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.

base_directory

```
base_directory = <path>
```

Using this option will change the base directory of the script. This only affects those files listed as relative, as they become relative to this new base directory rather than the actual directory of this script. This may cause errors as the files may not exist.

This option defaults to the directory of the script.

Example:

```
base_directory = C:\Start Here
```

directories

```
directories = <path(s)>
{directories
<path(s)>
}
```

When only a file name is given with no path information, meaning no drive, . (dots), or .. (dotdots) these directories are applied to find a file that exists. If all have been tried with no success, the `base_directory` is used allowing processing to continue. The error, if any, will be

displayed in the log file.

This option defaults to only `base_directory`.

Example:

```
directories = C:\Now Here
{directories
C:\Now Here
C:\And Here
}
```

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 `base_directory`) 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
}
```

MODEL_FILTERS

```
model_filters = <filter>
{model_filters
<filter(s)>
}
```

This is a global option used to determine what type of channels are output to the no op mapping file. This may contain one or more values, see the example.

Possible Values:

`all`
: All types of channels are displayed. This is the default if not specified.

`none`
: No channels will be displayed.

`<Channel Type>`

These are actual channel type names as used by the ELConsole, or other scripting tools. The values may differ, but shall always be lower case. Check the Add Channel menu option in ELConsole to determine what types are available.

Example:

```
model_filters = all
model_filters = 1-axis lookup table;2-axis lookup table
{model_filters
1-axis lookup table
math
if-else
}
# Do stuff.
```

OUTPUT_TYPES

```
output_types = <boolean>
```

This is a global option used to determine if the types field of an item is displayed in the output file (no op mapping file).

Possible Values:

`true`
: The channel types will be included in the no op mapping file. If not specified, this is the default value.

`false`
: The channel types will not be included in the no op mapping file.

Example:

```
output_types = true
# Do other stuff
```

OUTPUT_CHANNELS

```
output_channels = <boolean>
```

This is a global option used to determine if the raw channel number will be displayed in the output file (no op mapping file).

Possible Values:

`true`
: The raw channel value will be included in the no op mapping file. If not specified, this is the default value.

`false`
: The raw channel will not be included in the no op mapping file.

Example:

```
output_channels = true
# Do other stuff
```

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"]
```

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. If not specified, this will use the same value that `output_types` was set.

include_channel

If present, the file wil include the channel of the items in the model. If not specified, this will use the same value that `output_channels` was set.

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 informs the system to append the given file to the current model.

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
```

GENERATE_FIXED_FILE

`!generate_fixed_file <file>`

This informs the system to generate a fixed file and store it in the given `<file>`. Fixed files are ROM based models. A ROM based model is generally used for large models that need to make optimal usage of the target resource.

Example:

```
# Some Files
!generate_fixed_file MyRomModel.elrom
```

GENERATE_DYNAMIC_FILE

`!generate_dynamic_file <file>`

This informs the system to generate a dynamic file and store it in the given `<file>`. Dynamic models are the eeprom based counterpart to a ROM based model. The dynamic model generally stores all of the modifiable data of the model. The fixed and dynamic files are used for large models in a production environment.

Example:

```
# Some Files
!generate_dynamic_file MyDynamicModel.elnv
```

DEFER_FILE_CREATION

`defer_file_creation = <defer>`

This is a global option used to determine when the mapping file should be generated.

Possible Values

`self`

The mapping file will be generated at the end of the current script. This includes any conversions that were made prior to the calling of the script.

`root`

The mapping file will be generated only by the initial script being ran. The conversions will be passed back to the parent script as needed. This is the default value.

Example:

```
defer_file_creation = self
# Do stuff.
```

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.

resolve_externs

```
resolve_externs = <when>
!resolve_externs
{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
resolve_externs = after_each_file
{externs_to_resolve
Extern1 -> Extern2
}
Some File.elcfg

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

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

externs_to_resolve

```
externs_to_resolve = <extern_mapping>
{externs_to_resolve
<extern_mapping(s)>
}
```

This list holds name mappings. The first name is the generic name of an external name currently on the target. The name does not need to exist, however a resolution only occurs with existing names. The second name is the name of an actual item, another external name or a name that does not exist. Those items in the model that reference the extern, will be set to reference the

new item instead. If the second name does not exist on the system, the first named extern will be renamed that of the second name. This feature allows for more generic sub models, where resolving can be done at the highest level where the specific names would be known.

Example:

```
{externs_to_resolve
GenericName1 -> SpecificName1
GenericName2 -> SpecificName2
}
Some File.elcfg
!resolve_externs
```

REPLACE_ITEM

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

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.
- The <name> must be in quotes "<name" if "[" is in the name. Quoted names are always accepted.

Example1: 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"]
```

Example2: 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" ]
```

Example3: Constant value replacement. Requires type and constant_value.

```
!replace_item extern_name [type="Constant", constant_value="0.12345"]
```

Example4: Constant value replacement. Requires type and constant_value.

```
!replace_item extern_name [type="Constant", constant_value="0.12345",
```

```
ignore_if_extern_not_found]
```

CHANGE_ITEM

```
!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.

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

Additional common params:
name
clamp_flag
clamp_lower
clamp_upper

add_input - This one is special where an input is added to the item. Just give it a
name as an item or an extern.

delete_inputs - Delete all of the inputs to the item.

input0 - input19 - Assign a particular input to an item name or extern.

continue_if_item_missing - This optional parameter will allow the script processing to
continue if the item to be changed is missing. This is helpful for renaming a large
group of items, such as the EAL_required items, where many of the items may or may not
be present in the model.

modify - This command will have the change_item use a modify command when changing the
item. A modify is equivalent to "import calibration" as opposed to "append
configuration".

Other misc. Parameters:
initial_value
test_value
use_test_output_flag
thread_ID
iteration
priority
hidden_flag
plant_item_flag
no_commit_flag
force_nonmodifiable_clear_flag
disable_nonmodifiable_clear_flag

Additional item specific params:
(Math) math_expression
(Math) math_expression_append
(If/else) if_else_clear_expressions
(If/else) if_else_add_expression
(1DTable) axis_selector
(1DTable) axis_width
(1DTable) 1Dtable_csv - assign a .csv file to the table.
(1DTable/2Dtable) cell_precision
(1DTable/2Dtable) min_cell_value
```

```
(1DTable/2Dtable) max_cell_value
(1DTable/2Dtable) cell_adjust
(1DTable/2Dtable) interpolate
(2DTable) axis0_selector
(2DTable) axis0_width
(2DTable) axis1_selector
(2DTable) axis1_width
(2DTable) 2Dtable_csv - assign a .csv file to the table.
(AnalogConditioner) analog_in_min
(AnalogConditioner) analog_in_max
(AnalogConditioner) analog_clamp_override_flag
(AnalogConditioner) analog_clamp_override
(EngineDescriptor) eal_num_sparks
(EngineDescriptor) eal_num_coils
(EngineDescriptor) eal_num_injectors
(LambdaStagedFuelSubsystem) lsfs_num_primary_injectors
(LambdaStagedFuelSubsystem) lsfs_num_secondary_injectors
(LambdaStagedFuelSubsystem) lsfs_primary_injector0 ... 11
(LambdaStagedFuelSubsystem) lsfs_secondary_injector0 ... 11
(LambdaStagedFuelSubsystem) lsfs_primary_stoich
(LambdaStagedFuelSubsystem) lsfs_secondary_stoich
(LambdaStagedFuelSubsystem) lsfs_primary_fuel_density
(LambdaStagedFuelSubsystem) lsfs_secondary_fuel_density
(Timer) timer_mode="One-shot" - all modes supported
(Thermistor) thermistor_type="Resistance (Ohms)" - all supported
(Fixed 1Dtable) fixed_1DTable_type="LS1 Mass-Air-Flow Sensor" - all supported
(Unit conversion) unit_conversion_type="Celsius To Fahrenheit" - all support
 more to come ...
```

CREATE_ITEM

```
!create_item <item_name> [type="type name", <optional/required params>]
```

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.

```
Type names supported:
   type="No Operation"
   type="Engine Description"
   type="Passthrough"
   type="Analog In conditioner"
   type="Math Calculation"
   type="If-Else Condition"
   type="1-Axis Lookup Table"
   type="2-Axis Lookup Table"
   type="Running Average Calculation"
   type="Rate-Of-Change Calculation"
   type="Multiplexor"
   type="MAF Fuel Requirement"
   type="Limiter"
   type="Unit Converter"
   type="Thermistor"
   type="PID Controller"
   type="Speed Density"
   type="Timer"
```

```
type="Hardware IO"
type="Fixed 1-Axis Lookup Table"
type="Plant Hardware Writer"
type="Data Latch"
type="Lambda Staged Fuel Subsystem"
type="CJ125 Interface"
type="EAL Event"
type="HIP9011 Interface"
type="Model identifier"
type="CAN Packet"
type="Serial Stream"
type="Engine Generator"
type="Hardware Placeholder"
type="CJ125 Interface v2"
type="NMEA2000 Interface"
```

DELETE_ITEM

```
!delete_item <name> [<optional params>]
```

The delete_item 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.

!delete_item item_name
!delete_item item_name [continue_if_item_missing]
```

REPLACE_ITEMS

```
replace_items = <when>
!replace_items
{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.

Example:

```
#script1
replace_items = after_each_file
{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
```

```
#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
}
```

ITEMS_TO_REPLACE

```
items_to_replace = <item_mapping(s)>
{items_to_replace
<item_mapping(s)>
}
```

This command is obsolete. Please use !replace_item. This list stores name mappings to items currently on the target. The first name is that of the item being replaced by the item that matches the second name. The first item is generally of an `Hardware Placeholder` type, where the second item is generally of an `Hardware IO` type. This command will not work if the second item is a Predefined item type. Use the resolve_externs command to replace an items inputs with a Predefined item.

When processed, the script informs the target to do the replacement. This allows for the target to save memory, and item indices while also allowing for more generic development of sub models. Using this list, along with `externs_to_resolve` list, allows for the majority of sub models to be abstracted against the final platform, or model.

Example:

```
{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
```

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.

RENAME_ITEMS

```
rename_items = <when>
!rename_items
{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.

Example:

```
#script1
rename_items = after_each_file
{items_to_rename
OldName1 -> New Name 1
OldName2 -> New Name 2
}
SomeFile.elcfg

#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
}
```

ITEMS_TO_RENAME

```
items_to_rename = <item_mapping(s)>
{items_to_rename
<item_mapping(s)>
}
```

This command is only added to match other commands of the same type. This is meant to be obsolete, use the list style `rename_items` instead.

This will hold a list for the next raw `!rename_items` call to use.

Example:

```
{items_to_rename
OldName 1 -> New Name 1
OldName 2 -> New Name 2
}
Some File.elcfg
!rename_items
```

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.

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 = <when>
!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 NoOperation items. The NoOperation 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 NoOperation channels that have been converted will be added to the noop mapping file after the script has ran.

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
}
```

no_ops_to_convert

```
no_ops_to_convert = <noop(s)>
{no_ops_to_convert
<noop(s)>
}
```

This name holds the list of names that are to be converted on the next `convert_no_ops` command. The list will be overwritten at the point in the file where a redeclartion occurs.

When used in conjunction with the `convert_no_ops` command, this will cause all `No`

`Operation` items in the current model that are named in the list to be converted to constants. The mapping of where the `No Operation` items use to exist will be stored in the mapping output file.

Example:

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

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.

Example:

```
!unconvert_no_ops_from_file MyMappingFile.noop
```

MOVE_ITEMS_TO_PRIMARY_HEAP

```
move_items_to_primary_heap = <when>
!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.

Example:

```
#script1
move_items_to_primary_heap = after_each_file
{items_to_move_to_primary_heap
Item1
Item2
}
Some File.elcfg

#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
```

```
}
```

ITEMS_TO_MOVE_TO_PRIMARY_HEAP

```
items_to_move_to_primary_heap = <item(s)>
{items_to_move_to_primary_heap
<item(s)>
}
```

This resets the values of the shared list. The contents of this list is used on the next raw `move_items_to_primary_heap` call.

Example:

```
items_to_move_to_primary_heap = Item1;Item2
{items_to_move_to_primary_heap
Item1
Item2
}
```

MOVE_ITEMS_TO_SECONDARY_HEAP

```
move_items_to_secondary_heap = <when>
!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.

Example:

```
#script1
move_items_to_secondary_heap = after_each_file
{items_to_move_to_secondary_heap
Item1
Item2
}
Some File.elcfg

#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
}
```

ITEMS_TO_MOVE_TO_SECONDARY_HEAP

```
items_to_move_to_secondary_heap = <item(s)>
```

```
{items_to_move_to_secondary_heap
<item(s)>
}
```

This resets the values of the shared list. The contents of this list is used on the next raw `move_items_to_secondary_heap` call.

Example:

```
items_to_move_to_secondary_heap = Item1;Item2
{items_to_move_to_secondary_heap
Item1
Item2
}
```

SET_ITEMS_NO_COMMIT

```
set_items_no_commit = <when>
!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.

Example:

```
#script1
set_items_no_commit = after_each_file
{items_to_set_no_commit
Item1
Item2
}
Some File.elcfg

#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
}
```

ITEMS_TO_SET_NO_COMMIT

```
items_to_set_no_commit = <item(s)>
{items_to_set_no_commit
<item(s)>
}
```

This resets the values in the shared list. The contents of this list will be used on the next raw `set_items_no_commit` call.

Example:

```
items_to_set_no_commit = Item1;Item2
{items_to_set_no_commit
Item1
Item2
}
```

CLEAR_ITEMS_NO_COMMIT

```
clear_items_no_commit = <when>
!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.

Example:

```
#script1
clear_items_no_commit = after_each_file
{items_to_clear_no_commit
Item1
Item2
}
Some File.elcfg

#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
}
```

ITEMS_TO_CLEAR_NO_COMMIT

```
items_to_clear_no_commit = <item(s)>
{items_to_clear_no_commit
<item(s)>
}
```

This resets the values in the shared list. The contents of this list will be used on the next raw `clear_items_no_commit` call.

Example:

```
items_to_clear_no_commit = Item1;Item2
{items_to_clear_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.

Example:

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

Conditional script control

break_here

As command:

```
!break_here
```

This is used internally for debugging purposes, but is placed in the open incase future tools allow for more strict break step continue usage when running a script. Please DO NOT USE this command.

Example:

```
!break_here
```

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.
  }
}
```

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. When all other cases are false, the else supplies the default actions. Only one else may exist for any given if-block. See Conditionals.

Example:

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

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.

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
```

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 flo, or be used for determining the elements that will ultimatley exist in a list.

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

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
}
}
```

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(...)`

Basic constants: `"...", true, false, isroot`

As everything in the append script is string based, all conditions will generally compare strings. Whether the strings be in variables, or constants.

Comparators

`left == right`

This will compare the two sides for equality

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

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

`left != right`

This will compare the two sides to determine inequality.

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

`"Hello" != "World"` 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.

`left > right`

This will check if left is greater than right.

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

`"B" > "A"` 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...

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 `\`.

`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).

Controlling "when" to process a command

Some commands below have an option style that says when the script should automatically run; In the listing this will be noticed by a <when>.

The possible values for this case are the same, and causes the same thing to occur with the given command. The possible values are:

`after_each_file`
: The command will run after each file has been appended. This inserts the command version after each file when processing. Stops the need to explicitly state the command when it may be needed after every file has been processed.

`end_of_script`
: The command will run only once at the end of the script. This does not stop explicit runs from occuring, but does enforce the command to run at the end regardless.

`never`
: The command will never auto run. Explicitly stating when to run the command is required. This is the default value.

In most cases, using the "!" is the most direct approach to executing a command and ignoring the <when> usage can generally be avoided.

Example with convert_no_ops: (convert_no_ops is just an example of one of the supported commands in the system. This command can be used as a model optimization step where no operation items are removed from the model. Again, this is just used an example to better explain the <when> concept of controlling when a command is processed)

```
#script1
convert_no_ops = end_of_file
{no_ops_to_convert
NoOp1
NoOp2
}
# Alot of files.

#script2, same as script1
convert_no_ops = never
{no_ops_to_convert
NoOp1
NoOp2
}
# Alot of files.
!convert_no_ops

#script3
convert_no_ops = after_each_file
{no_ops_to_convert
NoOp1
NoOp2
}
File1
File2

#script4, same as script3
convert_no_ops = never
{no_ops_to_convert
NoOp1
NoOp2
}
File1
!convert_no_ops
File2
```

```
!convert_no_ops
```