EL Script

Table of Contents

Introduction

An EL script is a file that allows a developer to automate model creation and modification on the machine. The append script will also allow a user to output model information as needed.

A script is very simple in layout and features. The structure of a script will generally include comments, lists, commands (on lists), and files. Where as the most basic script will do nothing, or append a single file to a model.

The abilities of a script will be limited by the features of the connected target. A script can have a million files to append, but the target may limit memory and only the first few files may be appended where the rest will error.

All errors in the script are reported and stored in a log file that bears the same name, and is created in the same directory, of the running script.

Version support in the script is currently limited. As a script will always support older versions, there is no version check to determine if a script being ran holds newer features. Errors will be generated in the corresponding logs in the form of unknown file, or unknown command or option.

Basics

A script may contain comments, lists, commands (on lists), options, and files. Evey line in a script is trimmed, meaning leading and trailing whitespace is removed, before applying the following rules to determine type.

A line may be continued by ending with a \. This allows for better formatting/readability of the script for future use.

Example:

```
!export elcfg "MyModel.elcfg" [prefix=0_, \
                               suffix=_0, \
                               filter="USER_", \
                               user_only]
```

A comment is any line that starts with a #

This type of line, along with empty/blank lines, are ignored during processing.

A list is any line that starts with a {

A list can be useful for performing an action multiple times on a given set of items.

A command is any line that starts with a !

The ! (bang) commands are used to inform the script to do something at that spot in the file.

A option is any line that contains an =

The basic structure of an option is to have a name equal to a value. A list is a special command, so instead of attempting to place the entire contents of the list on a single line, it can break up each portion on its own line.

If the option holds the a list of values, each value shall be separated by a ;

(semicolon). This will be better explained later as it applies.

A file is any line that starts with a X: (where X is a drive), a . (dot), a /, or a .. (dotdot).

Scripts allow files to be stated in a relative position from the directory of the running script. However, disk boundaries cannot be crossed in this method.

Commands & Options

There are many names that are used by the script when a command or option is being used. Each name may be used multiple ways depending on need.

Simple Script Example

The following lines are a simple example of using a script to help your model development. Assume that you have designed 3 sub-models.

submodel_0.elcfg, submodel_1.elcfg and submodel_2.elcfg.

The script would simply list the submodels to be appended using the command "file".

```
!file submodel_0.elcfg
!file submodel_1.elcfg
!file submodel_2.elcfg
```

Maybe you have a base model that contains some core function for your system. The base model may contain several submodels. Rather than list all of the submodels in the top level script, you can create a base model script called base_model.elscr. The top level script can include another script with the command "include".

```
!include ..\Base\base_model.elscr
```

This document contains the various commands and methods which should assist a developer in maintaining sensible a model development environment.

INCLUDE

```
!include <file>
{include
<file(s)>
}
```

Using this command will process, in place, the script file name that follows the include command. The absolute or relative (from script file location)) location to a script file can be used.

Any cyclic includes will be ignored. Meaning, script A includes script B, script B includes script A. If the initial script to run was A, then the second A will be ignored. If the initial script to run was B, then the second B will be ignored.

Example:

```
!include My Other Script.elscr
!include ../My Other Script.elscr
!include ./My Other Script.elscr
!include C:/My Other Script.elscr

# The following are considered the same.
#This
```

```
!include Script1.elscr
!include Script2.elscr

#Or this
{include
Script1.elscr
Script2.elscr
}
```

CONNECT

```
!connect <target> <target_options>
```

This is used as a method for the append script to change the connection to be something new. Also, a good way to force a certain connection that is expected by the script.

Example:

```
# Connect to simulator, using option file.
!connect simulator ../My Simulator Options.ini

# Connect to usb target.
!connect usb
```

ERROR

```
!error <string> [<arg0>, <arg1>, ..., <argN>]
```

This is used to cause the script to quit processing and error out. Mainly used if some expectant state does not exist. For verbose output without stopping the script, use the echo or write command.

The string can be a format string, where the arguments are optional.

If the string does not start with a ", the entire contents to the end of the line are written to the log file.

Example:

```
{if !defined(a_variable_needed)
  !error Required, but not defined: a_variable_needed
}

# Writes (meaning would write to the log file): This is a [test]
!error This is a [test]
# Writes: This is a
!error "This is a " [test]
# Writes: This is a test
!error "This is a %s" [test]
```

ECHO

```
!echo <string> [<arg0>, <arg1>, ..., <argN>]
```

This is used to output strings to the script log. The script log writes a new line after writing the string. The string can be a format string, where the arguments are optional unless wanted.

If the string does not start with a ", then the entire contents to the end of the line are written to the log file.

Example:

```
!echo Hello there.
```

```
# Writes: This is a [test]
!echo This is a [test]
# Writes: This is a
!echo "This is a " [test]
# Writes: This is a test
!echo "This is a %s" [test]
```

WRITE

```
!write <variable> <string> [<arg0>, <arg1>, ..., <argN>]
```

This command is used to place a string, and possible formats to a variable of a file. The writing depends on wether or not the variable given is a handle to a file or a normal string. The strring can be a format string, where the arguments are optional unless needed/wanted.

Example:

```
#script1
!define MyVar
!write MyVar "This is a string.  In current script \"%s\"." [THISFILE]
# Write the contents of MyVar to the log file.
!write THISLOG "MyVar: %s\n" [MyVar]

#script2
!define MyVar
!write MyVar "This is like set."
# MyVar now holds the string:  This is like set.
!write MyVar "  And now appended."
# MyVar now holds the string:  This is like set.  And now appended.

#script3
!define MyVar
!write MyVar "This is a %s"
# MyVar contains: This is a %s
# Due to no arguments being present, the %s is not resolved.
```

WIZM

```
!wizm <option> <value>
```

This command is a way to manipulate the wizm file by adding in other data as expected. The only creation that occurs is arrays, each time a name is used the value is added as another element in the array. As the wizm file is in json formation, the value and name get properly escaped to become value json strings.

The only array that is implemented in the script by default is model_names, you can add to it without issue, however everytime a !file is processed a new entry is added by the script engine. Keeping that in mind may stop possible duplicates.

Example:

```
!wizm "names" "Some Feature"
!wizm "names" "Some Other Feature"
WizmOption = names
WizmValue = This new feature
!wizm WizmOption WizmValue
!export wizm "" [name = "MyWizm", extension = ".txt"]
# The generated wizm file will now have a new array named names, with 3 values
# eg: "names" : ["Some Feature", "Some Other Feature", "This new feature"]
```

DEBUG

```
!debug <action>
```

This command only works if you are using a debug build of the script engine. Otherwise all these commands are ignored.

Supported actions:

`break`

This will stop execution at that point in the script.

`enable_operation_output`

This tells the script engine to output the compiled code for each command .

`disable_operation_output`

This tells the script to not output the compiled code for each command.

LIST

```
{list <variable>
<string(s)>
}
```

This creates a list and adds the given elements to the list. This list can be used anywhere a list is expected.

Example:

```
{list Values
"No Op 1"
"No Op 2"
}
!convert_no_ops Values
```

PAIR_LIST

```
{pair_list <variable>
<string_pair(s)>
}
```

This creates a pair list and adds the given pairs to the list. This list can be used anywhere a pair list is expected.

Example:

```
{pair_list Values
"Old Name" -> "New Name"
}
!rename_items Values
```

PARAMETER_LIST

```
{parameter_list <variable>
<param(s)>
}
```

This creates a parameter list and adds the given parameters to the list. This list can be used anywhere a parameter list is expected.

Example:

```
{parameter_list ItemParams
  type : "No Operation"
  initial_value : 1.0
}
!item create "New Item" ItemParams
```

LOCAL

```
!local <variable>
{local
<command(s)>
}
```

This sets evey variable defined within to be a local. A local variable is one that can be shared to child scripts, but will not adjust the value of any parent scripts.

Example:

```
# Script1
!include Script1.elscr
{if defined(MyVar)
  !echo true
!else
  !echo false
}
# Output: false

# Script2
{local
!set MyVar 1
}
!include Script3.elscr
!echo MyVar
# Output: 1

# Script 3
{if defined(MyVar)
  !echo true
!else
  !echo false
}
# Output: true
```

EXCLUSIVE

```
!exclusive <variable>
{exclusive
<command(s)>
}
```

This will make each variable defined within to be exclusive to the script.

Example:

```
# Script1
!include Script2.elscr
{if defined(MyVar)
  !echo true
!else
  !echo false
}
```

```
# Output: false
# Script2
{exclusive
  !define MyVar
}
!include Script3.elscr

# Script3
{if defined(MyVar)
  !echo true
!else
  !echo false
}
# Output: false
```

GLOBAL

```
!global <variable>
{global
<command(s)>
}
```

This will make each variable defined within to be global.

Example:

```
# Script1
!include Script2.elscr
{if defined(MyVar)
  !echo true
!else
  !echo false
}
# Output: true

# Script2
{global
  !define MyVar
}
# Or
!define MyVar
!include Script3.elscr

# Script3
{if defined(MyVar)
  !echo true
!else
  !echo false
}
# Output: true
```

FILE INPUT/OUTPUT

IMPORT

```
!import <file_type> <file_path> [param0, param1, ... (optional)]
```

Import will read an input file and process based on the file_type. The file_path should be in quotes. The paramaters are optional and depend on the file_type. This support will grow as time passes.

Supported file_type:

`append`

Parameters:

`prefix = <desired_prefix>`

This string will be added to the beginning of all the channel names of the items in the imported model.

`suffix = <desired_suffix>`

This string will be added to the end of all the channel names of the items in the imported model. Brackets "[]" are used in the name and are understood to designate units for the channel. If brackets are present, the string will be appended to the name before the " [units]".

`user_only`

This will filter the Predefined items from the imported configuration. Only the user created items will be appended into the model.

`version_major, version_minor`

This will allow the script to function with different elcfg file versions. The elcfg files should have ".v96",".v97", etc ... appended to the names and the script will use the correct version of the file.

`modify`

Parameters: none

This will perform a modify of the model with the imported .elcfg file. This is analogous to import calibration in the ElConsole application. Items will not be downloaded but the modifiable data of the items will be applied to the same named items in the target.

`unconvert_noops`

Parameters: none

This is the inverse function of convert_noops. The noop mapping JSON structure can be applied to the model. The result will be that noops will be created which were previously replaced and the model will be connected to the noops. This will be helpful when working with models that have been optimized with noops replaced with constants.

`playback`

Parameters:

`time = <desired_time>`

The time is used as an offset for the time of the playback. The format is h:m:s.

`repeat`

The repeat parameter will allow the playback to be repeated.

`remove`

Parameters: none

This will remove from the target the items found in the file. This can be helpful when developing a scripting environment for a model.

Example:

```
!import append "MyModel.elcfg"
!import modify "MyModel.elcfg"
!import append "MyModel.elcfg" [prefix=0_, suffix=_0, user_only]
!import playback "logdata.elcsv" [time="0:0:2.000", repeat]
```

EXPORT

```
!export <file_type> <file_path> [param0, param1, ... (optional)]
```

Export will output a file based on the `file_type`. The paramaters are optional and depend on the exported `file_type`. The `file_path` should be in quotes and may be optional depending on type. Some types require `file_path` to point directly to a file, while others may expect `file_path` to point to a directory.

Supported `file_type`:

`elcfg`

`file_path`: Expected to point directly to a file.

Parameters:

`prefix = <desired_prefix>`

This string will be added to the beginning of all the channel names of the items in the exported model.

`suffix = <desired_suffix>`

This string will be added to the end of all the channel names of the items in the exported model. Brackets "[]" are used in the name and are understood to designate units for the channel. If brackets are present, the string will be appended to the name before the " [units]".

`user_only`

This will filter the Predefined items from the output configuration. Only the user created items will be exported to the model file.

`filter = <name_filter>`

This will filter based on the start of the channel names. With a well organized naming design, this parameter can be used in a variety of ways.

`fixed`

`file_path`: Expected to point directly to a file.

Parameters: none

`dynamic`

`file_path`: Expected to point directly to a file.

Parameters: none

`wizm`

`file_path`: Expected to point to a directory where to store the resultant file.

Parameters:

`name = <string>`

This string will hold the name of the file that is to be generated. The model id string of the model will be used if this is not specified or is specified empty.

`extension = <string>`

This string will hold the extension to use for the file being generated. If empty, or not specified, the system specific extension will be used.

`model_id = "0" "1" "2" or "3"`

The model id string to use when generating the file. This will affect the name of the file, if not overridden, as well as the string that is placed inside the file. If not specified, or empty, the model id used will be determined by the system.

`include_types`

If present, the file will include the types of the items in the model.

`include_channel`

If present, the file wil include the channel of the items in the model.

`filter = <filter>`

This will filter the items that are written to the file. Only those types that are listed in filter will be used. If not specified, or empty, this will use the filter set with `model_filters`.

Examples:

```
!export elcfg "MyModel.elcfg"
!export elcfg "MyModel.elcfg" [suffix=_0, user_only]
!export elcfg "MyModel.elcfg" [prefix=0_, suffix=_0, filter="USER_", user_only]
!export fixed "MyModel.elrom"
!export dynamic "MyModel.elnv"
!export wizm ""
!export wizm "" [model_id = "0", filter = "all"]
!export wizm "../" [name="wizard_file", extension=".wizm", filter = "all"]
```

FILE

```
!file <file>
{file
<file(s)>
}
```

This is deprecate as of `SCRIPT_VERSION` 3 and will be removed in a latrer build.

Use instead:

```
!import append <file>
```

Example:

```
# The following is the same way to append two files to the current model.
# Script 1
../Sub/Model1.elcfg
Model2.elcfg

# Script 2
!file ../Sub/Model1.elcfg
!file Model2.elcfg
# Script 3
{file
../Sub/Model1.elcfg
Model2.elcfg
}

# Script 4
file = ../Sub/Model1.elcfg
file = Model2.elcfg
```

OPEN

```
!open <variable> <file> [<options>]
```

This command is used to open a store a handle to a file. The handle can then be used as the variable a !write to add text to the opened file. The only option that is currently allowed is append, when set it will open a file for appending. The default behaviour is to create the file if it does not exist, or to clear out the file if it already exists.

This command may error, but no message would be responded as the isopen function in the conditionals can be used to check if a handle is open, thus allowing for a manual error to be reported if that is the expected case.

Example:

```
#script1
!open MyFile "MyTestFile.txt"
# A new variable has been defined to be a handle.
!wrte MyFile "Hello %s!" ["World"]
!close MyFile

#script2
!set MyVarHandle This is a text.
!open MyVarHandle "MyTestFile.txt" [append]
{if !isopen(MyVarHandle)
  !error Could not open file.
}
# MyVarHandle was promoted from a string to a handle.
!close MyVarHandle
# MyVarHandle is no longer a handle and is back to being a normal string.
```

CLOSE

```
!close <handle>
```

This command is used to close a file handle. The handle does not have to be open, however this command also demotes the handle back to a normal empty variable. An error would be reported if trying to close a variable that is not a handle. However, that error will not cause the script to stop processing. The error would be more apt to being a runtime warning.

Example:

```
!open MyFile "SomeFile.txt"
!close MyFile
```

MODEL MODIFICATION COMMANDS

DEVELOPMENT HELPERS

These commands are generally used to automate the development process. Sub-models can be designed as separate units and linked by name as needed. The external names may need to be resolved to items or other external names in the target. Hardware items and Predefined hardware inputs will need to be connected to sub-models so that sub-models can be designed as hardware independent units.

ITEM

```
!item <action> <name> [<parameter(s)>]
{item <action> <name>
<parameter(s)>
}
```

This command is a way to manipulate the model with relation to items. Each action is placed on the given item.

Currently supported actions:

`create`

This action will create an item with the specified parameters. When creating an item, a type parameter is required. All other parameters are optional.

Parameters:

`type`

The type of the item being created. This can be on of the Item Types allowed.

See Item Parameters for more information.

`delete`

This action will delete the item from the model. When deleting, an extern is created with the item_name to allow for linkage.

Parameters:

`continue_if_item_missing`

This being set informs the script to continue processing and not error if the item does not exist.

`change`

This action will modify an existing item. In normal model creation, this will delete and recreate the item. This action will fail if trying to modify an item on a hot (running) model. In order to make changes in the hot case, be sure to set the modify parameter.

Parameters:

`modify`

This parameter existsing means to run the modification on a hot model, or to

treat the model as being hot.

`continue_if_item_missing`

This being set informs the script to continue processing and not error if the item does not exist.

See Item Parameters for more information.

`replace`

This action will replace the given item with the parameters specified.

`move_to_primary_heap`

This will move the item to the primary heap.

`move_to_secondary_heap`

This will move the item to the secondary heap.

`set_no_commit`

This will set the item to be no commit.

`clear_no_commit`

This will remove the no commit flag from the item.

`resolve`

This will resolve the extern name with the given parameters.

Parameter:

`name`

This parameter is required as it holds the name of which to resolve the extern.

`replace_channel`

This will replace the hardware place holder with the hardware IO.

Parameter:

`name`

This parameter is required as it holds the name of the item that will replace the main item.

`rename`

This will rename an item with the given name. To rename an extern, use resolve.

Parameter:

`name`

This parameter is required, as it holds the new name.

ITEMS

```
{items <action> [<parameter(s)>]
```

```
<string(s)> or <string_pair(s)>
}
```

This command is special in that it allows doing the same action and parameter to multiple items. Some actions require item_pairs, so the list is expected to be in the pair format.

The parameters will be the same for each entry.

See Item for examples of supported actions.

Example:

```
# The following:
{items create [type="No Operation" initial_value=1000]
"No Op 1"
"No Op 2"
}
# Is the same as:
!item create "No Op 1" [type="No Operation" initial_value=1000]
!item create "No Op 2" [type="No Operation" initial_value=1000]
```

RESOLVE_EXTERNS

```
!resolve_externs <LISTVAR>
{resolve_externs
<extern_mapping(s)>
}
```

This command informs the system to resolve externs stored in the model with the given mapped names. If not given a list `{resolve_externs ...}` the latest `externs_to_resolve` list will be used. This command can also be used to replace an external name with a new external name which could come in handy.

Example:

```
#script1
{pair_list Values
"Extern1" -> "Extern2"
}
Some File.elcfg
!resolve_externs Values

#script2
{externs_to_resolve
"Extern1" -> "Extern2"
}
Some File.elcfg
!resolve_externs

#script3
Some File.elcfg
{resolve_externs
"Extern1" -> "Extern2"
}
```

REPLACE_ITEM

```
!replace_item <name> [type=<type>, <optional/required params>]
{replace_item <name>
  type : <type>
  <optional/required param(s)>
}
```

The replace_item command is currently used for replacing "hardware placeholders" and "external names" only. This command should replace the older commands "replace_items" and "items_to_replace". This command can be used to replace a hardware placeholder item when supporting hardware abstraction. This command can also be used when replacing externs with either "No Operation" items or "Constants". This command may come in handy when one might want to stub out a sub-model with constants or no ops in a model tree.

There are some rules for usage:

- The <name> must be a "HardwarePlaceholder" item for use with type="Hardware IO".
- The <name> must be an extern for other types. "No Operation" and "Constant". NOTE: The <name> does not need to be present if the "ignore_if_extern_not_found" parameter is used. This allows for common scripts to replace several externs that may (or may not) be present. If the extern is not present, the script will halt with an error without this parameter.

Optional command:

```
!item replace <name> [type=<type>, <optional/required params>]
```

Examples:

HardwareIO item creation. Requires type, name, pin, board_conditioning and cpu_mode. The pin, board_conditioning and cpu_mode should use the exact strings as used in the ElConsole "HardwareIO" item properties dialog.

```
{replace_item <placeholder_name>
  type : "Hardware IO"
  name : "my name"
  pin : "VR4+_In, C3-6"
  board_conditioning : "Variable reluctance sensor input"
  cpu_mode : "Falling Edge Decrement"
}
```

No Operation item creation. Requires type and initial_value. The name can be used but if omitted, the No Operation item will take the name of the replaced extern.

```
{replace_item <extern_name>
  type : "No Operation"
  name : "my name"
  initial_value : "0.12345"
}
```

Constant value replacement. Requires type and value.

```
!replace_item <extern_name> [type="Constant", value="0.12345"]
```

Constant value replacement ignore error. Requires type and value.

```
{replace_item <extern_name>
  type : "Constant"
  value : "0.12345"
  ignore_if_extern_not_found
}
```

CHANGE_ITEM

```
!change_item <name> [<optional/required params>]
{change_item <name>
<optional/required params>
}
```

The change_item command can be used to change the various fields in an item. The

specified fields will be changed in the item and the remaining fields of the item will be unchanged.

Parameters:

See Item Parameters for a full list, the following are change specific.

`continue_if_item_missing`

Set to skip the change if the item does not exist. Otherwise an error is generated.

`modify`

Set if you want to only modify an item, this is required if the model is currently running.

Alternate use:

```
!item change <name> [<optional/required param(s)>]
{item change <name>
  <optional/required param(s)
}
```

Examples:

```
!change_item item_name [thread_ID="0"]
{change_item item_name
  thread_ID : 0
}
!change_item item_name [thread_ID="0", priority="135"]
{change_item item_name
  thread_ID : 0
  priority : 135
}
!change_item item_name [iteration="0"]
{change_item item_name
  iteration : 0
}
```

CHANGE_ITEMS

```
!change_items [<optional/required param(s)>] <list_variable>
{change_items [<optional/required param(s)>]
<item(s)>
}
```

This is the same as change_item, however the same changes can be applied to multiple channels.

Alternate use:

```
!items change [<optional/required param(s)>] <list_variable>
{items change [<optional/required param(s)>]
<item(s)>
}
```

CREATE_ITEM

```
!create_item <item_name> [type="type name", <optional/required params>]
{create_item <item_name>
  type : "type name"
  <optional/required param(s)>
}
```

The create_item command can be used to create items in the system. The optional parameters can be specified to change the various fields of the items. The same list of optional parameters from !change_item are supported with the !create_item. A general approach may be to use !create_item to create the item and follo with !change_item to change the item as needed.

Alternate use:

```
!item create <name> [type="typename", <optional/required param(s)>]
{item create <name>
  type : "type name"
  <optional/required param(s)>
}
```

Examples:

```
!create_item "NewItem" [type="No Operation"]
{create_item "NewItem"
  type : "No Operation"
}
```

CREATE_ITEMS

```
!create_items [type="typename", <optional/required param(s)>] <list_variable>
{create_items [type="typename", <optional/required param(s)>]
<item(s)>
}
```

The same as create_item however the same parameters are used to create multiple items at a time.

Alternate use:

```
!items create [type="typename", <optional/required param(s)>] <list_variable>
{items create [type="typename", <optional/required param(s)>]
<name(s)>
}
```

DELETE_ITEM

```
!delete_item <name> [<optional params>]
{delete_item <name>
<optional param(s)>
}
```

This command can be used to delete items in the system. Unresolved externs will be generated if the item is an input to other items in the system.

Optional parameter:

`continue_if_item_missing`
: This optional parameter will allow the script processing to continue if the item to be changed is missing.

Alternate use:

```
!item delete <name> [<optional params>]
{item delete <name>
<optional_params>
}
```

Examples:

```
!delete_item <item_name>
```

```
!delete_item <item_name> [continue_if_item_missing]
{delete_item <item_name>
  continue_if_item_missing
}
```

DELETE_ITEMS

```
!delete_items [<optional param(s)>] <list_variable>
{delete_items [<optional param(s)>]
<item(s)>
}
```

The same as delete_item however this allows for deleting multiple items at a time.

Alternate use:

```
!items delete [<optional param(s)>] <list_variable>
{items delete [<optional param(s)>]
<item(s)>
}
```

REPLACE_ITEMS

```
!replace_items <PAIRLIST_VAR>
{replace_items
<item_mapping(s)>
}
```

This command is obsolete. Please use !replace_item. This command can be used to replace one item in the target with a different item in the target. This can be useful for platform specific hardware item resolution. Generally the base model will be hardware independent and will contain a HardwarePlaceholder items. These items can later be replaced with a HardwareIO items when building for a particular hardware platform. If not given a list `{replace_items ...}` the latest `items_to_replace` list will be used.

Alternate use:

```
{items replace_channel
<item_mapping(s)>
}
```

Example:

```
#script1
{pair_list Mappings
"HWIO_OUT_PWM_Throttle_Hbridge_in0" -> "Pin_HB0_In1_OUT_PWM"
"HWIO_OUT_PWN_Throttle_Hbridge_in1" -> "Pin_HB0_In2_OUT_PWM"
"HWIO_OUT_GPIO_Throttle_Hbridge_enable" -> "Pin_HB0_nDisable_OUT_GPIO"
"HWIO_IN_GPIO_Throttle_Hbridge_status" -> "Pin_HB0_Status_IN_GPIO"
}
Some File.elcfg
!replace_items Mappings

#script2
{items_to_replace
"HWIO_OUT_PWM_Throttle_Hbridge_in0" -> "Pin_HB0_In1_OUT_PWM"
"HWIO_OUT_PWN_Throttle_Hbridge_in1" -> "Pin_HB0_In2_OUT_PWM"
"HWIO_OUT_GPIO_Throttle_Hbridge_enable" -> "Pin_HB0_nDisable_OUT_GPIO"
"HWIO_IN_GPIO_Throttle_Hbridge_status" -> "Pin_HB0_Status_IN_GPIO"
}
Some File.elcfg
!replace_items
```

```
#script3
Some File.elcfg
{replace_items
"HWIO_OUT_PWM_Throttle_Hbridge_in0" -> "Pin_HB0_In1_OUT_PWM"
"HWIO_OUT_PWN_Throttle_Hbridge_in1" -> "Pin_HB0_In2_OUT_PWM"
"HWIO_OUT_GPIO_Throttle_Hbridge_enable" -> "Pin_HB0_nDisable_OUT_GPIO"
"HWIO_IN_GPIO_Throttle_Hbridge_status" -> "Pin_HB0_Status_IN_GPIO"
}
```

CLEAR_CONFIGURATION

```
!clear_configuration
```

This command will clear the model from the target. This may be helpful as an intermediate step where models are exported and imported during the build process.

Alternate use:

```
!model clear
```

RENAME_ITEMS

```
!rename_items <PAIRLIST_VAR>
{rename_items
<item_mapping(s)>
}
```

This command can be used to quickly change only the name of given items. A rename will only occur if the new name does not exist, and the old name is not a predefined item. All issues will be displayed in the log file of the script.

The change_item command can be used for the same effect, however this is used for large scale changes.

Alternate use:

```
{items rename
<item_mapping(s)>
}
```

Example:

```
#script1
{pair_list Mappings
"OldName1" -> "New Name 1"
"OldName2" -> "New Name 2"
}
SomeFile.elcfg
!rename_items Mappings

#script2
{items_to_rename
"OldName1" -> "New Name 1"
"OldName2" -> "New Name 2"
}
SomeFile.elcfg
!rename_items

#script3
SomeFile.elcfg
{rename_items
```

```
"OldName1" -> "New Name 1"
"OldName2" -> "New Name 2"
}
```

OPTIMIZE_PRIORITY128_CHANNELS

```
!optimize_priority128_channels
```

This command can be used to optimize the processing order in a model. The items in the model with a priority of 128 will be renumbered for efficiency. The command will sort the items in each thread and renumber them to allow the most efficient processing method. Items that are inputs to other items will be processed earlier so that the response to system changes will be improved. For example, if item X is an input to item Y and assuming item X and Y have priority 128, item X and Y will be renumbered so that item X has a lower channel number than item Y and it will therefore be processed first.

The reasoning for renumbering only those channels with a priority of 128 is to make sure not to disrupt the priority preference that a user may have intended.

This is deprecate as of SCRIPT_VERSION 3 and will be removed in a latrer build.

Alternate use:

```
!model optimize_128
```

RESOLVE_ALL_EXTERNS_TO_NOOPS

```
!resolve_all_externs_to_noops
```

This command is helpful for testing a submodel. This command will create a noop item for each of the unresolved externs in the system. This will allow quick testing of a model. This will also be helpful when building layouts with a submodel with unresolved channels.

This is deprecate as of SCRIPT_VERSION 3 and will be removed in a latrer build.

Alternate use:

```
!model resolve_externs_to_noops
```

RESOURCE OPTIMIZATION

These commands are used to optimize the resource usage on the target. The target eeprom and ram resource may become strained as the size of the model increases. One target platform may have different resource than another so these commands can be used to generate models as needed.

CONVERT_NO_OPS

```
!convert_no_ops
{convert_no_ops
<noop(s)>
}
```

This informs the system to do a conversion for the supplied NoOperation channels to constants. This is a space optimization method to free up the target resource usage in storing the No Operation items. The No Operation item will be removed from the system and replaced with a constant value in the item. The value of the constant will be the Initial Value of the NoOperation item that is being replaced. If not given a list `{convert_no_ops ...}` the latest `no_ops_to_convert` list will be used. The No Operation channels that have been converted will be added to the noop mapping file after the script has ran.

Alternate use:

```
{items convert_noops
<noop(s)>
}
```

Example:

```
# Script 1
{no_ops_to_convert
NoOp1
NoOp2
}
Some File.elcfg
!convert_no_ops

# Script 2
convert_no_ops = after_each_file
{no_ops_to_convert
NoOp1
NoOp2
}
Some File.elcfg

# Script 3
Some File.elcfg
{convert_no_ops
NoOp1
NoOp2
}
```

UNCONVERT_NO_OPS_FROM_FILE

```
unconvert_no_ops_from_file = <file>
!unconvert_no_ops_from_file <file>
```

This informs the system to take the given mapping file and attempt to generate No Operation channels in the model. Connectivity of the model to the newly generated No Operations will also

be applied.

Alternate use:

```
!import unconvert_noops <file>
```

Example:

```
!unconvert_no_ops_from_file MyMappingFile.noop
```

MOVE_ITEMS_TO_PRIMARY_HEAP

```
!move_items_to_primary_heap
{move_items_to_primary_heap
<item(s)>
}
```

This informs the system to place the given items in primary memory. The primary memory is quicker to access than secondary, however there is not as much available memory in the primary as in the secondary. If not given a list `{move_items_to_primary_heap ...}` the latest `items_to_move_to_primary_heap` list will be used.

Alternate use:

```
{items move_to_primary_heap
<item(s)>
}
```

Example:

```
#script1
{list items_to_move_to_primary_heap
Item1
Item2
}
Some File.elcfg
!move_items_to_primary_heap

#script2
{items_to_move_to_primary_heap
Item1
Item2
}
Some File.elcfg
!move_items_to_primary_heap

#script3
Some File.elcfg
{move_items_to_primary_heap
Item1
Item2
}
```

MOVE_ITEMS_TO_SECONDARY_HEAP

```
!move_items_to_secondary_heap
{move_items_to_secondary_heap
<item(s)>
}
```

This informs the system to place the given items in secondary memory. The primary memory is quicker to access than secondary, however there is not as much available memory in the primary as in the secondary. If not given a list `{move_items_to_secondary_heap ...}` the latest

`items_to_move_to_secondary_heap` list will be used.

Alternate use:

```
{items move_to_secondary_heap
<item(s)>
}
```

Example:

```
#script1
{list items_to_move_to_secondary_heap
Item1
Item2
}
Some File.elcfg
!move_items_to_secondary_heap

#script2
{items_to_move_to_secondary_heap
Item1
Item2
}
Some File.elcfg
!move_items_to_secondary_heap

#script3
Some File.elcfg
{move_items_to_secondary_heap
Item1
Item2
}
```

SET_ITEMS_NO_COMMIT

```
!set_items_no_commit
{set_items_no_commit
<item(s)>
}
```

This informs the system to place the given items as no commit. This essentially saves ram on the system as nothing in the given items is expected to change. The exception being the output value.

Alternate use:

```
{items set_no_commit
<item(s)>
}
```

Example:

```
#script1
{list items_to_set_no_commit
Item1
Item2
}
Some File.elcfg
!set_items_no_commit

#script2
{items_to_set_no_commit
Item1
```

```
Item2
}
Some File.elcfg
!set_items_no_commit

#script3
Some File.elcfg
{set_items_no_commit
Item1
Item2
}
```

CLEAR_ITEMS_NO_COMMIT

```
!clear_items_no_commit
{clear_items_no_commit
<item(s)>
}
```

This informs the system to clear the no commit feature for the item. This can be handy when dealing with models that are ROM models that are now being used as dynamic models and this flag needs to be cleared.

Alternate use:

```
{items clear_no_commit
<item(s)>
}
```

Example:

```
#script1
{list items_to_clear_no_commit
Item1
Item2
}
Some File.elcfg
!clear_items_no_commit

#script2
{items_to_clear_no_commit
Item1
Item2
}
Some File.elcfg
!clear_items_no_commit

#script3
Some File.elcfg
{clear_items_no_commit
Item1
Item2
}
```

RENUMBER_CHANNELS

```
!renumber_channels
```

This informs the system to renumber all the channels in the current model. Essentially, this is packing all the channels to be closer in memory. This step is not a necessary step in the build process but is mainly used for debug purposes.

Alternate use:

```
!model renumber_channels
```

Example:

```
# Bunch of files
# Some conversions
!renumber_channels
```

MODEL

```
!model <action>
```

This is a method that is used to affect the entire current model.

Actions supported:

`renumber_channels`

This will renumber all channels to fill in holes.

`optimize_128`

This will renumber and rearrange those items with priority 128. This rearrangment is done to allow for the best data propagation. This may be an extremely slow operation, use with caution.

`clear`

This will clear all user items from the model.

`resolve_externs_to_noops`

This will look through all the externs in the model, and for each one a No Operation item with the externs name will be created. This allows for fast testing of sub models.

`stop`

This tells the model to stop computation. If the model is currently stopped, this is ignored.

`run`

This tells the model to start computation. If the model is currently running, this is ignored.

CONDITIONAL SCRIPT CONTROL

DEFINE

```
!define <variable>
{define
<variable(s)>
}
```

This is used to define a single, or multiple, variables. The variables will be empty, and are generally used to specify features. Like checking the target then specifying which features are available. Further down the line, checking if a feature is defined determines if said feature is to be included.

Example:

```
!define my_variable
# ... some where down the line, possibly in a different included script
{if defined(my_variable)
  # A list of files, or other such my_variable specifics.
}
```

SET

```
!set <variable> <value>
<variable> = <value>
```

This is used to define a variable and set it to a certain value. Unlike define above, this is used if a variable may actually change throughout the running of the script.

Example:

```
!set my_variable "Something Wicked"
my_variable_2 = "This way comes"
# ... somewhere down the line, possibly in a different included script.
{if defined(my_variable) && defined(my_variable_2)
  {if my_variable == "Something Wicked" && my_variable_2 == "This way comes"
    # Do something.
  }
}
```

SET_PATH

```
!set_path <variable> <path>
```

This sets the value to the variable to be the resolved path.

Example:

```
!set_path my_path "../"
# Assuming scripts path is C:\Script
!echo ##my_path##
# Ouputs : C:\
```

DEFAULT

```
!default <variable> <value>
```

This is used to ensure that the given variable has a value. Only processes if variable is not already defined.

Examples:

```
# Script1
!set my_var "My Value"
!default my_var "Default Value"
!echo my_var
# Ouput: My Value

# Script2
!default my_var "Default Value"
!echo my_var
# Output: Default Value
```

DEFAULT_PATH

```
!default_path <variable> <path>
```

This is used to ensure that the given variable has a value. Only processes if variable is not already defined.

Examples:

```
# Script1
!set_path my_path "../Something"
!default_path my_path "../Default Path"
!echo my_path
# Output: "C:/Something"  Assuming C:/Script is where the script is located.

# Script2
!default my_path "../Default Path"
!echo my_path
# Output: "C:/Default Path" Assuming C:/Script is where the script is located.
```

IF

```
{if <condition>
...
}
```

This is used to allow for branching depending on the given criteria. This begins an if-block, an if-block may contain many elseif commands and upto one else command. See Conditionals for more information.

Example:

```
{if defined(TEST)
  {if TEST == "SomeThing"
    # Do something
  !elseif TEST == "SomethingElse"
    # Do something else
  !else
    # Do Default
  }
!else
  # Do Default
}
```

ELSEIF

```
!elseif <condition>
```

This is used to offer a different option while in an if-block. This command only has a definition while inside an if-block. One or more of these commands may exist inside an if-block, however this command may never follow an else command. See Conditionals.

Example:

```
{if defined(TEST)
  #Do stuff for TEST case
!elseif defined(TESTONE)
  #Do stuff for TESTONE case
}
```

ELSE

```
!else
```

This is used to offer a default for an if-block or select-block. When all other cases are false, the else supplies the default actions. Only one else may exist for any given if-block or select-block. See Conditionals.

Example:

```
{if defined(TEST)
  # Specific
!else
  # Default
}

{select TEST
  ...
!else
  # Do if no other case works.
  !break
}
```

LOOP

```
{loop <condition>
...
}
```

This is used to run a set of commands over and over again while the condition holds true.

A loop expects any command sequences to be inside. To allow for early termination of a loop, use the !break command.

Example:

```
!set TEST 1
{loop TEST < 5
  !echo "##TEST##"
  !increment TEST
}

{loop true
  # Do stuff that may define TEST.
  {if defined(TEST)
    !break
  }
}
```

BREAK

```
!break
```

Depending on when used, this will either exit a loop-block, or exit a select-block. The inner most block is broken when this command is used.

When used for a loop-block, this will jump processing to the end of the loop.

When used for a select-block, this will jump processing to the end of the select.

Example:

```
!set TEST 1
{loop TEST <= 5
  {select TEST
  !else
    {loop true
      # Break out of this inner loop.
      !break
    }
    # Break out of the select.
    !break
  }
  # Break out of the loop.
  !break
}
```

SELECT

```
{select <variable>
...
}
```

Used when you want something to occur when the variable is a certain value, or if you want multiple things to occur depending on the value.

Each case is specified by a !when, where you can include an !else if no cases exist.

Example:

```
{select TEST
  # Do not place commands between the select and the first !when or !else as they will
not be processed.
!when 1
  # Do something when value is 1.
  !break
!when 3
  # Do something when value is 3, then fallthrough.
!when 2
  # Do something when the value is 2, then fallthrough.
!else
  # Do something when the value is 3 2 or anything other than 1 and 4.
  !break
!when 4
  # Do something when the value is 4.
}
```

WHEN

```
!when <value>
```

Use this to determine the cases used for a select-block. The value must be a constant string or number. The value type (string or number) determines how equality is determined.

Example:

```
{select TEST
!when "StringValue"
  # Do this iff TEST == "StringValue".
  !break
!when 4
  # Do this iff number(TEST) == 4
  !break
}
```

INCREMENT

```
!increment <variable> <value>
```

This converts the value in variable to a number, increments the number by the given value then stores the result back in variable.

If no value is specified, the increment is 1.

Example:

```
!set MyVar 1
!increment MyVar
!echo MyVar
# Output: 2
!increment MyVar 2
!echo MyVar
# Output: 4
```

DECREMENT

```
!decrement <variable> <value>
```

This converts the value in variable to a number, decrements the number by the given value then stores the result back in variable.

If no value is specified, the decrement is 1.

Example:

```
!set MyVar 1
!decrement MyVar
!echo MyVar
# Output: 0
!decrement MyVar 2
!echo MyVar
# Output: -2
```

VARIABLES

Variables are a way to specify state in the script. There are two types of variables: System, and User. The main difference between the two is the defined nature. A system variable is always defined, where as a user variable may or may not be defined, depending on what command were actually ran in the script.

A variable may contain any number of letters, digits, or underscores however, a variable must start with a letter or underscore. The variables are case sensitive, so the variable _hi is different from the variable _Hi. System variables are always fully capitalized. This is to allow for quick differentiation between possible user variables. A system variable will be empty until if no data exists.

A variable should not be named the same as a command. So the variable if, define, error, etc should never be declared.

The currently supplied system variables are:

`PLATFORM`
: This will hold the platform string for the current connection. Eg: Zuma_11 or Venice.

`MODELID_0`
: This will hold the first string in the model id of the current connection. Will be empty if no model id is found.

`MODELID_1`
: This will hold the second string in the model id of the current connection. Will be empty if no model id is found.

`MODELID_2`
: This will hold the third string in the model id of the current connection. Will be empty if no model id is found.

`MODELID_3`
: This will hold the fourth string in the model id of the current connection. Will be empty if no model id is found.

`VERSIONMINOR`
: This will hold the minor version of the connected target. Will be empty if undetermined. If determined, this will hold a 4 character string padded with zeros. Eg: This will hold "0055" for the target 96.55.

`VERSIONMAJOR`
: This will hold the major version of the connected target. Will be empty if undetermined. If determined, this will hold a 3 character string padded with zeros. Eg: This will hold "096" for the target 96.55.

`THISFILE`
: This will hold the full path to the current script. This is meant to be a constant, and cannot be changed.

`THISLOG`
: This is a handle to the log file of the current script. This handle is managed internally, and cannot be opened or closed. Useful when wanting to write more information to the log, and echo is not enough.

`SCRIPTLOG`
: This if a handle to the log file of the root script (the top most script that is currently running). This handle is managed internally, and cannot be opened of closed. Useful when wanting to write more information to the main scripts log file. This will be the same handle as `THISLOG` when in the main script.

`SCRIPT_VERSION`
: This holds the version of the scripting engine being used. As of right now, this is equal to 3.

To define a variable, especially if no value is ever to be expected use the define command.

```
!define my_variable
```

To change the contents of a variable, or to define with a set value use the set command.

```
!set my_variable "This text"
my_variable = "This text"
```

To set the contents of a variable only if not previously set use the default command.

```
!default my_variable "My Contents."
```

There is no restriction on changing the value of the system variables, however what ever value is manually set will be overridden before the next required reference.

```
!set PLATFORM MyPlatform
{if PLATFORM == "MyPlatform"
# This may never occur, unless the connection really is MyPlatform
}
```

CONDITIONALS

Conditionals for append scripts are used to determine what commands may be ran depending on the current state of the script. A conditional may be used to break up normal script flow, or be used for determining the elements that will ultimatley exist in a list.

Only four commands exist for conditionals: `if`, `elseif`, `else`, and `loop`

An if statement determines the boundaries for a given if-block. As such, the typical list style is used:

```
{if true
# Everything between is part of the if block.
}
```

The elseif and else are commands, and are only valid while in an if-block. Essentially, when an if-block is being parsed, the commands that will be ran are between:

```
{if true
# The true case, until:
!else
# The false case.
}
```

Nothing in the false case would run in the last example, but everything up until the `!else` will be ran.

Nesting of if-blocks is allowed, the amount of nesting does not matter, however it is suggested that each nest is offset to allow for quick visible reference.

```
{if true
  #Nest1
  {if true
    #Nest2
    {if true
      #Nest3, etc etc
    }
  }
}
```

A conditional may be used to make choices between normal commands, or between list elements.

```
{if true
!file file1
!file file2
}
!file file3
{convert_no_ops
NoOp1
{if true
  NoOp2
}
}
```

A loop-block is self terminated, the loop continues while the given condition holds true.

```
{loop true
# Everything between is part of this loop-block.
# Continue running until broken or condition fails.
}
```

To exit a loop at any time regardless of the condition, place a !break statement.

```
{loop true
  # Do some stuff.
  # We do not want to continue anymore
  !break
}
```

Nesting of loop-blocks is allowed, however it is suggested that each nest is offset to allow for quick visible reference.

```
{loop true
  #Nest 1
  {loop true
     #Nest 2
    {loop true
      #Nest 3, etc etc
      !break
    }
    !break
  }
  !break
}
```

CONDITION

A condition is used as the test for the `if` or `elseif`. This determines if the follow commands will be ran.

Conditions are boolean based, where the entire condition needs to evaluate to a `true` or `false`. For simplicity, the append script supports the:

Basic comparators: `==, !=, <, >, <=, >=`

Basic joins: `||, &&`

Basic negator: `!`

Basic functions: `defined(...), empty(...), is_open(...), supported(...), action_supported(...), number(...)`

Basic constants: `"...", true, false, isroot, [+-]0-9*`

As everything in the append script is string based, comparing as numbers requires a cast, such as `number(...)`, or to have a number constant being compared against.

Comparators

```
left == right
```

This will compare the two sides for equality

Eg: `"Hello" == "Hello"` results in `true`.

`"Hello" == "World"` results in `false`.

`"5" == 5` results in `true`.

`"5" == 6` results in `false`.

`5 == "5"` results in `true`.

`5 == "6"` results in `false`.

`5 == 5` results in `true`.

`5 == 6` results in `false`.

`0 == ""` results in `true`.

`left != right`

This will compare the two sides to determine inequality.

Eg: `"Hello" != "Hello"` results in `false`.

`"Hello" != "World"` results in `true`.

`"5" != 6` results in `true`.

`"5" != 5` results in `false`.

`5 != "6"` results in `true`.

`5 != "5"` results in `false`.

`5 != 6` results in `true`.

`left < right`

This will check if left is less than right.

Eg: `"A" < "B"` results in `true`.

`"B" < "A"` results in `false`.

`5 < "6"` results in `true`.

`5 < 6` results in `true`.

`"6" < 5` results in `false`.

`6 < 5` results in `false`.

`left > right`

This will check if left is greater than right.

Eg: `"A" > "B"` results in `false`.

`"B" > "A"` results in `true`.

`5 > "6"` results in `false`.

`5 > 6` results in `false`.

`"6" > 5` results in `true`.

`6 > 5` results in `true`.

`left <= right`

This will check if left is less than or equal to right. This is equivalent to the condition: `left < right || left == right`

`left >= right`

This will check if left is greater than or equal to right. This is equivalent to the condition: `left > right || left == right`

Join

`left || right`

This will result in true if either left or right result in true.

Eg: `"A" == "B" || "A" == "A"` will result in true.

`left && right`

This will result in true iff left and right result in true.

Eg: `"A" <= "B" && "A" == "A"` will result in true.

Negator

`! right`

This will flip the result of right. So `!true` is `false` and `!false` is `true`.

Functions

`defined(variable)`

This will return true iff variable has been defined. This will always return `true` if the given variable is a system variable.

Eg: `defined(SYSTEMVARIABLE)` is true

`define(USERVARIABLE)` may be true or false...

`empty(variable/string)`

This will return true if the given variable holds an empty string, or if the given string is already empty.

Eg: `empty("")` is true

`empty("123")` is false

`isopen(variable)`

This will return true if the given variable is a handle to a file and if the file is currently opened.

Eg: `isopen(THISLOG)` is `true`

`isopen(SomeRandomVar)` may be `true` or `false`...

`number(variable/string)`

This will cast the given variable or string to be considered a number for comparisons.

Eg: `number("5")` is treated as 5

`supported(string)`

This will return true if the given string is a supported command in the script.

Eg: `supported("export")` is `true` if !export command is supported.

`supported("notsupported")` may be `true` or `false` depending on !notsupported being allowed.

Constants

`"..."`

This is a string constant. Surrounded by quotes, so any quote expected in the string should be escaped such that: `"\""` is actually the string `"`. Or `"\\"` is actually the

string `\`.

Eg: `"string", ""`

`true`

This just means `true`.

`false`

This just means `false`.

`isroot`

This is `true` iff the current script is the initial script ran (the parent, rather than a child).

`[-+]0-9*`

This is a number constant. No quotes and each character is the value between 0-9.

Eg: `5, 6, 190, 019, 223, 21243, 0, -1, -1728, -15`

ITEM PARAMETERS

The parameters that are generally accepted when doing an !item create or !item change command. Those listed below are mentioned if they are required, or if they are only for create or change.

COMMON

These parameters are common to all items or many types. In some instances a parameter is ignored. An example would be trying to place inputs into a No Operation item.

`type`

This is the type of the item. Required when creating an item.

This may be one of the following values:

```
"No Operation"
"Engine Description"
"Passthrough"
"Analog In conditioner"
"Math Calculation"
"If-Else Condition"
"1-Axis Lookup Table"
"2-Axis Lookup Table"
"Running Average Calculation"
"Rate-Of-Change Calculation"
"Multiplexor"
"MAF Fuel Requirement"
"Limiter"
"Performance Profiler"
"EmbeddedCode block"
"Unit Converter"
"Thermistor"
"PID Controller"
"OBDII Parameter IDs"
"OBDII MID TIDs"
"Speed Density"
"Timer"
"Hardware IO"
"Fixed 1-Axis Lookup Table"
"Plant Hardware Writer"
"Diagnostic Reporter"
"Data Latch"
"Lambda Staged Fuel Subsystem"
"CAN Transmitter"
"CJ125 Interface"
"EAL Event"
"HIP9011 Interface"
"External Communicator"
"Model identifier"
"CAN Packet"
"Serial Stream"
"Engine Generator"
"Hardware Placeholder"
"CJ125 Interface v2"
"NMEA2000 Interface"
```

`name`

The name of the item. This is only used when changing (modifying) and item to allow for a rename.

`clamp_flag`

The item will clamp its output.

`clamp_lower`

The lower end of the clamp value, output will not be below this value.

`clamp_upper`

The upper end of the clamp value, output will not be above this value.

`use_test_output_flag`

This tells the item to use its test output over actual computational output. Good for testing models.

`dirty_flag`

The item is considered dirty, will be reflashed during a commit and will force any tool to reload the item into its cache.

`plant_item_flag`

The item is part of a plant. Simulator only. Will not be placed into a hardware system.

`no_commit_flag`

The item will not be commited to flash memory. Exists as a constant in the system.

`force_nonmodifiable_clear_flag`

The item when downloaded will have all information cleared before retrieval. Useful when hiding certain model mechanics.

This is set be default on some systems, the force is to set the flag regardless.

`disable_nonmodifiable_clear_flag`

This will turn off the clear for this item.

`secondary_ram_flag`

The item will be stored in the external ram of the system. The external has more space, but processes slower. Use caution.

`thread_ID`

The item will be placed in the given threads processor. This only works for items that are processed, No Operation items are not processed so this would be ignored.

`iteration`

`thread_iteration`

How often this item is processed in the thread. A value of 1 means every time, 2 every other time etc etc.

`priority`

`thread_priority`

The priority in the thread, a higher priority means that the item is processed before others in the same thread. By nature, the items with the same priority are processed in the order in which they are stored.

`initial_value`

The value that the item is set to when the model before the model is first run.

`test_value`

The value used in leiu of output when use_test_output_flag is set.

`delete_inputs`

Clear all inputs to the item. This is for change (modify) only.

`inputs`

The inputs for the item. This is for create only.

This is meant to be a list, as such there are multiple ways to set the values.

Example:

```
# A single input
!item create "..." [inputs=["Input1"]]
# or
!item create "..." [inputs="Input1"]
# Many inputs
!item create "..." [inputs=["Input1", "Input2", ...]]
# or
!item create "..." [inputs="Input1", inputs="Input2", ...]
```

`input0 - input19`

This is a way to set an individual input. The number of inputs currently existing must match the value of X. If only 2 inputs exist then setting an input2 will fail. Inputs are 0 based, so input0 is the first input and Input19 is the last allowed input.

`add_input`

This will append the input to the item.

Example:

```
# A Single input
!item create "..." [inputs="Input1"]
# Want more later.
!item change "..." [add_input="Input2"]
# Item now has 2 inputs: Input1 Input2
# Multiple inputs
!item create "..." [inputs="Input1"]
!item change "..." [add_input=["Input2", "Input3"]]
# Or
!item change "..." [add_input="Input2", add_input="Input3"]
# Both cases:  Item now has 3 inputs: Input1 Input2 Input3
```

ENGINE DESCRIPTION

`num_sparks/ed_num_sparks`

The number of sparks supported.

`spark_timings`

A list of timings for the number of sparks. If num_sparks not specified, the number will be the number of items in the list.

`spark0_timing/ed_spark0_timing - spark9_timing/ed_spark9_timing`

Set the timing for the specified spark. Will error if the spark is out of range.

`error0_response/ed_error0_response - error9_response/ed_error9_response`

The error from the given spark...

`num_coils/ed_num_coils`

The number of coils supported.

`num_injectors/ed_num_injectors`

The number of injectors supported.

`injector_timings`

A list of timings for the number of injectors. If num_injectors is not specified, the number will be the number of items in the list.

`injector0_timing/ed_injector0_timing - injector11_timing/ed_injector11_timing`

Set the timing for the specified injector. Will error if the injector is out of range.

`coil_type/ed_ign_type`

The type of coil being used.

One of:

```
"Coil On Plug"
"Wasted Spark"
"Distributor"
```

`coil_invert/ed_ign_invert`

Invert the signal.

`inj_type/ed_inj_type`

The type of injector.

One of:

```
"Multiport low or high impedence"
"Gas Direct Injection"
```

`displacement/ed_displacement`

The displacement of the engine.

`num_cylinders/ed_num_cylinders`

The number of cylinders supported.

`cycles/ed_cycles`

The cycle of the engine.

One of:

```
"2 stroke"
"4 stroke"
```

`timing/ed_timing`

The timing type to use.

One of:

```
"Cam sensor single tooth"
"Honda600R Cam sensor 3 teeth"
"Cam with crank missing 1 tooth"
"Cam with crank missing 1 tooth single VVT"
"Cam with crank missing 2 teeth"
"Cam with crank missing 2 teeth single VVT"
"Honda Civic 5 tooth cam 12+1 crank"
"Honda RSX 2 cam 12+1 crank single VVT"
"Subaru WRX STi VVT"
"1st/2nd Generation Diamond Star Motors"
"Nissan QG18DE"
"Dodge 8 cylinder NGC VVT"
"Dodge Viper 1992-95"
"Dodge Viper 1996 and up"
"Mitsubishi EVO 8/9 VVT"
"GM LS7 4/60-2 VVT"
"BMW S54"
"BMW S65"
"Nissan VQ35 2 Cam VVT"
"Nissan VQ35 4 Cam VVT"
"Nissan VQ37"
"Toyota 2JZGTE"
"Dodge Viper 4th Gen"
"Honda J37A1"
"Subaru EZ36 4Cam VVT"
"Honda F22C"
"Porsche 997"
"Ford Coyote"
"Volkswagon 1.8"
"Lamborghini V10"
"GM LFX"
"Toyota 2GR-FE"
"Yamaha 1.8 4+1 crank"
"Nissan RB26DETT 6-180"
"Nissan SR20DET 4-180"
"Ford 4.6L 3V"
"Mazda L MZR"
"Toyota 3SGE"
"Chrysler 6.4L Hemi"
"GM Ecotec LE5"
"Mazda NB8C VVT"
```

`crank_pin/ed_crank_pin`

The pin on the board that is connected to the crank signal.

`crank_teeth/ed_crank_teeth`

The number of teeth on the crank.

`crank_edge_trigger/ed_crank_edge_trigger`

The edge on which to count a crank tooth.

One of:

```
"Rising edge"
"Falling edge"
```

`crank_nc/ed_crank_nc`

The noise cancelation filter to apply.

`cam0_pin/ed_cam0_pin - cam3_pin/ed_cam3_pin`

The pin on the board that holds the specified cam signal.

`cam0_teeth/ed_cam0_teeth - cam3_teeth/ed_cam3_teeth`

The number of teeth on the specified cam.

`cam0_edge_trigger/ed_cam0_edge_trigger - cam3_edge_trigger/ed_cam3_edge_trigger`

The edge on which to count a cam tooth.

One of:

```
"Rising edge"
"Falling edge"
```

`cam0_nc/ed_cam0_nc - cam3_nc/ed_cam3_nc`

The noise cancelation filter to apply to specified cam.

ANALOG IN CONDITIONER

`min/analog_in_min`

The minimum value for the given signal.

`max/analog_in_max`

The maximum value for the given signal.

`clampoverride_flag/analog_in_clampoverride_flag`

Tell the item to override the clamp values.

`clampoverride/analog_in_clampoverride`

The value to set when clamp is applied.

MATH CALCULATION

`expression/math_expression`

The expression to use for the item.

Example:

```
!item create "..." [expression="i0 + i1"]
```

`append_expression/math_expression_append`

Append the given expression to what already exists.

Example:

```
!item create "..." [expression="i0 + i1"]
!item change "..." [append_expression="+i2"]
# Actual expression being ran: i0 + i1 +i2
```

IF-ELSE CONDITION

`clear_expressions/if_else_clear_expressions`

Remove all the expressions from the item.

`expressions`

A list of expressions, up to 5. Each expressions matches that value returned when true.

`add_expression/if_else_add_expression`

Add a new expression to the item. Up to 5.

1-AXIS LOOKUP TABLE

`cell_precision`

How accurate the data is in the table. Either 8, 16, or 32

`min_cell_value`

The minimum value of the cell data.

`max_cell_value`

The maximum value of the cell data.

`cell_adjust`

The filter to apply to the cells

`interpolate`

The amount of interpolation applied to the table.

`csv/1Dtable_csv`

The file in which to read in the table data.

MoTec csv files are supported.

`axis_selector`

The input index to use as the axis data.

`width/axis_width`

The number of cells in the axis.

Note: This controls the entire size of the item as the number of axis points is also the number of actual table data.

2-AXIS LOOKUP TABLE

`cell_precision`

How accurate the data is in the table. Either 8, 16, or 32

`min_cell_value`

The minimum value of the cell data.

`max_cell_value`

The maximum value of the cell data.

`cell_adjust`

The filter to apply to the cells

`interpolate`

The amount of interpolation applied to the table.

`csv/2Dtable_csv`

The file in which to read in the table data.

MoTec csv files are supported.

`xaxis_selector/axis0_selector`

The index to the input used for the x-axis.

`width/axis0_width`

The number of elements on the x-axis of the table.

`yaxis_selector/axis1_selector`

The index of the input used for the y-axis.

`height/axis1_width`

The number of elements on the y-axis of the table.

LIMITER

`output_inversion/limiter_output_inversion`

TODO

`compare_inversion/limiter_compare_inversion`

TODO

UNIT CONVERTER

`sub_type/conversion_type/unit_conversion_type`

The type of conversion to apply.

One of:

```
"Celsius To Fahrenheit"
"Celsius To Kelvin"
"Fahrenheit To Celsius"
"Fahrenheit To Kelvin"
"Kelvin To Celsius"
"Kelvin To Fahrenheit"
```

```
"PSI To KPa"
"KPa To PSI"
```

THERMISTOR

sub_type/thermistor_type

The type of thermistor.

One of:

```
"Resistance (Ohms)"
"Coolant Temp (degC)"
"Air Temp (degC)"
"1GDSM Clt (degC)"
"GM12146312 Clt (degC)"
"Supra 1993-7 Clt (degC)"
"Honda 1990-01 Clt (degC)"
"Honda 2002-06 Clt (degC)"
"Subaru 2004-06 Clt (degC)"
"2GDSM Clt (degC)"
"Acura NSX 1994-7 Clt (degC)"
"GM25036751 Air (degC)"
"S2000 2000-5 Air (degC)"
"Honda 1990-01 Air (degC)"
"SubaruSTI 2004-6 Air (degC)"
"Toyota MAF Air (degC)"
"Nissan Air (degC)"
```

PID CONTROLLER

invert_enable/pid_invert_enable

TODO

derivative_uses_measured/pid_derivative_uses_measured

TODO

TIMER

mode/timer_mode

The mode of the timer.

One of:

```
"One-shot"
"Elapsed Time Transition"
"Signal Present For Time"
"Linear Transition Over Time reset=start"
"Linear Transition Over Time reset=end"
"Increment While Event Present"
"Signal Present Increment and Count"
```

invert_reset/timer_invert_reset

TODO

FIXED 1-AXIS LOOKUP TABLE

`sub_type/table_type/fixed_1DTable_type`

The type of the fixed table.

One of:

```
"LS1 Mass-Air-Flow Sensor"
"LS6 Mass-Air-Flow Sensor"
```

DATA LATCH

`edge/data_latch_edge`

The edge on which this latch is activated.

One of:

```
"Rising Edge"
"Falling Edge"
```

`initial_state/data_latch_initial_state`

The initial state of the latch.

One of:

```
"Zero"
"Non Zero"
```

`edge_counter/data_latch_edge_counter`

Tell the item to count the edges encountered.

LAMBDA STAGED FUEL SUBSYSTEM

`num_primary_injectors/lsfs_num_primary_injectors`

TODO

`primary_injectors`

TODO

`primary_injector0 - primary_injector11`

TODO

`primary_stoich/lsfs_primary_stoich`

TODO

`primary_fuel_density/lsfs_primary_fuel_density`

TODO

`num_secondary_injectors/lsfs_num_secondary_injectors`

TODO

`secondary_injectors`

TODO

`secondary_injector0 - secondary_injector11`

TODO

`secondary_stoich/lsfs_secondary_stoich`

TODO

`secondary_fuel_density/lsfs_secondary_fuel_density`

TODO

CJ125 Interface

CJ125 Interface v2

`device/cj125_device`

TODO

EAL Event

`sub_type/event_type/ealevent_type`

One of:

```
"Injector"
"Spark"
"Save data"
```

`edge/ealevent_edge`

One of:

```
"Rising"
"Falling"
```

`initial_state/ealevent_initial_state`

One of:

```
"Zero"
"Non Zero"
```

CAN Packet

`channel/CanPacket_channel`

The can channel for the item to use.

One of:

```
"A"
"B"
```

`direction/CanPacket_direction`

The direction of communication on this item.

One of:

```
"Transmit"
"Receive"
```

`sub_type/CanPacket_type`

The type of this can item.

One of:

```
"Periodic"
"Event"
```

`max_entries/CanPacket_max_entries`

The maximum number of entries in this item. Max limit 200.

`entry/CanPacket_entry`

A list of entry indices that are being set.

`entry_address/CPE_address`

`entry0_address - entry199_address`

11 or 29 bit address. NOTE: Use hex in script.

`entry_format/CPE_format`

`entry0_format - entry199_format`

One of:

```
"Standard 11-bit"
"Extended 29-bit"
```

`entry_end/CPE_end`

`entry0_end - entry199_end`

Can packet end for this address packet.

`entry_input/CPE_input`

`entry0_input - entry199_input`

The input to use for this entry.

`entry_data_type/CPE_data_type`

`entry0_data_type - entry199_data_type`

One of:

```
"float 2 int8"
"float 2 int16"
"float 2 int32"
"float 2 float"
"uint32 2 int8"
"uint32 2 int16"
"uint32 2 int32"
"uint32 2 float"
"int32 2 int8"
"int32 2 int16"
"int32 2 int32"
"int32 2 float"
"uint16 2 int32"
"uint16 2 float"
"int16 2 int32"
"int16 2 float"
"uint8 2 int32"
"uint8 2 float"
"int8 2 int32"
"int8 2 float"
```

```
            "float 2 uint8"
            "float 2 uint16"
            "float 2 uint32"
            "float 2 signmag8"
            "float 2 signmag16"
            "float 2 signmag32"
            "signmag8 2 float"
            "signmag16 2 float"
            "signmag32 2 float"
```

`entry_data_endianess/CPE_data_endianess`

`entry0_data_endianess - entry199_data_endianess`

One of:

```
   "Little endian"
   "Big endian"
```

`entry_data_minvalue/CPE_data_minvalue`

`entry0_data_minvalue - entry199_data_minvalue`

For interpolation of 8bit or 16bit.

`entry_data_maxvalue/CPE_data_maxvalue`

`entry0_data_maxvalue - entry199_data_maxvalue`

For interpolation of 8bit or 16bit.

`entry_data_bits/CPE_data_bits`

`entry0_data_bits - entry199_data_bits`

Number of bits to use for this entry.

`entry_data_shift/CPE_data_shift`

`entry0_data_shift - entry199_data_shift`

The bit shift in the 8-byte packet for this entry.

ENGINE GENERATOR

`timing/etg_timing`

TODO

`crank_pin/etg_crank_pin`

TODO

`crank_teeth/etg_crank_teeth`

TODO

`crank_invert/etg_crank_invert`

TODO

`cam0_pin/etg_cam0_pin - cam3_pin/etg_cam3_pin`

TODO

`cam0_teeth/etg_cam0_teeth - cam3_teeth/etg_cam3_teeth`

TODO

`cam0_invert/etg_cam0_invert - cam3_invert/etg_cam3_invert`

TODO