Advanced I/O Scripter User Manual for GreenWave ver. X.34

1. Features

1.1 Features Overview:

- ✓ 8 Programmable Scripts that can be up to 20K bytes each.
  - ✓ User Editable Script Name
  - ✓ Globally Enabled/Disabled (Controller restart not required)
- ✓ 2 Types of Scripts (4 Input Scripts and 4 Output Scripts)
  - ✓ Input Scripts run before the traffic engine after reading field inputs.
  - ✓ Output Scripts run after the traffic engine before setting field outputs.
- ✓ Ability to Edit Scripts from the Front Panel of the Controller (not recommended for large changes to Script text)
  - ✓ Shortcut menu to insert components into the script to avoid manual typing.
  - ✓ Scripts tested for syntax upon entry, with automatic jump to line with error.
  - ✓ Ability to enter custom text using the 16-character 0-9 A-F keypad in cellphone like style (T9 entry style + extra keys for shift, symbols, and space)
- ✓ Ability to Edit Scripts from Web Server (recommended method) and upload/download script files.
- ✓ NTCIP Access to Scripts
- ✓ Flexible Debug Print mechanism for troubleshooting Scripts real time with output history.
- ✓ 8 Debug Flags with optional timeout, displayed on Status Screens 1.1.1, 1.1.2 and 1.1.6.1, for troubleshooting Scripts (version X.34 and higher).
- ✓ Ability to Save / Load Scripts from an external USB storage device.

1.2 Advanced I/O Scripter Features:

- ✓ Scripted Logic Statements allow complex logic combination for mapping I/O Pins to Functions
- ✓ Mapped Controller Input Functions (ie. Detector 1 Input, Preempt 1 Input, Stop Time, etc.) and Output Functions (ie. Phase 1 Green, Overlap 1 Red, Phase 1 Check, etc.) can be used as testable conditions.
- ✓ Ability to Override Input Functions before Traffic Engine Runs
- ✓ Ability to Override Outputs immediately after Traffic Engine Completes
- ✓ Simple timing functions DELAY(x) and ELAPSED() which allow timing logic to be performed without the need to manage counters.
- ✓ Ability to check values internal to the traffic engine and override various traffic engine values.
- ✓ Ability to perform custom timing for an intersection by commanding the engine through internal traffic engine functions.
- ✓ Time of Day Enabling or Disabling by checking Time of Day status functions in the Script.

1.3 Advanced I/O Scripter Advanced Features:

The Advanced I/O Scripter uses the Lua programming language (see lua.org). This manual describes only the necessary subset of Lua features for successfully achieving most scripting needs.

	This icon is displayed within this document when referencing advanced features of the Lua language. These sections can be skipped without impeding basic understanding of using the Advanced I/O Scripter.

- ✓ Ability to create variables and use them across multiple scripts.
- ✓ Full capability of the Lua 5.1.5 Programming Language (Third Party Documents Readily Available, see www.lua.org)

Note Advanced features are not supported by ORIUX Product Support Dept.

2. Operation

2.1 Advanced I/O Scripter Operational Overview:

The Advanced I/O Scripting engine executes scripts that use certain reserved keywords to direct flow through the script and to perform operations and define variables. Functions exist to get mapped input and output values, override mapped input and output values, and get raw pin values. Separate scripts are required for overriding input values and for overriding output values. The order of script execution is defined further below in this section.

Script **syntax** refers to what may be typed into a script that will load into the controller without an error. Invalid script syntax leads to a syntax error. The controller will reject saving script data that has incorrect syntax.

2.2 Advanced I/O Scripter Syntax:

- A script is a sequence of statements that are evaluated from top to bottom.
- The basic statement types are:
  - An **Assignment** (ex, flag = true – assigns variable called “flag” equal to the value “true”).
  - An **if statemen**t (see below table description for if)
  - A call to a **function** (ex, PHGRN(2) – calls function that gets the value (true or false) of Phase 2 Green output).
- An expression evaluates to a value of one of these types:
  - An expression may evaluate to nil, true/false, a number, or a string.
  - A true or false expression can use:
    - the operators **<**, **>**, **<=**, **>=**, **~=**, and **==**
    - the keywords **and**, **or**, and **not** along with variables or functions that evaluate to true or false.
  - A number expression can use the operators **+**, **-**, ***, */*, **^**, and **%** (see table below) along with variables or functions that evaluate to numbers.
  - A string expression contains a sequence of one or more characters. Strings entered in the Script must be surrounded by single or double quotes.
- A Script can be broken to a new line **anywhere a whitespace occurs.** This may be needed to avoid exceeding the 40-character limit per row.

2.3 Common Reserved Keywords

Keywords or Operators	Description	Example

if
then
else
elseif
end
	The if statement uses the keywords if, then, else, elseif, and end. It tests expressions and executes statements.
	
	if *expression1* **then**
	statements
	elseif *expression2* **then**
	statements
	else
	Statements
	end
	
	The **elseif** and **else** are optional. Multiple **elseif** clauses can exist, but only one **else** clause is allowed.
		if expression1 then
		print(“expression1 is true”)
		elseif expression2 then
		print(“expression1 is false”)
		print('expression2 is true’)
		else
		print('expr1&2 are false')
		end
		
		if PHGRN(2) then
		setPHOMIT(1,on)
		elseif PHGRN(1) then
		setPHSCALL(3)
		elseif PHNXT(4) then
		setPHOMIT(3,on)
		end

	IMPORTANT: The *expression* in an if statement must evaluate to true or false. For example, if it evaluates to a numerical value, the expression will ALWAYS BE TRUE. See example to the right and Section 2.11.	This always prints true regardless of the value of Ring 1 Yellow Timer: if RINGYELTMR(1) then print(“true”) end
and	The 'and' keyword may be used in expressions to perform boolean logic. The 'and' evaluates sub-expressions on its left and right side. If both the left and right side evaluate to true the 'and' returns true, otherwise it returns false. NOTE: The 'and' is evaluated left side first then right side, returning true or false. If the expression to the left of the 'and' evaluates to false, the right side is not evaluated and the 'and' expression returns false.	print(true and false) Output is: false print(false and true) Output is: false print(true and true) Output is: true print(false and false) Output is: false
or	The 'or' keyword may be used in expressions to perform boolean logic. The 'or' evaluates sub-expressions on its left and right side. If the expression on either side evaluates to true the 'or' returns true, otherwise returns false. Note: The 'or' is evaluated left side	print(PHGRN(1) or PHGRN(2)) Output of the above print is true if either phase 1 or phase 2 is green, otherwise the output of the print is false.

	first then right side, returning true or false. If the expression to the left of the 'or' evaluates to true, the right side is not evaluated and the 'or' expression returns true.	
not	The 'not' keyword is used to switch a boolean expression on its right side to the opposite state. The 'not' keyword returns the opposite of the sub-expression to the right of the 'not' keyword.	'not true' evaluates to 'false' 'not false' evaluates to 'true' 'not PHGRN(2)' will evaluate to true if phase 2 is not green, and will be false if phase 2 is green.

true, false	The 'true' and 'false' keywords are the two states involved in boolean logic.
	An 'if' statement expression must always evaluate to a boolean value.
	Logical Operators: 'and', 'or', and 'not' have boolean expressions as input and output.
	Comparison Operators: '<', '>', '<=', and '>=' have numerical input and boolean output.
	Equality Operators: '==' and '~=' have numerical or boolean input and boolean output.
		Both of these will work:
		if DET(17) then
		setPHOMIT(3,on)
		end
		
		if DET(17) == true then
		setPHOMIT(3,on)
		end
		
		This will **not work** because true is not the same as number 1:
		If DET(17) == 1 then
		setPHOMIT(3,on)
		end

nil	The 'nil' keyword is used to indicate an undefined value. A variable name that has not been assigned to a value has a value of nil by default.	

=	This is the assignment operator. The item on the left side of the = is assigned to the value on the right side. **The value on the left side must be a user-defined variable**. The value on the **right side** can be any valid expression (could have another user-defined variable in that expression).
	
	IMPORTANT: do not confuse the single equal with the double-equal == equality operator described below.
		if SCRIPTSTART() then
		flag = true
		end
		The above sets the user-defined variable “flag” equal to true.
		
		NOT TO BE CONFUSED WITH
		if flag == true then
		setPHOMIT(1,on)
		end
		The above checks if “flag” is equal to true. If it is, setPHOMIT is executed, otherwise it is not executed.
		
		The following is invalid and will cause an error:
		if flag = true then
		setPHOMIT(1,on)
		end

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td><
<=
>
>=</td><td>These operators are used in comparisons; they require the expressions to the left and right side of the operator to evaluate to numbers.

If the value on the left side of the operator is
<table><tr><td><</td><td>Less than</td></tr><tr><td><=</td><td>Less than or equal to</td></tr><tr><td>></td><td>Greater than</td></tr><tr><td>>=</td><td>Greater than or equal to</td></tr></table>the value on the right side of the operator, then the comparison evaluates to true, otherwise it evaluates to false.</td><td>if ELAPSED() < 50 then
   print(“less than 5 seconds”,
      ”have elapsed”)
else
   print(“5 seconds or more”,
      “have elapsed”)
end</td></tr>
<tr><td>==
~=</td><td>These operators are used to test for equality; they require expressions to the left and right side of the operator to be of the same variable type.

The ‘==’ equality test evaluates to true only if the items on the left and right side are equal, otherwise it evaluates to false.

Conversely, the ‘~=’ equality test evaluates to true only if the values on the left and right side are not equal, otherwise it evaluates to true</td><td>print(5 ~= 7)
output is: true

print(not (5 == 7))
output is: true

The following example will always print true regardless of the values of A and B. This example shows that using not along with == achieves the same result as using ~=.
print((A ~= B) == (not (A == B)))
output is: true</td></tr>
<tr><td>+
-
*
/</td><td>Arithmetic operators perform the expected mathematic operations on expressions evaluating to numbers that appear to their left and right side.

<table><tr><td>+</td><td>Addition</td></tr><tr><td>-</td><td>Subtraction</td></tr><tr><td>*</td><td>Multiplication</td></tr><tr><td>/</td><td>Division</td></tr></table></td><td>print(1+2)
output is: 3

print(3-1)
output is: 2

print(4*7)
output is: 28

print(9/3)
output is: 3</td></tr>
<tr><td>^</td><td>Exponential operator</td><td>Print(2^4)
output is: 16</td></tr>
<tr><td>%</td><td>Modulus operator, returns the remainder of dividing the numerical expression on the left by the numerical expression on the right.</td><td>print(7%3)
output is: 1</td></tr>
</table>

2.4 Operator Precedence (order of operations)

The scripter evaluates keywords and operators in the following order from highest to lowest priority.

```
^
not    - (unary)
*      /      %
+      -
<      >      <=     >=     ~=     ==
and
or
```

You can use parenthesis to change the precedence of an expression. **ORIUX recommends always using parenthesis to make the script explicit instead of relying on the above built-in order of operations. Parenthesis must be used when combining "and" and "or".** For example, to omit Phase 1 when Phase 2 is green and either Detectors 11 or 12 have a call, you must use parenthesis as follows:

```
if (DET(11) or DET(12)) and
   PHGRN(2) then
  setPHOMIT(1,on)
end
```

The above highlighted parenthesis causes (DET(11) or DET(12)) to be evaluated first (resulting in a true or false value) that gets an "and" operation with PHGRN(2), yielding the correct result. Without the parenthesis highlighted above, DET(12) and PHGRN(2) would be evaluated first, then an **"or"** operation with DET(11), which is not the correct logic.

2.5 Advanced I/O Scripter: Advanced Reserved Keywords

The following keywords are not necessary for achieving the functionality necessary in most scripts. These keywords are reserved and must be used in accordance with the Lua language specification (see www.lua.org).

```
break, do, for, function, in, local, repeat, return, until, while
```

2.6 Script Comments

Scripts can have comments added to aid in readability and clarity.
A comment starts with a double dash `--` and can have any characters after it and the remainder of the Script line is considered a comment. **The forbidden words listed in Section 2.9 are not allowed to be in a comment.** Comments have no effect on the Script's operation. ORIUX recommends using comments in Scripts. The following are examples of comments:

Example of a comment on the entire line:

```
--omit ph 1 if ph 2 grn:
if PHGRN(2) then
 setPHOMIT(1,on)
end
```

Example of an “inline” comment (comment on same line as script contents):

```
if PHGRN(2) then
 setDET(3,on) --set det 3
end
```

Example of an “inline” comment mistake, setDET(3,on) is executed, but setDET(4,on) is part of the comment and is not executed:

```
if PHGRN(2) then
 setDET(3,on) --set det 3 setDET(4,on)
end
```

Example of multiple line comments (“comment block”):

```
-- multiple line comments are
-- allowed. The if statement below
-- omits ph 1 if ph 2 is grn:
if PHGRN(2) then
 setPHOMIT(1,on)
end
```

NOTE: The above PHGRN, setDET and setPHOMIT are three of the built-in functions described in Section 3.

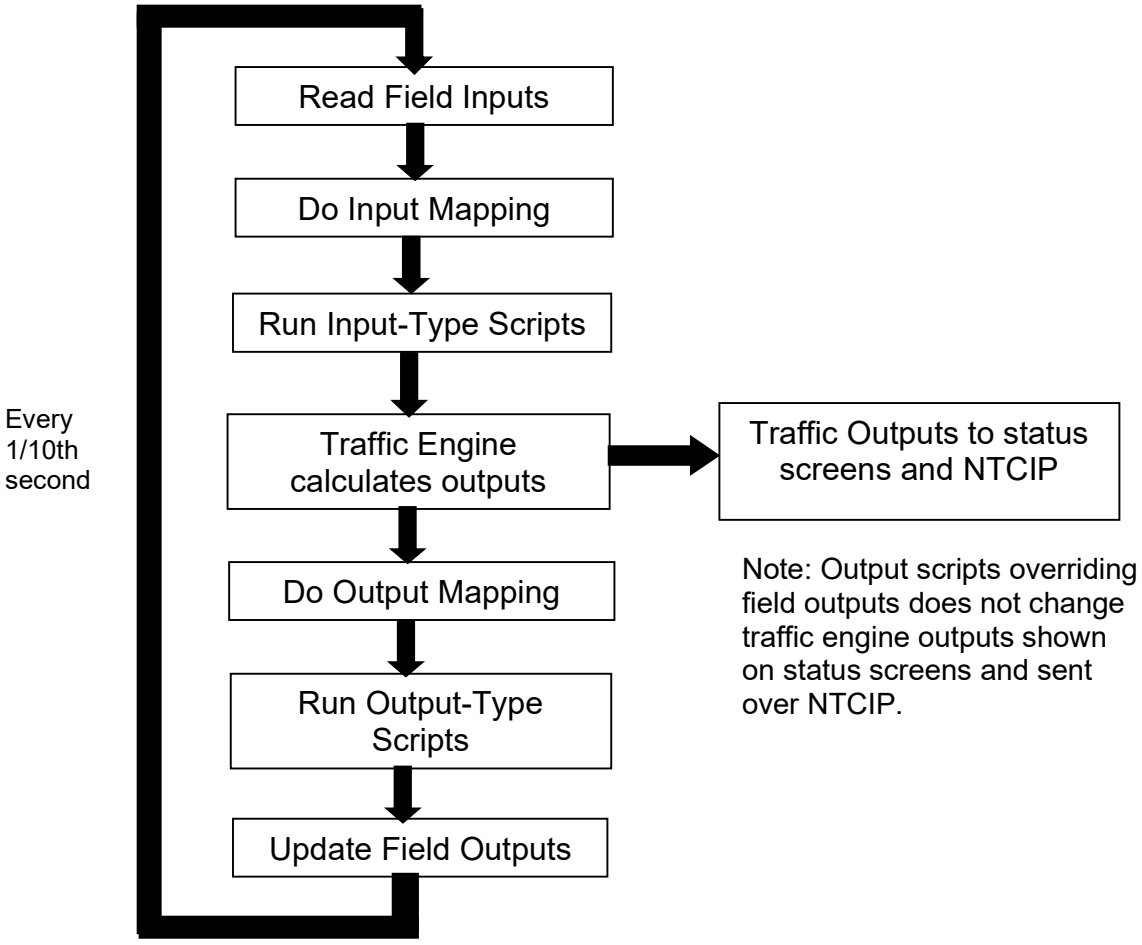
2.6 Advanced I/O Scripter: Execution Sequence

The following steps 1-6 run in order every tenth second:

1. Field Inputs are read from Hardware via Input Mapping (Screen 2.1.5.4).
2. Input Scripts are run – Script #1 first, Script #4 last.
3. Traffic Engine calculates outputs and internal status (ex, phase, ped and overlap signals, preempt state).
4. Output Mapping (Screen 2.1.5.4) runs using Traffic Engine's outputs.
5. Output Scripts are run – Script #1 first, Script #4 last.
6. Field Outputs are updated to reflect any changes from Output Scripts.

Diagram 1 below illustrates the above steps.

Diagram 1: Field I/O Traffic Engine and Script Execution



Input-Type Scripts Detecting Output Changes
Based on the above, an Input-Type Script will detect changes to Outputs 0.1 seconds after they are made. Example: based on Diagram 1, an Input-Type Script will run, and "see" Phase 1 RED. The Traffic Engine will run next and change Phase 1 to GREEN.

The Input-type Script will “see” Phase 1 green at the NEXT pass thru Diagram 1 (0.1 seconds later).

NOTES ABOUT OUTPUT SCRIPTS:

An Output Script that tries to set an Input has no effect on the Input because every tenth-second, all Inputs are read from the field and over-write any Inputs set by the **Output** Script.

A Script-overridden Output does not match the Traffic Engine’s Outputs shown on the controller’s Status Display the same way external cabinet logic does not match the Status Display (ex, controller’s Phase 2 status is red, and external logic turns Phase 2 green).

Output Mapping sets every Output to the value calculated by the Traffic Engine every tenth second, overriding any output script that previously set an output. Therefore, an Output Script needing to override an Output’s value must do so continuously during the required time frame.

NOTES ABOUT INPUT SCRIPTS:

An Input Script that tries to set an Output has no effect on the Output because every tenth-second, the Traffic Engine calculates and sets all outputs, which overwrites the outputs set by the **Input** Script, then the Output Scripts are executed.

Input Mapping reads every input value each tenth second, overriding any Input Scripts that previously set an input. Therefore, an Input Script needing to override an Input’s value must do so continuously during the required time frame.

An Input Script detects changes to Outputs after they have been ON for a tenth-second. Referring to Diagram 1, let’s say the Traffic Engine will turn Phase 2 ON and Phase 1 OFF:

The Input Scripts run. Phase 2 will be detected **OFF** and Phase 1 will be detected **ON**.
The Traffic Engine runs and turns Phase 1 OFF and Phase 2 ON.
The Output Scripts run. Phase 2 will be detected **ON** and Phase 1 will be detected **OFF**.

In the next 100ms, Phase 2 has been ON for 100ms:
The Input Scripts run. Phase 2 will be detected **ON** and Phase 1 will be detected **OFF**.
The Traffic Engine runs, and the next 100ms of Phase 2 is timed.
The Output Scripts run. Phase 2 will be detected **ON** and Phase 1 will be detected **OFF**.

Advanced I/O Scripter: Errors and Debugging

Two Categories of Errors

- Syntax Errors
  - A script cannot be saved in the controller if there is a syntax error.
  - When attempting to save a Script (Controller front panel or Web Interface), if a syntax error is present, a message box will be displayed explaining the syntax error and stating the Script line number with the error.
- Runtime Errors
  - Runtime Errors stop a Script from running and change the script status to an error state.
  - Pressing the E key on Controller Advanced I/O Scripts Page (Screen 2.1.9.2) will display a message explaining the line number and description of a runtime error. On the Web Interface, the Status column will show "Error" and the "Info" box will show the runtime error when the magnifying class icon is clicked (on the right side "Actions" column).

Script Debugging

- A simple method to debug a script has been made available by allowing any expression from the Scripter to be printed to the script output screen 2.1.9.2 (press D key as instructed), and to the Web Interface viewed by clicking on the "magnifying glass" icon. The Script output is also available from NTCIP as a scriptOutput object.
- The print function allows any expression to be printed **separated by commas**. Additionally, strings can be printed by inserting text to be displayed in quotes.
- For, example to display the tenth seconds the engine has been running since startup followed by a message to display what the state of phase 1 green is:

```
print(ENGUP(),": ","PHASE 1 GREEN IS ", PHGRN(1))
```

When the above **`print`** statement is executed, the value returned by ENGUP is printed (see Section 3.3), then a colon and space is printed, then PHASE 1 GREEN IS is printed followed by a space, then the value of PHGRN(1) is printed (which is true or false). As the Script runs, below is a small sample of the output from the above print:

```
123: PHASE 1 GREEN IS true
124: PHASE 1 GREEN IS true
125: PHASE 1 GREEN IS true
126: PHASE 1 GREEN IS false
127: PHASE 1 GREEN IS false
```

- To further simplify printing information and the time the traffic engine has been running since power-up the **`tprint`** function is provided which will accomplish the same result as the above **`print`** function with the following statement:

```
tprint("PHASE 1 GREEN IS ", PHGRN(1))
```

- The **`tprint`** function is useful to distinguish between prints during different Script executions. Prints with the same “engine up time” are for the same Script execution.

Bulti-In Script Debug Flags
The Scripter contains 8 debug Flags that can be set in any Script. The values of the 8 Flags are shown on Status Screen 1.1.1 in the “FLAGS” field and Status Screens 1.1.2, 1.1.6.1 and 1.1.6.2 in the “Script Flags” field. If a number is shown, the corresponding Flag is true, otherwise it is false.

- The function for setting the debug Flags is **setDBUGFLAG(x,y,z)** where **x** is the Flag number (1-8) and **y** is the value to set the Flag (0/false/off or 1/true/on) and **z** is a timeout value in tenth seconds. If **z** is 0, the timeout is disabled, causing the Flag to remain ON value until the Script turns it OFF.
- The timeout is applied only when turning a Flag ON.
- The timeout is reset every time a Script turns the corresponding Flag ON.
- Each Flag maintains its value set by a Script until 1) a Script sets it to another value, or 2) the timeout expires causing the Flag to be set to 0/false/off.
- The same Flag can be used in more than one Script, and in Input and/or Output Type Scripts.

Example:
To verify the Script correctly “sees” Phase 1 green, this Script contents could be written, using Flag #2 to debug the Script:

```
if PHGRN(1) then
  setDBUGFLAG(2,1,0)
  other script content goes here
else
  setDBUGFLAG(2,0,0)
end
```

When Phase 1 is green, Screen 1.1.1 will show “2” on the “FLAGS” row, indicating Flag #2 is true/ON, which indicates the Script “sees” Phase 1 green. When Phase 1 is not green (yellow or red), Screen 1.1.1 will show <blank> on the “FLAGS” row, indicating Flag #2 is false/OFF, which indicates the Script correctly does not “see” Phase 1 green.

Example Using the Flag Timeout:
To verify the Script correctly "sees" Phase 1 change to green, this Script contents could be written, using Flag #2 to debug the Script:

```
if TURNEDON(PHGRN(1)) then
  setDBUGFLAG(2,1,30)
  other script content goes here
end
```

When Phase 1 turns green, Flag #2 will be set ON for 30 tenth seconds (3 seconds) then automatically turn OFF. The Script does not need to explicitly turn Flag #2 OFF. Without the timeout, the Script would need to clear Flag #2, like this script below:

```
if TURNEDON(PHGRN(1)) then
  setDBUGFLAG(2,1,0)
  other script content goes here
else
  setDBUGFLAG(2,0,0)
end
```

The above if-statement will only be true for one pass thru the Script, which is 0.1 seconds. This results in Flag #2 showing active on the status screen for only 0.1 seconds. The timeout allows the Flag to show true/ON longer to avoid potentially missing the flicker of "2" (indicating Flag #2 is ON) on the status screen, without the user having to create more Script content to control how long the Flag shows a true/ON status.

The above examples are for illustrative purposes. In practice, your Script might have complex logic with mistakes, and these Debug Flags can help correct the Script and verify correct controller operation.

2.7 User-Defined Variables for Latches/Flags, Counters, Timers

User-defined variables must be used for **latches/flags** and **counters.** For **timers**, if the ELAPSED and DELAY functions described in Section 3.3 cannot do the job, then a variable must be used to act as a **timer**. The user can name variables according to preference, but it may not contain spaces and must begin with an alphanumeric character and contain only letters, numbers, and underscores. Variables maintain their values between script executions until set again or the controller is rebooted. Some variables have been pre-defined. For example, **ON** and **on** have been pre-defined to **true** and **OFF** and **off** have been pre-defined to **false**. This means you can use "ON" or "on" instead of "true" and use "OFF" or "off" instead of "false" according to preference.

Note All variables made by the user **should be initialized to a value** using the SCRIPTSTART function (shown in the examples in this document), otherwise a runtime error might occur.

Tip Variables made by the user are used to act as a "latch" or "flag" to remember certain conditions exist, and perform certain actions during those conditions, or as a "counter" to count the occurrence of an event (ex, count detector actuations), or as a "timer" to measure the duration of certain conditions (ex, detector presence timer).

Caution Variables created by the user can be set to and tested for any value in ANY of the 4 Input Scripts and 4 Outputs Scripts. The user must use a different variable name in each Script if the variables must be separately used otherwise unwanted side effects will occur.

Example Using a Variable for a Latch (or Flag):
Create and use the variable **`latch4`** as a true/false type variable to act as a "latch/flag" to remember Pedestrian Detector 1 went active:

```
if SCRIPTSTART() then
  latch4 = false
end

if PEDDET(1) then
  latch4 = true
end

if PHWLK(4) then
  latch4 = false
elseif latch4 then
  setPEDOMIT(8,true)
```

```
end
```

In the above script, the variable **latch4** is created and initialized to **false** when the script starts (see SCRIPTSTART in Section 3.3).

When Ped Detector 1 is active (see PEDDET in Section 3.1.7), **latch4** is set true and it maintains this value until Phase 4 Walk is on (see PHWLK in Section 3.1.1), where it gets set to **false**. While it is **true** and Phase 4 walk is NOT on, the script evaluates the **elseif** and because **latch4** is true, the expression becomes **elseif true**, causing the script to set a Phase 8 Ped Omit (see setPEDOMIT in Section 3.2.1).

Example Using a Variable for a Count-up Timer:
Suppose you want to activate Preemption Run 2 if Detector 12 is occupied for 2 minutes. You would create a variable to keep track of how long Detector 12 is occupied. In this example, the variable **Timer** is created and used as follows:

```
if SCRIPTSTART() then
  Timer = 0
end

if DET(12) then
  Timer = Timer + 1
else
  Timer = 0
end

if Timer >= 1200 then
  setPMTCALL(2,on)
  Timer = 0
end
```

In the above script, the variable **Timer** is created and initialized to 0 when the script starts (see SCRIPTSTART in Section 3.3).

The timer is controlled by the variable **Timer**. Because the script executes once every 0.1 seconds, Timer can be incremented by 1 with each script execution when DET(12) returns true, to create a tenth-second count-up timer. See DET in Section 3.1.7.

If the Timer reaches 1200, preempt 2 is called (see setPMTCALL function in Section 3.2.7) and the Timer is reset to 0 to start a fresh presence count (1200 is the tenth seconds equivalent of 2 minutes: 120 seconds in 2 minutes, times 10 because the timer must be in tenth seconds).

In this example, the DELAY() function can be used instead of the variable **Timer** – see Section 3.3 Example A.

Example Above Changed to Use a Variable for a Count-down Timer:

```
if SCRIPTSTART() then
  Timer = 1200
end

if DET(12) then
  if Timer ~= 0 then
    Timer = Timer - 1
  end
else
  Timer = 1200
end

if Timer == 0 then
  setPMTCALL(2,on)
  Timer = 1200
end
```

In the above script, the variable **Timer** is created and initialized to 1200 when the script starts (see SCRIPTSTART in Section 3.3).

The timer is controlled by the variable **Timer**. Because the script executes once every 0.1 seconds, Timer can be decreased by 1 with each script execution when DET(12) returns true and **Timer** is not already zero, to create a tenth-second count-down timer. In the second if-statement, if Detector 12 is true it does a "nested" if-statement to decrease **Timer** by 1 only if it is not zero – once zero, that indicates a timeout, and we want it to remain at 0 (instead of decreasing **Timer** to a negative number).

If the Timer reaches 0, preempt 2 is called and the Timer is reset to 1200 to start a fresh presence count (1200 is the tenth seconds equivalent of 2 minutes: 120 seconds in 2 minutes, times 10 because the timer must be in tenth seconds).

In this example, the DELAY() function can be used instead of the variable **Timer** – see Section 3.3 Example A.

Example Using a Variable for a Counter:
Suppose if Detector 8 has only 1 or 2 cars waiting during its associated Phase 4 red signal, we want to keep Phase 3 green past its force off point for 10 seconds and the force off point is at 40.0 seconds on the Local Cycle Timer. The script can use a variable to count the number of vehicles, as shown below.

```
if SCRIPTSTART() then
  Count = 0
end

if TURNEDON(DET(8)) then
  Count = Count + 1
end

if PHGRN(4) then
  Count = 0
end

if PHGRN(3) and
   Count <= 2 and
   LOCALCYCLETMR() >= 400 and
   LOCALCYCLETMR() <= 500 then
  INHCOORDFO(3)
end
```

In the above script, the variable **Count** is created and initialized to 0 when the script starts (see SCRIPTSTART in Section 3.3).

When Detector 8 turns on, the **Count** variable is incremented by 1 to count the vehicle (second “if statement”). See DET in Section 3.1.7.

When Phase 4 is green, the **Count** is set to 0 because we do not want to count cars during green (third “if statement”). See PHGRN in Section 3.1.1.

When Phase 3 is green and **Count** is less than or equal to 2 and the Local Cycle Timer value is greater than or equal to 40.0 and less than or equal to 50.0, then the force off for Phase 3 is inhibited/blocked. When this condition is no longer true, the force off inhibit is removed (because the last if-statement will be false) and the Coordination force off takes effect to terminate Phase 3. See LOCALCYCLETMR in Section 3.1.4 and INHCOORDFO in Section 3.2.1.

2.8 Advanced I/O Scripter: Scripts Using Values Overridden by Other Scripts

Input Script #1 runs first, and Input Script #4 runs last.
Then the Traffic Engine runs.
Then Output Script #1 runs first, and Output Script #4 runs last.

Caution If an Input Type Script overrides an Input, subsequent Script numbers **will use the input's overridden value**. If the subsequent Script requires the original input's value:

1. swap Scripts to make the last Script override the Input,
2. create a variable that the first Script sets equal to the original input's value and the second Script checks in an if-statement,
3. use one Script with statements properly ordered (subject to the restriction of Input Type scripts only setting inputs and Output Type scripts only setting outputs).

The same concept applies to an Output Script overriding an Output.

Oriux does not recommend using the same variable in more than one script.

Example using 2 Input Scripts:
Input Script #1:

```
if PHGRN(1) then
  setPHOMIT(2,ON)
end
```

Input Script #2:

```
if PHOMIT(2) then
  setDET(3,ON)
end
```

In the above example, Input Script #1 runs first and if Phase 1 is Green, Phase 2 is omitted. When Input Script #2 runs, the check for Phase 2 omit will reflect the Phase 2 omit set by Input Script #1. If Script #2 must use the original Phase 2 Omit Input value (before Script #1 runs), do one of the following:

1. Swap the scripts – put Script #1's contents in Script #2 and put Script #2's contents in Script #1.

2. Use a variable (ex, use a variable named "ph2omit") that Script #1 sets and Script #2 checks.
Input Script #1:

```
ph2omit = PHOMIT(2)
if PHGRN(1) then
  setPHOMIT(2,ON)
end
```

Not shown above, ph2omit would be initialized to false with the SCIPTSTART utility function described in Section 3.3.

Input Script #2:

```
if ph2omit then
  setDET(3,ON)
end
```

3. Use one script and change the order: check PHOMIT(2) before doing setPHOMIT(2,ON):
Input Script #1:

```
if PHOMIT(2) then
  setDET(3,ON)
end

if PHGRN(1) then
  setPHOMIT(2,ON)
end
```

2.9 Forbidden Words

The following words are not allowed to be in a script, either alone or with surrounding characters, even in comments:

```
ffi
dostring
loadstring
dofile
loadfile
loadlib
```

Using any of these in a Script will cause an error message.

For example, since “ffi” is not allowed, “tra**ffi**c” is also not allowed.

2.10 Output Function Recommendations

An Output function (for example, one that overrides a phase’s red output to on) must be used carefully. **Oriux does not recommend using Output functions to override the controller’s signal status (Phase, Ped, Overlap, Transit) beyond flashing a signal that would otherwise be steady on. Instead Oriux recommends using Input functions (Input Scripts).**

For example, to prevent a red Phase from turning green, use the Phase Omit Input function instead of using an Output function to override the Phase’s red to on (and green and yellow to off). See Section 3 for all the Scripter functions.

An example of a safe use of Output functions is to flash a signal red **that is normally steady red** if there is no controller setting to do this. In this case the controller's signal status is red, but the field will show a flashing red.

2.11 Clarification of True/False Expressions in If Statements

An **if** statement has the following format:

if *expression* **then**
  statements
end

For simplicity, optional **elseif** is not shown but the same concepts in this section apply. The *expression* must evaluate to an appropriate true/false value, otherwise the Script will not work correctly. Below are two recommended Rules to help prevent incorrect Scripts.

Rule 1: When using functions in an if-statement *expression* that return a number, always use comparison operators (`<`, `<=`, `>`, `>=`) or equality operators (`==`, `~=`) to create the appropriate true/false expression.

The following is an attempt to check if Ring 1's Yellow Timer is **not** zero. Since RINGYELTMR(x) returns a number (see Section 3.1.2), below is a correct if-statement that obeys Rule 1:

```
if RINGYELTMR(1) ~= 0 then
 statements
end
```

Depending on the Ring 1 Yellow timing status, this evaluates to "if X ~= 0 then" where X is the yellow time remaining in tenth seconds.
This true/false expression yields the **correct** result: if the Yellow Timer is 0, this evaluates to "if 0 ~= 0" which is false, and if the Yellow Timer is non-zero (20 for example), this evaluates to "if 20 ~= 0" which is true.

The following is a wrong way to check if Ring 1's Yellow Timer is **not** zero:

```
if RINGYELTMR(1) then
 statements
end
```

Depending on the Ring 1 Yellow timing status, this evaluates to "if X then" where X is the yellow time remaining in tenth seconds.
This true/false expression **is always true,** because all numeric values used in this context cause a true value, yielding the **wrong** result: if the Yellow Timer is 0, this evaluates to "if 0 then" which is true, and if the Yellow Timer is

non-zero (20 for example), this evaluates to “if 20 then” which is also true.

Rule 2: In an if-statement *expression*, numerical values should not be mixed with functions that return true or false.

The following if-statement is an attempt to check if Detector 17 is active. Since DET(x) is a function that returns true or false (see Section 3.1.7), below are two correct if-statements that obey Rule 2 (use either according to preference):

```
if DET(17) == true then
 statements
end
```

Depending on the status of Detector 17, this evaluates to “if true == true then” or, “if false == true then”, which yields the **correct** result.

```
if DET(17) then
 statements
end
```

Depending on the status of Detector 17, this evaluates to “if true then” or “if false then”, which yields the **correct** result.

The following is another attempt to check if Detector 17 is active, but will not work correctly because it violates Rule 2:

```
if DET(17) == 1 then
 statements
end
```

Depending on the status of Detector 17, this evaluates to “if true == 1 then” or “if false == 1 then”, both of which will NEVER be true because the numerical value 1 is not the same as true or false, which causes the *statements* to never be executed, which yields the **wrong** result.

3. Advanced I/O Scripter Functions

3.1 Testable Items ("get" functions)

Note All functions accepting **ANY** may instead use **any** or **0**.

Note All functions in Section 3.1 work in "input" or "output" type scripts.

3.1.1 PHASE

x = 1-16 (Phase number), or ANY to check "if any Phase number"

Functions that Get Output Function Values *– return true or false.*

PHGRN(x)	**GREEN_ON_PHASE(x)**
PHYEL(x)	**YELLOW_ON_PHASE(x)**
PHRED(x)	**RED_ON_PHASE(x)**
PHON(x)	**PHASE_ON_IS(x)**
PHWRN(x)	**WARNING_ON_PHASE(x)**

NOTE: PHWRN is an output normally used to control an electric sign.

PHNXT(x) **NEXT_ON_PHASE(x)**
Returns true if a Next Decision has been made for Phase x.

Caution PHNXT can change during vehicle clearance intervals if (1) a Preempt activates or (2) if Screen 2.1.7's PHASE NEXT CONTROL equals "nonPersistent(1)" and Phase demand changes. Make sure Scripts using PHNXT consider Preemption and PHASE NEXT CONTROL.

PHCHK(x) **CHECK_ON_PHASE(x)**
Returns false if: (1) closed Coordination Vehicle Permissive window or (2) an active Preempt prevents the phase from running. Note a Check stays on even if omitted.

PHWLK(x)	**WALK_ON_PHASE(x)**
PHFDW(x)	**FDW_ON_PHASE(x)**

NOTE: PHFDW is the pedestrian clearance output (steady ON during ped clear/FDW).

PHDWK(x) **DW_ON_PHASE(x)**

Functions that Get Input Function Values *– return true or false*

PHHOLD(x)	**HOLD_ON_PHASE(x)**
PHOMIT(x)	**OMIT_ON_PHASE(x)**
PEDOMIT(x)	**OMIT_ON_PED(x)**

Functions that Get Internal Traffic Status *– return true or false*

PHFORCEOFF(x) **FORCE_OFF_PHASE(x)**

This includes force-offs from external ring input, coordination, preemption, transit priority and logic scripter.

PHCALL(x) **CALL_ON_PHASE(x)**
Returns true if a Phase call exists without an omit from: Detectors, TSP, Screen 2.2.9 Recalls, Split Mode Recalls, TOD Recalls, Preempt, Exclusive Ped Soft Return Recalls, and Anti-Backup Dynamic Recalls, otherwise returns false.

PEDCALL(x) **CALL_ON_PED()**
Returns true if a Ped call exists without an omit from: Detectors, Screen 2.2.9 Recalls, Split Mode Recalls, TOD Recalls, and Preempt, otherwise returns false.

PHDETFAIL(x) **DETECTOR_FAIL_ON_PHASE(x)**
Returns true when a Phase's Detector is "failed."

PEDCHECK(x) **CHECK_ON_PED(x)**
Returns false if: (1) closed Coordination Pedestrian Permissive window or (2) an active Preempt prevents the PED from running. Note PED Check stays on even if Ped Omit.

PHWALKMODE(x)
Returns a string indicating the Walk Mode in effect for Phase x, as follows:

"NEMA"	green and walk start simultaneously
"DLYG"	green delayed (early walk, leading pedestrian interval/LPI)
"DLYW"	walk delayed (early green, leading green)
"DLYGW"	green and walk delayed
"GW"	green warning starts with walk

PHWALKMODE (x) example: apply phase 8 call if Phase 4 Walk Mode is "delayed walk". The Script must contain quotes around DLYW.

```
if PHWALKMODE(4) == "DLYW" then
  setPHSCALL(8)
end
```

Functions that get Phase Times in effect for Phase x (based on default, timing plan, commanded action, preempt). All functions return the time in tenth seconds. The "_T" indicates "Time".

PHMINGRN_T(x)	Minimum Green time
PHMAXGRN_T(x)	Maximum Green time
PHGRNWARN_T(x)	Green Warning time
PHMINWALK_T(x)	Minimum Walk time
PHPCLR_T(x)	Pedestrian Clearance time
PHSDWCLR_T(x)	Green Steady Don't Walk Clearance time
PHYELWARN_T(x)	Yellow Warning time
PHYEL_T(x)	Yellow Time
PHRED_T(x)	Red Clearance time
PHGRNDELAY_T(x)	Green Delay time
PHWALKDELAY_T(x)	Walk Delay time

3.1.2 RING

x = 1-4 (Ring number), or ANY to check "if any Ring number"

RING TIMERS (returns a number counting down in tenth seconds)

RINGMINGRNTMR(x)	**MINIMUM_GREEN_TIMER_FOR_RING(x)**
RINGMAXGRNTMR(x)	**MAXIMUM_GREEN_TIMER_FOR_RING(x)**
RINGGAPTMR(x)	**GAP_TIMER_FOR_RING(x)**
RINGYELWARNTMR(x)	**YELLOW_WARNING_TIMER_FOR_RING(x)**
RINGYELTMR(x)	**YELLOW_TIMER_FOR_RING(x)**
RINGREDTMR(x)	**RED_TIMER_FOR_RING(x)**
RINGWLKTMR(x)	**WALK_TIMER_FOR_RING(x)**
RINGFDWTMR(x)	**FDW_TIMER_FOR_RING(x)**

RINGSDWTMR(x) **SDW_TIMER_FOR_RING(x)**

The time remaining in the phase's steady don't walk clearance timer.

RINGWLKMODEWLKTMR(x)

This timer indicates the amount of time remaining in the delayed **green** or delayed **walk** interval, according to the Walk Mode in effect (Screen 2.2.3 or Time of Day), as follows:

Walk Mode in effect	**RINGWLKMODEWLKTMR definition**
nema	not applicable
GW	not applicable
dGrn	delayed green time remaining
dWlk	delayed walk time remaining
dWG	delayed walk time remaining

RINGWLKMODEGRNTMR(x)

This timer indicates the amount of time remaining in the delayed **green** interval ONLY when the Walk Mode in effect (Screen 2.2.3 or Time of Day) is set to **dWG**. In **dWG** mode, the walk and green are delayed using separate time settings and the concurrent **walk** delay time remaining is indicated by the above **RINGWLKMODEWLKTMR(x)**.

RING (returns ON/OFF)

Functions that Get Input Function Values *– return true or false*

INHMAX(x)	**INHIBIT_MAX_ON_RING(x)**
MAX2(x)	**MAX2_FOR_RING(x)**
OMITRED(x)	**OMIT_RED_ON_RING(x)**
REDREST(x)	**RED_REST_ON_RING(x)**
PEDRECYCLE(x)	**PED_RECYCLE_ON_RING(x)**
RINGFORCE (x)	**FORCE_OFF_RING(x)**
STOPTIME(x)	**STOP_TIME_ON_RING(x)**

Functions that Get Internal Traffic Status

RINGPHON(x) **PHASE_ON_IN_RING(x)**

Returns the phase number of the phase on in ring x or 0 if no phase on in ring.

RINGSTATE(x) **STATE_OF_RING(x)**

Returns the phase state for the phase that is on in Ring x corresponding to one of the following String values (first part is vehicle and second part is pedestrian):

"REDREST" No Phase is timing in the Ring.
"REDWALK" Red with ped in Walk.
"REDFDW" Red with ped in FDW/Ped Clearance.
"REDDW" Red with ped at Steady Don't Walk.
"YWNWALK" Yellow **Warning** with ped in Walk.
"YWNFDW" Yellow **Warning** with ped in FDW/Ped Clearance.
"YWNDW" Yellow **Warning** with ped at Steady Don't Walk.
"YELWALK" Yellow with ped in Walk.
"YELFDW" Yellow with ped in FDW/Ped Clearance.
"YELDW" Yellow with ped at Steady Don't Walk.
"GRNWALK" Green with ped in Walk.
"GRNFDW" Green with ped in FDW/Ped Clearance.
"GRNDW" Green with ped at Steady Don't Walk.
"DLYWALK" Green delayed while timing Walk.
"DLYFDW" Green delayed while timing Ped Clearance.
"GRNDLY" Walk delayed while timing Green.
"GWNWALK" Green **Warning** with ped in Walk.
"GWNFDW" Green **Warning** with ped in FDW/Ped Clearance.
"GWNDW" Green **Warning** with ped at Steady Don't Walk.
"UNUSED" Ring contains no Phases (not used).

RINGSTATE(x) example: apply phase 2 call if Ring 1 in Red Rest:

```
if RINGSTATE(1) == "REDREST" then
  setPHSCALL(2)
end
```

Script must contain quotes around REDREST.

RINGREMAIN(x) **STATE_TIME_FOR_RING(x)**
Returns the number of tenth seconds remaining in Ring x's active Phase's current interval. If Ring x is in green/steady don't walk, and phase is green, it returns the Minimum Green time remaining or if the phase is in Green Warning, it returns the Green Warning time remaining.

Functions that Get Output Function Values *– return true or false*

RINGSTATA(x)	**STATUS_BIT_A_RING(x)**
RINGSTATB(x)	**STATUS_BIT_B_RING(x)**
RINGSTATC(x)	**STATUS_BIT_C_RING(x)**

3.1.3 VEH AND PED OVERLAP

x = 1-32 (Veh Overlap number), 1-16 (Ped Overlap number), or ANY to check "if any Veh/Ped Overlap number"

Functions that Get Output Function Values *– return true or false*

OLGRN(x)	**GREEN_ON_OVERLAP(x)**
OLYEL(x)	**YELLOW_ON_OVERLAP(x)**
OLRED(x)	**RED_ON_OVERLAP(x)**
OLWRN(x)	**WARNING_ON_OVERLAP(x)**

NOTE: OLWRN is an output normally used to control an electric sign.

OLWLK(x)	**WALK_ON_PED_OVERLAP(x)**
OLFDW(x)	**FDW_ON_PED_OVERLAP(x)**

NOTE: OLFDW is the pedestrian clearance output (steady ON during ped clear/FDW)

OLDW(x)	**DONT_WALK_ON_PED_OVERLAP(x)**

Functions that Get Internal Traffic Status *– return true or false*

OLOMIT(x)	**OMIT_ON_OVERLAP(x)**
OLCALL(x)	**CALL_ON_OVERLAP(x)**
OLFORCE(x)	**FORCE_OFF_ON_OVERLAP(x)**
OLON(x)	**OVERLAP_ON_IS(x)**

Returns true if the Vehicle Overlap x is timing (including Red Clearance but NOT including Red Revert), otherwise returns false.

OLSTATE(x)	**OVERLAP_STATE(x)**

Returns one of the following String values:

"DARK"	Green, Yellow and Red outputs Off Note: This state is returned when Overlap is disabled or is enabled and the Overlap Type causes a Dark state)
"GRN"	green and following Parent Phases
"YEL"	Yellow Clearance
"RED"	Red Clearance
"OFF"	Red and not timing (red rest)
"LEADGRN"	Leading Green interval
"GRNDELAY"	Overlap Red while timing Delay (Overlap green is delayed)
"TRAILGRN"	Trail Green interval
"YELWRN"	Yellow Warning interval (Overlap is green, either steady or flashing, per Yellow Warning Flash Rate on Screen 2.2.2).

OLSTATE(x) example: apply phase 2 call if Overlap 1 is timing Leading Green:

```
if OLSTATE(1) == "LEADGRN" then
  setPHSCALL(2)
end
```

Script must contain quotes around LEADGRN.

Functions that get Overlap Times in effect for Vehicle Overlap x (based on default, commanded action, preempt). All functions return the time in tenth seconds. The "_T" indicates "Time".

VOLMINGRN_T(x)	Minimum Green time
VOLGRNWARN_T(x)	Minimum Green Warning time
VOLTRAILGRN_T(x)	Trail Green time
VOLALTTRAILGRN_T(x)	Alternate Trail Green time
VOLMAXTRAILGRN_T(x)	Maximum Trail Green time
VOLALTMAXTRAILGRN_T(x)	Alternate Maximum Trail Green time
VOLYELWARN_T(x)	Yellow Warning time
VOLYEL_T(x)	Yellow time
VOLRED_T(x)	Red Clearance time
VOLLEADDELAY_T(x)	Lead/Delay time

POLSTATE(x) **PED_OVERLAP_STATE(x)**
Returns one of the following String values:

"WALK"	in walk timing min walk or a Parent Phase is in walk.
"XFER"	in walk while transferring between Parent Phases.
"FDW"	ped clearance (flashing don't walk).
"TRLWLK"	running trail walk interval.
"DW"	in timed-out steady don't walk.
"DWCLR"	timing steady don't walk clearance.

POLSTATE(x) example: apply phase 2 call if Ped Overlap 1 is timing Trail Walk:

```
if POLSTATE(1) == "TRLWLK" then
  setPHSCALL(2)
end
```

Script must contain quotes around TRLWLK.

POLON(x)
Returns true if the Ped Overlap x is timing (including Steady Don't Walk clearance), otherwise returns false.

Functions that get Overlap Times in effect for Pedestrian Overlap x (based on default, commanded action, preempt). All functions return the time in tenth seconds. The "_T" indicates "Time".

POLWALK_T(x)	Minimum walk time
POLPEDCLEAR_T(x)	Pedestrian Clearance time
POLTRAILWALK_T(x)	Trail Walk time
POLGRNDWKCLEAR_T(x)	Green Steady Don't Walk Clearance time

3.1.4 COORDINATION

Empty parenthesis for all functions

COORDSTATE() **COORD_STATE_VALUE()**
Returns one of the following String values:

"FREE"	commanded free or unable to enter coordination yet.
"XFER"	waiting for Greens to turn Yellow to enter Coordination Mode
"1CYCLE"	less than 1 cycle since it got in sync.
"INSYNC"	has been in sync for more than 1 cycle.
"DWELL"	offset correcting by increasing the coordinated phase green time. See NOTE 2 below.
"FAST"	offset correcting by decreasing split times with "fast seconds."
"SLOW"	offset correcting by increasing split times with "slow seconds."
"RDC"	offset correcting by decreasing split times with "real seconds."
"ADD"	offset correcting by increasing split times with "real seconds."
"RETRY"	trying Coordination again after a "FAULT" (Cycle Fault)
"FAULT"	Cycle Fault from skipping a Phase call for 2 consecutive cycles.

“BADPLAN” Running free due to invalid pattern settings.
“FAILURE” Two consecutive Cycle Faults causing Free Coord Failure
“TSPACT” TSP Request active and cycling to or in the TSP Phases
“TSPREC+” TSP recovering offset by increasing split times w/ “real seconds.”
“TSPREC-” TSP recovering offset by decreasing split times w/ “real seconds.”

NOTE 1: “DWELL”, “FAST”, “SLOW” and “XFER” are displayed only when Screen 2.8.1 TSP ENABLE is set “OFF” and offset correction is active, and “ADD” and “RDC” are displayed only when TSP ENABLE is set “ON” and offset correction is active, and the offset correction is not caused by a TSP call.

NOTE 2 (dwelling): When Screen 2.8.1 TSP ENABLE is set “OFF”, the local cycle timer stops while dwelling, thereby increasing the cycle, and the “DWELL” string will be returned. When TSP ENABLE is set “ON”, instead of stopping the local cycle timer, the coordinated phase commanded split increases, which has the same effect of increasing the cycle, and the “ADD” string will be returned.

COORDSTATE() example: apply phase 3 Omit if Coordination is in offset correction and shortening the cycle by speeding up the Local Cycle Timer:

```
if COORDSTATE() == "FAST" then
  setPHOMIT(3)
end
```

Script must contain quotes around FAST.

Functions below return a number.

ACTPTN()	**ACTIVE_PATTERN_NUMBER()**
ACTSPLITNUM()	**ACTIVE_SPLIT_NUMBER()**

Functions below return a number represented as tenth seconds.

ACTCYCTIME()	**ACTIVE_CYCLE_TIME()**
ACTOFFSETTIME()	**ACTIVE_CYCLE_OFFSET()**
LOCALCYCLETMR()	**LOCAL_CYCLE_TIMER()**
MASTERCYCLETMR()	**MASTER_CYCLE_TIMER()**

OFFSERR() the current Offset Error (zero when in sync).

Functions that get calculated Coordination parameters for Phase x

FORCEPOINT(x)
Returns the fixed force off point Local Cycle Timer value in tenth seconds.

CMDSPLIT(x)
Returns the commanded Split time in tenth seconds, based on Pattern command and CIC. If TSP is enabled, this includes the temporary Commanded Splits during Offset Correction.

PATTSPLIT(x)
Returns the Pattern Split time in tenth seconds.

VPERMOPEN(x)
Returns true of Phase x Vehicle Permissive Window is open, otherwise false.

PPERMOPEN(x)
Returns true of Phase x Pedestrian Permissive Window is open, otherwise false.

3.1.5 UNIT
Empty parenthesis for all functions except ALARM(x)

Functions that Get Internal Traffic Status

ACTVSEQNUM() **ACTIVE_SEQUENCE_NUMBER()**
Returns the Sequence Number in effect.

GETSTFLSHTMR() **GET_STARTUP_FLASH_TIMER()**
Returns the startup flash time remaining in tenth seconds.

FLSHSTAT() **FLASH_STATUS ()**
Returns true if the controller is in Flash. Returns false when cycling to Flash Entry Phases.

DYNMAX()
Returns true if **any Phase that is timing** (including clearance) has a Dynamic Max green time in effect that is smaller or larger than the "base" maximum green time (Max 1, Max 2, or "Zero Cycle Time" Free Pattern Split Table Max), otherwise returns false.
Example: Omit Phase 3 if Phase 2 is timing and Dynamix Max is in effect for Phase 2:

```
if PHON(2) and DYNMAX() then
  setPHOMIT(3,true)
end
```

Functions that Get Input Function Values

AUTOFLSH() **AUTO_FLASH()**
Returns true if Flash Input active (could be cycling to the Flash Entry Phases and not yet in the flashing state).

MINRECALL() **MINIMUM_RECALL()**

EXTSTART() **EXTERNAL_START()**
EXTSTART returns true if the external start input is active, or the Monitor (TS2 MMU, or ATC Cabinet CMU, or C5000 System Module) sends the controller the restart command (at cabinet power-up, or after a fault is cleared, or the local flash input is deactivated).

GOFREE() **GO_FREE()**
GOFREE returns true if the "high priority free input" is active. This input has a higher priority than system/central commands (ex, Mizbat, NTCIP). Only preempt, soft flash input, or CVM flash input have higher priority.

GOFREELOPRIO()

GOFREELOPRIO returns true if the “low priority free input” is active. This input has a lower priority than system/central commands. Only the time-of-day schedule has a lower priority than this input.

CNA1()	**CALL_TO_NONACTUATED_1()**
CNA2()	**CALL_TO_NONACTUATED_2()**
INTADV()	**INTERVAL_ADVANCE()**
MCE()	**MANUAL_CONTROL_ENABLE()**
WRM()	**WALK_REST_MODIFIER()**
INDLAMP()	**INDICATOR_LAMP()**
TESTINA()	**TEST_INPUT_A()**
TESTINB()	**TEST_INPUT_B()**
TESTINC()	**TEST_INPUT_C()**

The above functions return true if the input is active, otherwise return false.

ALARM(x)	**ALARM(x) x = alarm number (1-8) or ANY to check “if any Alarm number.”**

Returns true if the associated Alarm input is active, otherwise returns false.

3.1.6 PREEMPT / TRANSIT PRIORITY

x = 1-32 (Preempt Run number), 1-8 TSP Run number, or ANY to check “if any Preempt/TSP Run number”

Functions that Get Input Function Values *– return true or false*

PMTCALL(x)	**CALL_ON_PREEMPT(x)**
PMTGATE(x)	**PMT_GATE_INPUT(x)**
TSPCHKIN(x)	**TSP_CHECK_IN_ON_RUN(x)**
TSPCHKOUT(x)	**TSP_CHECK_OUT_ON_RUN(x)**
TSPCANCEL(x)	**TSP_ADVANCE_CANCEL_ON_RUN(x)**

Functions that Get Output Function Values *– return true or false*

PMTSTAT (x)	**ACTIVE_PREEMPT(x)**
TSPOUT(x)	**TSP_RUN_OUTPUT(x)**
QJMP(x)	**TSP_QJUMP_ON_RUN(x)**

Transit Signals (vertical bar, clearance triangle, horizontal bar):

TSPVBAR(x)	**VERT_BAR_ON_TSP(x)**
TSPCLT(x)	**CLR_TRIANGLE_ON_TSP(x)**
TSPHBAR(x)	**HORIZ_BAR_ON_TSP(x)**

Functions that Get Internal Traffic Status

TSPEAT(x): returns the estimated arrival time for TSP Run x. For field input calls, this is the Arrival Time programmed in the TSP Run Configuration. For NTCIP remote calls, this is the time specified in the NTCIP message to the controller.

TSPSIGSTAT(x): returns the signal status of TSP Run x in one of these strings:

- **“SHB”** steady horizontal bar
- **“FHB”** flashing horizontal bar
- **“SVB”** steady vertical bar
- **“CLR”** clearance – flashing vertical bar or triangle, in the applicable mode.

TSPSIGSTAT(x) example: apply Phase 6 Hold if TSP Run 4 signal is in Steady Vertical Bar:

```
if TSPSIGSTAT(4) == "SVB" then
  setPHHOLD(6)
end
```

Script must contain quotes around SVB.

TSPRUNSTAT(x): returns the TSP Run Status as one of these strings, corresponding to the "Run Status" shown on Status Screens 1.1.2 and 1.1.6.1:

"REQS"	TSP Request (call)
"RSVC"	Timing Reservice Time (TSP call locked out)
"SUCC"	Success. TSP call dropped during green extension
"CSUC"	Clearance Success. TSP call dropped during vehicle clearance after maximum green extension occurred.
"REMV"	TSP call removed without green extension occurring.
"CLRF"	Clearance Failure: TSP call remains after maximum green extension and vehicle clearance ends.
"DETF"	TSP detector Fail time exceeded (TSP call locked out)
"DETC"	Failed TSP detector was reset because TSP input turned off.
"DELY"	Timing input delay
"EXTD"	Timing input extend (NOT green extension)
"OVRD"	Priority override from Preempt or Flash
"GEXT"	Timing TSP green extension
"INVL"	TSP run invalid because no TSP Phases or TSP Phases are not mutually compatible.

REMTSP(x): returns true if TSP Run x has a remote call thru communications or thru the Script **setREMTSP** function, otherwise returns false.

The functions below return the amount of tenth seconds remaining:

PMTDELAYTMER(x)	**DELAY_TIMER_FOR_PREEMPT(x)**
PMTMAXPRESTMR(x)	**MAX_PRESENCE_TIMER_FOR_PREEMPT(x)**
PMTMINTMR(x)	**MIN_TIMER_FOR_PREEMPT(x)**
PMTDWELLTMR(x)	**DWELL_TIMER_FOR_PREEMPT(x)**
PMTTRACKGRNTMR(x)	**TRACK_GREEN_TIMER_FOR_PREEMPT(x)**

PMTRUNSTATE(x) **PREEMPT_RUN_STATE(x)**

Returns one of the following String values:

"IDLE"	Preempt not active and no Preempt call.
"NOT ACTIVE"	Same as "IDLE" above.
"NOT ACTIVE W CALL"	Preempt call and service not started (timing delay).
"ENTRY"	Preempt active clearing to first preempt phases.
"TRACK"	Preempt active timing track clearance.
"DWELL"	Preempt active in dwell/cycling stage.
"LINK"	Preempt active and Linked Preempt in service.
"EXIT"	Preempt call dropped and waiting for exit phases.
"MAX PRES"	Preempt in a max presence failure.

PMTRUNSTATE(x) example: activate Alarm 6 if Preempt 4 is in the Max Presence timeout condition:

```
if PMTRUNSTATE(4) == "MAX PRES" then
  setALARM(6,on)
end
```

Script must contain quotes around MAX PRES.

3.1.7 DETECTORS

x = 1-64 (Veh Detector number), 1-8 (Ped Detector number) or ANY to check "if any Veh/Ped Detector number"

Functions that Get Input Function Values *– return true or false*

DET(x)	**CALL_ON_DETECTOR(x)**
PEDDET(x)	**CALL_ON_PED_DETECTOR(x)**

Functions that Get Internal Traffic Status

DETFAIL(x)	**FAILURE_ON_DETECTOR(x)**
DETDELAYTMR(x)	**DELAY_TIMER_FOR_DETECTOR(x)**
DETEXTENDTMR(x)	**EXTEND_TIMER_FOR_DETECTOR(x)**
PEDDETABSFAIL(x)	**ABSENCE_FAIL_ON_PED_DETECTOR(x)**
PEDDETLOCKFAIL(x)	**LOCKED_FAIL_ON_PED_DETECTOR(x)**
PEDDETERRATICFAIL(x)	**ERRATIC_FAIL_ON_PED_DETECTOR(x)**

DETVOL(x) **VOLUME_ON_DETECTOR(x)**
Returns the vehicle detector volume count for enabled Detector Number x collected during the last NTCIP Log Period (see Screen 2.5.8).

DETOCCY(x) **OCCUPANCY_ON_DETECTOR(x)**
Returns the vehicle detector occupancy for enabled Detector x collected during the last NTCIP Log Period (see Screen 2.5.8). During a detector fault, it returns:
210 = Max Presence Fault
211 = No Activity Fault
212 = Open Loop Fault
213 = Shorted Loop Fault
214 = Excessive Change Fault
216 = Watchdog Fault
217 = Erratic count Fault
Note: The highest number is reported when there is more than one fault active.

3.1.8 TIME OF DAY

Empty parenthesis for all functions except TODCMDACTION(x)

TODYEAR()	**TOD_YEAR()**

Returns 4-digit year (ex, 2012)

TODMON()	**TOD_MONTH()**

Returns 1 – 12 (January = 1 – December = 12)

TODDOM() **TOD_DAY_OF_MONTH()**
Returns 1-31

TODDOW() **TOD_DAY_OF_WEEK()**
Returns 0 – 6 (Sunday = 0 – Saturday = 6)

TODHOUR() **TOD_HOUR()**
TODMIN() **TOD_MIN()**
TODSEC() **TOD_SEC()**
TODTENTHS() **TOD_TENTH_SECOND()**
TODDAYPLAN() **TOD_ACTIVE_DAY_PLAN()**
TODACTNPLAN() **TOD_ACTIVE_ACTION_PLAN()**
TODTSPPLAN() **TOD_ACTIVE_TSP_PLAN()**
TODEVENT() **TOD_ACTIVE_EVENT()**

BACKUPTIMER() **BACKUP_TIMER()**
NTCIP-defined timer, see NTCIP 1202 for more information.

TODCMDACTION(x) **TOD_CMD_ACTION_ACTIVE(x)**
x = 1-8 (TOD Commanded Action Number) or ANY or to check "if any TOD Commanded Action number"
Returns true if Commanded Action x active, otherwise returns false.

TIMGPLAN()
Returns the timing plan number (1-4) in effect or returns 0 if no timing plan is in effect. The value returned reflects the time-of-day programming and the **setTIMGPLAN(x)** script function explained later.

3.1.9 LOAD SWITCH CHANNEL

x = Load Switch Channel number (Screen 2.1.4) 1-32, or ANY to check "if any Load Switch number"

Functions that Get Output Function Values *– return true or false*

LSWRED(x) **RED_ON_LSW(x)**
LSWYEL(x) **YELLOW_ON_LSW(x)**
LSWGRN(x) **GREEN_ON_LSW(x)**

3.1.10 INTERVAL BASED CONTROL

x = Circuit Output Number 1-64, or ANY to check "if any Circuit Output number"

Functions that Get Output Function Values *– return true or false*

CCTOUT(x) **CIRCUIT_OUTPUT(x)**
OUTTIMPLAN(x) **CIRCUIT_OUTPUT_TIME_PLAN(x)**
OUTOFFSET(x) **CIRCUIT_OUTPUT_OFFSET(x)**
OUTSIGPLN(x) **CIRCUIT_OUTPUT_SIGNAL_PLAN(x)**
INTIMPLAN(x)
INOFFSET(x)
INSIGPLN(x)
DIAL2()

DIAL3()
DIAL4()
HMCONLINE()
HMCADVANCE()

3.1.11 UNCATEGORIZED TESTABLE OUTPUT FUNCTIONS

The following functions are listed for completeness but are not documented in detail currently. Contact ORIUX for information.

OADDR(x) – For Illinois Commerce Commission systems only.
DETRESETSLOT(x)
REDMONOFF() – returns the value of the output to disable red monitoring.
TBCAUX(x) – returns the value of TBC Aux Output x (where x is 1 – 3).
WATCHDOG()
PMTSTAT(x) – returns the value of Preempt Output x (where x is 1 – 6).
SPFUNCOUT(x) – returns the value of Special Function Output x (where x is 1 – 8).
SYSCNTRL() – returns the value of the System Control Status Output (where a true value means System Control is active)
VOLTMON()
FAULTMON()
FLSHLOGIC() – returns the value of the Flashing Logic Output (returns true or false).
SYSCOORD()
SYSHWIC()
HWICFLSH()
ICCSAFETY() – For Illinois Commerce Commission systems only.

3.1.12 UNCATEGORIZED TESTABLE INPUT FUNCTION ITEMS

The following functions are listed for completeness but are not documented in detail currently. Contact ORIUX for information.

IADDR(x) – For Illinois Commerce Commission systems only.
CABALARM()
TIMESRC()
SPFUNCIN(x)
SYSADDR(x)
DIMMEN()
TBCONLINE()
RESYNC()
CABFLASH()
ALTSEQ(x)
MMUFLASH()
ABSZERO()
PREEMPTMIR() – For Illinois Commerce Commission systems only.
ICCSAFELOOP() – For Illinois Commerce Commission systems only.

3.2 Settable Items (“set” functions)

Note All functions accepting **ON** may instead use **on**, **true, TRUE** or **1**. All Functions accepting **OFF** may instead use **off**, **false, FALSE** or **0**.

Note Some functions in Section 3.2 work for “input” or “output” type scripts, others work only during “input” type scripts, and others work only during “output” type scripts. Each sub-section indicates which types work.

3.2.1 PHASE

x = 1-16 (Phase number)
y = ON or OFF (some functions do not contain y)

Functions that Override Output Function Values

These work in an “output” type script.

setPHGRN(x, y)	**SET_GREEN_ON_PHASE(x, y)**
setPHYEL(x, y)	**SET_YELLOW_ON_PHASE(x,y)**
setPHRED(x, y)	**SET_RED_ON_PHASE(x,y)**
setPHWLK(x, y)	**SET_WALK_ON_PHASE(x,y)**
setPHFDW(x, y)	**SET_FDW_ON_PHASE(x,y)**

NOTE: PHFDW is the pedestrian clearance output (steady ON during ped clear/FDW).

setPHDW (x, y)	**SET_DWK_ON_PHASE(x,y)**

Note The above functions will NOT also set the corresponding Channel/Load Switch output. See Section 3.2.3 LOAD SWITCH for setting both the Phase output and Load Switch output using one function.

setPHCHK(x, y)	**SET_CHECK_ON_PHASE(x,y)**
setPHNXT(x, y)	**SET_NEXT_ON_PHASE(x,y)**
setPHWRN(x,y)	**SET_WARNING_ON_PHASE(x,y)**

NOTE: PHWRN is an output normally used to control an electric sign.

Functions that Override Input Functions Values

These work in an “input” type script.

setPHHOLD (x,y)	**SET_HOLD_ON_PHASE(x,y)**
setPHOMIT(x,y)	**SET_OMIT_ON_PHASE(x,y)**
setPEDOMIT(x,y)	**SET_OMIT_ON_PED(x,y)**

Functions that Override Internal Traffic Input Status

These work in an “input” or “output” type script.

setPHSCALL(x) **SET_CALL_ON_PHASE(x)**
All Phase Omit sources and active Preempt Runs stop the call commanded by this function.

setPEDCALL(x) **SET_CALL_ON_PED(x)**
All Ped Omit sources and active Preempt Runs stop the call commanded by this function.

OVRDPHCALL(x,y) **OVRD_CALL_ON_PHASE(x,y)**
Forces a Phase x Call ON or OFF (overriding all Phase Omit sources). Overriding a Phase call to OFF cannot stop the following phases from turning on:
during Preemption, Track, Dwell and Exit Phases.
during Flash Entry Phases (including changing to Interval Mode).

FRZRINGVEH will prevent OVRDPHCALL from turning the phase green.

OVRDPEDCALL(x,y) **OVRD_CALL_ON_PED(x,y)**
Same as OVRDPHCALL() except forces a Phase x PED call ON or OFF.

Caution OVRDPHCALL and OVRDPEDCALL can (1) delay Preempt if it forces a call on an intermediate phase and (2) make a phase/ped run outside of its Coordination Vehicle/Ped Permissive window.

This will remove a ped call latched in memory.

INHCOORDFO(x) **INH_COORD_FORCE_OFF(x)**
Prevents Coordination from Forcing-off phase x. Coordination maintains its Force Off in the background and upon removal of INHCOORDFO(x), Phase x will Force-terminate. The coordination status screen 1.1.2 shows “L” on the “Hold-FO” row when this function prevents Coordination from Forcing-off a phase.

Caution INHCOORDFO can make a phase to over-run its Force Off Point and skip remaining phases and/or return late to the Coordinated Phases. When used responsibly the user can achieve more efficient Phase service. Example: if Phase 3 is green and the Local Cycle Timer is within a desired range and Phase 4 has 3 or less cars waiting, the user can postpone Phase 3’s force off to let Phase 3 steal time from Phase 4.

INHGRNDLY(x)
This function inhibits/omits the green delay interval for Phase x as follows:
If the Walk Mode is set to dGrn (Delay Green), then the Phase will NOT time a leading walk interval. If the Walk Mode is set to dWG (Delay Walk and Green) and the green delay time is greater than the walk delay time, then the Phase Green and Walk will start at the same time, when the walk delay time ends. This function does not affect a Phase already timing and therefore must be applied by the script before Phase x starts timing. When the controller turns Phase x ON, it checks if this script function is active to determine whether to start the Phase with a delayed green interval. Below is an example

of a structure that can be used if you want to inhibit the green delay for Phase 2 during "some condition" using an Input-type script:

```
if not PHON(2) and
   < some condition > then
  INHGRNDLY(2)
end
```

The function PHON(2) will return **false** when Phase 2 is first turned on, because the Traffic Engine has not run yet and therefore has not updated the Phase 2 ON status to true. See Section 2.6.

3.2.2 VEH AND PED OVERLAP

x = 1-16 (Ped Overlap number) or 1-32 (Veh Overlap number)
y = ON or OFF

Functions that Override Output Function Values

These work in an "output" type script.

setOLGRN(x,y)	**SET_GREEN_ON_OVERLAP(x,y)**
setOLYEL(x,y)	**SET_YELLOW_ON_OVERLAP(x,y)**
setOLRED(x,y)	**SET_RED_ON_OVERLAP(x,y)**
setOLDW(x, y)	**SET_DWK_ON_PED_OVERLAP(x,y)**
setOLFDW(x, y)	**SET_FDW_ON_PED_OVERLAP(x,y)**

NOTE: OLFDW is the pedestrian clearance output (steady ON during ped clear/FDW).

setOLWLK(x, y)	**SET_WLK_ON_PED_OVERLAP(x,y)**

Note The above functions will NOT also set the corresponding Channel/Load Switch output. See Section 3.2.3 LOAD SWITCH for setting both the Overlap output and Load Switch output using one function.

setOLWRN(x,y)	**SET_WARNING_ON_OVERLAP(x,y)**

NOTE: OLWRN is an output normally used to control an electric sign.

Functions that Override Internal Traffic Input Status

These work in an "input" or "output" type script.

setOLFORCE(x) **SET_FORCE_OFF_ON_OVERLAP(x)**
Forces a green Overlap to yellow (abandons green Parent Phases) when Min Green interval finishes. FRZOL overrides setOLFORCE. Using setOLFORCE during a Parent Phase green terminates the Overlap green, times Overlap clearances, and reverts to green unless setOLOMIT is active.

setOLOMIT(x) **SET_OMIT_ON_OVERLAP(x)**
Prevents a red Overlap from turning green. Has no effect on an Overlap that is timing.

setOLCALL(x) **SET_CALL_ON_OVERLAP(x)**

Turns a red Overlap green when no conflicting phases are On or Next Decisions. A FRZOL without ADVOL prevents a red Overlap turning green if an Overlap call is applied using this function. Once in green, this function will maintain Overlap green unless a setOLFORCE(x) or ADVOL(x) or setINTADV() occurs. When the setOLCALL(x) is removed, normal Overlap operation resumes subject to other functions in this section that affect Vehicle Overlaps.

FRZOL(x) **FREEZE_OVERLAP(x)**
Stops Overlap's current interval with its timer suspended, continues its programmed flashing (Screen 2.2.9.1 programmed FLASH RATE, FLSH GRN and FLSH RED settings), and prevents conflicting Phases from turning on. The only way to advance the Overlap during FRZOL is to activate ADVOL. FRZOL inhibits Lead and Early Lead Overlap intervals making the Parent Phases begin at their normal time.

ADVOL(x) **ADVANCE_OVERLAP(x)**
Advances Overlap to its next interval (that would normally occur) and has higher priority than FRZOL(x), truncates minimum green and clearance times. A red Overlap with setOLOMIT will not advance to green. A green Overlap with ADVOL(x) will not advance to yellow if its Parent Phases are not ready to go yellow. Use setOLFORCE and ADVOL to always advance from green to yellow (abandon green Parent Phases and override Overlap Min Green time).

setPOLFORCE(x)
Forces a Ped Overlap in walk to advance to ped clearance (abandons Parent Phases in walk) when Minimum Walk time finishes. FRZPOL overrides setPOLFORCE. Using setPOLFORCE during a Parent Phase walk terminates the Ped Overlap and will revert to walk unless setPOLOMIT is active.

setPOLOMIT(x)
Prevents a Overlap in steady don't walk from turning to walk but has no effect on a Ped Overlap that is timing.

setPOLCALL(x)
Places a call on Ped Overlap to turn on the walk and extend the walk. The Ped Overlap will turn to walk when no conflicting phases are On or Phase Next Decisions. A FRZPOL without ADVPOL prevents a Ped Overlap in steady don't walk from turning to walk if a setPOLCALL(x) is applied. Once in walk, this function will maintain Ped Overlap walk until a setPOLFORCE(x) or ADVPOL(x) or setINTADV() occurs. When the setPOLCALL(x) is removed, normal Ped Overlap operation resumes subject to other functions in this section that affect Ped Overlaps.
The Ped Overlap call is NOT latched in memory. The script must do setPOLCALL until the walk turns on, otherwise the walk will not be serviced.

FRZPOL(x)
Stops Ped Overlap's current interval with its timer suspended. The only way to advance the Ped Overlap during FRZPOL is to activate ADVPOL.

ADVPOL(x)
Advances Ped Overlap to its next interval (that would normally occur) and has higher priority than FRZPOL(x), truncates minimum walk and ped clearance (FDW and SDW) times. A Ped Overlap in steady don't walk with setPOLOMIT will not advance to walk. A

Ped Overlap in walk with ADVPOL(x) will not advance to ped clearance if its Parent Phases are not ready to advance to ped clearance. Use setPOLFORCE and ADVPOL to always advance from walk to ped clearance (abandon Parent Phases in walk and override Ped Overlap Minimum Walk time).

Caution Activating ADVOL for multiple consecutive Script executions will advance each interval after 0.1 seconds and cause intersection Flash via a Short Yellow Fault.

Caution setPOLCALL (during walk), setOLCALL (during green), setOLOMIT, setPOLOMIT, FRZPOL and FRZOL can (1) delay Preempt and (2) extend Coordination Split times causing potential problems.

3.2.3 LOAD SWITCH/CHANNEL

x = Load Switch Number 1-32 (Screen 2.1.4)
y = ON or OFF

Functions that Override Output Function Values

These work in an "output" type script.

setLSWGRN(x,y)	**SET_GREEN_ON_LSW(x,y)**
setLSWYEL(x,y)	**SET_YELLOW_ON_LSW(x,y)**
setLSWRED(x,y)	**SET_RED_ON_LSW(x,y)**

Note The above functions will **not** set the corresponding output assigned to the Channel/Load Switch (ex, Phase, Overlap). Instead use the functions below if needed.

Sets Load Switch/Channel x and **Vehicle Phase z** both to value y (on/off).
setLSWPHGRN(x,z,y)
setLSWPHYEL(x,z,y)
setLSWPHRED(x,z,y)

Sets Load Switch/Channel x and **Pedestrian Phase z** both to value y (on/off).
setLSWPHWLK(x,z,y)
setLSWPHPCL(x,z,y) Ped Clearance output
setLSWPHDW(x,z,y)

Sets Load Switch/Channel x and **Vehicle Overlap z** both to value y (on/off).
setLSWOLGRN(x,z,y)
setLSWOLYEL(x,z,y)
setLSWOLRED(x,z,y)

Sets Load Switch/Channel x and **Pedestrian Overlap z** both to value y (on/off).
setLSWOLWLK(x,z,y)
setLSWOLPCL(x,z,y) Ped Clearance output
setLSWOLDW(x,z,y)

Sets Load Switch/Channel x and **TSP Run z** both to value y (on/off).
setLSWQJ(x,z,y) TSP Queue Jump signal
setLSWVBAR(x,z,y) TSP Vertical Bar signal
setLSWCLT(x,z,y) TSP Clearance Triangle signal
setLSWHBAR(x,z,y) TSP Horizontal Bar signal

3.2.4 DETECTORS

x = Detector Number
y = ON or OFF

Functions that Override Input Functions Values

These work in an "input" type script.

setDET(x,y)	**SET_CALL_ON_VEH_DETECTOR(x,y)**
setPEDDET(x,y)	**SET_CALL_ON_PED_DETECTOR(x,y)**

3.2.5 RING

x = (1-4) Ring Number
y = ON or OFF (some functions do not contain y)

Functions that Override Input Functions Values

These work in an "input" type script.

setINHMAX (x,y)	**SET_MAX_INHIBIT_ON_RING(x,y)**
setMAX2 (x,y)	**SET_MAX2_ON_RING(x,y)**
setOMITRED(x,y)	**SET_OMIT_RED_ON_RING(x,y)**
setREDREST(x,y)	**SET_RED_REST_ON_RING(x,y)**
setPEDRECYCLE (x,y)	**SET_PED_RECYCLE_ON_RING(x,y)**
setRINGFORCE (x,y)	**SET_FORCE_OFF_ON_RING(x,y)**
setSTOPTIME (x,y)	**SET_STOP_TIME_ON_RING(x,y)**

Functions that Override Output Functions Values

These work in an "output" type script.

setRINGSTATA(x, y)	**SET_STATUS_BIT_A_RING(x,y)**
setRINGSTATB(x, y)	**SET_STATUS_BIT_B_RING(x,y)**
setRINGSTATC(x, y)	**SET_STATUS_BIT_C_RING(x,y)**

Functions that Override Internal Traffic Input Status

These work in an "input" or "output" type script.

FRZRINGVEH(x) **FREEZE_VEH_STATE_FOR_RING(x)**
Freezes Ring x's vehicle interval with its timers suspended (unlike STOPTIME, this does not drop to Free operation).

FRZRINGPED(x) **FREEZE_PED_STATE_FOR_RING(x)**
Freezes Ring x's pedestrian interval with its timers suspended (unlike STOPTIME, this does not drop to Free operation).

Caution FRZRINGVEH and FRZRINGPED can delay Preempt and extend Coordination Split times.

ADVRING(x) **AVANCE_RING(x)**
A per-Ring version of NEMA's Interval Advance function, except it has priority over Preempt, Flash Entry Phase transition and Stop Time.

Caution Activating ADVRING for multiple consecutive Script executions will advance each interval after 0.1 seconds and cause intersection Flash via a Short Yellow Fault.

3.2.6 UNIT
x = ON or OFF

Functions that Override Input Functions Values

These work in an "input" type script.

setALARM(y, x)	**SET_ALARM(y, x) y = Alarm Number**
setTESTINA(x)	**SET_TEST_INPUT_A(x)**
setTESTINB(x)	**SET_TEST_INPUT_B(x)**
setTESTINC(x)	**SET_TEST_INPUT_C(x)**
setINDLAMP(x)	**SET_LAMP_INDICATOR(x)**
setEXTSTART(x)	**SET_EXTERNAL_START(x)**
setAUTOFLSH(x)	**SET_AUTO_FLASH(x)**
setDIMMING(x)	**SET_DIMMING_ENABLE(x)**
setCNA1(x)	**SET_CALL_TO_NONACTUATED_1(x)**
setCNA2(x)	**SET_CALL_TO_NONACTUATED_2(x)**
setMINRECALL(x)	**SET_MINIMUM_RECALL(x)**
setINTADV(x)	**SET_INTERVAL_ADVANCE(x)**
setMCE(x)	**SET_MANUAL_CONTROL_ENABLE(x)**
setWRM(x)	**SET_WALK_REST_MODIFIER(x)**

setGOFREE(x) **SET_GO_FREE(x)**
setGOFREE overrides/sets the "high priority free input". This input has a higher priority than system/central commands (ex, Mizbat, NTCIP). Only preempt, soft flash input, or CVM flash input have higher priority.

setGOFREELOPRIO(x)

setGOFREELOPRIO overrides/sets the “low priority free input”. This input has a lower priority than system/central commands. Only the time-of-day schedule has a lower priority than this input.

Functions that Override Output Functions Values

These work in an “output” type script.

setFLSHSTAT(x) **SET_FLASH_STATUS(x)**

Functions that Override Internal Traffic Input Status

These work in an “input” or “output” type script.

setPATTN(x) **SET_ PATTERN(x)**
Forces Pattern x to run, and Preempt, CVM Flash Input, Soft Flash Input and External Free Input have higher priority. To make setPATTN() have the highest priority, the I/O Script can set the higher-priority inputs “OFF”. When the script selects a pattern, the time-of-day status screen 1.1.3’s “Current Command” displays “LOGIC SCRIPT”.

setPHNEXTRED (x) **SET_PHASE_NEXT_RED_CLEAR(x)**
When x = ON the Next Phase Decision can change during vehicle clearance intervals if phase demand changes.
When x = OFF the Next Phase Decision occurs at the NEMA-defined Yellow beginning and cannot be changed unless Preempt occurs.

When Screen 2.1.7’s Phase Next Control is “end of red” (nonPersistent), the Logic Script can temporarily force “end of green” when Green Overlaps exist to prevent an Overlap turning Yellow and red-reverting Green if the Phase Next Decision changes to a Parent Phase after the Overlap turns yellow. The Screen 2.1.7 setting Phase Next **Overlap** Control also addresses the Overlap response, but is unit-wide affecting all Overlaps.

setTIMGPLAN(x)
When x = 0, any timing plan in effect by time-of-day programming is disabled and all Phases use the standard settings.
When x = 1 – 4, this timing plan is forced active and takes effect immediately.

3.2.7 PREEMPT / TRANSIT PRIORITY

x = 1-32 (Preempt Run number) or 1-8 (TSP Run number)
y = ON or OFF

Functions that Override Input Functions Values

These work in an “input” type script.

setPMTCALL(x,y)	**SET_CALL_ON_PREEMPT(x,y)**
setTSPCHKIN(x,y)	**SET_TSP_CHECK_IN_ON_RUN(x,y)**
setTSPCHKOUT(x,y)	**SET_TSP_CHECK_OUT_ON_RUN(x,y)**
setTSPCANCEL(x,y)	**SET_TSP_ADVANCE_CANCEL_ON_RUN(x,y)**
setPMTGATE(x,y)	**SET_PREEMPT_GATE(x,y)**

setREMTSP(x,y,z)
This does the equivalent of a remote TSP call thru communications.
x = TSP Run number (1-8)
y = Action (0 = clear call, 1 = apply call, 2 = log TSP call but do not act, 3 = log TSP call cleared)
z = ETA in seconds (0 - 65535)
Important: If y is 0, this will remove a remote TSP call thru communications. If the remote system periodically sends the remote TSP call, then the controller will continue to receive remote TSP calls.

INHREMTSP(x)
This function blocks/inhibits a remote TSP call thru communications from taking effect, but does not remove the call. The call is held in a pending status while the Arrival timer and Fail timer count, and the Arrival timer will be updated if periodic remote TSP calls are received for Run x (i.e., an "update" call request to reset the timeout).
x = TSP Run number (1-8)

Functions that Override Output Functions Values

These work in an "output" type script.

setQJMP(x, y)	**SET_QJUMP_ON_RUN (x,y)**

Transit Signals (vertical bar, clearance triangle, horizontal bar):

setTSPVBAR(x,y)	**SET_VERT_BAR_ON_TSP(x,y)**
setTSPCLT(x,y)	**SET_CLR_TRIANGLE_ON_TSP(x,y)**
setTSPHBAR(x,y)	**SET_HORIZ_BAR_ON_TSP(x,y)**

Note The above functions will NOT set the corresponding Channel/Load Switch output. See Section 3.2.3 LOAD SWITCH for setting both the Transit Signal output and Load Switch output using one function.

setTSPOUT(x, y)	**SET_TSP_OUT_ON_RUN(x,y)**
setPMTSTAT(x, y)	**SET_OUT_ON_PREEMPT(x,y)**

Functions that SET Internal Traffic Status
This works in an "input" or "output" type script.
ovrdTSPEAT(x,y)
sets/overrides the estimated arrival time for TSP run x to the value y, where y is in TENTH seconds. This overrides Screen 2.8.4 Arrival Time for a field input TSP call and the Estimated Arrival Time encoded in the NTCIP TSP call message.

3.2.8 INTERVAL MODE

x = Circuit Output Number
y = ON or OFF

These work in an "output" type script.

setCCTOUT(x, y)	**SET_CIRCUIT_OUTPUT(x,y)**

Circuit Outputs programmed on screens 2.1.A.1, 2.1.A.2 and 2.1.A.3

These work in an "input" type script.

setINTIMPLAN(x,y) **SET_INTERVAL_TIME_PLAN(x,y)**
setINOFFSET(x,y) **SET_INTERVAL_OFFSET(x,y)**
setINSIGPLN(x,y) **SET_INTERVAL_SIGNAL_PLAN(x,y)**
setDIAL2(y)
setDIAL3(y)
setDIAL4(y)
setHMCONLINE(y)
setHMCADVANCE(y)
setABSZERO(y)

3.2.9 UNCATEGORIZED OVERRIDABLE OUTPUT FUNCTIONS

setBUZZ(x,y)
Sets the front panel buzzer to the value x (0/off/false, 1/on/true) with optional timeout value y in tenth seconds. If setBUZZ is called with x true and y/timeout 0, the buzzer will remain ON until the Script calls setBUZZ with x false to turn the buzzer OFF. If setBUZZ is called with a non-zero timeout/y, then the buzzer will turn off after the timeout (unless a Script sets the buzzer ON before the timeout expires) or if a Script turns the buzzer off before the timeout expires. The timeout is reset each time setBUZZ is called by a Script.

The following functions are listed for completeness but are not documented in detail currently. Contact ORIUX for information.

These work in an "output" type script.
In the functions below, the value y is on/1 or off/0.

setOADDR(x, y) – For Illinois Commerce Commission systems only.
setICCSAFETY(y) – For Illinois Commerce Commission systems only.

setREDMONOFF(y) – sets the Red Monitor Output to value y.
setTBCAUX(x) – sets TBC Aux Output x (1-3) to value y.
setWATCHDOG(x, y) – overrides the Watchdog Output to value y.
setPHON(x, y) – overrides the Phase-On Output for Phase x to value y.
setSPFUNCOUT(x, y) – sets Special Function Output x (1-8) to value y.
setSYSCNTRL(y) – sets the System Control Status output to value y.
setVOLTMON(y) – sets the CVM Output to value y. Must be toggled otherwise controller will go into a CVM Fault.
setFAULTMON(y) – sets the Fault Monitor Output to value y. Must be toggled, otherwise, controller will go into a Fault Monitor/CVM Fault.
setFLSHLOGIC(y) – sets the Flashing Logic Output to value y.
setSYSCOORD(y) – sets the System Coord output to value y.
setSYSHWIC(y) – sets the System Hardwire Interconnect output to value y.
setHWICFLSH(y) – sets the System Hardwire Interconnect Flash status output to the value y.

3.2.10 UNCATEGORIZED OVERRIDABLE INPUT FUNCTIONS

The following functions are listed for completeness but are not documented in detail currently. Contact ORIUX if further information is needed.

These work in an "input" type script.

setIADDR(x,y) – For Illinois Commerce Commission systems only.
setIPARITY(y) – For Illinois Commerce Commission systems only.
setICCSAFETYLOOP(y) – For Illinois Commerce Commission systems only.

setCABALARM(y) – sets the Cabinet Alarm Status input to value y.
setSPFUNCIN(x,y) – sets the Special Function status input x to value y.
setSYSADDR(x,y) – sets the System Address bit x to value y.
setTBCONLINE(y) – sets the TBC Online Status input to value y.
setCABFLASH(y) – sets the Cabinet Flash Status input to value y.
setMMUFLASH(y) – sets the MMU Flash Status input to value y.

setTIMESRC(y) – sets internal time source to Clock Chip (when y is 1/on) or AC Frequency (when y is 0/off).

3.3 TIMING AND UTILITY FUNCTIONS

The functions in this section work in an "input" or "output" type script.

Caution Only one of these functions DELAY, ELAPSED, TURNEDON, TURNEDOFF, and TOGGLE described in this section can be used on the same script line or in the same if-statement, otherwise unwanted side effects may occur.

DELAY(x) **DELAYFOR(x)**
x = Delay Time in tenth seconds
DELAY causes the script to exit **early (ALL remaining script statements not executed)** to provide a delay timing effect (delay time x) when the nearest prior if-statement is executed and tests TRUE. After each script **early** exit (while timing DELAY) the **script re-starts (in the next tenth-second) on line # 1.** Therefore, the required delay (time x) will NOT be timed unless the DELAY statement is executed for consecutive script executions equal to the delay time (value x) – see Example B. If x is 0, it causes no DELAY and has the same effect if the DELAY(0) is removed from the script. Other scripts are not affected by DELAY(x).

Caution DELAY(x) should not be used inside a user-defined function (see section 3.5) otherwise the script might not work correctly.

Caution While DELAY(x) is timing, all remaining script lines are not executed (script lines after the line with DELAY). If remaining script lines must be executed during the DELAY timing, the DELAY function cannot be used. See ELAPSED below.

Example A: activate Preempt 2 if Detector 12 is active for 1200 tenth seconds (12 seconds):

```
if DET(12) then
  DELAY(1200)
  setPMTCALL(2,on)
end
```

setPMTCALL(2,on) does not execute until Detector 12 has been active/true for 1200 tenth seconds or more.

The same result can be achieved using ELAPSED() described later:

```
if DET(12) then
  if ELAPSED() >= 1200 then
    setPMTCALL(2,on)
  end
end
```

Example B: a pitfall when using DELAY. Let's say you want to activate the TSP Run 3 Check-Out input 5 seconds after Detector 12 turns on, and you create this script:

```
if TURNEDON(DET(12)) then
  DELAY(50)
  setTSPCHKOUT(3,on)
end
```

The result is not what is required. When Detector 12 is **false**, the script executes (every 0.1 seconds) and the if-statement tests **false**. When Detector 12 transitions from false to true, the if-statement tests **true**, and DELAY(50) is executed. The internal count-up timer associated with DELAY starts with a value of 1, and because the internal timer is not at 50, the script exits early.
In the next script execution (0.1 seconds later), the script starts on line #1 and the if-statement tests **false** (because Detector 12 did NOT transition from false to true), which causes **DELAY to not be executed**, which causes the count-up timer to NOT time. The script bypasses the if-statement and reaches the end. **The result is setTSPCHKOUT(3,on) is never executed.**

Here is an example script using DELAY without the above pitfall (by using a flag/latch):

```
if SCRIPTSTART() then
  Flag = false
end

if TURNEDON(DET(12)) then
  Flag = true
end

if Flag == true then
  DELAY(50)
  setTSPCHKOUT(3,on)
  Flag = false
end
```

The above yields the required result. The variable **`Flag`** is initialized to false when the script starts (SCRIPTSTART). When Detector 12 transitions from false to true, the **`Flag`** is set to true to latch the condition even if Detector 12 is only active momentarily.
The third if-statement checks if **`Flag`** is true, which it will be continuously until it is set to false after the DELAY(50) times out, and setTSPCHKOUT(3,on) is executed.

ELAPSED()
ELAPSED() creates an internal counter that starts counting at 0 and increments by 1 each time ELAPSED is executed during Script execution. The counter resets to zero if the ELAPSED() statement was not executed during the previous Script execution.

ELAPSED() returns the number of consecutive tenth-seconds that the nearest prior if-condition tested TRUE. ELAPSED returns zero the first time the if-condition tests true (see Example 1a).

Caution ELAPSED() should not be used inside a user-defined function (see section 3.5) otherwise the script might not work correctly.

Example 1a: omit phase 2 the first tenth-second phase 1 turns green.

```
if PHGRN(1) then
  if ELAPSED() == 0 then
    setPHOMIT(2,ON)
  end
end
```

Example 1b: apply Phase 5 Hold if Phase 1's green ran longer than 100 tenth seconds.

```
if PHGRN(1) then
  if ELAPSED() > 100 then
    setPHHOLD(5,ON)
  end
end
```

Caution To avoid incorrect results when using ELAPSED within an if statement, ELAPSED() must be used either in an if statement as the only expression (ex, if ELAPSED() > 100 then) like the above Examples 1a and 1b, or in an assignment statement (ex, time = ELAPSED() with a variable 'time' used to check if the elapsed time is within a certain range) like in Example 2 below.

Example 2: use ELAPSED to apply a Phase 6 hold if Phase 1 green has run longer than 100 tenth seconds but less than or equal to 200 tenth seconds:

```
if SCRIPTSTART() then
  time = 0
end

if PHGRN(1) then
  time = ELAPSED()
  if time > 100 and time <= 200 then
    setPHHOLD(6,ON)
  end
end
```

You must use the above structure with a variable (**`time`** in the example) to hold the value of ELAPSED and use **`time`** to check for the appropriate range.
DO NOT DO THIS – IT WILL CAUSE THE WRONG RESULT:

```
if PHGRN(1) and ELAPSED() > 100
  and ELAPSED() <= 200 then
  setPHHOLD(6,ON)
end
```

TOGGLE(x,y,z)
TOGGLE() returns true(ON) or false(OFF) in an alternating fashion beginning with x's value (which should also be *true* or *false*). The duration of the *true* portion will be y tenth seconds and the duration of the *false* portion will be z tenth seconds.

Caution TOGGLE(x,y,z) should not be used inside a user-defined function (see section 3.5) otherwise the script might not work correctly.

Example flashing Special Function Output #4 5 tenth seconds on and 3 tenth seconds off with the flashing starting *true(ON)*:

```
if TOGGLE(ON,5,3) then
  setSPFUNCOUT(4,ON)
else
  setSPFUNCOUT(4,OFF)
end
```

Here is a condensed equivalent that produces the same result:

```
setSPFUNCOUT(4,TOGGLE(ON,5,3))
```

The scripter first executes **`TOGGLE(ON,5,3)`**, which returns true or false:
if true, then true gets substituted as follows:

```
setSPFUNCOUT(4,true)
```

if false, then false gets substituted as follows:

```
setSPFUNCOUT(4,false)
```

To flash any output or pulse any input, the following template can be used:

```
<set input or output function name>(X,TOGGLE(ON,A,B))
```

Where the "set input or output function name" is one defined in this document. In the above example, it would be setSPFUNCOUT. And X is the number (the special function output number in this example, which is 4). A is the number of tenth seconds in the "on" cycle and B is the number of tenth seconds in the "off" cycle. Substituting setSPFUNCOUT into the template yields this script text:

```
setSPFUNCOUT(4,TOGGLE(ON,5,3))
```

TURNEDON(x) and TURNEDOFF(x) FUNCTIONS TO SIMPLIFY SCRIPTS CHECKING IF A CONDITION CHANGED

TURNEDON(x) returns true when the expression (condition) passed as x changes from false to true.

TURNEDOFF(x) returns true when the expression (condition) passed as x changes from true to false.

```
if TURNEDON(DET(2)) then
  [ statements ]
end
```

The above **`[ statements ]`** will execute only when Detector 2 turns on (input changes from inactive to active).

The expression x can be more complicated provided x **evaluates to a true or false** value. For example:

```
if TURNEDON(DET(2) and not DET(3)) then
  [ statements ]
end
```

The above **`[ statements ]`** will execute only when the if-statement is true, which is when **a)** Detector 2 turns on and Detector 3 is not on or **b)** Detector 2 turns on and Detector 3 turns off at the same time, or **c)** Detector 2 is on and Detector 3 turns off.

The expression x can check if a status has changed. For example:

```
if TURNEDON(PMTRUNSTATE(2) ==
   "TRACK") then
  [ statements ]
end
```

The above **`[ statements ]`** will execute only when the Preempt #2 Run State changes to "TRACK" (Track Clearance) from a different State. Note the line ends with == because the if-statement requires more characters than the **40-character limit per Script row**.

Tip Without TURNEDON() and TURNEDOFF() the user would have to create a variable to keep track of the condition's prior state and each condition would require its own variable.

Note The expression passed as "x" should be limited to ONLY the functions described in this manual that "get" (or return) true or false values (ex, PHOMIT(x), PHGRN(x), TOGGLE(x,y,z), otherwise incorrect results may occur.

Caution TURNEDON(x) and TURNEDOFF(x) should not be used inside a user-defined function (see section 3.5) otherwise the script might not work correctly.

SCRIPTSTART()
Returns true the first time the Script executes either after power-up or when the Script is enabled and returns false during all other Script executions. It is expected the user will use SCRIPTSTART in an "if" statement at the beginning of the Script to initialize any user-defined variables:

```
if SCRIPTSTART() then
  [statements]
end
```

SCRIPTSTART() does not return true if a cabinet restart occurs (external start input, MMU/CMU/C5000 monitor reset) with the controller power remaining on. If the user wants to re-initialize variables upon External Start, it is also necessary to check External Start. Depending on the script requirements, the user can check for the External Start input transitioning from true to false using TURNEDOFF() or check External Start without TURNEDOFF() or check for the External Start input transitioning from false to true (not shown below). **Examples:**

```
if SCRIPTSTART() or
   TURNEDOFF(EXTSTART())
  then
  [statements]
end
```

```
if SCRIPTSTART() or
   EXTSTART() then
  [statements]
end
```

ENGUP()
Returns the number of tenth seconds the controller/engine has been running since power-up. This can be used as a general-purpose **count-up timer** with user-defined variables when **DELAY** and **ELAPSED** cannot satisfy the script requirements. **Note:** This does not get reset if a cabinet restart occurs (external start input, MMU/CMU/C5000 System Module Monitor reset) with the controller power remaining on.

Example: To force Pattern #12 to run the first 5 minutes after preemption #1, you can use the following INPUT type script example:

```
if SCRIPTSTART() then
  stamp = nil
end

if TURNEDOFF(PMTCALL(1)) then
  stamp = ENGUP()
end

if (stamp ~= nil) and
   ((ENGUP() - stamp) <= 3000) then
  setPATTN(12)
end
```

The script uses a variable called **`stamp`**. This variable will have a value of **nil** when the script starts (SCRIPTSTART) to **disable** the 5-minute timer until the first time Preempt

#1 input turns off (after Preempt #1 service) . A value of **nil** means the **`stamp`** does not have a valid value (and the 5-minute timer is **disabled**). When preempt 1 input turns off, the **`stamp`** is set to a valid value to **enable** the 5-minute timer. Therefore, **`stamp`** is checked for not being equal to **nil** (`stamp ~= nil`) before it is used in the last "if-statement" to set Pattern 12 active. The **`stamp`** variable allows the script to "remember" how long ago the Preempt #1 call/input dropped. Since the engine up timer (ENGUP) keeps counting, at any point in time, the amount of time since the preempt call dropped is equal to the engine up timer value minus the stamp (if stamp is **enabled**).

Since the script executes once every tenth-second, 3000 is the number of tenth seconds in a 5-minute interval (5 minutes X 60 secs/minute X 10 tenth seconds/second).

3.4 GET INPUT PIN FUNCTIONS

The following functions allow the user to get the input value directly being read from a pin on a connector. The parameter to these functions (except AUXSW) is a connector pin capable of being an input, in the form of a number or a string in quotes with the pin letter. Examples:

1. To read pin q from the NEMA MS-A Connector, you would type this in the script: `DEVA("q")`. The pin letter must be in quotes to not confuse it with a variable name.
2. To read pin 3 from Detector BIU #1, you would type this in the script: `BIU9(3)`. No quotes around the number 3.

All the functions in this section:

- return true if the input pin is active, or false if not active.
- work in an "input" or "output" type script.

BIU1(x) **NEMA T/F BIU 1**
BIU2(x) **NEMA T/F BIU 2**
BIU3(x) **NEMA T/F BIU 3**
BIU4(x) **NEMA T/F BIU 4**
The previous 4 functions return the value of the Terminal/Facility BIU pin x. Valid values for pin x are 16-51.

BIU9(x) **NEMA Detector BIU 1**
BIU10(x) **NEMA Detector BIU 2**
BIU11(x) **NEMA Detector BIU 3**
BIU12(x) **NEMA Detector BIU 4**
The previous 4 functions return the value of the Detector BIU pin x. Valid values for pin x are 1-51.

DEVA(x) **NEMA MS-A Connector**
Valid input pin values for x are K, L, M, N, P, R, S, T, U, V, f, g, h, i, j, k, m, n, p, q, v, w, x, y, z, AA, BB, EE, FF, GG, and HH.

DEVB(x) **NEMA MS-B Connector**
Valid input pin values for x: A, B, C, D, E, F, G, H, J, W, X, Y, Z, a, b, c, d, e, CC, and DD.

DEVC(x) **NEMA MS-C Connector**
Valid input pin values for x: P, R, S, T, U, V, W, X, Y, Z, a, b, m, n, p, q, r, s, t, u, v, EE.

HMC(x) **HMC MS Connector**
Valid input pin values for x are 1,3, 4, 6, 7, 9, 10, 12, 14, 16, 19, 20, 25, 26, 35, 36, 38, 55, 56, 57.

HMCSET(x) **HMC Set Connector**
The only valid input pin value for the HMCSET connector is x=10.

LMD40MSA(x) **LMD40 MS-A Connector**
Valid input pin values for x are A, D, E, F, G, H, J, K, L, M, N, P, R, S, y, z, AA, BB, EE, FF, GG.

LMD40MSB(x) **LMD40 MS-B Connector**
Valid input pin values for x are U and V.

LMD40CPC(x) **LMD40 CPC D Connector**
Valid input pin values for x are 1-17, 19-24, 30-34, 37-42, and 49.

LMD40SUB(x) **LMD40 Sub Connector**
Valid input pin values for x are 1-15.

LMD9200CPC(x) **LMD9200 CPC D Connector**
Valid input pin values for x are 2, 4-27, 29-34, 37-42, 49, and 63.

CLOOPCOOR(x) **3000E D-Module Coord Connector**
Valid input pin values for x are A, B, G, H, J, K, L, M, P, R, S, T, W, X, Y, Z, a, and b.

CLOOPPRE(x) **3000E D-Module Preempt Connector**
Valid input pin values for x:1-20, 24-25.

CLOOPAUX(x) **3000E D-Module Aux Connector**
Valid input pin values for x: 4, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 27, 28, 29, 35, 36.

M820A(x) **Multisonic 820A D Connector**
Valid input pin values for x are A, B, M, P, Y, Z, a, b, c, d, e, f, g, p, q, r, s, t, u, v, w, x, y, z, AA, BB, CC, DD, EE, FF, GG, HH, and JJ.

TRACO(x) **Traconex 390 D Connector**
Valid input pin values for x are 3, 4, 6, 7, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 25, 26 30, 31, 35, 36, 37, 38, 39, 40 47, 49, 50, 55, 56, 57, 58, and 60.

INTIN1(x) **International C-3000 Controller Input Connector #1**
Valid input values for x are 1-32.

INTIN2(x) **International C-3000 Controller Input Connector #2**
Valid input values for x are 33-64.

IN2AC1(x) **ATC-2000 2A (or 2E) C1 Connector**
Valid input pin # values for x are 39-82.

IN2AC11(x) **ATC-2000 2A (or 2E) C11 Connector**
Valid input pin # values for x are 10-30.

AUXSW() **ATC-2000 Front Panel Aux Switch**

C5000IN(x) **C-5000 Controller Inputs**
Valid input pin # values for x are 0-63.
0-31 are CPC Inputs 1-32.
32-43 are Opto-Inputs A1-4,B1-4,C1-4 in that order.
48-63 are miscellaneous inputs. See C-5000 User Manual.

SIUOUT1(x) **ATC Cabinet Output SIU # 1**
SIUOUT2(x) **ATC Cabinet Output SIU # 2**
SIUIN1(x) **ATC Cabinet Input SIU # 1**
SIUIN2(x) **ATC Cabinet Input SIU # 2**
SIUIN3(x) **ATC Cabinet Input SIU # 3**
SIUIN4(x) **ATC Cabinet Input SIU # 4**
SIUIN5(x) **ATC Cabinet Input SIU # 5**
Valid input pin # values for x are 0-57 (54-57 are Opto-Inputs 1-4).
If x is an invalid pin #, the scripter will report an error in this format:
SIU 'X' Pin Index Invalid
where X = 1-7 corresponding to Output SIUs 1,2 and Input SIUs 1-5 in that order.

3.5 DEFINING YOUR OWN FUNCTIONS

Note The information in this section is supplied for completeness and as a reference. For more information visit www.lua.org.

In addition to the built-in functions (ex, PHFDW(x)) available, the Scripter allows the user to define and use their own functions, useful if an action must be done more than once in the same script but for different Phase, Overlap, etc. Defining a function prevents having to copy chunks of script text and change only the values used in each chunk. This is an advanced topic requiring an understanding of function parameters and substitution, and best illustrated by example:

Suppose you want to write an Input Type Script that omits the “left turn” phase when the opposing thru “phase” is green. This can be done without a Script using the Anti-Backup and Recall feature, but a Script is used for illustration purposes. For a typical intersection layout:

omit phase 1 when phase 2 is green
omit phase 3 when phase 4 is green
omit phase 5 when phase 6 is green
omit phase 7 when phase 8 is green

We can make a simple Script that does the four actions above, by creating the script chunk for phase pair 1 & 2, then copying that script chunk 3 times, changing each chunk to phase pair 3 & 4, 5 & 6, and 7 & 8, as follows:

```
if PHGRN(2) then
  setPHOMIT(1,on)
end

if PHGRN(4) then
  setPHOMIT(3,on)
end

if PHGRN(6) then
  setPHOMIT(5,on)
end

if PHGRN(8) then
  setPHOMIT(7,on)
end
```

Instead of repeating the chunk 4 times and changing the numbers, let’s make a function that can omit any left turn and thru phase pair by writing this chunk:

```
function  omitLeftPhase(left,thru)
  if PHGRN(thru) then
    setPHOMIT(left,on)
  end
end
```

The **function** keyword must be first, then a space (or newline), then the function name **omitLeftPhase** then a left parenthesis, then the parameters **left** and **thru** **each separated by a comma**, then a right parenthesis, then the script contents for this function, then a closing **end** to match the **function** keyword. The names above are examples. You can use any valid name for the function and the parameters according to preference.

The above defined function that has no effect until it is executed by "calling the function" by writing a statement with: the function name, left parenthesis, values "passed in" separated by commas, then right parenthesis, like this:

```
omitLeftPhase(3,4)
```

The script executes the function **omitLeftPhase** with the number 3 substituted for the first parameter **left** and the number 4 substituted for the second parameter **thru**. Therefore, this "call" to the function **omitLeftPhase** does this after parameter substitution:

```
if PHGRN(4) then
  setPHOMIT(3,on)
end
```

The above "call" to the function omits phase 3 if phase 4 is green
You would "call" this function for each left turn and thru phase pair:

```
omitLeftPhase(1,2)
omitLeftPhase(3,4)
omitLeftPhase(5,6)
omitLeftPhase(7,8)
```

Below is an example of an entire script with comments in grey:

```
--script to omit a left turn phase
--when opposing thru phase is green

--function defined to omit the left
--turn phase when thru phase is grn.
-- left = left turn phase #
```

```
-- thru = phase # opposing left ph
function omitLeftPhase(left,thru)
  if PHGRN(thru) then
    setPHOMIT(left,on)
  end
end

--call the function for each left
--turn and thru phase pair:
omitLeftPhase(1,2)
omitLeftPhase(3,4)
omitLeftPhase(5,6)
omitLeftPhase(7,8)
```

If the above example were more complicated and a function were not used, the copied chunks could be lengthy, resulting in a relatively long script and requiring extra care to make the correct changes to each copied chunk.

Caution The utility functions DELAY, ELAPSED, TOGGLE, TURNEDON and TURNEDOFF should not be used inside a user-defined function otherwise the script might not work correctly. Instead, a variable should be used to keep track of these conditions.

3.6 USING DATA SLOTS WITH PEER-TO-PEER COMMUNICATIONS

GreenWave has a Peer-to-Peer Communications feature (controller-to-controller) allowing information to be transmitted between controllers (Peers) to achieve special functionality. The information is transmitted using Data Slots using the Scripter to:

1) **set** the Data Slot values of Peers using the setDATASLOT function below.
2) **get** the Data Slot values of Peers using the DATASLOT function below.

A Script is the only way to set or get a Data Slot.

Each Controller in the Peer network has their own 16 Data Slots that can each have a value from 0 to 255. You define your own meanings for each Data Slot and its value(s) and not all Data Slots and values have to be used.

A Data Slot can act as a “virtual relay” from one Peer Controller to another by using Data Slot values 0 and 1, where 0 means no relay is set, and 1 means a relay is set. For example, Peer #1 could set the “virtual relay” to 1 and Peer #2 could check if the “virtual relay” is set to 1, and perform some action, then reset the ”virtual relay” to 0 (see “Example Using a Data Slot for a Virtual Relay”).

A Data Slot can act as a “state” when using more than only the values 0 and 1. For example, a Data Slot value of 0 can indicate a “done” or “idle” status, a value of 1 can indicate a “waiting status”, and a value of 2 can indicate an “active” status. During appropriate conditions (depending on the application) the Data Slot value would be set to values 0, 1 or 2 (by a Script).

A Data Slot can act as a time value or count value or an arbitrary value according to the application. For example, the Data Slot value might be the amount of time to wait before the receiving Peer sets a Transit Priority call.

Note The Peer-to-Peer feature must be enabled on Process Control screen 2.1.5.6 page 8. Ethernet or Serial communications can be used. For **Ethernet** Peers, the Peer IP Addresses must be set up on screen 2.1.5.0. For **Serial Port** Peers, the serial port must be assigned on the lower half of screen 2.1.5.6. Contact Oriux for the Peer-to-Peer programming example in **Application Note** titled Programming for Light Rail Transit, part number 99-0670 and the Scripts created by Oriux that the **Application Note** refers to.

Note All Data Slots start with a value of zero when the Controller powers up. When a Data Slot is set by a Script, the value remains until the Data Slot is set again by a Script.

Below are the Scripter functions related to the Peer-to-Peer Communications feature. They can be used in Input or Output Script Types.

setDATASLOT(x,y,z)
Sets the Data Slot value for the Peer x and Data Slot y to value z.

Possible values for x are:
- **0** - this Controller (instead of the values below, see Example 3.6.1)
- **1 to 16** - the Ethernet Peer Controller Numbers
- **252** - Serial Port 2
- **253** - Serial Port 3
- **254** - Serial Port 4
- **255** - Serial Port 5

Possible values for y are 1 to 16 (each controller has 16 Data Slots).

Possible values for z are 0 to 255.

Example 3.6.1: to set Ethernet Peer Controller #3's Data Slot #4 to the value of 6, the Script for Controller #3 could do either of these.

`setDATASLOT(0,4,6)` OR `setDATASLOT(3,4,6)`

The script for the other Peer Controllers must do this:

`setDATASLOT(3,4,6)`

DATASLOT(x,y)
Gets (returns) the Data Slot value for the Peer x and Data Slot y.

Possible values for x are:
- **0** - this Controller (instead of the values below, see Example 3.6.2)
- **1 to 16** - the Ethernet Peer Controller Numbers
- **252** - Serial Port 2
- **253** - Serial Port 3
- **254** - Serial Port 4
- **255** - Serial Port 5

Possible values for y are 1 to 16 (each controller has 16 Data Slots).
Status Screen 1.2.5.1 shows the Data Slot values for this Controller.
Status Screen 1.2.5.2 Pages 1 – 16 show the Data Slot values for each Peer Controller (1 – 16) using Ethernet communications.
Status Screen 1.2.5.3 Pages 1 – 4 show the Data Slot values for Serial Ports 2,3,4,5 respectively.

Example 3.6.2: to get the value of Ethernet Peer Controller #3's Data Slot #4, the Script for Controller #3 could do either of these.

```
DATASLOT(0,4)    OR    DATASLOT(3,4)
```

The script for the other Peer Controllers must do this:

```
DATASLOT(3,4)
```

PEERCOMMSTAT(x,y)
Gets (returns) the Data Slot communications status for the Peer x and Data Slot y. Status Screen 1.2.5.2 and 1.2.5.3 "Status" columns show this status.

Possible values for x are:

0	- this Controller (instead of the values below, see Example 3.6.3)
1 to 16	- the Ethernet Peer Controller Numbers
252	- Serial Port 2
253	- Serial Port 3
254	- Serial Port 4
255	- Serial Port 5

Possible values for y are 1 to 16 (each controller has 16 Data Slots).

The possible return values are:
0 = N/A no communications attempt was made to set the Data Slot.
1 = Queued Data Slot is in the sending queue to be sent.
2 = Sending trying to send Data Slot to peer but has not been successful.
3 = Sent the Peer received and accepted the Data Slot value.
NOTE: If the status never goes to 3/Sent, then that Data Slot value never got to its Peer, due to a setup error or communications error, and PEERSYNCSTAT is likely to be 1/syncing.

Example 3.6.3: to get the Peer Communications Status of Ethernet Peer Controller #3's Data Slot #4, the Script for Controller #3 could do either of these.

```
PEERCOMMSTAT(0,4)    OR    PEERCOMMSTAT (3,4)
```

The script for the other Peer Controllers must do this:

```
PEERCOMMSTAT (3,4)
```

PEERSYNCSTAT(x)
Gets (returns) the sync status for the Peer x. Status Screen 1.2.5.2 and 1.2.5.3 "SYNC STAT" shows this status. Every 800 milliseconds each controller sends a "sync" message to all its enabled **Ethernet** Peers, and every second each

controller sends a “sync” message to all its enabled **Serial Ports**. If a response to the “sync” message is not received after 10 seconds, the sync status goes to “syncing”, indicating a communications failure and the controller is forever retrying the “sync” message. If communications recover the **PEERSYNCSTAT** will revert to “synced”.

Possible values for x are:

0	- this Controller (instead of the values below, see Example 3.6.4)
1 to 16	- the Ethernet Peer Controller Numbers
252	- Serial Port 2
253	- Serial Port 3
254	- Serial Port 4
255	- Serial Port 5

The possible return values are:

0 = N/A	no communications because peer is disabled in database.
1 = Syncing	attempting to send a “sync” message to sync to this peer but not successful (at startup or if a communication failure).
2 = Synced	synced to this peer (responding to “sync” messages).

NOTE: If the status never goes to 2/Synced, then there is most likely a setup error or communications error.

Example 3.6.4: to get the Peer Sync Status of Ethernet Peer Controller #3, the Script for Controller #3 could do either of these.

```
PEERSYNCSTAT(0)
```

OR

```
PEERSYNCSTAT (3)
```

The script for the other Peer Controllers must do this:

```
PEERSYNCSTAT (3)
```

Example Using a Data Slot for a Virtual Relay:
If Detector 12 is active for Peer #1, send a Preempt 3 call to Peer #2. To accomplish this, Peer #1 and Peer #2 would have these Input Scripts:

Input Script for Peer #1:

```
if DET(12) then
  setDATASLOT(2,1,1)
end
```

Input Script for Peer #2:

```
if DATASLOT(0,1) == 1 then
  setPMTCALL(3,on)
  setDATASLOT(0,1,0)
end
```

Data Slot 1 acts as a “virtual relay” to send a Preempt 3 call from Peer #1 to #2.

In the above Input Script for Peer #1, if Detector 12 is active, Data Slot 1 for Peer #2 is set to a value of **1** to relay the Preempt 3 call to Peer #2. A Data Slot value of 1 means Peer #2 should set its Preempt call and a value of 0 means no Preempt call.

In the above Input Script for Peer #2, if Peer #2’s Data Slot 1 has a value of **1**, the Preempt 3 call is set and Peer #2’s Data Slot 1 is reset to 0. The Data Slot must be reset to 0 otherwise the Preempt 3 call will be applied forever, because the Data Slot maintains its value until it is set again.
NOTE: `DATASLOT(0,1)`, the **0** means “this” controller. `DATASLOT(2,1)` also works because “this” controller is Peer #**2**. Same for `setDATASLOT(0,1,0)`. Using `setDATASLOT(2,1,0)` works.

Detecting Peer Communications Errors

Regardless of any enabled Scripts that set or get Data Slots, communication between Peers (“sync” message sent) occurs automatically every 800 milliseconds (to Ethernet Peers) and every second (to Serial Ports) to detect communication failures between the Peers.

If a response to the “sync” message occurs within 10 seconds or less, the **PEERSYNCSTAT** will remain “synced”, otherwise it will be “syncing” to indicate a communication failure and the controller is forever attempting to “sync” the peer. If communications recover the **PEERSYNCSTAT** will revert to “synced”.

Central System Notification of Peer Communications Errors

When the **PEERSYNCSTAT** is 1/syncing:

1) bit 9 in the NTCIP controllerAlarmStatus1 object will be set to 1 (see MIB).
2) the Script can be modified to set one of the 8 user-defined Alarms.

Either of these can be used to indicate a Peer Communications failure for Central System notification.

4. Using I/O Scripter for External Input Logic

By using the functions in the “GET INPUT PIN FUNCTIONS” section along with the appropriate set function, the user can achieve the equivalent of external cabinet input logic.

For example, suppose someone wanted to use external cabinet logic to cause Ring 2 Inhibit max termination to be applied by OR-ing the MS-C connector pins a and b; in that case the following Script could be applied:

```
if DEVC(“a”) or DEVC(“b”) then
  setINHMAX(2,on)
else
  setINHMAX(2,off)
end
```

Or the following advanced equivalent:

```
setINHMAX(2,DEVC(“a”) or DEVC(“b”))
```

5. Using I/O Scripter for External Output Logic

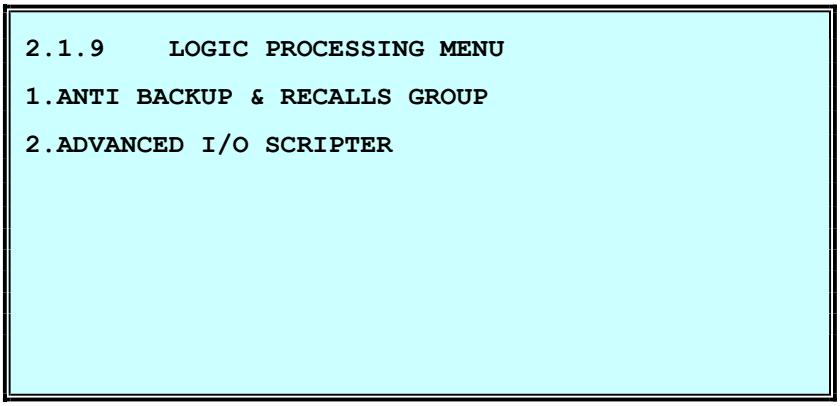
By getting the value of traffic outputs using the “testable” output functions under subsections below section 3.1 and using the “set” functions from section 3.2, the user may achieve the equivalent of external cabinet output logic.

For example, suppose the user wants to cause the special function output 1 to be on when phase 2 is on or when there is a next on phase 2 while phase 6 is on. The following Script could be applied:

```
if (PHNXT(2) and PHON(6)) or
   PHON(2) then
  setSPFUNCOUT(1,on)
else
  setSPFUNCOUT(1,off)
end
```

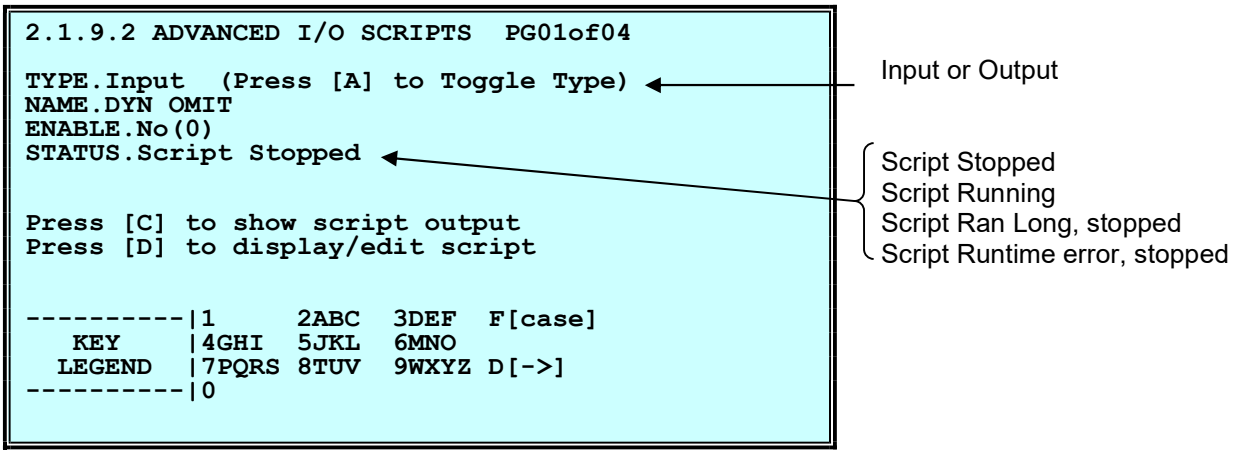
6. Advanced I/O Scripter User Interface

6.1 Logic Processing Screen 2.1.9



Pressing [2] will navigate to the Advanced I/O Scripter Menu.

6.2 Advanced I/O Scripter Screen 2.1.9.2



TYPE:
There are 4 Input Script pages and 4 Output Script pages independently accessed by pressing [A] to switch between “Input” and “Output” TYPE and using [UP] and [DWN] keys to go to the desired script number.

NAME:
Identifies the Script using a 35-character-maximum name (numbers, letters and spaces allowed). ORIUX recommends giving each Script a meaningful name. When entering Edit Mode, the Cursor is at the next character position after a pre-existing Script Name or at the first position if no Script Name exists.

ENABLE and STATUS:
Setting ENABLE to No(0) stops the Script from running and makes STATUS display “Script Stopped”.
Setting ENABLE to Yes(1) attempts to run the Script making STATUS display one of the following messages (see Script Error Types section):

(1) “Script Running” if no errors,
(2) “Script Ran Long, stopped”,
(3) “Script runtime error, stopped”

KEY LEGEND:
Pressing [F] toggles the Key Legend’s letters between Uppercase and Lowercase mode. Press Keys 1-9 and 0,A,B,C,E to toggle between the character group until the desired character appears. Upon relaxing the Key (leaving it released at least 1.5 seconds), the selected character inserts and the cursor moves to the next position.

Lowercase Mode:

```
----------|1     2abc  3def  F[CASE]
   KEY    |4ghi  5jkl  6mno
  LEGEND  |7pqrs 8tuv  9wxyz D[->]
----------|0
```

Uppercase Mode:

```
----------|1     2ABC  3DEF  F[case]
   KEY    |4GHI  5JKL  6MNO
  LEGEND  |7PQRS 8TUV  9WXYZ D[->]
----------|0
```

6.3 CREATING OR EDITING A SCRIPT

1. **Select the Script Number:** When not in Edit Mode and on Screen 2.1.9.2, page down to the desired Script Number.
2. **Select Input or Output Script Type:** Use the [A] key to toggle between TYPE.Input and TYPE.Output to match the Script type being created.
3. **Enter Script Name:** Use [*][E] to enter Edit Mode. Use the Key Legend to enter a NAME. ORIUX recommends giving every Script a meaningful name. Press [*][E] to save the script's NAME.
4. **Edit the Script's Text:** Press [D] to move to the Display/Edit Script screen. Press [*][E] to enter Edit Mode. The screen will look like Figure 6.3.1 below. Use the Key Legend and Shortcut Menu [MNU] to generate the Script's text. Press [*][E] to save the Script's text to the database. If an error message box occurs, see the Script Error Types section.

[Arrow Keys] navigate thru the Script's text.
[UP] and [DWN] Keys navigate through a multi-page script.
[NXT] moves cursor to one column after the line's last character.
[HME] moves the cursor to the beginning of the line.
[CLR/ESC] moves the cursor and its right-side characters one column to the left, erasing the character under the cursor. When pressed on the first column, it moves to the previous line.
[ENT] moves every character to the right of the cursor (including the character the cursor is on) to the next line with the cursor on the first column.

Note During Edit Mode on the Display/Edit Script screen, the user cannot return to Menu 2.1.9.2 unless they load their Script changes by pressing [*][E] or cancel their Script changes by pressing [*][C].

5. Press [PRV] to return to Screen 2.1.9.2.
6. **Enable the Script:** Press [*][E] to enter Edit Mode.

Caution Enable and test all Scripts in a safe environment because a Script with its Status "Running" and containing incorrect logic statements can cause

intersection flash. The user must evaluate all Script statements against valid scenarios to ensure (1) no Runtime Errors occur and (2) the Script runs without undesirable results.

7. Set ENABLE to Yes(1). The STATUS reads “Script Running” if the script is error-free. If the STATUS reads anything else, see Section Script Error Types section.
8. Evaluate the Script to ensure abnormal scenarios (ex, Preemption, External Start, Hardware Inputs, varying Phase demand) do not cause undesirable results.

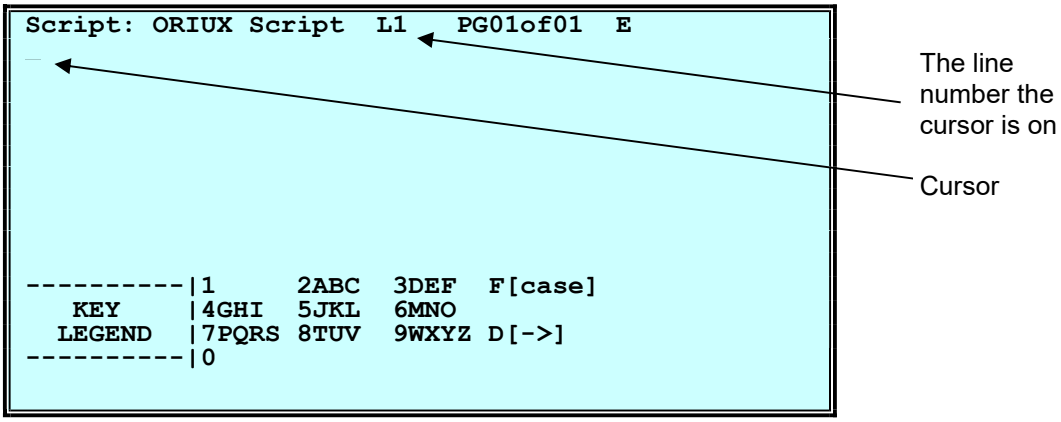


Figure 6.3.1 The Display/Edit Script Screen

The Key Legend (shown in the Display/Edit Script screen) and the Shortcut Menu (shown below and launched by pressing MNU) generates Script text.

6.4 THE SHORTCUT MENU

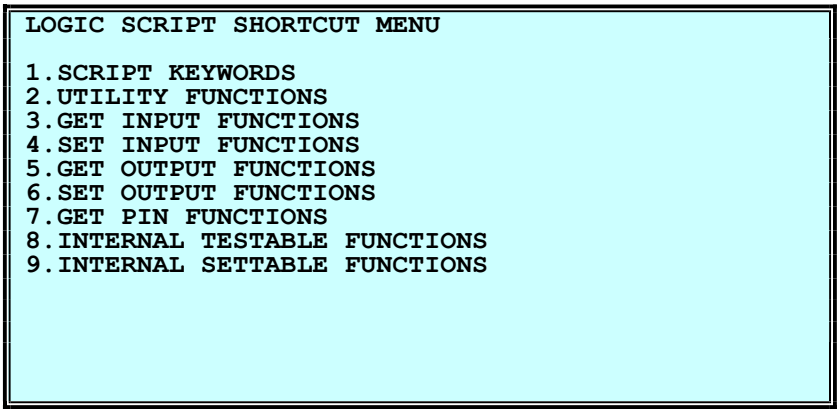
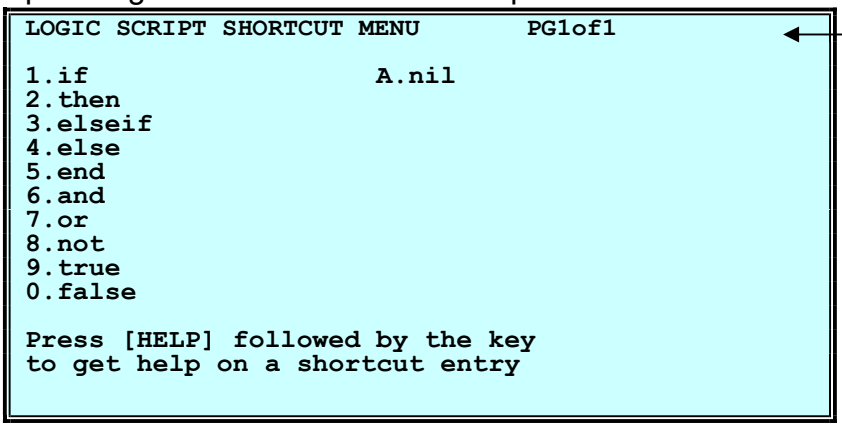


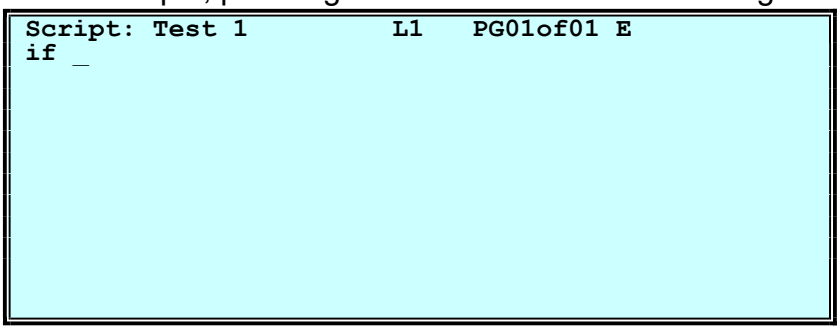
Figure 6.4.1. Shortcut Menu.

Pressing [PRV] returns to the Logic Script Menu or pressing a Number key opens the Sub-Menu for the selected item (all other keys ignored). For example, pressing 1 from the Menu above opens the Sub-Menu below:



Pressing [PRV] returns to the previous Logic Script Shortcut Menu.
Pressing [HLP] toggles ‘H’ and ‘ ‘ in the top right corner.
Pressing keys [0-9,A-F,YES,NO,HME] when displaying ‘H’ opens the Help Screen for the selected item (the above screen limits its selections to [1-9,A]).
Pressing keys [0-9,A-F,YES,NO,HME] when displaying ‘ ‘ selects an item, moves to the Display/Edit Script screen, inserts the selected item into the Logic Script text, and positions the cursor appropriately.

For example, pressing 1 from the above Menu changes to the Menu below:



The Display/Edit Script screen below shows a completed Script example.

```
Script: DYN OMIT          L1   PG01of01
if PHGRN(2) or PHNXT(2) then
  setPHOMIT(1,ON)
end

if PHGRN(6) or PHNXT(6) then
  setPHOMIT(5,ON)
end



----------|1     2ABC  3DEF  F[case]
   KEY    |4GHI  5JKL  6MNO
  LEGEND  |7PQRS 8TUV  9WXYZ D[->]
----------|0
```

6.5 STATEMENTS REQUIRING MORE THAN ONE LINE

If a Script statement requires more than one line, it must be broken only where whitespaces are allowed, otherwise a Syntax Error or Runtime Error occurs. The Script below is invalid because it breaks the PHCHK function:

```
if PHGRN(2) and PHGRN(6) and PHCH
  K(4) then setPHOMIT(1,ON)
end
```

A corrected Script is shown below:

```
if PHGRN(2) and PHGRN(6) and
  PHCHK(4) then setPHOMIT(1,ON)
end
```

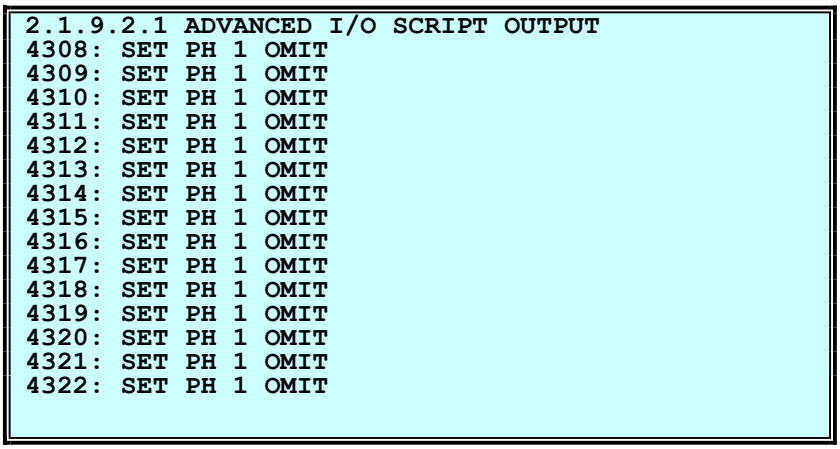
6.6 SCRIPT OUTPUT

Pressing [C] from Screen 2.1.9.2 opens Screen 2.1.9.2.1, displaying the script output generated by user-positioned print() and tprint() statements.

Inserting tprint(“SET PH 1 OMIT”) inside the above script’s if statement yields this new if-statement:

```
if PHGRN(2) and PHGRN(6) and
  PHCHK(4) then
  setPHOMIT(1,ON)
  tprint(”SET PH 1 OMIT”)
end
```

and generates the following Script Output when Phases 2 and 6 are green and phase 4 has a call (check):



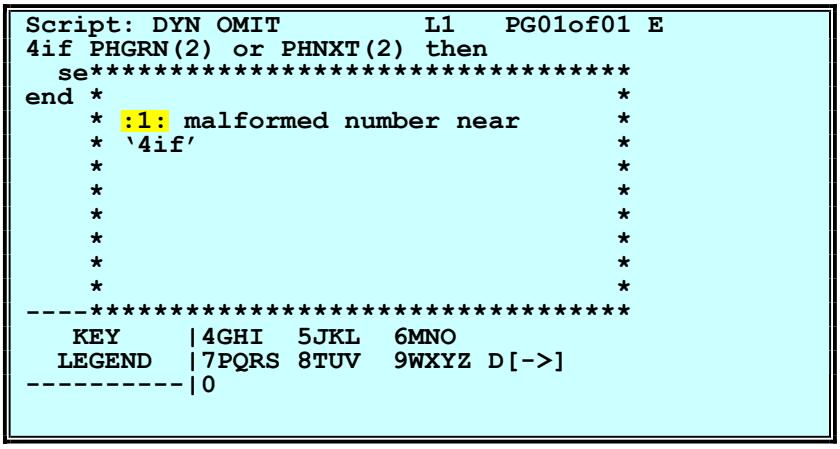
Pressing [ENT] toggles between pausing and updating the Script Output to prevent it from scrolling off the screen (the Script remains running). A flashing “P” is shown on the top right when in “pause mode”.

6.7 SCRIPT ERROR TYPES

The Message Box displays Script errors. Emphasis should be on the number between the colons (the Script line number containing the error) and an optional expression in single quotes (the Script’s text containing the error).

Syntax Errors

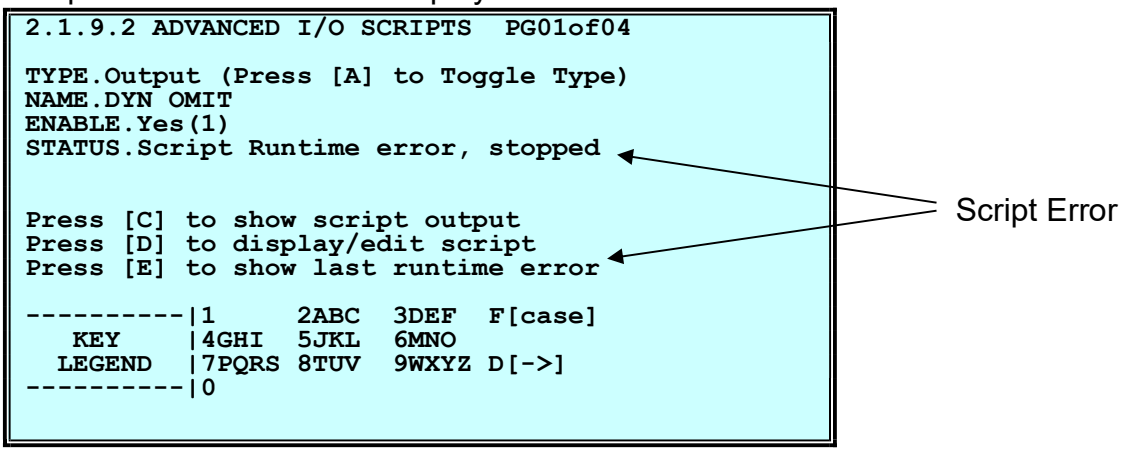
Syntax Errors occur when attempting to save a Script containing errors (by pressing [*][E]). See Section 2.2. For example, if the user accidentally types “4if” instead of “if”, the following Message Box opens when [*][E] is pressed.



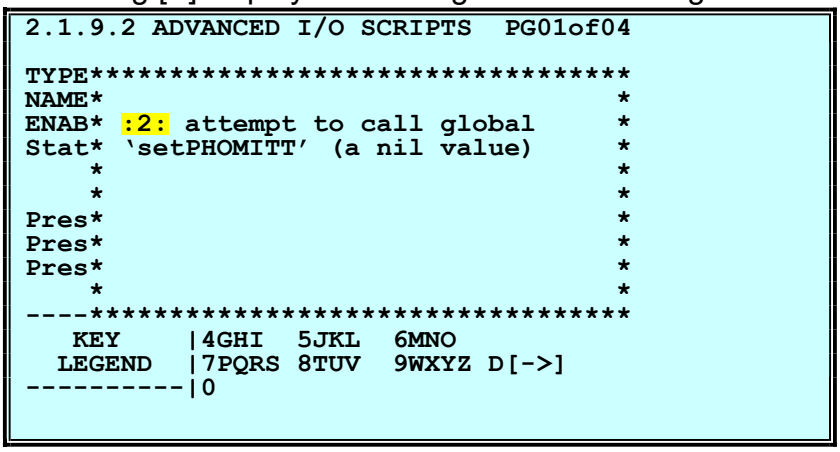
The message above means Script Line Number 1 (denoted by :1:) has an error in the script’s text ‘4if’. Press [CLR/ESC] to exit the Message Box and fix the syntax error.

Runtime Errors

Runtime Errors cannot be detected until the Script runs and the error makes the Script stop. For example, instead of typing setPHOMIT(1,ON), the user mistakenly types setPHOMITT(1,ON). The Script generates a Runtime Error and stops and Screen 2.1.9.2 displays:



Pressing [E] displays a Message Box describing the error:



The message above means script line number 2 (denoted by :2:) has an error in the script's text 'setPHOMITT' (an extra "T"). Press [CLR/ESC] to exit the Message Box and fix the runtime error.

Script Ran Long Errors occur if a Script did not finish in its allocated time and makes the Script stop and display "STATUS.Script Ran Long, stopped". Add print() or tprint() statements to the Script to troubleshoot.

The Main Status Display shows I/O Scripter Status on the right side of Row 6:

R (Running): displays “Y” if an error-free Script running, otherwise displays an “N”.
E (Error): displays “Y” if a Script stopped via Runtime/Ran Long errors, otherwise displays an “N”.

Note Runtime Errors and Script Ran Long Errors do not cause intersection Flash. The Script stops and the Controller operates normally.

6.8 FREQUENTLY ASKED QUESTIONS AND TROUBLSEHOOTING

1. Does controller have to be restarted when changing Script ENABLE from No(0) to Yes(1)? No.
2. Script has no effect.
    a. Is Screen 2.1.9.2 Script Enable set to “Yes(1)”?
    b. Does Screen 2.1.9.2’s Status show an error? Show running?
    c. Is the correct Script Type chosen? If the Script sets an Input, it must be an Input Script. If the Script sets an Output, it must be an Output Script.
3. An Output Script is running without error, but the outputs are not being changed by the Script.
    a. Verify on status screen 1.2.4 “inputs/outputs pin status” that the script is turning the correct outputs on or off. If not, check the script contents.
    b. If using a TS2 Type 1, or International module, or ATC Cabinet I/O, or C5000 (with Load Switch/Channels in the I/O Map), the script must use setLSWGRN, setLSWYEL and setLSWRED. You can also use the functions that set both the “LSW” (Channel) and its assigned output (ex, Phase, Overlap) using one function. **For example**, if the script must set Vehicle Overlap 1/A’s yellow ON and green OFF and red OFF, and Vehicle Overlap A is assigned to channel #9, the script can do setLSWYEL(9,on), setLSWGRN(9,off) and setLSWRED(9, off) and setOLYEL(1,on), setOLGRN(1,off) and setOLRED(1,off). You could also use the single function (recommended by ORIUX): setLSWOLRED(9,1,off), setLSWOLYEL(9,1,on), setLSWOLGRN(9,1,off), where 9 is for Channel 9, and 1 is for Vehicle Overlap 1/A.
    c. If using a TS2 Type 1, or International module, or ATC Cabinet I/O, or C5000, is the script setting the correct Channel/load switch?
    d. If using a TS2 Type 2, HMC-1000, LMD-40, 2070 2A, is the I/O Mapping correct?

4. An output script is causing Field Check Faults (TS2 MMU or ATC Cabinet CMU).
    a. Make sure the script is setting the LSW/Channels (see question 3 above). If using a TS 2 Type 2 cabinet, the default I/O mapping has Vehicle Phase, Ped Phase, and Vehicle Overlap (not **Channel** GYR), but the MMU receives **Channel** status from the Controller on Port 1, so the script must also set the Channel outputs otherwise there will be a **mismatch** between the Channel status and the MSA,B,C Connector status.
5. Does a vehicle Phase, pedestrian Phase, Overlap or pedestrian Overlap have to be configured in the sequence in order for an output script to manipulate those outputs? No.
6. Can the Script's contents be changed while it runs? Yes
7. If Input Script #1 is used, is Output Script #1 available? Yes, Input Script #1 and Output Script #1 are separately stored Scripts. Use the [A] key to toggle between displaying/editing Input Script #1's data and Output Script #1's data. (Same for scripts 2-4)
8. What is the Script Output used for? To troubleshoot the script by using the tprint() and print() functions. Watching the controller's outputs and the Script Output should help when troubleshooting a misbehaving script.
9. Does Preemption override a script? No. The script must be designed to handle Preemption. Preemption can modify phase times, change the order of phases (overriding next phase decision), or skip phases.
10. Can the Script be Time of Day enabled and disabled? Yes. When setting Screen 2.1.9.2's ENABLE to Yes(1), the Script always runs. To Time of Day disable, add the appropriate statements in the Script. For example, the script below Omits Phase 7 every Sunday ("== 0" equals Sunday):

```
Script: Sunday Omit7   L1   PG01of01
if TODDOW() == 0 then
  setPHOMIT(7,ON)
end









----------|1     2ABC  3DEF  F[case]
   KEY    |4GHI  5JKL  6MNO
  LEGEND  |7PQRS 8TUV  9WXYZ D[->]
----------|0
```

7. Advanced I/O Scripter USB Interface

7.1 USB Device's I/O Scripter Folder and File Structure

The USB's Root directory must contain the ORIUX Signature file "ASTC_DATA_DISK" and the folder named **ATC_LINUX**. Contact ORIUX Product Support to qualify your USB Device.

The USB's **USTC_scripts** folder (inside the **ATC_LINUX** folder) contains folders:

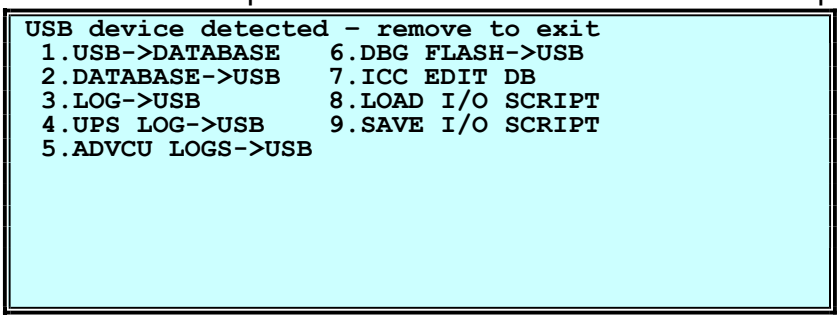
misc folder created when saving Script to USB (USB Menu Choice 9 SAVE I/O SCRIPT).

cabWXYZ folder created when saving Controller Database to USB (USB Menu Choice 2 DATABASE->USB) with Screen 2.1.5.3 Cabinet Address equal to WXYZ.

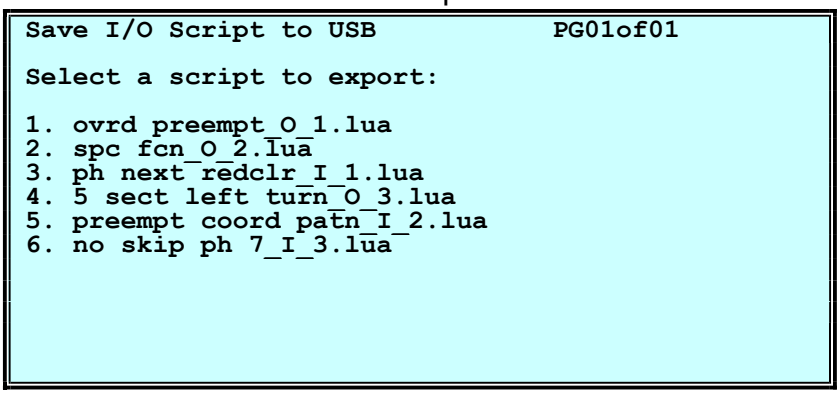
The controller automatically creates the required folder structure on the USB device.

7.2 Saving a Script from Controller to USB Device

1. Insert a qualified USB Device. The menu below opens:



2. Select 9. Save I/O Script.



3. Script files have the following format:

   [Screen 2.1.9.2's NAME]**_T_N.lua**

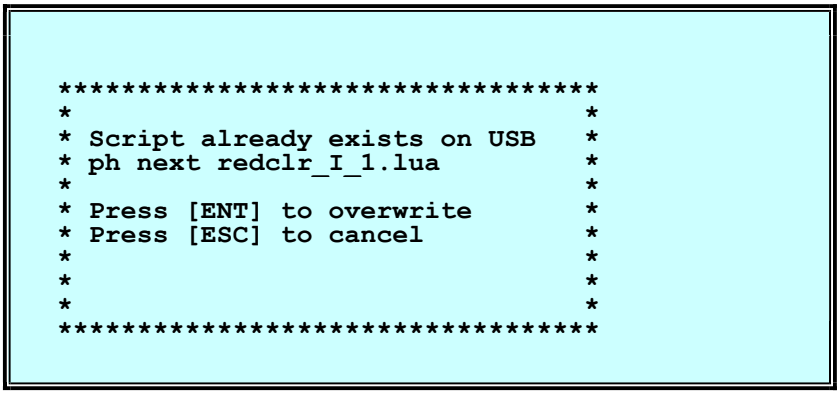
   T (Type) is "O" for Output Script or "I" for Input Script

N is the Script/Slot Number 1-4

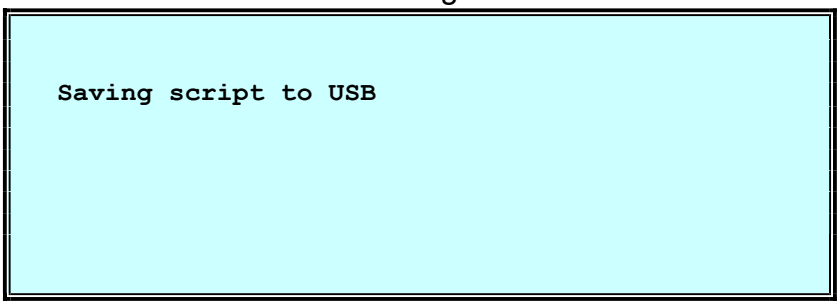
Example: In menu selection #1 above, “ovrd preempt” appears on Screen 2.1.9.2’s NAME field, O indicates an Output Script, 1 indicates Output Script/Slot #1.

4. Press the corresponding key to select the Script file.

5. If the USB contains the same file, the Message Box allows the user to cancel or overwrite the file:

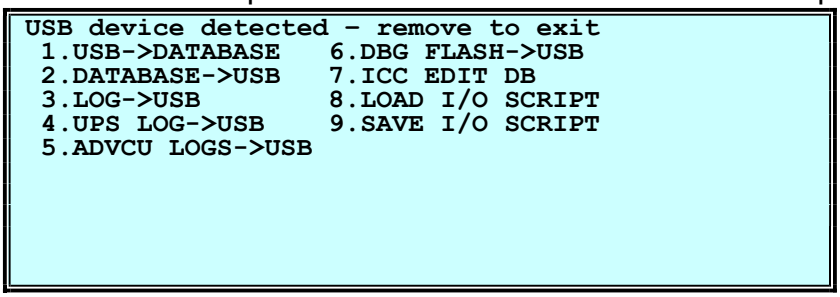


If the USB does not contain the same file, the screen below occurs until the USB finishes saving the file.

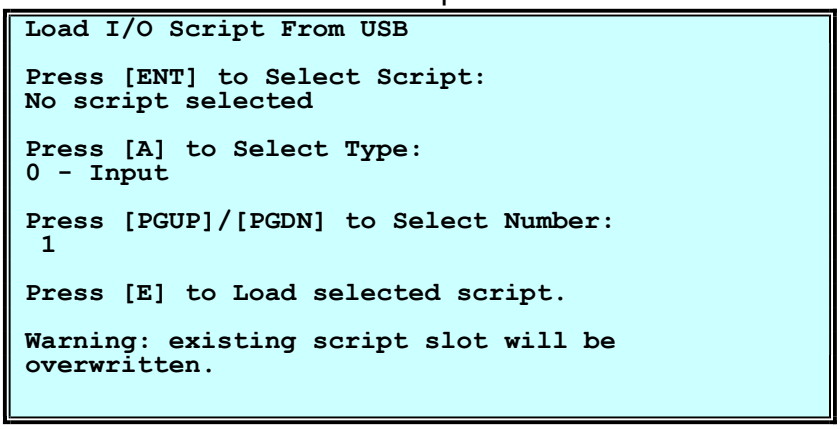


7.3 Loading a Script from USB Device to Controller

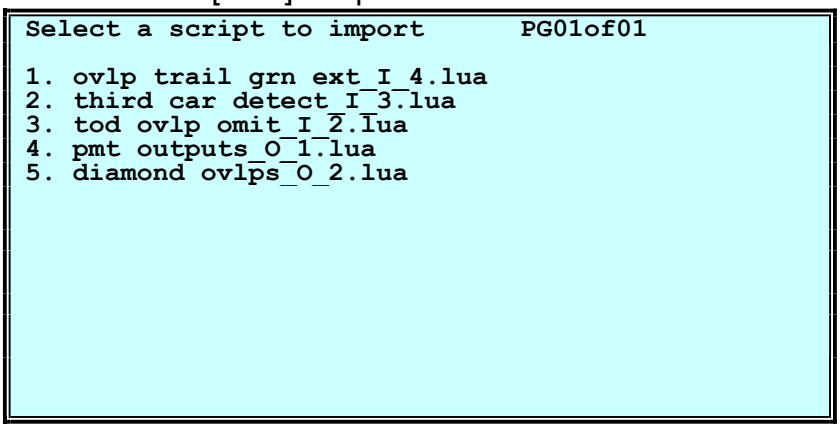
1. Insert a qualified USB Device. The menu below opens:



2. Select 8. Load I/O Script.



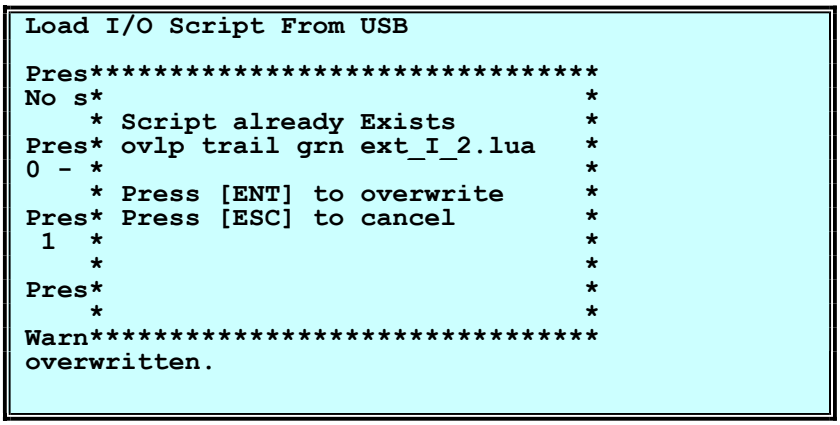
3. Press [ENT] to open the selection menu. Press number to select Script:



4. Screen returns to the 'Load I/O Script From USB' menu. Press [A] to select whether the controller stores the selected Script as an Input or Output Script Type. The [A] key toggles the display between "0 – Input" and "1 – Output".

5. Press [PGUP] and [PGDN] to select which Script Number/Slot (1-4) the controller will store the selected Script.

6. Press [E] to load the selected Script. The following Message Box appears if the Controller has an existing Script stored at the selected Script Slot (Script Number and Script Type determine the Slot). If the Message Box does not appear, the Script loaded successfully.


Load I/O Script From USB
Script already Exists
ovlp trail grn ext_I_2.lua
Press [ENT] to overwrite
Press [ESC] to cancel
overwritten.

8 Scripter Web Interface

This section describes how to use the Scripter Utility in the Web Interface.

This is the recommended method to work with Scripts.

The images in this section might be different than those displayed on your Controller's Web Interface, due to version changes over time. However, all images closely resemble all firmware versions and are adequate for the understanding of the content in this section.

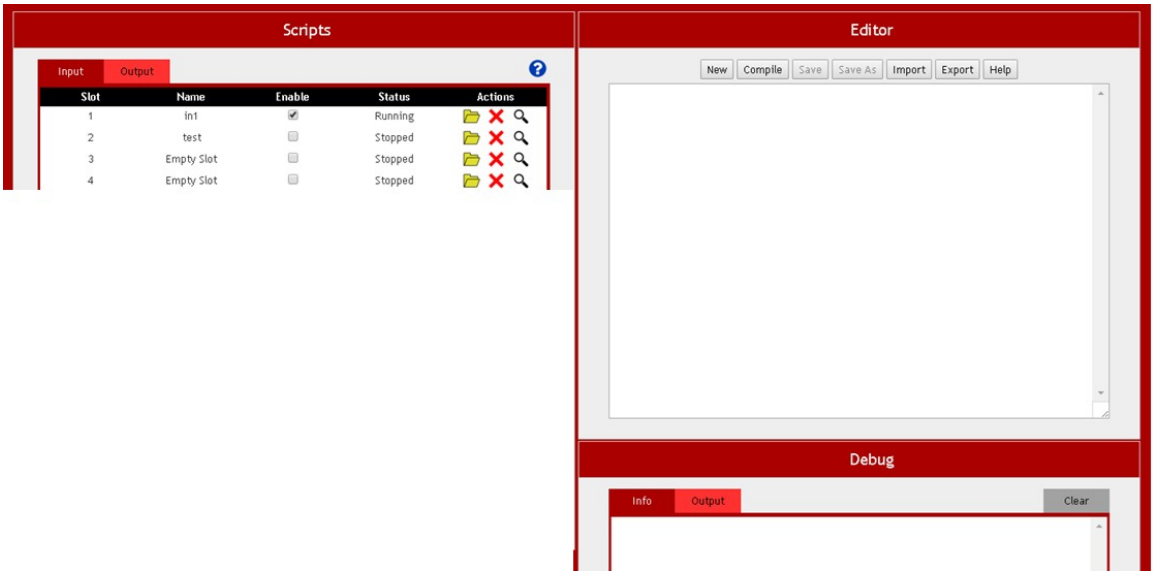
8.1 Overview

The Scripter Utility in the Web Interface aids in creating, editing, debugging and viewing Scripts on the Controller. This utility supports features of a standard text editor.

The Scripter utility is divided into three sections:

- **Scripts Section**
- **Editor Section**
- **Debug Section**

Each of these sections can be minimized by clicking the header when not in used and opened when needed by clicking the header once more.



8.1.1 Scripts Section

The scripts section is composed of two tabs: the Input Scripts tab and the Output Scripts tab. The Input Scripts tab displays a list of 4 input slots where the name and status of the scripts as well as action buttons to

perform different actions on each of the scripts. The Output Scripts tab displays the same elements for the 4 Output scripts.

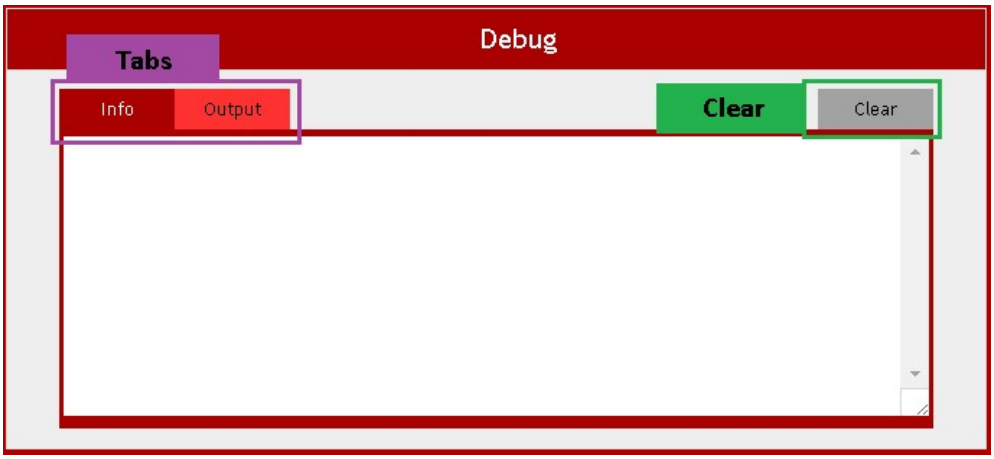
To switch between Input and Output Scripts, click the desired tab. To perform the desired action, press the action button that corresponds to the desired action. A tooltip will show when the cursor is held on top of the button that displays the type of action it performs.

8.1.2 Editor Section

The editor section is composed of a toolbar and a text area. To use the text area, click the area, then use the keyboard to create or edit a script. To perform an action, click the corresponding button in the toolbar. Certain actions will display extra elements such as dialogs or menus. Follow the instructions in each case.

8.1.3 Debug Section

The debug section is composed of two tabs: the Info tab and the Output tab. The Info tab displays a text area where messages will be displayed when compiling or saving Scripts. The Output tab displays the output of a script when Debug mode is enabled (discussed later). The clear button is located at the top right corner of the tab container and will clear the text area in the selected tab of the Debug section when pressed.

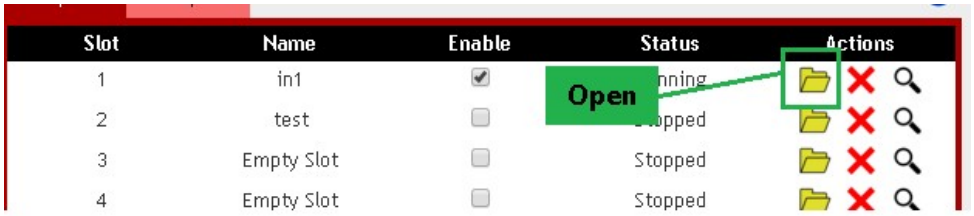


To switch between tabs, click the desired tab. To clear the text area, click on the Clear Button.

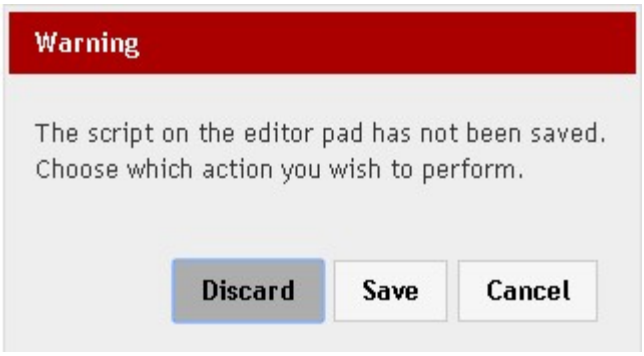
8.2 Opening and Saving Script Files

8.2.1 Opening a Script

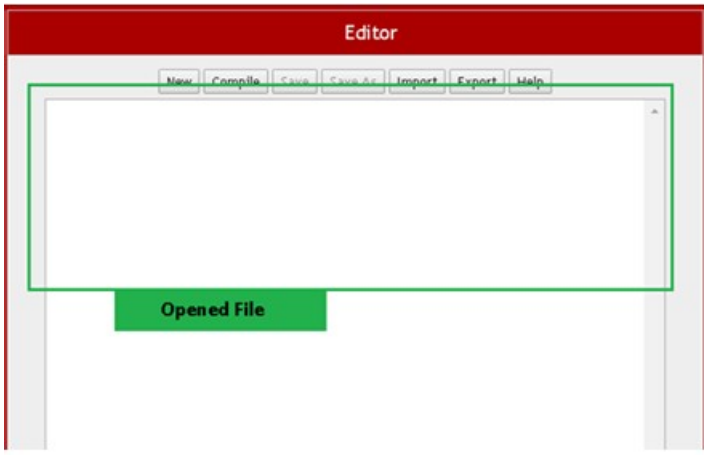
To open an existing script on the Controller, go to the Scripts section. Locate the script in the input or output scripts tab and click the open button in the action buttons section.

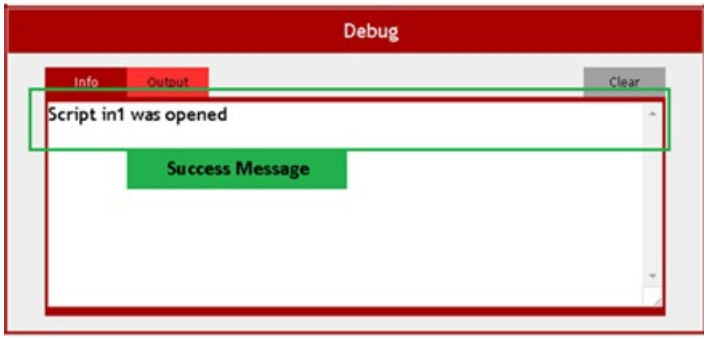


If the Scripter detects unsaved content in the Editor text area, a warning dialog will pop up. Click the Save button if you wish to save the content in the text area, click the Discard button if you wish to discard the content and click Cancel if you wish to exit the dialog and cancel the open operation.



The open operation will be finalized by displaying the contents of the screen on the Editor text area. If an error occurs or the operation is invalid, an error message will be displayed on the Info tab text area in the Debug section.



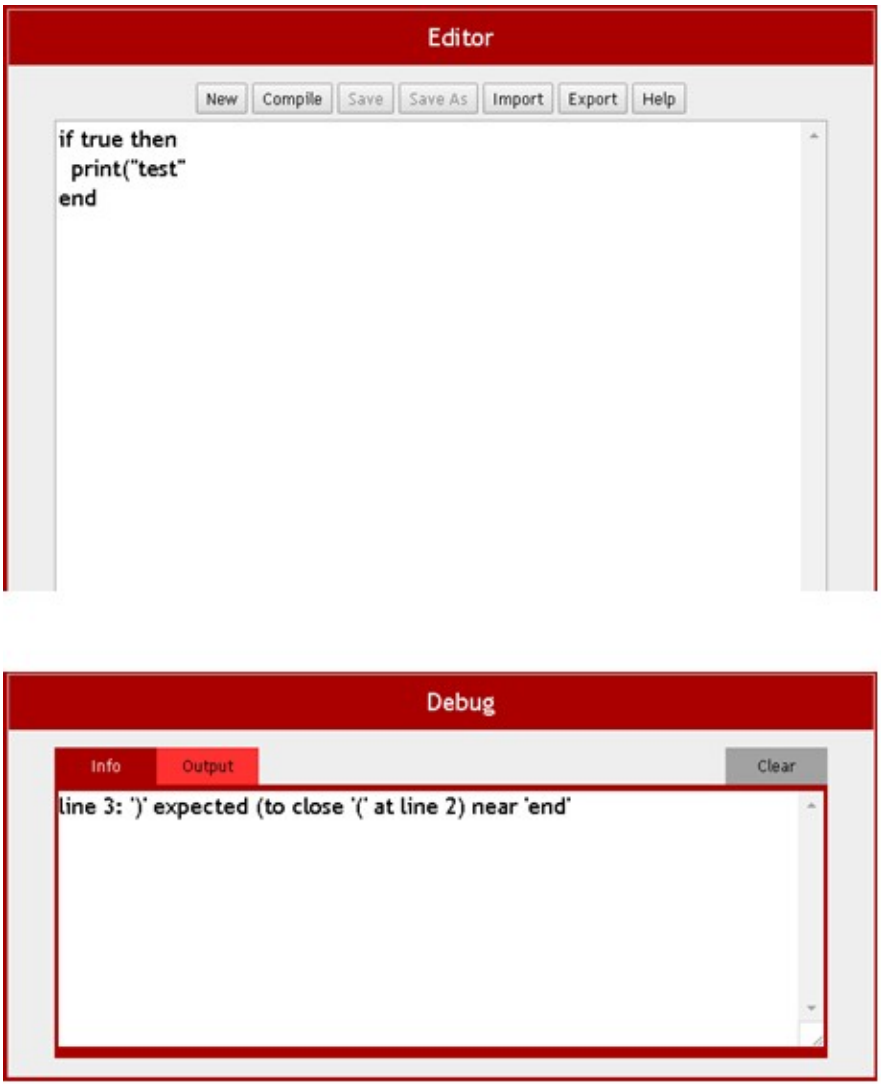


8.2.2 Saving a Script

8.2.2.1 Before Saving

To save, compilation must be performed first. To compile a script, click the Compile button on the toolbar in the Editor Section. The scripter will grab the script currently in the Editor text area and compile it. The results of the compilation will be displayed on the Info tab text area.

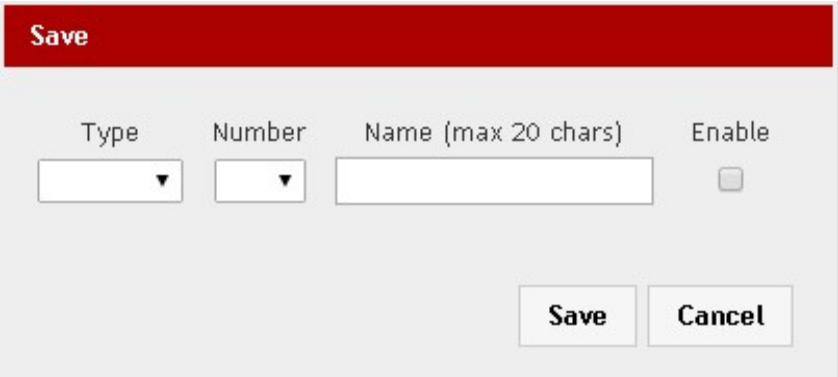
If the compilation is successful, a success message will be displayed and the Save buttons will be enabled. If there was an error during compilation, an error message detailing the line and reason for the error will be displayed. The user then can change the script and attempt compilation again. The example Script below contains a missing right parenthesis on the second line:



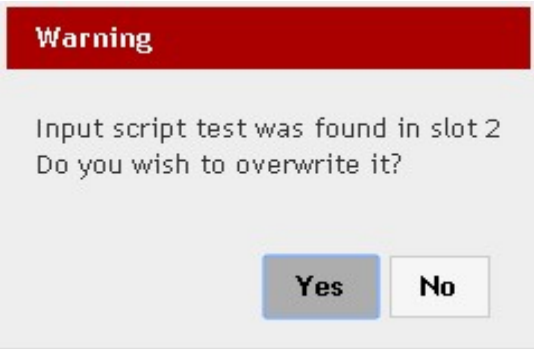
After a successful compilation, the script can be saved.

8.2.2.2 Saving a New Script

To save the script, click the Save button. The save dialog will be displayed. Fill in the information requested in the dialog. Select the “Type” of script and slot “Number”, type the “Name” of the script and select whether the script will be enabled or not. Once the information has been entered, press the Save button in the dialog to save the script. Press Cancel if you want to cancel the process.



If there is a script in the slot already, an Overwrite dialog will appear. Click Yes to overwrite the script in the slot with the new script. Click No to go back to the Save dialog to change the information in the Save dialog to avoid conflicts.



If the save was successful, a successful message will be displayed on the Info tab text area. The Scripts section will reflect the new information in the script.

8.2.2.3 Saving an Opened Script

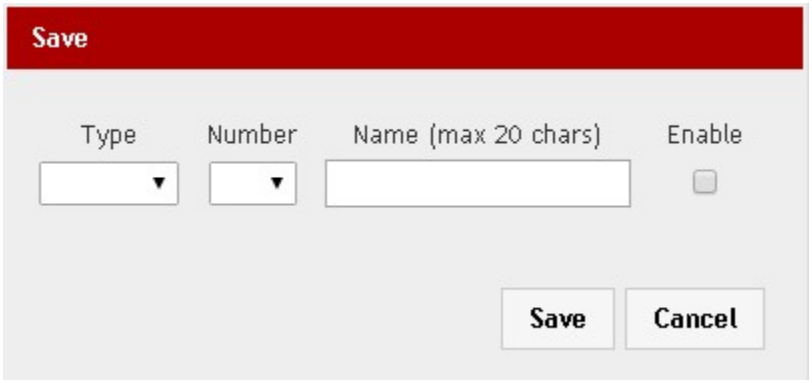
If a script has been previously opened, the user can save any changes to the opened script. To do this, click the Save Button.

If the save was successful, a successful message will be displayed on the Info tab text area.

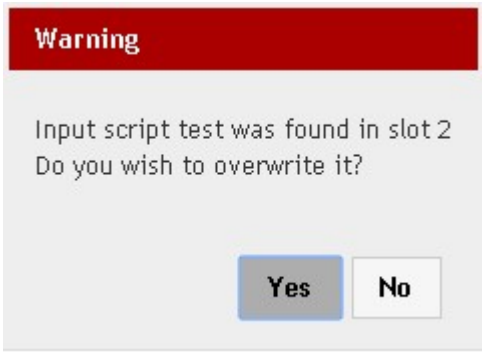
8.2.2.4 Saving a Script in a New Slot

If the user wants to save an opened script in a new slot, click the “Save As” button. The save dialog will be displayed. Fill in the information requested in the dialog. Select the “Type” of script and slot “Number”, type the “Name” of the script and select whether the script will be enabled or not. Once the information has been entered, press the Save

button in the dialog to save the script. Press Cancel if you want to cancel the process.



If there is a script in the slot already, an Overwrite dialog will appear. Click Yes to overwrite the script in the slot with the new script. Click No to go back to the Save dialog to change the information in the Save dialog.



If the save was successful, a successful message will be displayed on the Info tab text area. The Scripts section will reflect the new information in the script.

8.3 Performing Actions on Scripts

8.3.1 Deleting a Script

To delete an existing Script, go to the Scripts section. Locate the script in the input or output scripts tab and click the “delete” button in the action buttons section.

The delete operation will be finalized by displaying a success message in the Info tab text area in the Debug Section. If an error occurs or the operation is invalid, an error message will be displayed in the text area instead.

8.3.2 Enabling and Disabling a Script

To delete a file, go to the Scripts section. Locate the script in the input or output scripts tab and click the enable checkbox in the enable section.

If the enable box is checked, the script will be enabled. If the enable box is unchecked, the script will be stopped. The script status in the status section of the script table will update accordingly.

8.4 Debugging a Script

8.4.1 Script Status

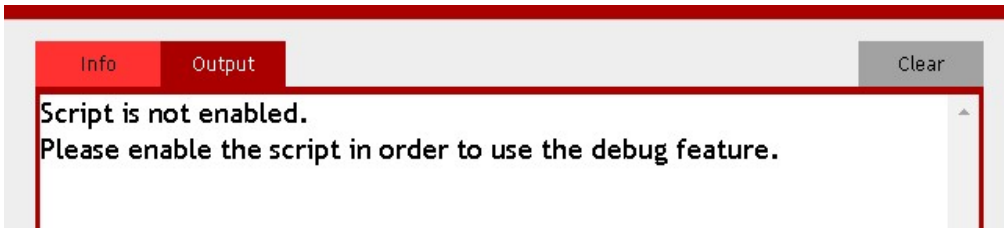
To correctly debug a script, you must understand the script status shown in the scripts table of the Scripts section. These are the possible statuses for a script:

- **Running**: The script is currently running without any errors
- **Stopped**: The script was disabled by the user or has not been enabled previously.
- **Error**: The script had an error when running. Some errors cannot be detected until the Script is run and the error occurs after some time, based on the Script's logic.
- **Ran Too Long**: The script ran without error but its duration to execute exceeded the limit. The script is now stopped. To clear a "Ran Too Long" error, disable the script, then enable the script again. To clear an Error status, it is necessary to correct the error and save the script. To

determine the cause of this error, debug the script or contact Oriux Product Support.

8.4.2 Debugging

To debug a script, the script must be enabled and running or trying to run on the Controller.



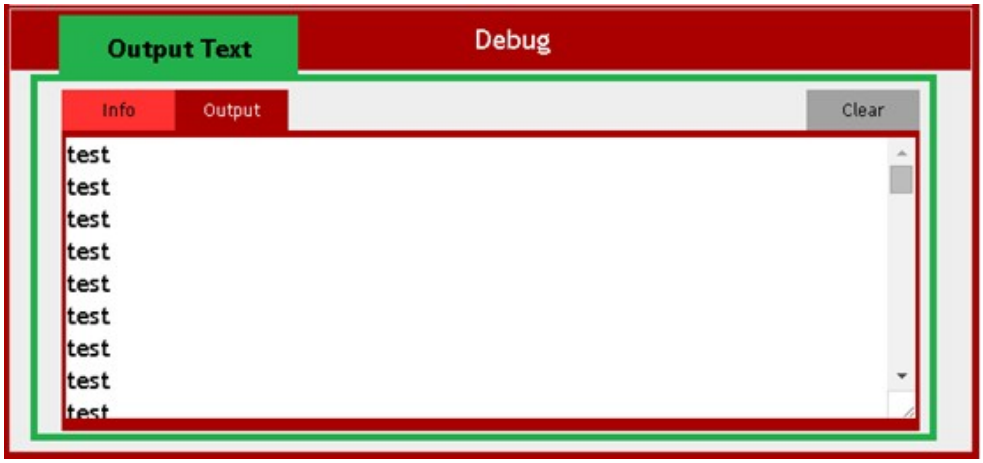
To debug a script, go to the Scripts section. Locate the script in the input or output scripts tab and click the debug button in the action buttons section. This will switch the Debug button for the Stop Debug button and will switch to the Output tab in the Debug section.

If the script status is Running, the Output text area will show the most recent 100 lines of output.

If the script produces more than 100 lines of code, the older lines will be erased, and the new ones will be inserted below.

If the script status is Error, the Output text area will show the error message of the executing script. This will give more information so that the user can correct the errors in the script.

If the script status is Ran Too Long, the Output text area will display an error that will inform the user of this.

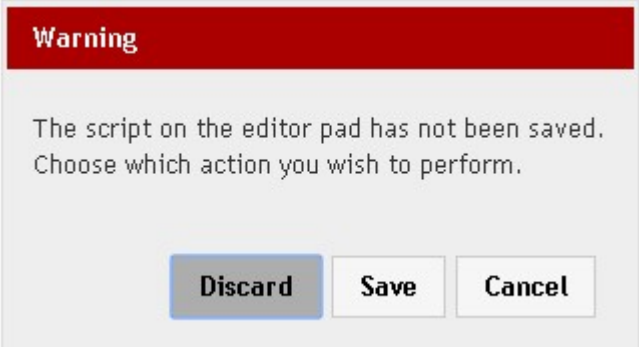


To stop debugging, go to the Scripts section. Locate the script in the input or output scripts tab and click the “stop” debug button in the action buttons section. This will stop updating the Output text area with the output for the script.

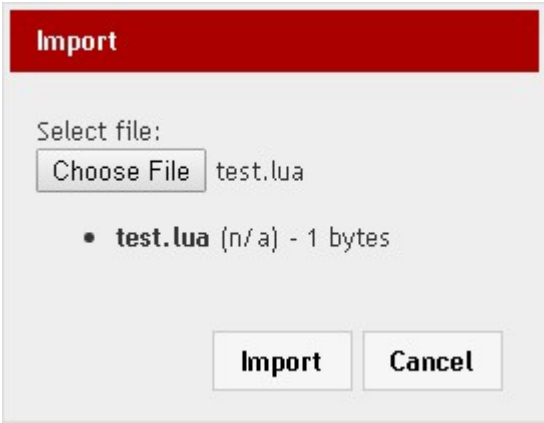
8.5 Importing and Exporting Scripts

8.5.1 Importing a Script

To import a script, click on the Import button in the Editor toolbar. If the scripter detects unsaved content in the Editor text area, a warning dialog will popup. Click the Save button if you wish to save the content in the text area, click the Discard button if you wish to discard the content and click Cancel if you wish to exit the dialog and cancel the open operation.



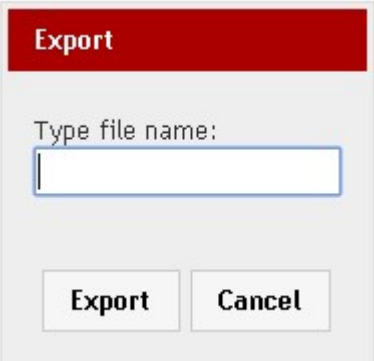
Next the Import dialog will be opened. Click on the Choose file button and a File Dialog will open. Choose the file you wish to import. Only Lua files (.lua) and plain text files (.txt) can be imported. If the file has been accepted, it will appear below the chosen file button. If the file was rejected a failure message will appear. Click the Import button to import the selected file.



The imported file content will be loaded onto the Editor text area. From there, the script can be edited or saved into a slot. If the import failed a message will be displayed on the Info tab text area.

8.5.2 Exporting a Script

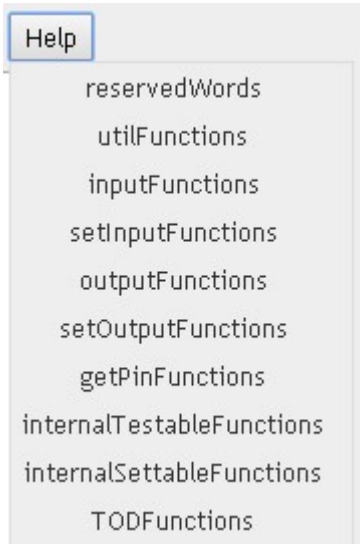
To export a script, click on the Export button in the Editor toolbar. If the scripter detects the Editor text area as empty, a failure message will be shown in the Info text area. If the Editor text area has content, the export dialog will pop up.



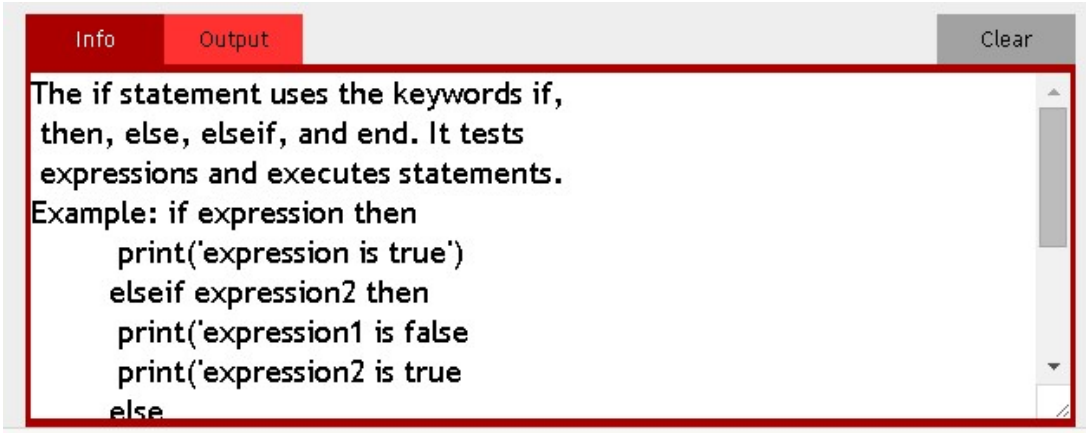
In the export dialog, type the name of the file without extensions. The exported file will always be a Lua file. Click the Export button in the dialog to complete the export. The file download will occur shortly after this.

8.6 Function Help

The scripter built-in functions to perform special actions in the controller. To access these functions, click in the Help button in the Editor toolbar. This will display the Special Functions Help menu, which facilitates creating or editing scripts.



The Special Functions submenu is divided into subsections according to the type of function. Hover over each section to display a submenu listing the functions the section contains. In this function submenu, a function help message will be displayed in the Info text area when the mouse is hovered above it. For example, this is displayed if you hover over the “if” keyword:



Clicking the function will insert it in the Editor text area.