

Reliatronics Inc.
1858 Ranch Road 3232
Johnson City, TX 78636
(830)868-9400 / 868-9900 fax
sales@reliatronics.com
www.relaitronics.com

RiState

State logic programming for the RTU3200

Release 07/02/2010	Misc corrections
Release 02/16/2010	Add reference to "task execution rate"
Release 06/11/2009	IO Attribute "StateOn" was sometimes referenced as "On"
Release 06/10/2009	Major release

Table of Contents

Introduction	4
What are the steps to developing a program?	4
What is state logic?	4
Definition	4
Simple example of state logic	5
A more useful example of state logic	5
Adding a little to our example	6
Structure of a program	8
Basic structure	8
Task and state definitions	8
Task Execution Rate	9
If, then, else	10
Changing states	10
Operators and expressions	13
The assignment operator (equal sign)	13
Arithmetic operators	13
Relational operators	13
Bitwise operators	14
The () operator	14
Compound assignments	14
Operator precedence	15
Variables	16
Predefined local variables	16
User defined local variables	17
The “v_” prefix	18
Expanding the scope of local variables	18
I/O variables	18
IO General Types	18
IO Reference	19
IO Attributes	19
Common IO variable definitions	20
The pre-processor	21
#option option_name	21
#define identifier character_sequence	21
Programming Tools	23

General	23
Editor	23
Compiler	23
What is a program signature?	24
Debugger	25
RTU Firmware Files	26
<i>Example program</i>	28

Introduction

What are the steps to developing a program?

To develop a program, you need the "[RTU3200 Configuration Program](#)", version 6.0 or later, an RTU3200, a means to power the RTU3200, and a communications cable to connect your Windows PC, to the RTU.

There are four basic steps to developing a RiState program.

1. [Create and edit a source file](#). The source file includes all of your programming instructions, in a plain text format. You include programming statements like "if/then/else", operators like "+" and "-", general purpose variables, and variables that give access to the RTU's real time database.
2. [Build, or compile](#) the source file, into a binary file that the RTU can execute. This is called an output file, and contains a list of instructions for the RTU. This file is also very compact, since text names for variables and the like are replaced with numerical pointers, and anything the RTU doesn't need for execution, like comments, spaces, tabs, etc. is not included.

The build process also sets up tables of task and state names, variable names, etc. in the configuration program, to allow several debugging features (see item 4).

3. Upload the output file to a connected RTU. While developing a program, you will most likely upload your recently compiled output file from the [debugger](#) tool. However, you can also upload output files using the "[RTU Firmware Files](#)" menu item, on the configuration program's main menu.
4. Debug the program with the [debugger](#) built into the configuration program. The debugger shows real time program information in English, using the tables and lists, built by the build process (step 2).

See the [Programming Tools](#) section for details.

What is state logic?

State logic describes the operation of a state machine, which, in the context of a state logic program, is a model of behavior defined by some number of states, transitions between those states, and actions executed within those states.

Definition

A state machine is generally defined by the following set of attributes:

- States – A state machine is said to be in one, and only one state, at a given time. The current active state of the state machine, embodies information about the current situation, of a system.

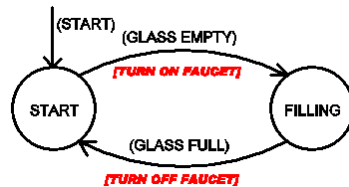
- Events - An event is an input message to the state machine, in the form of a logical or mathematical expression that is evaluated within the current, active state.
- Transitions - A transition is a change from one state to another, usually triggered by an event.
- Actions - An action is an activity performed by the state machine. The action can be performed as a result of a logical or mathematical expression while in a state, or upon the transition from one state to another.

Only one state, of a state machine, is active at a time. Input events, calculation results, timer timeouts and other input stimuli and combinations of other stimuli, cause transitions from one state, to another. Conditions programmed into each state, may activate outputs, trigger additional computations, and cause other outputs conditions to become true.

State machines are easier to visualize, than describe, and the best way to visualize a state machine is with a [state diagram](#).

Simple example of state logic

Our first example is very simple: we illustrate the logic required to fill a glass with water.



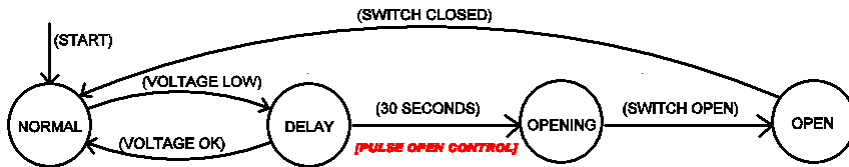
The status of two input conditions (“GLASS EMPTY” and “GLASS FULL”) control the transition from one state to another (“START” and “FILLING”). For now, we will ignore any complexities that involve the input conditions (such as the human brain recognizing a visual image of an empty glass), and concentrate on the sequence that must be executed, to accomplish our goal.

Two actions execute (“TURN ON FAUCET” and “TURN OFF FAUCET”), as the state transitions occur.

The arrow with the input condition “(START)” comes from nowhere, and goes to the state names “START”. This indicates that the “START” state is the active state, when the program first runs. In a RiState program, the initial active state is the first state defined in a task.

A more useful example of state logic

We can program a simple form of auto-sectionalizing, by opening a switch, when the feeder goes dead. In our example, the state machine returns to the normal state, if the switch is closed (by some means other than this program).



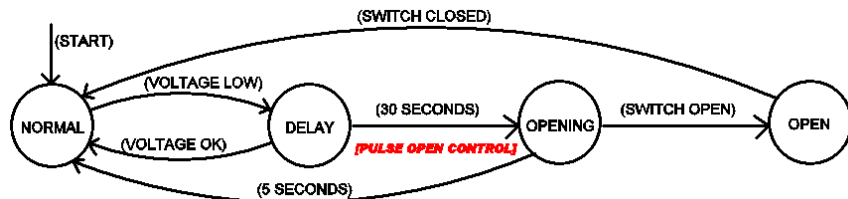
If the feeder voltage goes low, we go to the “DELAY” state. If, while in the delay state, the voltage returns to an “OK” condition, then we return to “NORMAL”, however, if “30 SECONDS” go by while in the “DELAY” state, we “PULSE OPEN CONTROL”, and go to the “OPENING” state. Once the switch is fully open, we go to the “OPEN” state, and remain there until the switch is closed (by some action taken outside of this program).

Again, we concentrate on the basic logic, and ignore the complexities of the input conditions. We might conclude that the voltage is low, by sampling a status input connected to a relay contact whose coil is energized by the power line, or by comparing one of more of the phase voltages to a fixed threshold voltage. We will deal with those details, as we explain statements, expressions, and operators in the following sections.

Adding a little to our example

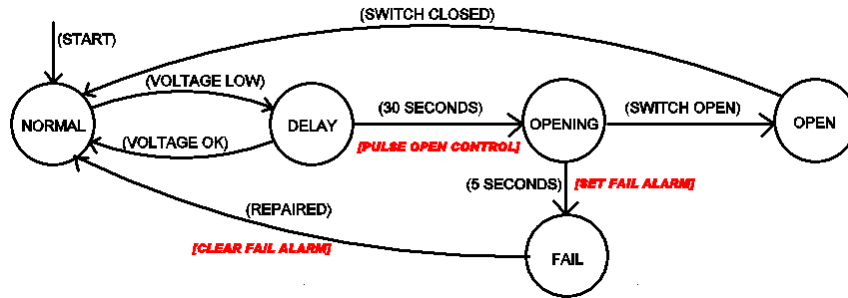
The example above certainly seems reasonable; however, experience tells us that the real world is often unreasonable. What if the switch doesn’t open? The program pulses the open command once, as it enters the “OPENING” state, but if the switch is frozen shut, or broken, or something, the state machine will remain in the “OPENING” state forever.

We could go back to the start state, if the state machine stayed in the “OPENING” state too long.



Now, if the switch doesn’t open within 5 seconds we go back to “NORMAL”. If the voltage is still low, we go to “DELAY”, then pulse the open control and go to “OPENING” once again, after 30 seconds. The state machine will sequence through those three states, every 35 seconds, until either the switch opens, or the voltage is no longer low.

We might add an additional state, to indicate a failed condition:



The fail alarm could be anything from a flashing light at the switch operator, to a status point annunciated on a remote computer host. The “REPAIRED” input could be from a local pushbutton switch, or even a signal sent from a remote source.

Structure of a program

Basic structure

A **program** is set of instructions, stored in a file, created by a program editor and compiler, that can be uploaded to an RTU3200, and executed. The program is stored in the RTU's non-volatile memory, and executes automatically anytime the RTU is powered up.

The program is stored as a single unit in the RTU, as well as a single file on the PC running the Windows application that compiles the program. Programs are identified by their file name.

The RTU executes one and only one program at a given time, but that program may be composed of numerous tasks.

A **task** is a collection of instructions that compose a single state machine. Multiple tasks of a program all run concurrently, and can operate completely independently of one another. In a practical program, one task might perform a periodic load test on a battery system, another task might operate a security system, and a third task might perform an auto-sectionalizing algorithm similar to that described above.

Tasks may be interrelated. For example, one task might perform a test, which determines a piece of equipment needs attention, another task might execute a special algorithm, as a result of that need.

Each task is further divided into multiple states. At any time, each task has one, and only one **active state**.

Each state is composed of a series of statements, expressions, and operators that execute when, and only when, that state is active.

Each task, runs over and over, in a continuous loop, at a rate determined by the [task execution rate](#) argument.

Task and state definitions

You write a program, by adding text, to a simple text file. There's a simple text editor included with the RTU3200 Configuration Program, described in the "[Programming Tools](#)" section.

All of the tasks in a program, are given a name, by prefixing the name with "Task:". The name must start with a alpha character (a-z or A-Z). It is composed of alpha characters, numerals, and the underscore character ("_").

```
Task:MyTask_1
  // Instructions for MyTask_1 task go here
```

compiler.

Note that the second line in the example is a comment. The double slashes ("/") define the start of a comment, so anything that follows (until the end of the line) is ignored by the

Everything that follows the task definition, is part of the task. The task ends, whenever another task is defined, or at the end of the program file.

```
Task:BatteryTest
  // Instructions for BatteryTest task go here
Task:AutoOpen
  // Instructions for AutoOpen task go here
```

This example shows two tasks defined.

Next, lets add some state definitions to a program.

```
Task:AutoOpen
  State:Normal
    // Statements for Start state go here
  State:Delay
    // Statements for Delay state go here
  State:Opening
    // Statements for Opening state go here
  State:Open
    // Statements for Open state go here
  State:Fail
    // Statements for Fail state go here

Task:SomeOtherTask
  State:FirstStateOfSomeOtherTask
    // Statements for Start state go here
```

Note the indentions we used in the example. It's not necessary to indent like that, but it makes your program much easier to read. The indentions make it obvious that that the five defined states all belong to the AutoOpen task.

Also note that there is a state named "Start", in both tasks of our example. State names are local to a task, and tasks are independent from one another so this is allowed.

Each state name within a task must be unique, but state names can be duplicated in different tasks.

Task Execution Rate

By default, each task repeats execution on 100 millisecond intervals. However, intervals of 10 milliseconds, 100 milliseconds, 1 second, or 1 minute can be specified, for each individual task.

```
Task:MyTask1, 0 // run this task every 10 milliseconds
  // Task program statements go here

Task:MyTask2, 1 // run this task every 100 milliseconds
  // Task program statements go here

Task:MyTask3, 2 // run this task every 1 second
  // Task program statements go here

Task:MyTask4, 3 // run this task every 1 minute
  // Task program statements go here

Task:MyTask5 // no rate specified, so run at the default
  // rate of 100 milliseconds
  // Task program statements go here
```

Simply add a comma, and the number 0 through 3, following the task name, to specify a task execution rate.

If, then, else

Just about all programming languages share the [conditional programming](#) feature of “if, then, and else”. Our programming language adds the additional keywords “elseif” and “endif”. The programming structure is straight forward, as described below:

```
if (if_condition) then
    (if_statements)
elseif (elseif_condition) then
    (elseif_statements)
else
    (else_statements)
endif
```

Conditions are a list of statements, that equate to a logical, or true/false result. This is described in more detail in the next section.

We used several lines, in the example above, however, you can put multiple statements on a single line, as shown below:

```
if (if_condition) then (if_statements) else (else_statements) endif
```

Here are the rules:

- “if” must be first
- There must be an “endif” to terminate the whole “if/then/else” set of statements.
- Every “if” and “elseif” has a matching “then”
- “elseif” is optional, and you can use as many as you please
- “else” is optional, but you can only use one

If, then, else statements can be nested, i.e:

```
If (expression) then
    if (expression) then
        statements
    endif
endif
```

The “statements” execute if both expressions are true.

Changing states

We know that a task has one and only active state, and that only the statements of an active state are executed. We know how to set up conditional statements using if,then,else, so we’re getting pretty close to having a working program structure.

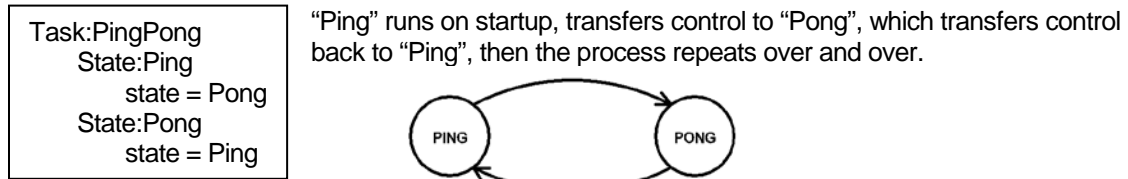
The active state is selected as follows:

- The first state defined within a task, is the active state when the program first runs.
- The active state is changed, to the state name equated to the variable “State”.

The variable "State" is a [predefined local variable](#), described in the next section. It can be set to a new state with the simple expression "State = NewStateName".

In our "AutoOpen" task example, "Start" is the first state defined, so "Start" will be the active state when the task first runs.

An example of a useless program is illustrated below.



For a program to be useful, the state change should be performed, as a result of some condition.

Note that the execution of the expressions within a state, stops, when a new state is selected. For example:

```
State = my_new_state           // my_new_state becomes the new active state
expression_following_state_change // before this expression executes
```

A useful example:

```
Task:AutoOpen
State:Start
  If (voltage_low) then State = Delay endif

State:Delay
  if (voltage_OK) then state = Start
  elseif (StateSeconds >= 30) then
    open_switch
    State = Opening
  endif

State:Opening
  if (switch_open) then State = Open
  elseif (StateSeconds >= 5) then
    turn_alarm_on
    State = Fail
  endif

State:Open
  if (switch_closed) then State = Start endif
```

The expressions shown with the bolded font, are symbols, that we use to represent a group of programming instructions.

There is no instruction “voltage_low” for example, but later on, we will show you how to use the [preprocessor](#), to substitute programming instructions for symbols.

The next section describes how we build “real” expressions, using the RTU’s database and other resources.

We also introduced an additional predefined local variable “StateSeconds”, and the “greater then or equal” [relational operator](#) “>=”, which will be described later, as well.

Operators and expressions

An expression is some sequence of instructions, that equate to a numerical value. Operators are used in conjunction with variables, to construct expressions.

The assignment operator (equal sign)

This is just the good old equal sign, and uses the format: "*variable_name = expression*"

`v_A = 2`, makes the variable `v_A` equal to the number 2.

`v_temp_calculation = v_A + 1`, makes the variable "v_temp_calculation" equal to the variable `v_A`, + 1

`v_A = v_B = v_C = 25`, puts 25 into the variable `v_C`, then puts the value of `v_C` (25), into `v_B`, then puts `v_B` into `v_A`.

Note that the general purpose variables defined in the examples above, all have the "[v_](#)" prefix. Details about the use of this prefix are described below in the "[User defined variables](#)" section.

Arithmetic operators

Operator	Action
-	Subtraction, also unary minus*
+	Addition
*	Multiplication
/	Division

*The unary minus multiplies the expression that follows by -1. For example, the expression `v_A = -5`, multiplies the number 5 by -1, and puts the result into the variable "v_A".

Relational operators

Relational operators compare two expressions, and produce a true (1) or false (0) result. They are used only in the conditional portion of an "if" or "elseif" statement.

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
=	Compare Equal
==	Compare Equal (alternate)
<>	Not equal

These operators compare two numeric values, and produce a true/false result.

Examples:

`(25 > 2)` returns true, or 1

`(25 <> 25)` returns false, or 0

Notice that the equal sign can be used both as an assignment operator (see above) and to compare two expressions for equality.

For example:

```
If v_a = 1 then v_b = 2 endif
```

The variable "v_a" is compared to the number "1", and if equal, the variable "v_b" is set to the value of the number "2".

The double equal sign ("==") may be used to compare two expressions as well.

Logical relational operators compare two Boolean (i.e. true/false) expressions, and produce a true/false result.

Operator	Action
&&	AND
	OR
!	NOT

Examples:

(25 > 2) && (2 < 99) returns true, or 1

Any expression evaluated by an 'if' or 'elseif' statement must contain a relational operator.

For example: "if (Io.BinIn.A0.StateOn) then" will cause a compile error. Use the format "if (Io.BinIn.A0.StateOn > 0) then", etc.

Bitwise operators

Bitwise operators are used to test, set, or shift the actual bits in a storage element (variable or expression result).

Operator	Action
&	AND
	OR
^	Exclusive OR *XOR)
~	Complement (NOT)
>>	Shift right
<<	Shift Left

The () operator

Parentheses increase the precedence of the operations inside them. [Operator precedence](#) is described below.

Compound assignments

We can combine the equate (equal sign) with the arithmetic or bit wise operators, therefore, instead of writing:

(v_A =v_ A + 2) is the same as (v_A += 2)

The supported compound assignments are: -=, +=, *=, /=, &=, |=, ^=, ~=, >>=, and <<=.

Operator precedence

Operators of the same precedence are evaluated from left to right, but some operators have a higher precedence than others. Operators of high precedence are evaluated first. From highest to lowest, the following list shows the order of evaluation:

```
()  
!, ~  
*, /  
+, -  
<<, >>  
<, <=, >, >=  
==, <>  
&  
&  
|  
&&  
||
```

Examples:

$2 + 3 * 4 = 14$ since "*" has a higher precedence than "+", multiply 3 by 4, then add 2 to get 14

$(2 + 3) * 4 = 20$ evaluate (2+3) first, since parenthesis have the highest precedence, then multiply by 4

Variables

Variables are portions of memory, used to store a numeric value.

Predefined local variables

Predefined local variables are defined by the RiState firmware, and are used to store and/or change parameters having to do with the execution of a program.

Variable	Description
State	Name of the current state
StateSeconds	Number of seconds current state has been active
StateMinutes	Number of minutes current state has been active
StateHours	Number of hours current state has been active
TaskSeconds	Number of seconds task has been active
TaskMinutes	Number of minutes task has been active
TaskHours	Number of hours task has been active
InternalIndications	The 16 DNP internal indications flags
SecondsSinceMidnight	The number of seconds since 12:00 AM
SecondsSinceComm	The number of seconds since this RTU has received a communications.

Equate "State" to the name of the next state to be activated. (i.e." State = NextState", causes "NextState" to become the active state.)

Note that the time related variables are dependant on one another. For example:

Executing this expression	Affects the following variables
StateSeconds = 0	StateSeconds, StateMinutes, and StateHours are all set to zero
TaskSeconds = 0	TaskSeconds, TaskMinutes, and TaskHours are all set to zero
StateSeconds = 3600	StateSeconds set to 3600, StateMinutes set to 60, StateHours set to 1

Internal Indication Description	Mask
IIN1_BROADCAST	1
IIN1_CLASS1	2
IIN1_CLASS2	4
IIN1_CLASS3	8
IIN1_TIME	16
IIN1_LOCAL	32
IIN1_TROUBLE	64
IIN1_RESTART	128
IIN2_FUNCTION_UNKNOWN	256
IIN2_OBJECT_UNKNOWN	512
IIN2_PARAMETER_ERROR	1024
IIN2_BUFFER_FULL	2048
IIN2_BUSY	4096
IIN2_CORRUPT	8192

Internal Indications are returned by the RTU in any DNP message, and provide information about the communications status of the RTU. An indication is active, when it's bit within the variable "InternalIndications" is set. You can logically and the variable with a mask, to check that bit.

For example :

```

if ((InternalIndications & 128) > 0) then
.....
// The RTU has been restarted

```

SecondsSinceMidnight is a special timer that is cleared to zero at mid-night, then increments once each second. This variable is cleared to zero, on power-up, and remains zeroed until the RTU is time synchronized by an external host. Once synchronized, the firmware calculates the correct seconds since midnight from the DNP time supplied by the host, then increments that value once each second.

SecondsSinceComm is cleared each time the RTU receives a valid communications from a host, and increments once each second thereafter.

User defined local variables

User defined variables allow the programmer to store numeric values in the course of program execution, and are defined automatically, anytime the variable name is referenced.

For example, in the simple expression below:

v_A = 1

Space in the RTU's memory is allocated for a variable, with the name "v_A". Then the number one (1) is stored in that memory location.

Variable names must start with an alpha character (a-z or A-Z), and be composed only of alpha characters (a-z, A-Z), numerals (0-9), and the underscore character ("_"). A keyword (see appendix) cannot be used as a variable name.

Some sample variable names are listed below:

v_CalculationTotal = v_temp_1 + v_R2D2 + v_x

Local values are local only to the task for which they are assigned, so setting the value of a variable in one task, had no effect on a variable with the same name, in another task.

User defined variables are stored in a 32 bit signed format, in the range of -2^{32} to $2^{32}-1$.

The "v_" prefix

Note that the user defined local variable examples, all have a "v_" prefix. The use of this prefix makes it clear, that the text used to name a variable, is indeed intended to describe a user defined variable. Since we follow the same rules naming tasks, states, and pre-processor symbols (see below), the "v_" (or "V_") prefix makes the programmer's intent clear. If the compiler recognizes a user defined variable, without the prefix, it will issue a warning (but continue to compile). These warnings can be turned off with the option "#option variable_prefix_off..

Expanding the scope of local variables

Local variables are normally only referenced within the task for which they are assigned. However, it's oftentimes useful to reference a variable outside of it's task. To do this, just prefix the variable name with the task name of the defining task, with a period inserted ".".

```
Task:my_task_1
  State:start
    v_var1 = 1
    // additional statements follow
    // .
    // .
Task:my_task_2
  State:Init
    If my_task_1.v_var1 > 0 then
    // additional statements follow
```

The variable v_var1 is first used, and therefore defined in task "my_task_1". However, a different task is able to reference that variable by prefixing the variable name with "my_task_1..

Note that predefined local variables can be referenced globally as well. The second task of our example could have referenced "my_task_1.State" or "my_task_1.StateSeconds".

I/O variables

IO variables allow access to the RTU's database. They are global in nature, since the database is common (global) to all tasks.

IO variables follow the naming convention "io.general_type.reference.attribute". Note that there are three text items, separated by a period ".".

The variable name always starts with "io.", however, case is not important, so "io.", "IO." Or even "iO." are acceptable. (Just don't use the number zero "0" in place of the letter oh "O").

IO General Types

General types are not case sensitive, so the acceptable names can have any combination of upper and lower case letters.

General Type	Recommend name	Acceptable names (shown in lower case)
Binary Inputs	BinIn	binaryin, binin, binaryinput, bininput, obj01, obj1
Binary Outputs	BinOut	binaryout binout, binoutput, obj10
Counters	Counter	counter, obj20
Analog Inputs	Analn	analogin, anain, analoginput, anainput, obj30

Analog Outputs	AnaOut	analogout, anaout, anaoutput, obj40
----------------	--------	-------------------------------------

There is nothing magical about the recommend name, but those are the names we use in our examples.

IO Reference

To reference a particular data point, of data type, include either the absolute address of the point, or the communications index. (See the RTU3200 User's Manual for a comprehensive description of each).

To specify an address, prefix the point number with "A" or "a". To specify an index, prefix the point number with "I" or "i". Numbers without a prefix, are assumed to be a address reference.

Examples:

Reference	Description
A0., a0., 0.	Address 0
I25., i25.	Index 25

IO Attributes

Attributes allow us to differentiate between different details, of a general type. For example, in one instance, you might want to see if a binary input is turned on. In another instance, you might want to see if the same binary input is on-line. Attributes allow us to make that distinction.

Attribute	Value range	Details	Applies to:
Value	-2^{15} to $2^{15}-1$ -2^{31} to $2^{31}-1$		<i>AnaIn, AnaOut Counter</i>
Pulse	0 to $2^{32}-1$		<i>BinaryOut only</i>
Flag	0 to 255		<i>All, status flag as reported via DNP</i>
OnLine	0 or 1	Bit 0 of flag	<i>All</i>
Restart	0 or 1	Bit 1 of flag	<i>All</i>
CommLost	0 or 1	Bit 2 of flag	<i>All</i>
RemoteForced	0 or 1	Bit 3 of flag	<i>All</i>
LocalForced	0 or 1	Bit 4 of flag	<i>All</i>
Chatter	0 or 1	Bit 5 of flag	<i>BinaryIn only</i>
OverRange	0 or 1	Bit 5 of flag	<i>AnaIn, Counter</i>
RefError	0 or 1	Bit 6 of flag	<i>AnaIn only</i>
<i>On, StateOn</i>	<i>0 or 1</i>	<i>Bit 7 of flag</i>	<i>BinaryIn, BinaryOut</i>

Common IO variable definitions

Example Instruction	Description
Io.BinIn.A5.StateOn	= 1 if binary input at address 5 is on = 0 if binary input at address 5 is off
Io.BinIn.I0.Online	= 1 if binary input at index 0 is on line = 0 if binary input at index 0 is off line
Io.AnalIn.A0.Value	Value of the first analog input (Battery voltage)
Io.BinOut.A1.StateOn = 1	Turn the binary output at address 1 on
Io.BinOut.A0.StateOn = 0	Turn the binary output at address 0 off
Io.BinOut.A10.Pulse = 200	Pulse the binary output at address 10 on, for 200 mSec
Io.AnalOut.I0.Value = 1234	Set the analog output at index 0 to the value 1234

The pre-processor

The pre-processor is the first thing that runs, when you build a program. The input to the pre-processor takes the text file you have edited, and performs certain modifications and checks.

One function of the pre-processor is to remove blank lines, comments, and other text, that the compiler does not need.

You can include special instructions in your program, that the pre-processor will use to control the compile process. These are called pre-processor directives.

Pre-processor directives start with '#', and are listed below:

```
#option option_name
```

Some options are:

```
#option ignorecase
```

The pre-processor will convert all variable names to lowercase, so the compiler will treat the variable "v_myvariable" the same as "v_MyVariable".

```
#option comparecase
```

The pre-processor will not change any variable names, so the compiler will treat the variable "v_myvariable" and the variable "v_MyVariable" as two unique variables. This is the default, if no option is specified.

```
#option variable_prefix_on
```

The compiler will issue a warning, when a user defined variable is not prefixed with "v_" or "V_". This is the default, if no option is specified. Basically, the compiler assumes that any text string that does not match a keyword, preprocessor symbol, or io variable format, must be a user defined variable. If you intended to reference a preprocessor symbol in your code, but accidentally misspelled a letter, then the compiler would not match the symbol, but assign a new variable instead. Although this is not an error, it's not what you intended either, so the compiler will issue a warning, unless it see the "v_" prefix.

```
#option variable_prefix_off
```

No variable prefix warning will be issues.

```
#define identifier character_sequence
```

You can make a program more readable, and easier to maintain, by defining certain character_sequences in a easy to find part of your program (like the top). For example, lets say you have a fixed number that appears over and over again in your program. You can define that number at the top of your program, and use it multiple times in your program.

A common example:

```
#define TRUE 1  
  
#define FALSE 0
```

Then, in the body of your program, use the expression:

```
If Io.BinIn.A0.StateOn == TRUE then .....
```

Another example:

```
#define LOW_VOLTAGE_THRESHOLD 1234
```

Later on in your program you may have something like:

```
If Io.Analn.A1.Value < LOW_VOLTAGE_THRESHOLD then .....  
If Io.Analn.A2.Value < LOW_VOLTAGE_THRESHOLD then .....  
If Io.Analn.A3.Value < LOW_VOLTAGE_THRESHOLD then .....
```

If you decide to change this threshold to 4321, then you only have to make one change, at the top of your program (and hopefully, next to a comment that reminds you what this all means).

The #define directive works with any character sequence, and isn't limited to numbers. For example you could write:

```
#define LOW_VOLTAGE_THRESHOLD 1234  
#define VoltagePhaseA Io.Analn.A1.Value  
#define VoltagePhaseB Io.Analn.A2.Value  
#define VoltagePhaseC Io.Analn.A3.Value
```

Then your program could read:

```
If VoltagePhaseA < LOW_VOLTAGE_THRESHOLD then .....  
If VoltagePhaseA < LOW_VOLTAGE_THRESHOLD then .....  
If VoltagePhaseA < LOW_VOLTAGE_THRESHOLD then .....
```

Upon evaluating a #define directive, the pre-processor performs a text substitution for the remainder of the file. This means that you can apply multiple #define directives to the same text.

For example:

```
#define NBR_OF_APPLES 25  
#define NBR_OF_ORANGES 10  
#define NBR_FRUITS (NBR_OF_APPLES + NBR_OF_ORANGES)  
  
fruit_basket += NBR_FRUITS // add apples and oranges to basket
```

Programming Tools

General

All of the tools required to create and install a RiState program, are contained in the RTU3200 configuration program. Download this program, free of charge, from our website.

The link is at: http://www.reliatronics.com/files/rtu3200_configuration.zip.

This is a standard Windows application installation. Unzip the files, then run "setup.exe" to install.

Editor

The first step in writing a program, is to create and edit a text file, with your programming instructions. RiState programs use the file extension ".sl" (state logic). This file is called a "source file".

To start the editor, simply click on the menu item "RiState" at the top of the RTU3200 Configuration Program main screen, then select "Edit".

This file is just a plain text file. In fact, you can edit the file with a text editor like Notepad or Wordpad if you wish. However, the text would have to be pasted into the RiState editor window (or the file opened by the RiState editor) to actually build the application.

Compiler

The RTU executes a binary version of the program, that compiled by the RiState Windows application. This binary file is given the file extension ".s32" when it's created by the compiler. This binary file is called an "output file".

To compile a program, click on the menu item "Build" of the "State Logic Edit" window.

If you have changed any text with the "Program Edit Window", and not saved your changes to your source file, then you will be prompted to save those changes.

Look at the "Message Window" of the editor screen. It will clear when you click "Build", then will log the success or failure of the compiler, along with messages that will help you identify errors in your program. "Fatal Errors" are errors that prevent the compiler from completing, and indicate problems that must be fixed in your program. "Warnings" are not fatal, but indicate situations that might indicate a problem with your program.

If the program compiles without errors, the message "N Task(s) compiled, no errors." Will be visible in the message window (N is the number of tasks compiled).

Next, you will be prompted to save the output file. If you are developing a new program, then you will want to save the output file to the PC's disk drive, so that it can be uploaded to a connected RTU for testing.

If you answer “Yes” to save the output file, you will be prompted to append the program signature to the output file name.

What is a program signature?

The program signature is a unique four character identifier, that is calculated based on the final binary image of this file. (It's actually a cyclic redundancy check, or CRC). The slightest change to the output file, will cause the signature to change.

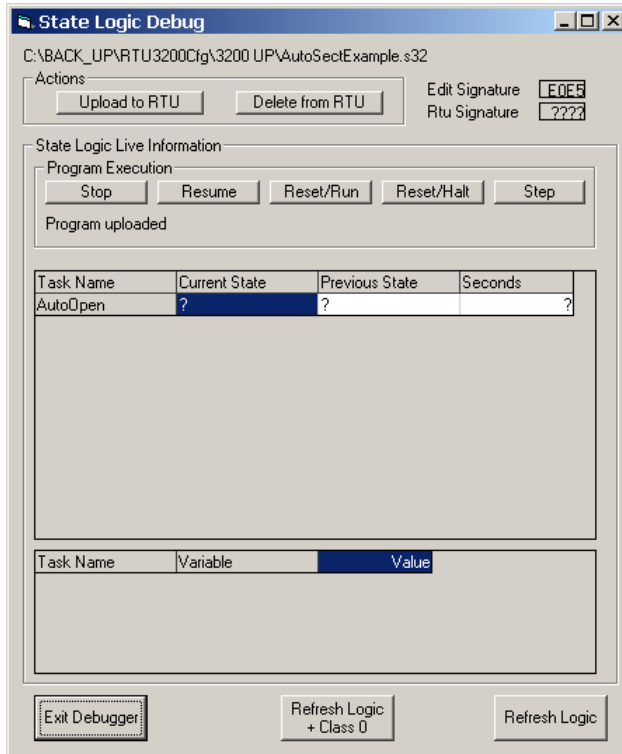
Changes to the source file may not change the signature. For example, simply adding a comment to a source file, would not change an output file, and wouldn't change the signature. If you change anything that the RTU executes though, the program signature will change.

You would most likely append the program signature to the name of an output file that you will archive. That way, you can just read the RiState file name from an RTU at some later date, compare it's file name to the filename in your archive, and if they match, you have an almost absolute guarantee that you have the correct file in that particular RTU.

During development of a program, you may save and upload the output program many times. You're only concerned about the latest version of the output file, and have no need to append the program signature to it's name.

You may not want, or need to save the output file, if you are simply debugging a program already uploaded to an RTU. For example, a connected RTU might have a RiState program that had been previously uploaded, and you want to use the debugger to see the current states, and variable values. Things like the task and state names, variable names, etc. are not contained in the output file that the RTU executes, but if you compile the source file, then those names will be understood by the debugger, and the debugger can display those names as they are written in your source file. When you click on the debugger's “Refresh” button, the items identified on the debugger screen will be updated with live data. If the program signature of the file in the RTU, does not match the program signature of the compiled file, you will receive a warning.

Debugger

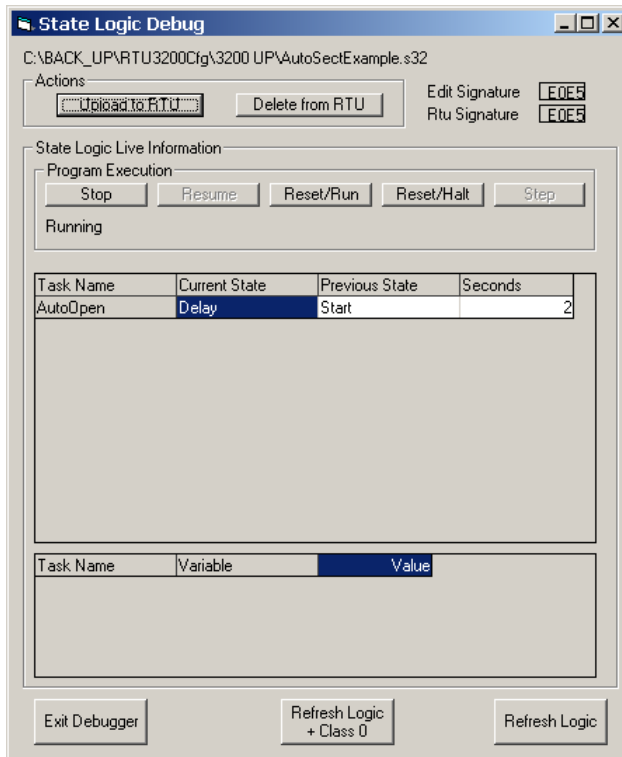


Click on the “Debug” menu item, on the program edit screen, to see the debugger window.

Note the “Edit Signature” in the top, right hand corner. This is the program signature calculated for the program, that was just built, who’s source file is in shown on the editor screen.

The program signature of the RiState file in the connected RTU is not know at this time, so the “RTU Signature” is indicated by “?????”.

If you click the “Upload to RTU” button, and a RTU is connected and powered up, the debugger will upload the output file just built, to the RTU.



After the output file is uploaded to the RTU, the debugger reads the RTU signature, and displays it. It should match the “Edit Signature”.

You may not want to upload a program to the RTU, but instead, debug a program uploaded the day, or week, or year before. In that case, build the source file, but don’t save the output file. Instead of uploading the file, click on “Refresh Logic”. If the signature matches, then you know that the file in the RTU, and your source file match. You can debug, with data from the RTU, but names and descriptions from your source file.

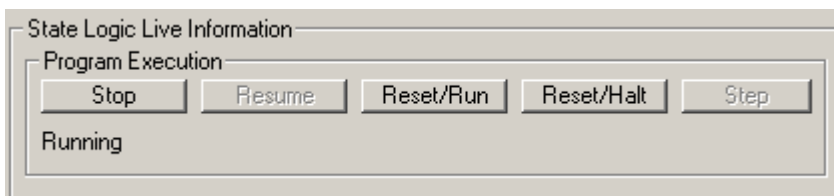
If they don’t match, a pop-up message will warn you.

Click on the “Refresh Logic” button, to update the current state, previous state, state seconds, and variable list fields.

You can also “Refresh Logic + Class 0”, to update the debug window, and the live data tree of the main configuration screen.

Notice the “Program Execution” frame of the debugger below. When you upload a output file, the RTU automatically begins execution the file. The active buttons are:

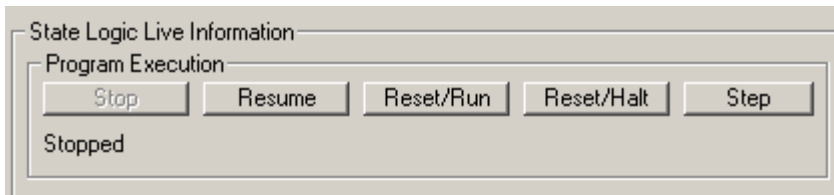
- Stop – Halts execution of the RiState program.
- Reset/Run – This starts the program execution form the beginning of the RiState program. Note that the RTU is NOT reset, just the RiState program.
- Reset/Halt – Resets the RiState program, but does not start execution. The program will be halted at the very beginning.



If you stop execution of the program (by clicking on either “Stop” or “Reset/Halt”, the active buttons will change as shown below.

Two new active buttons are described below:

- Resume – Causes the program to continue running, from the point that it was stopped.
- Step – Causes the program to execute the instructions within the active state of each task, one and only one time. The task and state timers will all increment by one second as well.



Any RiState program, saved in an RTU will run automatically, from the start, whenever the RTU is power up and/or reset, even if the program were stopped by the debugger.

RTU Firmware Files

You can upload RiState files to a connected RTU, by selecting the “RTU Firmware Files” menu item, on the configuration program’s main screen.

The files stored on the connected RTU will be retrieved, then displayed on the file screen.

If no RiState program is stored on the connected RTU, the file description will read "null.s32", and the file length = 0 bytes.

null.s32	time unknown	0	Update
----------	--------------	---	--------

To upload a file from your PC's disk, click on "Update", then select the file from the Window's file selection dialog. After the file is uploaded, the files screen changes, to show the name of the uploaded file, the file time, and length.

autosectexample_e0e5.s32	6/9/2009:16:19:02.000	153	Update
--------------------------	-----------------------	-----	--------

Note from the example above, that a RiState output file, with the signature appended to the file name ("e0e5") was uploaded. This makes it easy to read the file data at a later date (perhaps years from now), and be assured that the file in the RTU, matches the file data archived.

Example program

Back in the first two sections of this document, we outlined a program structure, but didn't include the actual argument details. We will add those details now:

```
#option variable_prefix_on    // this is the default option, so this statement is not required
#define TRUE 1
#define FALSE 0

// Assume the switch controller open contact is wired to status input address 0, close to address 1
#define switch_open    ((Io.BinIn.A0.StateOn == TRUE) && (Io.BinIn.A1.StateOn == FALSE))
#define switch_closed  ((Io.BinIn.A0.StateOn == FALSE) && (Io.BinIn.A1.StateOn == TRUE))

// Assume the switch controller open control is wired to relay address 0,
// and the close control (not used in this example) to address 1,
// and an alarm or some sort, to output address 2
#define open_switch    (Io.BinOut.A0.Pulse = 200)    // open the switch by pulsing output A0
#define close_switch   (Io.BinOut.A1.Pulse = 200)    // not used in this example
#define turn_alarm_on  (Io.BinaryOut.A2.StateOn = TRUE)
#define turn_alarm_off (Io.BinaryOut.A2.StateOn = FALSE)

// Assume a push button switch is wired to the status input at address 3.
// This switch will be pressed, to tell the program that a failed switch operator, has been repaired
#define repaired      (Io.BinaryIn.A2.StateOn == TRUE)

// Connect the contact of a line-powered relay coil to the status input at address 3.
// This contact tells the program that power has been lost.
// Note - We give several other examples for voltage_low and voltage_OK, at the end of this program
#define voltage_low    (Io.BinaryIn.A3.StateOn == FALSE)
#define voltage_OK     (Io.BinaryIn.A3.StateOn == TRUE)
```

```
Task:AutoOpen
  State:Start
    If (voltage_low) then State = Delay endif

  State:Delay
    if (voltage_OK) then state = Start
    elseif (StateSeconds >= 30) then
      open_switch
      State = Opening
    endif

  State:Opening
    if (switch_open) then State = Open
    elseif (StateSeconds >= 5) then
      turn_alarm_on
      State = Fail
    endif

  State:Open
    if (switch_closed) then State = Start endif
```

```

State:Fail
  If (repaired) then
    turn_alarm_off
    State = Start
  endif

```

// Other examples for voltage_low and voltage_OK

```

// Using the binary point Low Line, V1 set by a ACVx module (any phase < normal threshold)
//#define voltage_low      (Io.BinIn.A32.StateOn == TRUE)
//#define voltage_OK      (Io.BinIn.A32.StateOn == FALSE)

```

```

// Using the binary point Dead Line, V1 set by a ACVx module (all phases < dead threshold)
//#define voltage_low      (Io.BinIn.A33.StateOn == TRUE)
//#define voltage_OK      (Io.BinIn.A33.StateOn == FALSE)

```

```

// Compare RMS voltage at V1a to 95/105 volts
//#define voltage_low      (Io.AnalIn.A1.Value < 19456) // 204.8 cnt/volt * 95
//#define voltage_OK      (Io.AnalIn.A1.Value < 21504) // 204.8 cnt/volt * 105

```

```

Task:AutoOpen
  State:Start
    If (voltage_low) then state = Delay
  State:Delay
    if (voltage_OK) then state = Start
    elseif (StateSeconds >= 30) then
      (pulse_output)
      state = Opening
    endif
  State:Opening
    if (switch_open) then State = Open
    elseif (StateSeconds >= 5) then
      (turn_alarm_on)
      State = Fail
    endif
  State:Open
    if (switch_closed) then State = Start
  State:Fail
    If (repaired) then
      (turn_alarm_off)
      State = Start
    endif

```

We need to replace the following expressions with "real" code:

```

(voltage_low), (voltage_OK),
(switch_open),
(switch_closed), (repaired)

```

In addition, we need to replace some actions as well:

```

(pulse_output), (turn_alarm_on),
(turn_alarm_off)

```

.....

//The next section will show the user how to define identifiers, based on IO, for use in the program, i.e.:

```
// assuming open contact wired to status input address 0, close to address 1`
#define switch_open    (Io.BinIn.A0.StateOn && ! Io.BinIn.A1.StateOn)
#define switch_closed  (Io.BinIn.A1.StateOn && ! Io.BinIn.A0.StateOn)
// assuming open control wired to relay address 0, close to address 1, alarm to address 2
#define pulse_output   (Io.BinOut.A0.Pulse = 200)
#define turn_alarm_on  (Io.BinOut.A2.StateOn = 1)
#define turn_alarm_off (Io.BinOut.A2.StateOn = 0)
```

//Then use those identifiers in the program, i.e.:

```
State:Opening
  if (switch_open) then
    State = Open
  elseif (StateSeconds >= 5) then
    (turn_alarm_on)
    State = Fail
  endif
```