includes time outside the calculations of the VS Math Model that is due to external software.

Table 15. Variables based on real clock time.

Variable	Definition
`T_Real_Elapsed`	Elapsed real time since the start of the simulation
`T_Real_Last`	Total real time used by the Math Model plus the calling program for the last timestep
`T_Real_Step`	Real time used by the Math Model for the last timestep

Viewing Results from a Run with Restored States

There are several options for viewing results from a run in which the model was backed up in time using the `RESTORE_STATE` command. For example, suppose the run covered 10.0 simulated seconds, with the last 2.0 seconds having been repeated 10 times using the `SAVE_STATE` and `RESTORE_STATE` commands. This means that the simulation time was always in the range of 0.0 to 10.0s, but the total simulation time was 10 + 2*10 = 30s.

1. If the output variables were written to the VS or ERD file as the run proceeded, then all 30s of simulation outputs would be available for viewing, including the ten repeats.
    a. The variable `Time`, used by the animator, would go from 0 to 30.
    b. The variable `T_Stamp`, available for use in plots, would always be in the range of 0 to 10. Plots could be made using either `Time` (0 to 30) or `T_Stamp` (0 to 10). Plots made using `T_Stamp` would show overlays of 11 results for the last 2.0 seconds of the run.
2. If the output variables were written to a memory buffer as the run proceeded, then only the most recent version is available for viewing. (This option is set using the system parameter `OPT_BUFFER_WRITE`.) Both `Time` and `T_Stamp` would go from 0 to 10. Results from the last 2 seconds would be taken from the final iteration.

Linearize

Note This feature is not applicable to SuspensionSim.

VS linearization commands produce text files with linearized descriptions of the state of the VS Math Model that can be used for frequency-domain analyses such as bode and root locus plots. You will need the MATLAB software from The Math Works to analyze the linearized model and the MATLAB *Control System Toolbox* to view bode (frequency response) plots. The general approach is documented in the tech memo *Linear Analysis in VehicleSim Models*, along with examples.

Prior to the availability of high-speed digital computers, researchers and engineers typically studied dynamic systems by analytic frequency-domain methods such as eigenvalues and bode plotting. These methods have been used in automotive dynamic analyses since the 1950s, and in motorcycle dynamic analyses since the 1970s. The classical frequency-domain methods are all based on system models with linear equations of motion.

Although modern nonlinear vehicle dynamics models such as those in BikeSim, CarSim, and TruckSim run much faster than real time, the linear analysis capability remains valuable for engineers, especially for motorcycle studies.

Consider a dynamic VS Math Model where the variables of interest are organized into three arrays: **x** is an array of *n* state variables in the system, **y** is an array of *m* output variables, and **u** is an array of *r* control inputs. A linear dynamic system described with ordinary differential equations (ODEs) is typically represented with equations of the form:

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)\end{aligned} \tag{1}$$

where **A**, **B**, **C**, and **D** are matrices whose dimensions are matched to the arrays: **A** is *n* x *n*, **B** is *n* rows x *r* columns, **C** is *m* x *n*, and **D** is *m* x *r*.

Traditional control theory defines frequency analysis and stability criteria that are inherent properties of linear systems. MATLAB has built-in functions for calculating frequency response characteristics when provided **A**, **B**, **C**, and **D** matrices from Eq. 1.

The equations in a VS Math Model do not fit the form assumed in Eq. 1. As mentioned in *VS Math Models Manual*, not all the state variables in a VS Math Model are calculated with ODEs. Those state variables that are defined with ODEs have equations that are generally not linear combinations of the other state variables. They contain nonlinear force equations, nonlinear motions involving trigonometric functions of angles, friction and other hysteresis, lock conditions, and many other physically realistic characteristics that are nonlinear.

The ODEs for a VS Math Model have nonlinear equations of motion that can be expressed in a more generic, potentially nonlinear form:

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{f}\{\mathbf{x}(t),\, \mathbf{u}(t)\} \\ \mathbf{y}(t) &= \mathbf{g}\{\mathbf{x}(t),\, \mathbf{u}(t)\}\end{aligned} \tag{2}$$

where **f** and **g** are nonlinear array functions of time. In this expression of the model equations, **x** is an array of the state variables from the VS Math Model that are defined with ODEs, and **u** is an array of activated control variables.

Although the full model equations are nonlinear, the behavior of the full system can be approximated as a linear system at any time during a simulation run using *perturbation.* (See the *Appendix* in *Linear Analysis in VehicleSim Models* for details on the calculations.) At any time during the simulation run, the `LINEARIZE` VS Command can be used to apply a built-in perturbation algorithm to create **A**, **B**, **C**, and **D** matrices, and write them into a MATLAB M-file. After the simulation run has finished, M-files generated during the run can be loaded into MATLAB for analysis.

Linearization options are controlled with four VS Commands (Table 16), along with a parameter `OPT_LINEARIZATION` that determines whether linearization applies for all state variables in the model or a subset.

Table 16. VS Commands used in linearization.

Command	**Action**
`LINEARIZE`	Trigger the linearization and write matrices to M-File.
`LINEAR_SV`	Specify a state variable to perturb and linearize.
`LINEAR_CONTROL`	Specify a control variable to perturb and linearize.
`LINEAR_OUTPUT`	Specify an output variable for linearization.

LINEARIZE

`LINEARIZE` *pathname*

Whenever the VS Math Model reads the command `LINEARIZE`, it writes the linear matrices in the text file with the specified *pathname* using the MATLAB M-file format. *Pathname* can be a global pathname, a pathname relative to the current VS Database working directory, or a dynamic string.

Here is an example using a relative pathname:

```
LINEARIZE Extensions\Linearization\Linear_200kph_Road_to_Steer.m
```

Here is an example using a dynamic string:

```
LINEARIZE "Extensions\Linearization\"+"Linear_" + VX_TARG*3.6 + "kph.m"
```

In this case, files might be generated with the names that include the current value of a variable named `VS_TARG`: `Linear_0kph.m`, `Linear_10kph.m`, …. `Linear_200kph.m`.

The `LINEARIZE` command generates a request in the VS Math Model to make the analysis and write the file. The actual work takes place at the end of the timestep in which the command is encountered. This means that other inputs that are read from the Parsfile at the same time will be processed before the linearization calculations are made. For example, if a VS Event is used to apply the `LINEARIZE` command, and the same Event dataset changes some parameters, those parameter changes are made before the `LINEARIZE` calculations are made, regardless of the sequence of the specifications in the dataset.

If you want to apply the `LINEARIZE` command and then change some properties in the Math Model, you should make the parameter changes in another Event dataset that is triggered after the `LINEARIZE` command is applied.

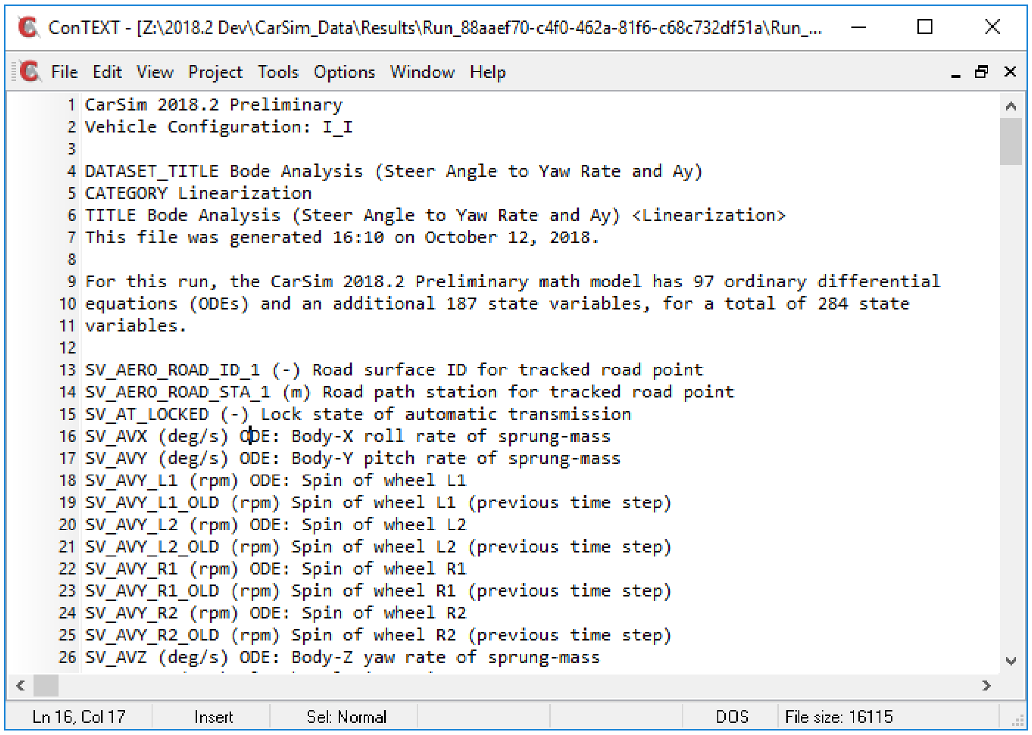
LINEAR_SV

`LINEAR_SV` *state_variable* [*perturbation*]

VS Math Models have two options for identifying the state variables to linearize, controlled by the parameter `OPT_LINEARIZATION`. If set to zero (the default), then all of the state variables that are defined with ODEs within the VS Math Model are used as state variables for the linear analysis. On the other hand, a non-zero value for `OPT_LINEARIZATION` specifies that only the state variables identified by the `LINEAR_SV` command be perturbed for the linearization.

Be sure to set `OPT_LINEARIZATION` = 1 before you specify state variables using the `LINEAR_SV` command. If `OPT_LINEARIZATION` = 0, the `LINEAR_SV` command is ignored.

State_variable should be one of the state variables indicated with ODE in the description obtained using the **View** button in the lower-right corner of the **Run Control** screen with the file type set to State Variables. For example, lines 16-18 in Figure 11 identify state variables calculated via numerical integration of ODEs.


ConTEXT - [Z:\2018.2 Dev\CarSim_Data\Results\Run_88aaef70-c4f0-462a-81f6-c68c732df51a\Run_...
File Edit View Project Tools Options Window Help
Ln 16, Col 17
Insert
Sel: Normal
DOS
File size: 16115

```
 1 CarSim 2018.2 Preliminary
 2 Vehicle Configuration: I_I
 3
 4 DATASET_TITLE Bode Analysis (Steer Angle to Yaw Rate and Ay)
 5 CATEGORY Linearization
 6 TITLE Bode Analysis (Steer Angle to Yaw Rate and Ay) <Linearization>
 7 This file was generated 16:10 on October 12, 2018.
 8
 9 For this run, the CarSim 2018.2 Preliminary math model has 97 ordinary differential
10 equations (ODEs) and an additional 187 state variables, for a total of 284 state
11 variables.
12
13 SV_AERO_ROAD_ID_1 (-) Road surface ID for tracked road point
14 SV_AERO_ROAD_STA_1 (m) Road path station for tracked road point
15 SV_AT_LOCKED (-) Lock state of automatic transmission
16 SV_AVX (deg/s) ODE: Body-X roll rate of sprung-mass
17 SV_AVY (deg/s) ODE: Body-Y pitch rate of sprung-mass
18 SV_AVY_L1 (rpm) ODE: Spin of wheel L1
19 SV_AVY_L1_OLD (rpm) Spin of wheel L1 (previous time step)
20 SV_AVY_L2 (rpm) ODE: Spin of wheel L2
21 SV_AVY_L2_OLD (rpm) Spin of wheel L2 (previous time step)
22 SV_AVY_R1 (rpm) ODE: Spin of wheel R1
23 SV_AVY_R1_OLD (rpm) Spin of wheel R1 (previous time step)
24 SV_AVY_R2 (rpm) ODE: Spin of wheel R2
25 SV_AVY_R2_OLD (rpm) Spin of wheel R2 (previous time step)
26 SV_AVZ (deg/s) ODE: Body-Z yaw rate of sprung-mass
```

Figure 11. Part of a State Variables file obtained with the View button on the Run Control screen.

The optional *perturbation* is a number used to perturb *state_variable* during the linearization process. If not specified, a default of 0.001 is used. The units of *perturbation* are always the internal SI units, regardless of the current user display units for *state_variable*.

LINEAR_CONTROL

`LINEAR_CONTROL` *Import perturbation*

Use this command to identify variables for the **u** array. *Import* should be an Import variable indicated with the prefix `IMP_` in the VS Math Model text file (*base*`_imports.txt`).

The optional *perturbation* is a number used to perturb *Import* during the linearization process. If not specified, a default of 0.001 is used. The units of *perturbation* are always the internal SI units, regardless of the current user display units for *Import*.

It is OK to perform a linearization without specifying any control variables; however, if there is no **u** array, then there will be no **B** or **D** matrices.

LINEAR_OUTPUT

`LINEAR_OUTPUT` *Export*

Use this command to identify variables for the **y** array. *Export* should be an output variable with the prefix `EXP_`.

It is OK to perform a linearization without specifying any output variables; however, if there is no **y** array, then there will be no **C** or **D** matrices.

Limits Linearization can be negatively affected by user-defined VS Commands equations or external software. Equations must solely depend on state variables, or else the linearization process can produce unpredictable results. See the section *Limits to Restoring Saves States*.

5. Configurable Functions

VS Expressions can include basic arithmetic and Boolean operators (+, -, *, /, ^,<,>,etc.) and the math functions listed earlier in Table 2 (page 17).

VS Math Models often use Configurable Functions to define potentially nonlinear relationships between one or two independent variables and a dependent variable that is calculated by a method specified at runtime. Most interactions involving Configurable Functions are handled with GUI controls, as described in *VS Browser Reference Manual*.

Nearly all the keywords used to configure the functions from a Parsfile are commands. Although they are commands, most support basic use of Configurable Functions and are documented in *VS Math Models Manual*. This chapter adds information about advanced uses of Configurable Functions.

Calculating the Value from a Configurable Function

Here is a short review of the calculation options for Configurable Functions; please see *VS Math Models Manual* for more detail.

Some Configurable functions calculate a dependent variable F from a single independent variable X with one of four methods:

1. Constant: F = constant
2. Linear relationship: $F = \textit{coefficient} \bullet X$
3. 1D table-lookup: $F = f_1(X)$
4. Custom user-defined equation: $F = f_1(X)$

Other Configurable Functions can calculate the dependent variable from two independent variables. In the most general cases, the calculations are based on a table with columns and rows involving two independent variables: *row* variable X, and *column* variable $Xcol$. Up to four additional methods are available to calculate the output variable F:

5. 2D table-lookup: $F = f_2(Xcol, X)$
6. Sum of two single-variable functions: $F = f_r(\mathrm{X}) + f_c(Xcol)$
7. Product of two single-variable functions: $F = \mathrm{f}_\mathrm{r}(X) \bullet f_c(Xcol)$
8. Custom user-defined equation: $F = f_2(Xcol, X)$

Table 17 summarizes the calculation methods available in Configurable Functions.

For a given function name *func* (e.g., `FD`, `ROAD_DZ`), Table 17 shows that there may be up to five commands that configure the function to use a type of calculation. The commands for a constant or a coefficient set the function up with no table interpolation. The command for an equation sets the function up to evaluate a formulation involving a row variable, possibly a column variable, and possibly any other variables of interest that are in the model. The table options involve both interpolation within the range of tabular data provided as input, and extrapolation beyond the range.

Table 17. Summary of calculation methods for Configurable Functions

Type	**Command**	**Argument**	**Interpolation**	**Extrapolation**
Constant	*func*`_CONSTANT`	Value	—	Flat
Coefficient	*func*`_COEFFICIENT`	Value	—	Linear
1D Table	*func*`_TABLE`	keyword	Linear	Linear, Flat, Loop
			Spline	Linear, Flat, Loop
			Step	Flat, Loop
2D Table	*func*`_CARPET`	keyword	Linear	Linear, loop, From Zero
			Step	Flat
			Spline	Linear
			Variable-width linear	Linear
			Variable-width step	Flat
Equation	*func*`_EQUATION`	Formula	—	—

Examples for each type of calculation method are shown in *VS Browser (GUI and Database) Reference Manual.*

Up to six parameters are available to provide scaling and offsets for the calculated value and for the independent variable(s) in a Configurable Function (Table 18).

Table 18. Configurable function modifiers and keywords.

Keyword / Suffix	**Example: MU_ROAD**	**Description**
`_GAIN`	`MU_ROAD_GAIN 1`	Gain and offset for calculated value
`_OFFSET`	`MU_ROAD_OFFSET 0`	
scale_xrow_keyword	`SSCALE_MU_ROAD 1`	Scale and offset for primary independent variable
start_xrow_keyword	`SSTART_MU_ROAD 0`	
scale_xcol_keyword	`L_SCALE_MU_ROAD 1`	Scale and offset for secondary independent variable
start_xcol_ keyword	`L_START_MU_ROAD 0`	

Not all of the options listed in Table 17 and Table 18 can be applied for every installed Configurable Function. The Echo file always indicates which of the options are supported.

For example, some Configurable Functions do not support `CONSTANT` values and others do not support linear `COEFFICENT` values.

Using Configurable Functions in Formulas

Configurable Functions built into the VS Math Models can be used in algebraic expressions.

Syntax

The syntax for using a Configurable Function is:

function_name (*col_var*, *row_var*, *index*)

where *function_name* is the root name of the Configurable Function, *col_var* and *row_var* are two expressions used as inputs to the function, and *index* indicates which dataset to use. (The argument is rounded internally to the nearest integer.) This syntax fits the most general case of a Configurable Function that can have multiple datasets (such as one dataset per wheel of the vehicle), with a calculated output dependent on up to two independent variables.

Even if the Configurable Function in the VS model can never make use of the first argument (*col_var*) or the last argument (*index*), they must still be included when using the function in a symbolic expression. The VS Math Model will check the value of *index* to ensure it is valid, and will generate an error message if it is not.

Indexing and ID Numbers

Some Configurable Functions support only one dataset, e.g., throttle as a function of time (if there is a single vehicle in the simulation). To use the `THROTTLE` function, you would provide an *index* of 1, e.g., `THROTTLE(0,T,1)`.

Other Configurable Functions support multiple datasets with a single index, e.g., a transmission downshift table calculates a rotational speed as function of one independent variable (throttle) and one index (gear to downshift to, `IGEAR`). To use the `DOWNSHIFT_TRANS` function, you provide a form such as `DOWNSHIFT_TRANS(0,Throttle,3)` to obtain the transmissions speed for a downshift from gear 4 to 3.

Many of the Configurable Functions in a VS Math Model show multiple indices when listed in an Echo file, e.g., the Configurable Function FD for specifying damping force as a function of stroke rate (Figure 12).

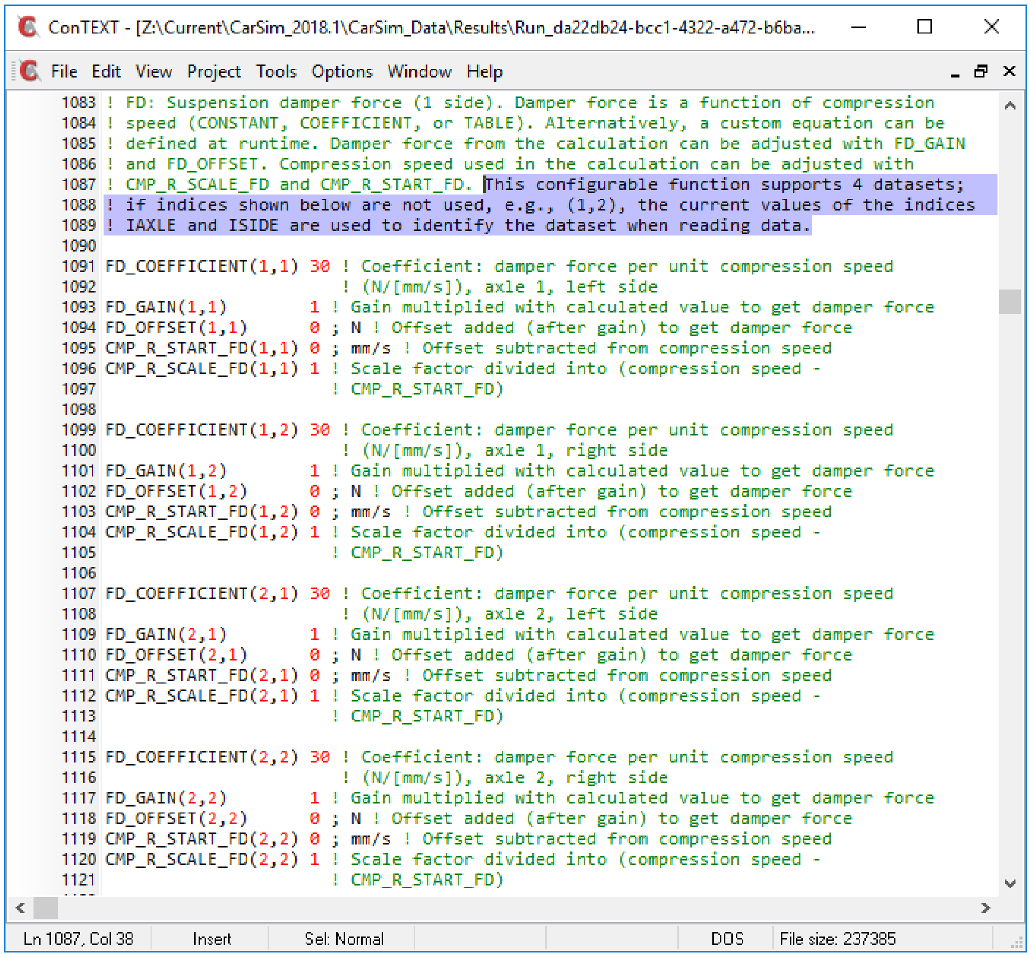
The highlighted text indicates that there are four shock absorber datasets this model, so the valid values for *index* are numbers that round off to integer values ranging from 1 to 4. To determine the *index* for a particular camber dataset, you should look at the Echo file generated by the VS Math Model. The Configurable Functions are always printed in the sequence of their internal indices. For this example, the sequence for the camber Configurable Functions in CarSim is:

1. (1,1) (Front-left corner)
2. (1,2) (Front-right corner)
3. (2,1) (Rear-left corner)
4. (2,2) (Rear-right corner)

For example, an *index* value of 3 would specify the table for the rear-left wheel, shown in the Echo file with indices (2,1). Consider the VS Command:

```
define_output fd_aux = fd(0,new_rate,3);n;...
  Extra damper force at rear-left wheel
```

This will define a new output variable calculated using the FD function for ID 3, using an independent variable `new_rate` (presumably defined using another VS Command).



ConTEXT - [Z:\Current\CarSim_2018.1\CarSim_Data\Results\Run_da22db24-bcc1-4322-a472-b6ba...
File Edit View Project Tools Options Window Help
Ln 1087, Col 38
Insert
Sel: Normal
DOS
File size: 237385

```
! FD: Suspension damper force (1 side). Damper force is a function of compression
! speed (CONSTANT, COEFFICIENT, or TABLE). Alternatively, a custom equation can be
! defined at runtime. Damper force from the calculation can be adjusted with FD_GAIN
! and FD_OFFSET. Compression speed used in the calculation can be adjusted with
! CMP_R_SCALE_FD and CMP_R_START_FD. This configurable function supports 4 datasets;
! if indices shown below are not used, e.g., (1,2), the current values of the indices
! IAXLE and ISIDE are used to identify the dataset when reading data.

FD_COEFFICIENT(1,1) 30 ! Coefficient: damper force per unit compression speed
                       ! (N/[mm/s]), axle 1, left side
FD_GAIN(1,1)        1 ! Gain multiplied with calculated value to get damper force
FD_OFFSET(1,1)      0 ; N ! Offset added (after gain) to get damper force
CMP_R_START_FD(1,1) 0 ; mm/s ! Offset subtracted from compression speed
CMP_R_SCALE_FD(1,1) 1 ! Scale factor divided into (compression speed -
                      ! CMP_R_START_FD)

FD_COEFFICIENT(1,2) 30 ! Coefficient: damper force per unit compression speed
                       ! (N/[mm/s]), axle 1, right side
FD_GAIN(1,2)        1 ! Gain multiplied with calculated value to get damper force
FD_OFFSET(1,2)      0 ; N ! Offset added (after gain) to get damper force
CMP_R_START_FD(1,2) 0 ; mm/s ! Offset subtracted from compression speed
CMP_R_SCALE_FD(1,2) 1 ! Scale factor divided into (compression speed -
                      ! CMP_R_START_FD)

FD_COEFFICIENT(2,1) 30 ! Coefficient: damper force per unit compression speed
                       ! (N/[mm/s]), axle 2, left side
FD_GAIN(2,1)        1 ! Gain multiplied with calculated value to get damper force
FD_OFFSET(2,1)      0 ; N ! Offset added (after gain) to get damper force
CMP_R_START_FD(2,1) 0 ; mm/s ! Offset subtracted from compression speed
CMP_R_SCALE_FD(2,1) 1 ! Scale factor divided into (compression speed -
                      ! CMP_R_START_FD)

FD_COEFFICIENT(2,2) 30 ! Coefficient: damper force per unit compression speed
                       ! (N/[mm/s]), axle 2, right side
FD_GAIN(2,2)        1 ! Gain multiplied with calculated value to get damper force
FD_OFFSET(2,2)      0 ; N ! Offset added (after gain) to get damper force
CMP_R_START_FD(2,2) 0 ; mm/s ! Offset subtracted from compression speed
CMP_R_SCALE_FD(2,2) 1 ! Scale factor divided into (compression speed -
                      ! CMP_R_START_FD)
```

Figure 12. Part of an Echo file showing FD data.

Setting Units

A Configurable function involves two or three types of variables (output, row, and column for functions with two independent variables).

The units associated with these variables can be changed using the commands `SET_UNITS`, `SET_UNITS_TABLE_ROW`, and `SET_UNITS_TABLE_COL` (described earlier, page 32). Units for scalar values can be set in-line, as done with other parameters in the VS Math Model. If the table has parameters for offsetting the variables, then the in-line method can be used to set the units for all three types of variables. If not, the commands `SET_UNITS`, `SET_UNITS_TABLE_ROW`, and `SET_UNITS_TABLE_COL` can be used to set the units for the output variable, row variable, and column variable, respectively.

If any of these commands are used, the keyword should be for the table or carpet plot (e.g., `FD_TABLE` or `ROAD_DZ_CARPET`). If there are multiple tables in a group, then the units must be set individually for each table.

Be aware that each specific Configurable Function has at most three sets of units, each associated with one of the three variable types (output, row, column). If the units are changed for one type, the change applies to all parameters associated with that type of variable.

As an example, consider the `ROAD_DZ` Configurable Function used to specify incremental road elevation as a function of longitudinal and lateral position, all with units of meters. Suppose that we want to see the data with vertical in inches and horizontal (station and lateral position) in feet. The following VS Commands are placed in the miscellaneous field on the **Run Control** screen, where they are processed after the road data have been loaded (with metric units):

```
SET_UNITS ROAD_DZ_OFFSET in
SET_UNITS_TABLE_ROW ROAD_DZ_CARPET ft
SET_UNITS_TABLE_COL ROAD_DZ_CARPET ft
```

The first command sets the output to have units of inches, and the other two set the column and row variables to have units of feet. Figure 13 shows part of an Echo file showing a dataset for `ROAD_DZ` made with two 1D tables, after the above `SET_UNITS` commands were applied.

In the Echo file, the `SET_UNITS` commands are written just before they are needed, in case the Echo file is used to repeat or continue the simulation. The first table and the `ROAD_DZ_OFFSET` parameter both have units of inches. The independent variables in both tables have units of feet, as do the offsets `SSTART_ROAD_DZ` and `L_START_ROAD_DZ`.

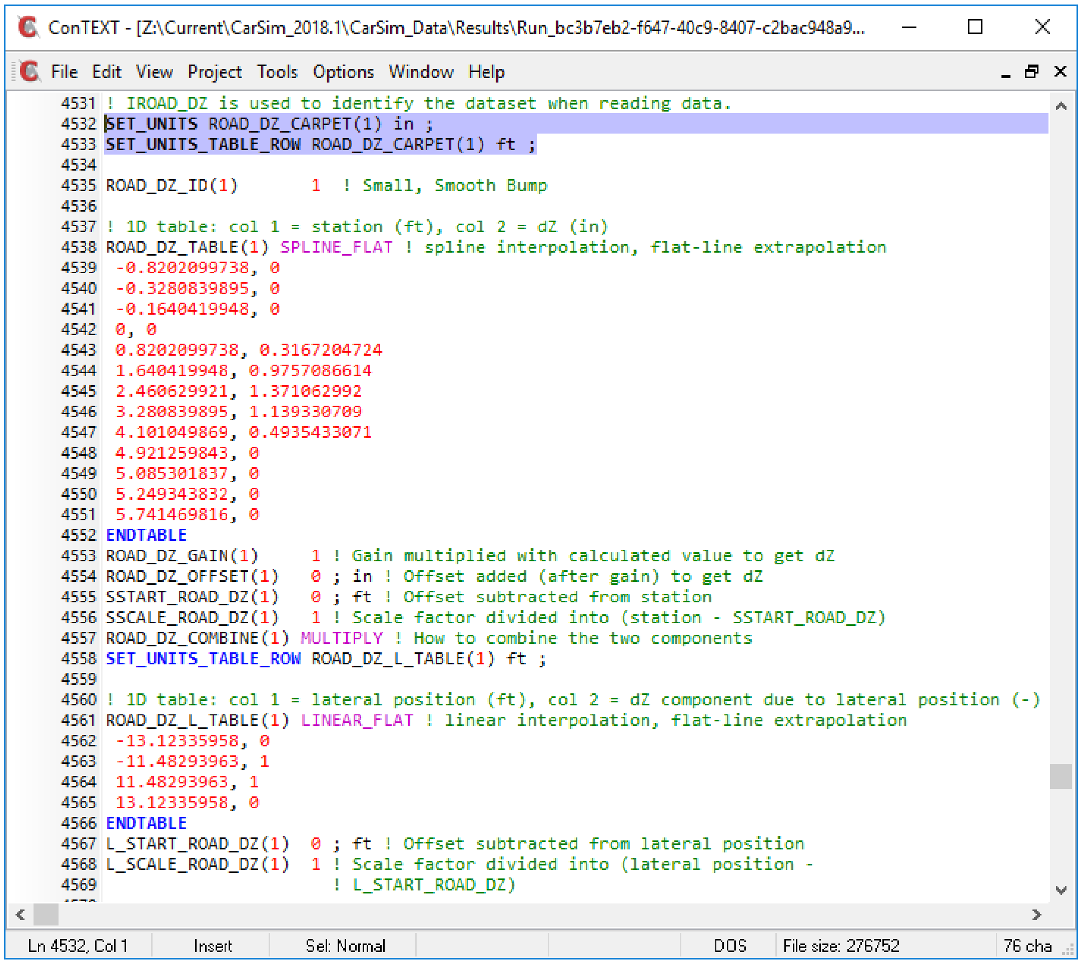
Because the dataset is made of two 1D tables that are multiplied (see `ROAD_DZ_COMBINE` on line 4557), the units for the second table `ROAD_DZ_L_TABLE` are automatically set to be dimensionless (line 4560).

Custom Equations for Configurable Functions

Some of the Configurable Functions allow an equation to be specified at runtime using a command with the suffix `_EQUATION`. The equation is processed and stored internally when the Parsfile is processed. The equation is then applied whenever the Configurable Function is used in the VS Math Model equations.

The equation can make use of any variables in the VS Math Model that have keywords, including parameters, output variable, Import variables, and state variables. These can be built-in variables or new variables defined at runtime with VS Commands. In addition to all of the regular keywords, the VS Math Model also recognizes two keywords that are specific to Configurable Functions:

1. The symbol "`X`" is recognized as the row variable for the table. In the friction example, this would be station S.
2. The symbol "`XCOL`" is recognized as the column variable for the table. In the friction example, this would be lateral position L. However, the `XCOL` variable will only be useful if the table supports 2D-lookup with `_CARPET` options. If the `_CARPET` options are not supported, then the `XCOL` keyword will use a default value of 0.

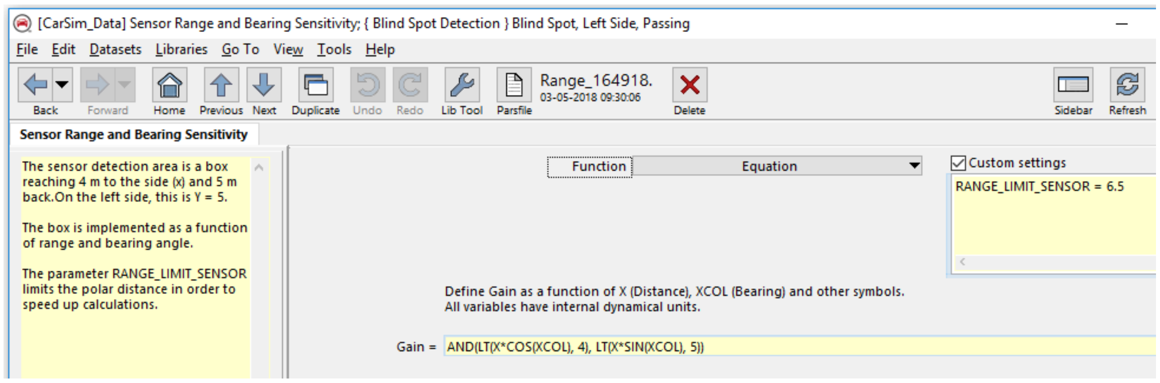

ConTEXT - [Z:\Current\CarSim_2018.1\CarSim_Data\Results\Run_bc3b7eb2-f647-40c9-8407-c2bac948a9...
File Edit View Project Tools Options Window Help
Ln 4532, Col 1
Insert
Sel: Normal
DOS
File size: 276752
76 cha

```
! IROAD_DZ is used to identify the dataset when reading data.
SET_UNITS ROAD_DZ_CARPET(1) in ;
SET_UNITS_TABLE_ROW ROAD_DZ_CARPET(1) ft ;

ROAD_DZ_ID(1)       1  ! Small, Smooth Bump

! 1D table: col 1 = station (ft), col 2 = dZ (in)
ROAD_DZ_TABLE(1) SPLINE_FLAT ! spline interpolation, flat-line extrapolation
 -0.8202099738, 0
 -0.3280839895, 0
 -0.1640419948, 0
 0, 0
 0.8202099738, 0.3167204724
 1.640419948, 0.9757086614
 2.460629921, 1.371062992
 3.280839895, 1.139330709
 4.101049869, 0.4935433071
 4.921259843, 0
 5.085301837, 0
 5.249343832, 0
 5.741469816, 0
ENDTABLE
ROAD_DZ_GAIN(1)     1 ! Gain multiplied with calculated value to get dZ
ROAD_DZ_OFFSET(1)   0 ; in ! Offset added (after gain) to get dZ
SSTART_ROAD_DZ(1)   0 ; ft ! Offset subtracted from station
SSCALE_ROAD_DZ(1)   1 ! Scale factor divided into (station - SSTART_ROAD_DZ)
ROAD_DZ_COMBINE(1) MULTIPLY ! How to combine the two components
SET_UNITS_TABLE_ROW ROAD_DZ_L_TABLE(1) ft ;

! 1D table: col 1 = lateral position (ft), col 2 = dZ component due to lateral position (-)
ROAD_DZ_L_TABLE(1) LINEAR_FLAT ! linear interpolation, flat-line extrapolation
 -13.12335958, 0
 -11.48293963, 1
 11.48293963, 1
 13.12335958, 0
ENDTABLE
L_START_ROAD_DZ(1)  0 ; ft ! Offset subtracted from lateral position
L_SCALE_ROAD_DZ(1)  1 ! Scale factor divided into (lateral position -
                      ! L_START_ROAD_DZ)
```

Figure 13. Part of Echo file showing units associated with a Configurable Function.

For example, the ADAS range and tracking sensors use a Configurable Functions `SENSOR_ANTANA_RANGE` that provides a detection gain based on range and bearing. It is typically set to 1 for combinations of range and bearing where a target is detected, or 0 when not. The sensor also has minimum and maximum bearing limits, and a range limit. The parameters alone define an area of detection shaped like a slice of pie, with the radius of the pie being the sensor range limit, and the angles of the two straight edges matching the bearing limits of the sensor. The Configurable Function can further limit the area of detection using a combination of bearing and range. For example, a sensor being used for blind-spot detection is set in one example to define a detection area shaped like a rectangle going 4m in the X direction of the sensor aim (lateral to the vehicle) and 5m back, in the Y direction of the sensor aim. An easy way to define this is with an equation (Figure 14).

Suppose you want to extend a model to add sensitivity to a new variable. For example, consider the Configurable Function used to relate force to compression rate in the damper in a VS vehicle model. In the example listing from an Echo file (see Figure 12, page 73), lines 1084 and 1085 indicate that this Configurable Function supports the Equation option.


[CarSim_Data] Sensor Range and Bearing Sensitivity; { Blind Spot Detection } Blind Spot, Left Side, Passing
File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile Delete Sidebar Refresh
Range_164918.
03-05-2018 09:30:06
Sensor Range and Bearing Sensitivity

The sensor detection area is a box reaching 4 m to the side (x) and 5 m back.On the left side, this is Y = 5.

The box is implemented as a function of range and bearing angle.

The parameter RANGE_LIMIT_SENSOR limits the polar distance in order to speed up calculations.

Function: Equation

☑ Custom settings

RANGE_LIMIT_SENSOR = 6.5

Define Gain as a function of X (Distance), XCOL (Bearing) and other symbols.
All variables have internal dynamical units.

Gain = AND(LT(X*COS(XCOL), 4), LT(X*SIN(XCOL), 5))

Figure 14. Equation used to define rectangular detection area for ADAS sensor.

Suppose you want to make the force dependent on both the compression rate and the compression, using an equation involving both variables. The root name for the damper force function is `"FD"`; use the keyword `FD_EQUATION(1,1)` to provide an equation for the damper on axle 1, side 1. In the equation, include variables using the symbols `X` (compression rate) and `CMPD_L1` (damper compression for axle 1, side 1).

The option to replace a Configurable Function with a new equation is not available for all Configurable Functions. If the option is not mentioned in the Echo file, then it is not available. The equation option is always disabled if the Configurable Function supports internal inverse calculations or derivative calculations. In this case, a potential workaround is to use an equation to generate a spreadsheet of tabular data that can be pasted into a dataset screen. The VS Math Model will use the tabular data, and support inverse or derivative calculation if needed.

Methods for Extending a Configurable Function

Given the many options for calculating a Configurable Function, combined with options for setting scalar values during an Event or with an equation added at runtime, there are many potential methods for extending the Configurable Functions during a run.

Using VS Commands

Here are some recommended methods that use VS Events:

1. If the calculated value should be a constant whose value is occasionally changed via Events, use the `_CONSTANT` keyword in Parsfiles linked to Events.

2. If the calculated value should be linearly related to the main independent variable (the row variable) with a coefficient that is mainly constant, but which is occasionally changed via Events, use the `_COEFFICIENT` keyword in Parsfiles linked to Events.

3. If the relationship between the function value and the independent variable(s) involves a distinct waveform that should be rescaled or offset occasionally via Events, use the `_GAIN`, `_OFFSET`, *start_xrow*, *scale_xrow*, *start_xcol*, and *scale_xcol* keywords in Parsfiles linked to Events.

4. If the function value should be sensitive to a variable other than the one or two independent variables that are built in, then an equation can be added at runtime.

a. If the `_EQUATION` option is supported for the function, then the symbolic equation can be provided using `X` and possibly `XCOL` as names for the built-in independent variables, as described in the previous section.

b. If the `_EQUATION` option is not supported for the function, and the other variables act to scale the value or add an offset, then use the `EQ_IN` VS Command to update the `_GAIN` or `_OFFSET` modifiers for the function with an equation involving the other variables.

   If the derivative or inverse is also calculated for the Configurable Function, be aware that there might be an error introduced with these modifications. For example, if the `_GAIN` value changes rapidly in response to another variable, that effect will not be included in the internal calculations for the derivative. (The internal calculations assume that `_GAIN` and `_OFFSET` are constant.)

c. If the `_EQUATION` option is not supported for the function, and the dependence on the other variable(s) is significant, then set the function to a constant using the `_CONSTANT` keyword and use the `EQ_IN` VS Command to update the constant dynamically.

   If the derivative or inverse is also calculated for the Configurable Function, this method will certainly introduce error into the auxiliary calculations. You will need to determine if the error is acceptable in return for the convenience of making a radical change to the model.

When adding equations, be aware of the distinction between internal SI units and user units. Any equations involving variables should be based on SI units (m, radians, m/s^2, etc.)

Extend Configurable Functions with Imported Variables

VS Math Models have many Import variables built in and ready to use for integrating the VS Math Model with external models from Simulink, LabVIEW, and other modeling software. The built-in variables cover common controls (steering, throttle, etc.) and major forces and moments (suspension springs and dampers, tire forces and moments, etc.). The built-in Import variables are documented in machine-generated text and spreadsheet files, as described in *VS Math Models Manual*.

In addition to these built-in options, it is possible to use the `DEFINE_IMPORT` command in combination with `EQ_IN` or `EQ_OUT` commands to create your own Import variables and use them to manipulate Configurable Functions.

Here are some suggested methods:

1. Always check the machine-generated documentation to make sure there isn't already an Import variable that does exactly what you want. From the lower-right corner of the **Run Control** screen of the VS Browser (e.g., `carsim.exe`) use the **View** button to view **Imports into math model** (either text or Excel). For example, the damper forces for all wheels are already installed and can be used as is.

2. If the Import variable does not exist, but a Configurable Function does exist that calculates the variable, then look at an Echo file (again, use the **View** button on the **Run Control**

screen). Read the comments (e.g., the part of the Echo file that lists the options, such as shown in Figure 12 (page 73) and Figure 13 (page 75)).

3. If the function allows the dependent variable to be represented with a constant, and there is no mention of a derivative function, then:

   a. Use the `DEFINE_IMPORT` command to define the variable that will be imported. E.g.,

   ```
   DEFINE_IMPORT IMP_MY_DEMO
   ```

   b. Specify that the function is a constant with some numerical default value. Suppose the Import variable for damper force didn't exist and we wanted to replace the force `FD` for the left-front wheel. To do this, specify the constant value either using the library screen for the function (if there is one), or enter a line in a miscellaneous field, such as:

   ```
   FD_CONSTANT(1,1) = 0;
   ```

   The value assigned to the constant here doesn't matter because it will be replaced using an equation that will be added in the next step.

Note When an input line begins with the name of a constant associated with a Configurable Function, it has two effects on the VS Math Model: (1) it sets the mode of the Configurable Function to use a constant, and (2) it sets the value for that constant. In this example, the name `FD_CONSTANT(1,1)` is used to accomplish the first effect (setting the mode). The numerical value used here (e.g., 0) serves only as a place holder to ensure that the line of input is properly processed by the VS Math Model.

   c. Add an equation to update the constant every timestep using the newly defined Import variable. E.g.,

   ```
   EQ_IN FD_CONSTANT(1,1) = IMP_MY_DEMO;
   ```

   The result is that every timestep a variable `IMP_MY_DEMO` is imported and used to update `FD_CONSTANT(1,1)`, which is applied internally in the VS Math Model.

Note This next option is described for the benefit of advanced users with some expertise in knowing how multibody equations are formulated and solved.

4. If the Echo file mentions a derivative function, then method 3 would be incomplete. It would properly update the dependent variable, but would not update any derivatives of that variable. For example, consider road wheel camber, related to suspension jounce (`Jnc_L1`, `Jnc_R1`, etc.) in independent suspensions in CarSim and TruckSim by a Configurable Function `CAMBER`. The derivative (∂camber/∂jounce) has a significant effect on jacking forces in the suspensions. Importing a variable such as camber is more

complicated, but possible. The main complexity is that two variables must be imported: (1) the variable of interest (e.g., camber), and (2) the partial derivative of that variable with respect to the independent variable in the Configurable Function (e.g., ∂ camber/∂jounce). Here is the basic method:

a. Use the `DEFINE_IMPORT` command to define the two variables that will be imported. E.g.,

   ```
   DEFINE_IMPORT IMP_MY_CAMBER
   DEFINE_IMPORT IMP_MY_CAMBER_DERIV
   ```

b. Specify that the function calculates the dependent variable using a linear coefficient. Specify the coefficient either using the library screen for the function (if there is one), or enter a line in a miscellaneous field, such as:

   ```
   CAMBER_COEFFICIENT(1,1) = 0;
   ```

   In this case, the value assigned to the coefficient doesn't matter (it will be replaced in the next step). The intended effect is that the Configurable Function is now using the linear coefficient mode of calculation, and will support internal calculation of the partial derivative.

c. Add an equation to update the coefficient every timestep using the newly defined Import variable for the partial derivative, e.g.,

   ```
   EQ_IN CAMBER_COEFFICIENT(1,1) = IMP_MY_CAMBER_DERIV;
   ```

   The result is that every timestep a variable `IMP_MY_CAMBER_DERIV` is imported and used to update `CAMBER_COEFFICIENT(1,1)`. This ensures that the partial derivative is calculated correctly.

d. Add an equation to update the offset for the function every timestep using both Import variables, and the current value of the independent variable, to account for the difference between the correct value and the value calculated with a linear coefficient. For this camber example, the independent variable for the left-front wheel is `Jnc_L1`, so the equation for the camber offset would be:

   ```
   EQ_IN CAMBER_OFFSET(1,1) = IMP_MY_CAMBER ...
                    - Jnc_L1*IMP_MY_CAMBER_DERIV;
   ```

   The two equations (c. and d.) set both the camber and the derivative to match the imported values.

Making New Configurable Functions

A VS Command is available for creating a new set of Configurable Functions.

DEFINE_TABLE

```
DEFINE_TABLE  name n
```

Use this command to define a set of Configurable Functions that can then be used in other VS Commands. Functions added with this command support the same options as the built-in Configurable Functions (they can be 1D tables, 2D tables, constants, or coefficients), and can be used in VS Command equations the same way as the built-in Configurable Functions.

The argument *name* is the name of the function that should be used in equations, and the argument *n* is the number of independent datasets that can be accessed with the new Configurable Function. If *n* is greater than 1, then a dataset ID parameter is automatically installed with a name obtained by prefixing "I" to *name*. For example, consider the command:

```
define_table my_func 2
```

This command installs a function `MY_FUNC` into the VS Math Model, with two independent datasets. It also installed an index parameter `IMY_FUNC` to set the context for reading the two datasets.

When specifying the data, you can also specify the interpolation method. For example, here is a simple specification of the data for two Configurable Functions:

```
imy_func 1
my_func_table linear
0,0
1,0
2,1
endtable

imy_func 2
my_func_coefficient 2.1
```

Configurable Function data can be specified anywhere in Parsfiles sent to the VS Math Model, with the only restriction being that the `DEFINE_TABLE` command must be read before the data for the new Configurable Function. The **Generic Table** screen of the VS Browser is usually the most convenient screen for specifying tabular data using new functions installed with `DEFINE_TABLE`.

You might define output variables that use the new functions with commands such as:

```
define_output my_out1 = my_func(0,t,1); units = m;
define_output my_out2 = my_func(0,t,2); units = m;
```

As with other Configurable Functions, the first argument to the function is ignored unless you have provided 2D data.

In general, table entries cannot be updated after the table has been defined (either with a `DEFINE_TABLE` or by loading it from a file). The exception is tables (not carpets) of type `STEP` which can be assigned to after definition:

```
my_out(0,1,1) = t+25
```

This is useful when a table of input values needs to be provided a Python call within the VS framework.

When you install a new function with DEFINE_TABLE, the VS Command automatically installs keywords based on the new name with standard suffixes: _TABLE, _CARPET _CONSTANT, _COEFICIENT, _GAIN, _OFFSET, _START_X, and _SCALE_X.

All Configurable Functions installed with DEFINE_TABLE are initially set up with user display units of '-' (dimensionless) for the output variable and both independent variables. Units can be set as with other Configurable functions, using the SET_UNITS commands described earlier (page 33). Recall that when using the SET_UNITS command, the keyword for the output of a Configurable Function must include the suffix _TABLE or _CARPET.

Alert The root name should be the name of the function should not include a table suffix such as _TABLE or _CARPET. If the name includes a suffix, an error message is generated.

A feature of some built-in 2D carpet tables is that they can be decomposed into two 1D table functions whose outputs are added or multiplied. For example, the function SPEED_TARGET provides target speed as a function of both time and longitudinal distance (station) along a path, using a dataset with the keyword SPEED_TARGET_CARPET. Alternatively, target speed data may be specified with two 1D tables with the keyword SPEED_TARGET_TABLE (speed v. time) and SPEED_TARGET_S_TABLE (speed v. station). With the option, the target speed is taken using two internal 1D functions and either adding or multiplying the results. This capability is not supported for 2D Carpet functions created with the DEFINE_TABLE command. If the intent is to have two 1D functions added or multiplied, DEFINE_TABLE may be used to create the individual functions, and then combine the results by addition or multiplication in VS Command equations.

Using External Files

VS Math Models can obtain data from a VS or ERD file and copy the information into a Configurable Function table. One application is to take simulation output files from VS Math Models in CarSim, TruckSim, or BikeSim, and copy time histories of forces, moments, and motions into SuspensionSim for study of movements and Joint forces and moments.

Another application is to use data from other sources to build tables that can be loaded directly from a separate binary file, rather than require entry into a VS screen or text Parsfile. This is particularly useful for datasets which are very large, such as an elevation map for a large road surface.

Accessing Tabular Data from VS or ERD Binary Files

Table 19 lists commands for loading and copying data from binary VS or ERD files.

Note These commands won't work with output CSV files because they do not contain enough information to do unit conversion.

Table 19. VS Commands to access data from a VS/ERD File.

Command	**Description**
`LOAD_TABLE_FILE` *filename*	Load data from a VS or ERD file into memory
`FILE_TO_TABLE` *table chx chy type*	Copy X and Y data from a loaded file to a table
`FILE_TO_CARPET` *table*	Copy data from a loaded file for values in a carpet
`ECHO_LOADED_TABLE` *table*	Display loaded data in the Echo file
`SAVE_TABLE` *table filename*	Save a table to file in .vstb format.

`LOAD_TABLE_FILE`

`LOAD_TABLE_FILE` *filename*

When this command is encountered, it opens the file named *filename* as a VS or ERD header file. If the specified file exists, the VS Math Model checks the format, extracts the names of the sampled variables, extracts the units for the variables, and then loads all the data from the associated binary file into memory.

The contents of one file are held in memory. If the command is encountered again, memory is cleared and the new file is processed and loaded.

Once a file has been loaded into memory, data can be copied into tables or carpets (2D tables) using the following commands.

If the intent is to move data from the file to a 1D tables, then the source can be any VS or ERD file, including those made as outputs of simulations with VS Math Models. However, if the intent is to obtain data for 2D (carpet) tables, then the file must be a VS or ERD file created specifically for this purpose. The next subsection, **Create Data Files for Tables**, provides some instructions for doing this.

`FILE_TO_TABLE`

`FILE_TO_TABLE` *table chx chy type*

Table is a keyword for 1D tabular data for one of the Configurable Functions (e.g., for SuspensionSim, `SS_TRANSLATE_TABLE(2)`). If the keyword *table* does not include a specific number, then the current values of the system index parameter for that Configurable function is used. Beside the various built-in Configurable Functions, it is also possible to define new functions at runtime with the VS Commands `DEFINE_TABLE`.

The parameters *chx* and *chy* are names of output variables that will be extracted from the VS/ERD file and used to set the X and Y values in the table. For example, if the vertical position of the left wheel center in a SuspensionSim model will be controlled to match motions produced by CarSim, then *Chx* would be `Time` and *Chy* would be `Jnc_L1` (Jounce for the left wheel on axle 1). As another example, if a Load is applied at the center of tire contact in the Y direction based on a time history of lateral tire force output from CarSim, then *Chy* would be `Fy_L1`.

The parameter *type* designates the type of table to be created. Allowable types are `LINEAR_LOOP`, `LINEAR`, `SPLINE`, `SPLINE_FLAT`, and `LINEAR_FLAT` (default). If no *type* is provided, then the default (`LINEAR_FLAT`) will be used.

FILE_TO_CARPET

`FILE_TO_CARPET` *table*

Table is a keyword for tabular data for one of the 2D Configurable Functions (e.g., `ROAD_DZ_CARPET`). If the keyword *table* is not a 2D table or 2D table group, an error will occur and the load will not proceed. This command will import all channels in the file, so it requires that the binary file be created for this purpose.

This command has been used to import 3D ground elevation and friction data for the Mcity ADAS examples included in recent versions of BikeSim, CarSim, and TruckSim.

ECHO_LOADED_TABLE

`ECHO_LOADED_TABLE` *table*

Table is a keyword for tabular data for a Configurable Functions that has been loaded with either `FILE_TO_TABLE` or `FILE_TO_CARPET`. If this command is used, the Echo File generated by CarSim will show the data extracted from the VS/ERD file in the same manner as if the data were provided with Parsfiles. Otherwise, the loaded data will not be written out. Instead, two commands are written: (1) `LOAD_TABLE_FILE` (with the pathname of the file), and (2) the command `FILE_TO_TABLE` or `FILE_TO_CARPET`, with the appropriate arguments.

SAVE_TABLE

`SAVE_TABLE` *table filename*

Table is a keyword for tabular data for a Configurable Function (e.g. `ROAD_DZ_CARPET` or `LTARG_TABLE`). The table or carpet referenced is then saved in table format (`.vstb`) as a new file indicated by *Filename*. The saved file can then be loaded (perhaps in a different simulation) using the `LOAD_TABLE_FILE` command.

Create Data Files for Tables

A VS/ERD output file can be used to load 1D tables. In order to import 2D table, it is usually necessary to create custom VS or ERD file for this purpose. This is most easily done with the `csv_to_erd` utility. A csv (comma separated values) file is created with the following format for a table:

```
Chan1Name,  Chan2Name,  Chan3Name,  …
Data11,     Data12,     Data13,     …
Data21,     Data22,     Data23,     …
Data31,     Data32,     Data33,     …
```

Run the utility on the .csv file, using a nominal time value (it is not used) and the units used by the channels (or most of the channels):

```
csv_to_erd.exe  myfile.csv 0.05 mm
```

This will result in two files, `myfile.erd` and `myfile.bin`. If one or more channels does not have the correct units, the .erd file can be manually edited to correct the unit type.

2D tables pose special challenges because in addition to the grid of data that must be stored, separate 1D arrays of indices (X and Y) must be stored.

For a 2D table (carpet) the following format is used:

```
Unused,     Chan1Name,  Chan2Name,  …
Unused,     X1Value,    X2Value,    …
Y1Value,    Data11,     Data12,     …
Y2Value,    Data21,     Data22,     …
Y3Value,    Data31,     Data32,     …
```

Run the utility on the .csv file, again using a nominal time value and the units used by 2-D grid of values:

```
csv_to_erd.exe  my2dfile.csv 0.05 mm
```

The two files resulting, `my2dfile.erd` and `my2dfile.bin` can then be used directly or converted into .vs format with the VS/ERD File Utility (available under 'Tools').

Using the Embedded Python Utility

When running under Windows OS or non-RT Linux, VS Math Models can load Python and run Python code from within the model. VS parameters and variables (numbers) can be placed in a table for use as input to the Python routine, and another table can return outputs (also numbers) back for use by other VS Commands. A text string is also available as input to the Python module, and the interpreter itself has full capability for reading and writing files, accessing network resources, etc. Using the utility is further explained in *Extending a Model with Embedded Python*.

The motivation for adding the Python Utility is that the user now has access to a fully functional programming language to extend model behavior, without having to use a third-party package such as Simulink or compiled code such as a C wrapper.

Accessing VS Embedded Python Utility

Table 20 lists commands for loading Python and applying routines form within the VS Math Model.

Table 20. VS Commands to access Embedded Python Utility.

Command	**Description**
`OPT_ENABLE_PYTHON`	Enable access to the Python Utility.
`RUN_PYTHON_STRING` *string*	Execute string as Python call. (Useful for `import` statements and other simple Python commands.)
`RUN_PYTHON_PROG` *function signal in_table out_table*	VS Command that calls the Python Utility.
`PYTHON`*(cond,function,signal,in_table,out_table)*	Call Python as an equation (`EQ_XXX`)
`SET_PYTHON_INSTALL_32`, `SET_PYTHON_INSTALL_64`	Reference external Python to be used instead of installation provided with product.

OPT_ENABLE_PYTHON

`OPT_ENABLE_PYTHON` = *1*

`OPT_ENABLE_PYTHON` = *0*

The embedded Python option is not loaded by default, so it must be explicitly enabled for the Python calls to work. If not enabled, then the Python calls will be ignored. As a result, this command needs to be present in any dataset using the embedded Python interpreter.

RUN_PYTHON_STRING

`RUN_PYTHON_STRING` *string*

String is just a literal string that can be passed to the Embedded Python Utility and executed directly, as if it was typed after a Python interpreter prompt. This command is most useful when setting up the Utility for operation, such as using the `import` command to load a module for use by the system.

This command could also be used to expand the search path for a Python session, but the system needs basic access to the "Lib" directory to function. The interpreter needs immediate access to the Lib directory to start up properly.

RUN_PYTHON_PROG

`RUN_PYTHON_PROG` *function, signal, in_table, out_table*

Function is a function known to the Python Utility. For example, if the user imported a steering module **steering.py** which contains the function `steer1`, then they may access `steering.steer1` as a function. *Signal* is just another string which can function as an optional input to the Python function call. *In_table* and *out_table* are step tables used to hold input and output data, respectively. Before the call, parameters or variables need to be loaded into the input table. After the call, new values in the output table can be accessed and used to update VS variables if desired.

PYTHON( )

```
PYTHON(cond, function, signal, in_table, out_table)
```

This is a function to access the Python utility for use with the VS Equation Commands (`EQ_INIT`, `EQ_DYN`, etc.). As such, it is not a VS Command per se, but a function in support of the VS Equation system. The function has the same inputs as `RUN_PYTHON_PROG` with an additional input, `cond`. When `cond` is greater than 0, then the routine is executed, otherwise it is not. The remaining inputs act as previously described.

SET_PYTHON_INSTALL_32, SET_PYTHON_INSTALL_64

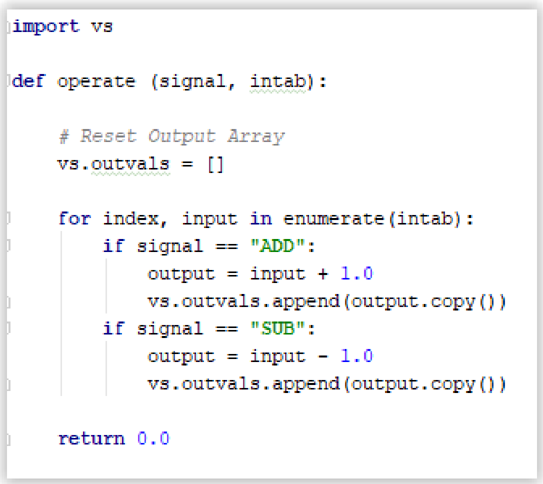
`SET_PYTHON_INSTALL_32` *"C:\Program Files (x86)\Python310\python310.dll"*

If the user wishes to use their own Python installation to run with the VS solver, they can use one of these set commands. For Windows one can just reference the 32-bit or 64-bit .dll of the installed Python. For Linux users, the set command should reference the shared library for the Python install (e.g., libpython3.10.so.1.0) The shared file is often not provided with a standard Linux Python install, so it may have to be built. The shared library can be accessed from its nominal location in the installed '`lib`' directory, a location it shares with the '`pkgconfig`' and '`pythonx.y`' subdirectories, as in: "`/usr/jsmith/py_install/lib/libpython3.10.so.1.0`".

Each version of Carsim/Trucksim/Bikesim has a specific version of Embedded Python associated with their solvers. For Version 2022.1`, this Python version is 3.10.2. Ideally, an external Python to be run with the solvers should be the same version, however, other versions of Python close to the nominal version may very well work as well (but cannot be guaranteed to do so).

Format of Python Routine used by Utility

The Python routines that are to be used by the Embedded Python Utility need to be in a particular format to function. This is so that they can access the inputs provided to them and return the outputs to the solver. A simple example that explains the format is shown below (Figure 15).



```
import vs

def operate (signal, intab):

    # Reset Output Array
    vs.outvals = []

    for index, input in enumerate(intab):
        if signal == "ADD":
            output = input + 1.0
            vs.outvals.append(output.copy())
        if signal == "SUB":
            output = input - 1.0
            vs.outvals.append(output.copy())

    return 0.0
```

Figure 15. Example of simple Python routine that can be run by the utility.

In this example, we see that the signal string is the first parameter of the call. The second parameter is a list of floating-point numbers, which represents the inputs sent to the function. In this example 1.0 is either added or subtracted from the input parameters, with the results placed in the output. The output is created by accessing the output list `vs.outvals`. To get access to this list, the routine should import `vs`, which is a module that is part of the utility.

VS Command Functions directly accessible from Python Routine

The vs module also contains several functions which can access values from a simulation. Individual values can also be accessed or updated. Certain table entries can also be updated as well. Further explanation on how to access simulation variables can be found in *Extending a Model with Embedded Python*. Other functions related to VS paths and roads (for solvers that include paths and roads) are also supported as is a `system()` operating system call and the VS statement call, which allows for other VS commands to be called from the embedded Python. Table 21 lists support functions implemented in Python in the vs module for accessing variables and tables.

Table 21. VS Functions accessible with the Embedded Python Utility.

Command	**Description**
`vs.statement(key, buffer, stop)`	Execute string as VS command.
`vs.system(string)`	Execute string as system shell call.
`vs.getval(varstring)`	Return value of VS variable named by string.
`vs.getvar(varstring)`	Return variable object of VS variable named by string.
`vs.gettype(varstring)`	Return type of VS variable named by string.
`vs.getunits(varstring)`	Return units of VS variable named by string.
`vs.setval(varstring, value)`	Set value of VS variable named by string.
`vs.gettab(tabstring)`	Return table object of VS table named by string.
`vs.gettabtype(tabstring)`	Return table type of VS table named by string.
`vs.gettabval(tabstring, ind1, ind2, ind3)`	Return table entry value of VS table named by string and indices.
`vs.settabval(tabstring, ind1, ind2, ind3, value)`	Set value of VS table named by string and indices.
`vs.print(string)`	Diagnostic to output information from a Python routine. Can pause simulation.
`vs.debug(value)`	Diagnostic to print out more information (value = 1). Can be turned off (value = 0).
`vs.var(“VAR_NAME”).value()`	Using the Python variable object, get or set the VS variable’s value.
`vs.var(“VAR_NAME”).setvalue(value)`	
`vs.tab(“TAB_NAME”).entry(p1, p2, p3).value()`	Using the Python table object, get or set the entry value.
`vs.tab(“TAB_NAME”).entry(p1, p2, p3).setvalue(value)`	
`vs.tab(“TAB_NAME”).entry(p1, p2, p3).defvalue()`	Using the Python table object, get the original value used in table definition.
`vs.tab(“TAB_NAME”).type()`	Using the Python table object, get the table type.
`vs.tab(“TAB_NAME”).num_rows()`	Using the Python table object, get the number of rows.
`vs.tab(“TAB_NAME”).num_cols()`	Using the Python table object, get the number of columns.
`vs.tab(“TAB_NAME”).num_index()`	Using the Python table object, get the number of data sets.

6. The Simulation Process

A VS simulation run on Windows occurs in five major steps:

1. Load the selected VS Solver library into memory.
2. Construct the VS Math Model and initialize.
3. Run the simulated test.
4. Terminate the run.
5. Unload the VS Solver library.

The full process has many steps that involve interactions between various factors:

- how variables are calculated using equations built into the VS Math Model,
- how information is obtained from input files,
- how variables are exchanged with external software such as Simulink,
- how equations are processed that are added at runtime with VS Commands, and
- how externally defined functions that might be installed using the API are applied.

The *VS API Manual* has the definitive description of the simulation process that includes all of the above factors; the *VS Math Model Manual* has a simplified description that excludes VS Commands and API interactions.

This chapter describes the process for basic use of the VS Math Model, including potential extensions made with VS Commands.

The following subsections describe the behavior of the VS Math Model after it has been loaded into memory, up until the run is terminated and the solver can be unloaded from Windows.

Initialize

Before a simulation run can be started, the solver determines the model layout, allocates memory, sets default values, reads input files, and prepares for the run. Figure 16 shows a schematic timeline for the initialization steps that are described below. In this figure, actions taken by the VS Math Model using built-in code are indicated with blue and gray blocks; equations added with VS Commands are indicated with tan blocks.

1. Start up and read input files:
    a. Read a Simfile to obtain names of input and output files; allocate memory, create an internal database, and define some core system variables.
    b. Begin reading the input Parsfile and obtain the model layout. From that information, activate modules based on the layout, extend the database with model parameters and variables, and continue to read from input Parsfiles. After the Parsfiles have been read, determine how many variables will be imported and exported, and allocate the Import and Export arrays accordingly.

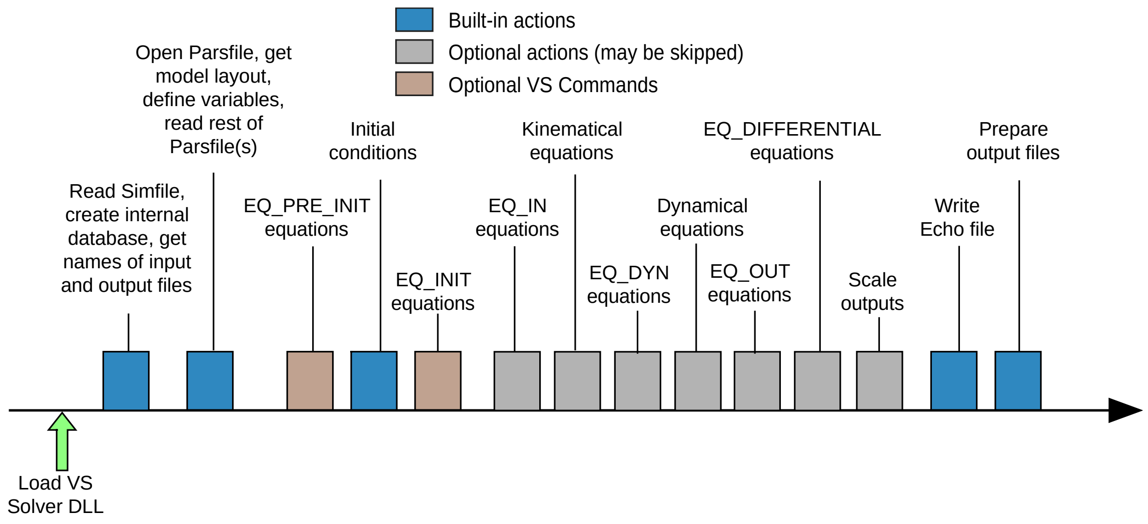


Figure 16. Schematic timeline for the VS Math Model initialization.

2. Always perform initialization calculations that depend on the values read in step 1b:

    a. Process equations provided with the VS Command `EQ_PRE_INIT`.

    b. Calculate initial conditions for built-in variables.

    c. Process equations provided with the VS Command `EQ_INIT`.

3. Optionally apply the model equations used each timestep, to prepare some parameters with calculated values that will appear in the Echo file, and output variables that might be used by external software such as Simulink. This step can be disabled with the parameter `OPT_SKIP_INIT_DYN`.

    a. Apply equations provided at runtime with the VS Command `EQ_IN`.

    b. Perform built-in kinematical calculations.

    c. Apply equations provided at runtime with the VS Command `EQ_DYN`.

    d. Perform built-in dynamical calculations.

    e. Apply equations provided at runtime with the VS Commands `EQ_OUT` and `EQ_DIFFERENTIAL`.

    f. Convert all potential outputs from dynamical units to user units.

4. Write an Echo file that lists all the model parameters, Configurable Functions, and VS Commands. The values are labeled and shown with user units.

5. Prepare to record output variables during the simulation run:

    a. Write the text ERD or VS header file with information about the variables to be recorded (names, units, etc.).

    b. If values will be written to a binary file during the run, then create the binary file. If values will be written to file when the run finishes, then allocate memory for a buffer in which to store the values as the run progresses. (The option to store results in memory is selected with the parameter `OPT_BUFFER_WRITE`).

At the end of step 5, the model is still not fully initialized. A few more steps are taken during the first set of operations at a major step, as described in the next subsection.

Run

The simulation is made by incrementing time by a small step ΔT (parameter `TSTEP`) and performing calculations to update the state variables and output variables for the new time T. The following actions are performed at each timestep.

Operations at the Full Step (Conventional)

Figure 17 shows a schematic timeline for the main steps of the specified timestep. In the figure, ΔT is `TSTEP` for the Euler and AB-2 integration methods; it is `TSTEP/2` for the AM and RK methods.

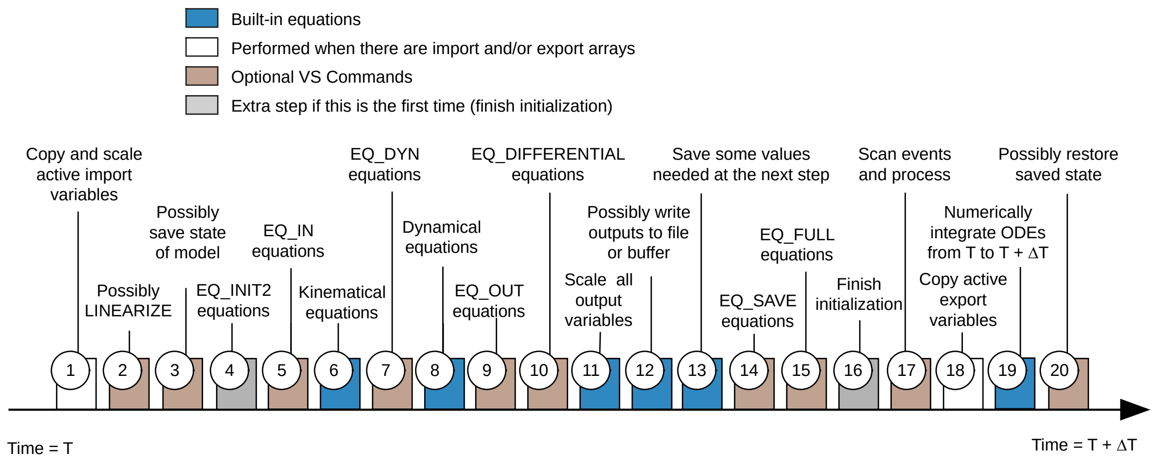

Built-in equations
Performed when there are import and/or export arrays
Optional VS Commands
Extra step if this is the first time (finish initialization)
Copy and scale active import variables
Possibly LINEARIZE
Possibly save state of model
EQ_INIT2 equations
EQ_IN equations
Kinematical equations
EQ_DYN equations
Dynamical equations
EQ_OUT equations
EQ_DIFFERENTIAL equations
Scale all output variables
Possibly write outputs to file or buffer
Save some values needed at the next step
EQ_SAVE equations
EQ_FULL equations
Finish initialization
Scan events and process
Copy active export variables
Numerically integrate ODEs from T to T + ΔT
Possibly restore saved state
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Time = T
Time = T + ΔT

Figure 17. Schematic timeline for the main timestep, using the conventional sequence.

The following actions are performed at each timestep `TSTEP`.

1. If there is an external workspace such as Simulink, the Import variables are copied from that workspace and converted from user units to internal dynamical units.
2. If requested via the command `LINEARIZE`, perform the linearization calculations, and generate an output M-file.
3. Check for requests to save the current state of the entire VS Math Model. (These requests are generated from the use of the VS Commands `SAVE_STATE` and `START_SAVE_TIMER`.) If there is a request, then allocate memory and save the state together with the current time.
4. If this is the first timestep, process equations provided with the VS Command `EQ_INIT2`.
5. Process equations provided with the VS Command `EQ_IN`.
6. Apply built-in kinematical equations. These include derivatives of generalized coordinates and output variables calculated from state variables.
7. Process equations provided with the VS Command `EQ_DYN`.

(8) Apply built-in dynamical equations. These include derivatives of generalized speeds and any output variables that have not already been calculated.

(9) Process equations provided with the VS Command `EQ_OUT`.

(10) Process equations provided with the VS Command `EQ_DIFFERENTIAL`. The results are saved in the internal array of derivatives of state variables.

(11) Convert all potential outputs from internal SI units to user units.

(12) Determine if the output values should be saved for this step. If so, write output variables to file or save in memory for later writing.

(13) If the run is not finishing, save values of some variables for use in the next timestep. These are used for model parts that need to compare current values with old values, e.g., to determine reversals.

(14) Process equations provided with the VS Command `EQ_SAVE`.

(15) Process equations provided with the VS Command `EQ_FULL_STEP`.

(16) If this is the first timestep, finish the initialization by writing outputs to file or saving in a buffer for later writing, and initialize counters for VS Events and writing to file.

(17) Scan a list of potential VS Events to see if any Events have occurred. If an Event occurred, either terminate the run or obtain new data and continue:

   a. If no Parsfile was specified when the Event was defined, then terminate the run.

   b. If a Parsfile name was associated with the Event, then the program goes back to step 1b to read the new Parsfile. It then continues from there.

(18) If there is an external workspace such as Simulink, copy output variables into an array that is shared with the external software.

(19) Numerically integrate the derivatives of the state variables. If the integration method is Euler or AB-2, the values are updated to time `T + TSTEP`; for the AM and RK methods they are updated to time `T + TSTEP/2`. (For details about the available numerical integration methods, please see the tech memo *Numerical Integration Methods in VS Math Models*.)

(20) Check for requests to restore a previous state, generated with the VS Command `RESTORE_STATE`. If there is a request and the state was saved near the requested time, then restore the state variables and the simulation time T that was associated with the saved state.

Operations at the Half Step

The VS Math Models support several numerical integration methods, as described the tech memo *Numerical Integration Methods in VS Math Models*. The method is set at runtime with the parameter `OPT_INT_METHOD`. Four of the methods require breaking the integration into two steps: the first goes from `T` to `T+TSTEP/2`, and the second continues to `T+TSTEP`. Figure 18 shows the timeline for the half-step calculations. In comparing this to Figure 17, it is clear that many of the steps applied at the major step are skipped at the half step. Referencing the numbered items in the previous subsection, the deleted steps are:

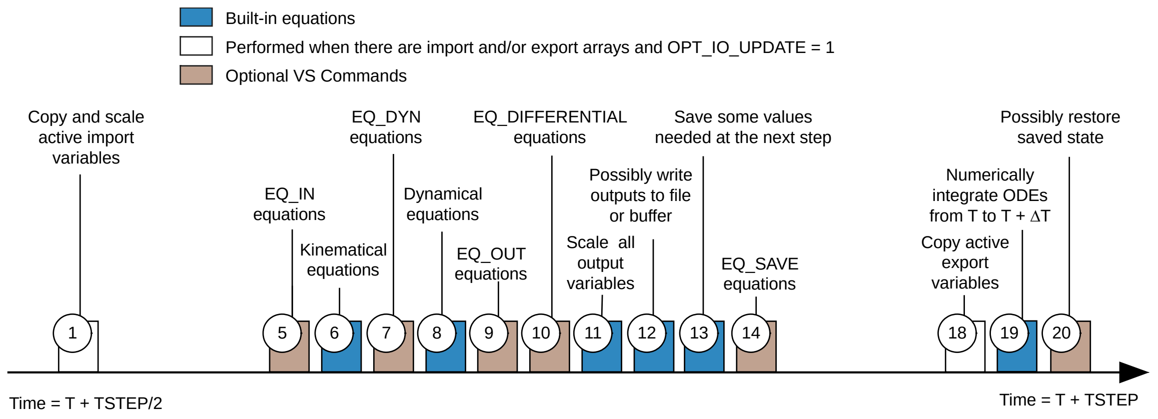

Built-in equations
Performed when there are import and/or export arrays and OPT_IO_UPDATE = 1
Optional VS Commands
Copy and scale active import variables
EQ_IN equations
Kinematical equations
EQ_DYN equations
Dynamical equations
EQ_OUT equations
EQ_DIFFERENTIAL equations
Scale all output variables
Possibly write outputs to file or buffer
Save some values needed at the next step
EQ_SAVE equations
Copy active export variables
Numerically integrate ODEs from T to T + ΔT
Possibly restore saved state
Time = T + TSTEP/2
Time = T + TSTEP

Figure 18. Schematic timeline for actions taken at the half-step.

- If the parameter `OPT_IO_UPDATE` is zero, skip steps ① (import) and ⑱ (export).
- Always skip steps ②, ③, ④, ⑮, ⑯, and ⑰.

If the integration method is Euler or AB-2, there are no operations at the half step.

Operations at the Full Step (Synchronized)

An alternative timing sequence is used internally when the system parameter `OPT_IO_SYNC_FM` is assigned a nonzero value. When this parameter is nonzero, forces and moments calculated externally are sychronized with the rest of the model by using a different calculation sequence for time T (Figure 19).

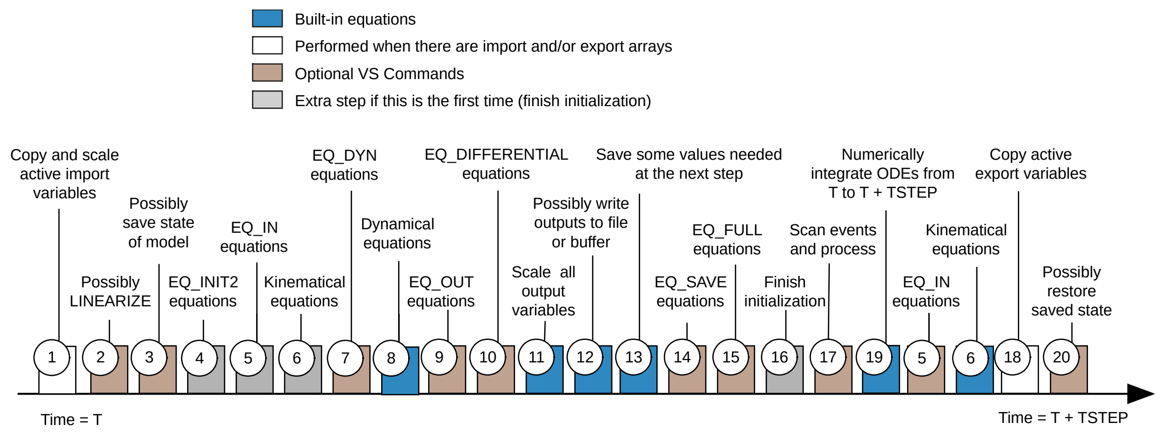

Built-in equations
Performed when there are import and/or export arrays
Optional VS Commands
Extra step if this is the first time (finish initialization)
Copy and scale active import variables
Possibly LINEARIZE
Possibly save state of model
EQ_INIT2 equations
EQ_IN equations
Kinematical equations
EQ_DYN equations
Dynamical equations
EQ_OUT equations
EQ_DIFFERENTIAL equations
Scale all output variables
Possibly write outputs to file or buffer
Save some values needed at the next step
EQ_SAVE equations
EQ_FULL equations
Finish initialization
Scan events and process
Numerically integrate ODEs from T to T + TSTEP
EQ_IN equations
Kinematical equations
Copy active export variables
Possibly restore saved state
Time = T
Time = T + TSTEP

Figure 19. Schematic timeline for the main timestep, using the "synchronized" sequence.

With this sequence, the steps for applying the user-defined `EQ_IN` equations ⑤ and kinematical equations ⑥ are shown twice. If this is the first-timestep, they are applied at the start of the timestep immediately after the import of external variables as the final step in the initialization. After that first step, they are not applied at the start, because the values of the kinematical variables were done the previous step. Instead, after the import of external variables ① and optional Linearize ② and Save options ③, the calculations resume with the dynamical equations (⑦, ⑧). Going from the dynamical equations to the scanning for Events ⑰, the sequence of actions is the same as described earlier (Figure 17, page 90). However, in this version, the next step is to calculate all ODE state variables for the next timestep ⑲. After the state variables are known for time `T + TSTEP`, the kinematical equations are applied (⑤ and ⑥). Activated outputs are now copied to the Export array ⑱.

With this approach, the variables calculated in the kinematical functions apply for time `T` + `TSTEP`, while all other variables (forces, accelerations, miscellaneous outputs) have values calculated for time `T`. The next timestep, forces and moments imported from the external model will be synchronized with the new time `T` + `TSTEP`.

Terminate

When the run ends, the solver cleans up and shuts down.

1. Process equations provided with the command `EQ_END`.
2. Set parameter values for the possibility that a future run will continue from where this one stopped. For example, set the start time to the current time, and turn off some initialization options to allow the VS Math Model state variables to be left intact.
3. Finish the writing of outputs:
    a. If variables were saved in memory (e.g., for RT operation), write to file.
    b. Close the file where output variables were written.
4. Optionally write an End Parsfile that can be read as input to continue the run later.
5. Free memory that was explicitly allocated during the initialization and the run.

7. More Information

This document has described how a VS Math Model can be extended using VS Commands, building on the more basic information provided in the *VS Math Models* manual.

Some other documents provide additional useful information about VS Math Models:

- *VS Visualizer Reference Manual* describes the interactive use of VS Visualizer. It includes an appendix that describes the contents of the ERD and BIN files created by the VS Math Models.
- *Numerical Integration Methods in VS Math Models* is a tech memo that describes the numerical integration methods in detail and includes a discussion of more advanced topics related to numerical integration.
- *System Parameters in VS Math Models* describes system-level parameters that are available in all VS Math Models, such as timestep, starting and stopping conditions, etc.
- *VS API: Accessing and Extending VS Math Model Programs* describes how VS Math Models can be run under other programs written in C or other programming languages. This document is included in the VS SDK packages.

Appendix: Deprecated Functions

The earliest versions of VS Commands had limited syntax. Table 22 lists functions that can be used in VS Expressions, but which are less convenient than other options.

Table 22. Deprecated functions in VS symbolic expressions.

Function	**Instead, Use**	**Description**
`ADD`(*x*, *y*)	x + y	*x* + *y*
`DIV`(*x*, *y*)	*x*/*y*	*x*/*y*
`MUL`(*x*, *y*)	*x***y*	*x***y*
`FABS`(*x*)	`ABS`(*x*)	Absolute value. `ABS` is also valid with VS Visualizer.
`FMOD`(*x*, *y*)	*x* % *y*	Remainder of *x*/*y* (with range check)
`NINT`(*x*)	`INT`(*x*)	Nearest integer to *x*. `INT` is also valid with VS Visualizer.
`POW`(*x*, *y*)	*x*^*y*	x^y (with range check on x if y is a fraction)
`POWER`(*x*, *y*)		
`SUB`(*x*, *y*)	*x*−*y*	*x*–*y*
`IF_GT_0_THEN`(*x*, *y*, *z*)	`IF` (*x*>0, *y*, *z*)	If *x* > 0 Then *y* Else *z*
`IF_NOT_0_THEN`(*x*, *y*, *z*)	`IF` (*x*, *y*, *z*)	If $x \neq 0$ Then *y* Else *z*

The two `IF_` functions can be replaced with the `IF` special function. Unlike `IF`, the two `IF_` functions in Table 22 always evaluate all three arguments, sometimes causing errors for illegal operations such as dividing by zero.

Two of the deprecated functions (`FABS` and `NINT`) are simply alternate names for functions listed in Table 2 (`ABS` and `INT`, respectively), which are also recognized by VS Visualizer.

Table 23 lists deprecated Boolean functions that can all be replaced with logical operators that are more compact, easier to read, and are supported by VS Visualizer.

Table 23. Deprecated Boolean functions supported in VS symbolic expressions.

Function	**Instead, Use**	**Description**
`AND`(x, y)	x & y	Return 1.0 if both x and y are not zero; otherwise return 0.
`OR`(x, y)	x | y	Return 1.0 if either x and y are not zero; otherwise return 0.
`GT`(x, y)	$x > y$	Return 1.0 if $x > y$; otherwise return 0.
`LT`(x, y)	$x < y$	Return 1.0 if $x < y$; otherwise return 0.
`GE`(x, y)	x >= y	Return 1.0 if $x \geq y$; otherwise return 0.
`LE`(x, y)	x <= y	Return 1.0 if $x \leq y$; otherwise return 0.
`EQ`(x, y)	x == y	Return 1.0 if x is exactly equal to y; otherwise return 0.
`NE`(x, y)	x ~= y	Return 1.0 if x is not exactly equal to y; otherwise return 0.
`NOT`(x)	~x	Return 1.0 if x is zero; otherwise return 0.