

IMPORTANT INFORMATION

Terms Used in This Manual

IMPORTANT	Indicates actions or procedures which may affect instrument operation or may lead to an instrument response which is not planned.
-----------	---

Note	Indicates actions or procedures which may affect instrument operation or may lead to an instrument response which is not planned.
------	---

Tip	Provides extra information about using the program.
-----	---

**Mark of Schlumberger.*

Other company, product, and service names are the properties of their respective owners.

Copyright © 2017 Schlumberger Limited.

All Rights Reserved.

Manual No 50331443, Rev. 01

December 2017

Table of Contents

Section 1—Introduction	5
About the Scanner Logic IDE	5
Downloading and Installing Scanner Logic IDE	5
Installation Requirements.....	5
Section 2—Navigating the Interface	7
Overview	7
IDE Layout	7
Program Information	9
Resources	9
Registers	12
User HMI Fields	13
Results Section	15
Menu Structure	15
Toolbars	16
Document Outline	16
Status Bar	17
The Editor	17
Starting A New Script File	17
Saving a Script File	19
Opening a Script File.....	20
File Auto Recovery.....	20
Program Structure.....	20
Editor Tools and Features.....	21
Scanner Logic Script Compiler	27
Target Platform	27
Compiling a Scanner Logic Program	28
Logic Script Error Types.....	30
Section 3—Managing Device Connections	37
Creating a New Connection with IP Address and Port Known	37
Creating a New Connection with Unknown IP Address.....	38
Removing Device Connection(s)	39
Installing a Scanner Logic Program on a Scanner	40
Uploading a Program to a Scanner via the IDE	41
Uploading a Program to a Scanner via the Web Interface	41
Downloading a Program from a Scanner via the IDE	43
Downloading a Program from a Scanner via the Web Interface.....	43
Uninstalling a Program from the Scanner via the IDE	44
Uninstalling a Program from the Scanner via the Web Interface.....	45
Section 4—Using the Program	47
Script Terminology.....	47
TUTORIAL: Creating a Simple Program	47
Problem	47
Coding the Program	54
Section 5—Debugging Scripts	71
Starting a Debug Session.....	71

Reviewing Device Connections	71
Selecting Target Platform and Device Connection	71
Debug Session Start Sequence	72
Script Execution Status	74
Watch Tree.....	74
System Load Charts.....	75
Debug Status Bar	76
Stepping through Program Code	77
Debug Commands	77
Breakpoints	78
Stop Debugging	79
Appendix A—Other Programs	81
Downloading and Installing ScanFlash	81
Appendix B—Sample Program Solution.....	83
Appendix C—Parser Error Messages.....	91
Appendix D—Runtime Error Codes.....	95

SECTION 1—INTRODUCTION

ABOUT THE SCANNER LOGIC IDE

The Scanner Logic Integrated Development Environment (IDE) allows you to create a script file (SLOGIC file), compile it into a binary program file (SLBIN file), and upload the program to Scanner 3100 devices. Additionally, you can debug operations in the IDE using a graphical format, showing you immediate and upcoming script sections to be debugged. Coupled with the Scanner 3100, the IDE allows users to experience the Scanner's full potential as an automated logic controller.

The programming language used by the IDE is a high-level procedural language designed to build logic controller programs. In this way, the program resembles a state machine. The programming language elements are intentionally limited to a subset of features common to general purpose programming languages. "Helper" forms – dialogs attached to buttons to guide your selections – enable users to build programs successfully with minimal knowledge of the programming language. This makes the Scanner Logic IDE easy to learn and use. The binary (SLBIN) program files are executed by the Scanner 3100 device without affecting other device programming, thereby ensuring the metrological integrity of the Scanner 3100.

Note This manual provides instruction in the use of the programming interface (IDE) and the creation of programmable logic control programs executed by the Scanner 3100. For a detailed description of the programming language, see the Scanner Logic Programmer Manual, which can be accessed from the **Help>Documents** menu in the IDE.

DOWNLOADING AND INSTALLING SCANNER LOGIC IDE

Installation Requirements

Before downloading the Scanner Logic IDE, ensure that your system meets the following minimum requirements:

TABLE 1.1—MINIMUM SYSTEM REQUIREMENTS

System Parameter	Requirement(s)
Operating System	Windows 7 or later
Computer/Processor	1 GHz or faster 32-bit (x86) or 64-bit (x64) processor
Memory	1 GB RAM (32-bit) or 2 GB RAM (64-bit)
Hard Disk Space	150 MB for program files, adequate space for data files
Display	DirectX 9 graphics device with WDDM 1.0 or later driver. Minimum resolution: 1400 × 1024 pixels.

IMPORTANT Before installing Scanner IDE, verify that you have local administrator rights to the computer on which the program will be installed. If you do NOT have local administrator rights or if the installation is blocked, contact your Information Technology department for assistance.

To download and install the IDE,

1. Access the SCANNER 3100 website at <http://www.cameron.slb.com/flowcomputers>.
2. Select **Scanner Model 3100 Flow Computer**.
3. Locate the Scanner Logic IDE under the “Software” heading to the right of the page.
4. Right-click, choose **SAVE LINK AS...**, and select the location to which you want to store the file. By default, the file will be saved to C:\USERNAME\Downloads.
5. Browse to the Installation file and double-click to open.
6. Select **Setup.exe** and run the installation program. By default, the files will be stored to C:\Cameron Data\Scanner Logic IDE.
7. Click the Scanner Logic IDE desktop icon to begin using the application.



SECTION 2—NAVIGATING THE INTERFACE

OVERVIEW

The Scanner Logic IDE is designed to create programs that are uploaded to a Scanner 3100. These programs depend on the Scanner 3100's inputs/outputs configuration for register names and Modbus map locations. You can create a program in the IDE without being connected to a Scanner, but register names and locations can only be populated when connected. For more information about connecting to the Scanner 3100, see [Section 3](#).

Interface components are presented in the order of recommended use, guiding the user through the process to create the script (SLOGIC) file that will be compiled into an SLBIN file that can be executed by the Scanner 3100. References will be made to the Scanner 3100 Web Interface, which is an web-based bridge between the IDE and the Scanner device. For more detailed information about the web interface, consult the Scanner 3100 Web Interface User Manual. For reference, we recommend opening the web interface while scripting.

IDE LAYOUT

Upon opening the Scanner IDE for the first time, the default layout shown in [Figure 2.1](#) will appear. The tools necessary for viewing, creating, modifying, and debugging scripts are available from the menu bar and task bar at the top of the screen and the main areas of the screen, including the following:

- **Resources** tabs and **Results** tabs, which provide access to the helper forms used for scripting
- **Editor** ([Figure 2.1](#)).
 - Create a new script.
 - View and edit an existing script.
- **Start Page** ([Figure 2.2](#))
 - Start a new script or edit a recently-accessed script.
 - Access the Scanner Logic Programmer and Scanner Logic IDE user manuals.
 - Select and view script examples.
 - Access and modify sample applications.
- **Device Connections** ([Figure 3.6](#)).
 - Connect to the Scanner 3100 device that will use the Scanner Logic Script program.

Note Each tab in the IDE can be expanded and repositioned (drag and drop) on the fly to maximize utility. Tabs can also be floated over the screen. To return the IDE to its original layout, choose **View>Restore Default Layout**.

Tip Use <CTRL> + <TAB> to cycle through the open panels in the IDE.

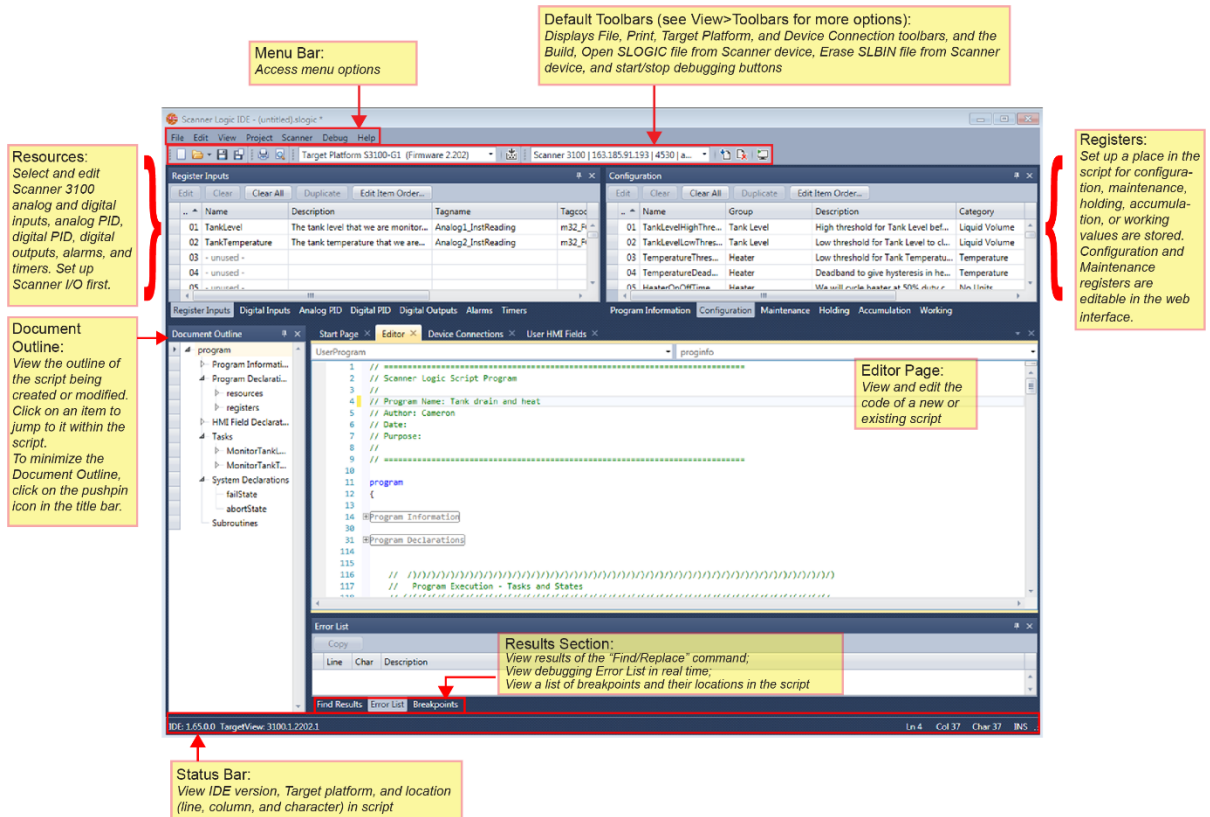


Figure 2.1—Default IDE layout after File>New is selected



Figure 2.2—Start Page

Program Information

The IDE “Program Information” tab allows you to enter program information that uniquely identifies the Scanner Logic Script program, the first step to creating a new program. All the program information entered will be presented in the Scanner web interface. See the Scanner 3100 Web Interface User Manual for more information.

Tip Take the time to communicate the purpose and strategy of your program in the Program Description field. The program description contains enough space to give an overview of the program’s function and can store revision notes as the program evolves.

The Program Information also defines user access levels for features within the Scanner web interface. Online viewing of the program source code, access to the Program Control webpage, and ability to edit the values on the User HMI page can be limited or prevented by configuring the “Access_” parameters:

- **nousers**. Users are prohibited from viewing and/or editing the program.
- **adminusers**. Users with Administrator permissions can view and/or edit the program.
- **configusers**. Users with Configuration Editor permissions (or greater) can view and/or edit the program.
- **maintusers**. Users with Calibration Tech permissions (or greater) can view and/or edit the program.
- **allusers**. Users with Download Access permissions (or greater) can view and/or edit the program.

The compiler validates and incorporates the information in the `proginfo` object into the binary script (SLBIN) output file. This information appears on the IDE “Program Information” tab, as shown in [Figure 2.3](#).

Property	Value
ProgramName	Sample Program
ProgramAuthor	Paige Turner
ProgramOwner	The Oil Company
ProgramVersion	1.000
ProgramCreationDate	10/28/2017
Access_OnlineSource	nousers
Access_OnlineControls	nousers
Access_WriteHMI	adminusers
ProgramDescription	This is a sample program.

Figure 2.3—Program Information tab

Resources

A **resource** object in the IDE provides an interface to inputs, outputs, and device registers of the Scanner 3100. Resource objects have user-assigned identifier names and individual index numbers based on the resource quantity available. Resource objects of the same type are grouped together

within a resource tab. See [Table 2.1](#) for resource types and [Table 2.2](#) for Program Information Resource Dialog Contents.

Each resource object type has its own set of parameters, properties, and methods, collectively referred to as object members. Parameters are defined when creating a resource object. Each parameter setting is guided by the tools in the parameter grids within the IDE. The script author uses properties and methods to access a resource. The properties contain all the runtime data within a resource, while the method calls are used to perform resource specific operations. For a complete list of parameters, properties, and methods for all resources, refer to the Scanner Logic Programmer Manual.

Note In most places throughout the IDE only the resources declared by the script will be displayed. The Parameter Grids are an exception, where an undeclared resource is displayed with the Name “unused.”

TABLE 2.1—RESOURCE DESCRIPTIONS

Resource Type	Usage	Qty	Description
Register Input	Input	32	Allows program to read input values from the Scanner 3100 device.
Digital Input	Input	6	Maps to the digital input ports of the Scanner 3100 to allow reading the state of the ports.
Analog PID	Output	2	Provides a PID controller object whose output can be mapped to a Scanner 3100 analog output port.
Digital PID	Output	1	Provides a PID controller object whose output can be mapped to a Scanner 3100 digital valve controller output.
Digital Outputs	Output	6	Provides a digital output proxy object whose output can be mapped to a Scanner 3100 digital output port
Alarms	Output	32	Provides a controllable alarm object that is accessible by the Scanner 3100 to be used in the same ways as Scanner 3100 device alarms.
Timers	Utility	8	Provides an object that can keep track of the time (in sec) between and execution of the start and stop methods.

As an example, the tabbed dialog for Register Inputs includes the information in [Table 2.2](#), as shown in [Figure 2.4](#).

TABLE 2.2—REGISTER INPUT RESOURCE DIALOG CONTENTS

Column Header	Type Code	Description
Name		A user-provided name for the register item.
Description	<str256>	A user-provided string describing the register item.
Tagname	<tagname>	The descriptive string describing the register tag name.

Column Header	Type Code	Description
Tagcode	<tagcode>	The complete Scanner 3100 register tag descriptor is used source of the input/output.
Category	<category>	The unit category of the S3100 input/output source. Must be selected if the input/output source has a dynamic category, otherwise the category will be set automatically.
Units	<unit>	Numerator of the desired measurement unit for the value imported from the Scanner 3100 register. Also includes the denominator if required for the specified measurement unit category.
Rate	<rate>	Rate scalar unit desired for the value imported from the Scanner 3100 register. If the rate is not specified, then the value is interpreted as not being a rate value.

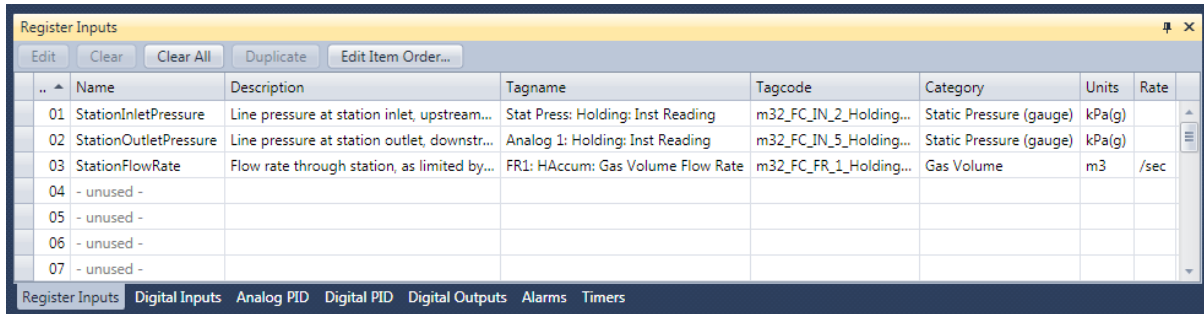


Figure 2.4—Resource Register Inputs tab

Parameter Grids

The parameter grids (*Figure 2.4*) that appear for each Resource or Register type are populated by user entry into the fields and by the document’s parsing in the Editor. The column headers indicate the parameters associated with the program’s language objects. To edit a row in the parameter grid, double-click on the row and click **Edit**.

To copy a row, highlight it and click **Duplicate**. This feature is enabled if an item is selected and an unused resource slot is available. The copy will appear in the first unused resource slot. When you duplicate a row, the entry name will be the original name with an incrementally numbered postfix (e.g. “StationInletPressure” becomes “StationInletPressure1”).

To remove an entry from the Editor script, select the entry and click **Clear**. To remove all the entries, click **Clear All**.

The position and ordering of items within the parameter grids are important. Some interfaces of the Scanner host system only refer to a resource object by its index number. To reposition or regroup the objects in a parameter grid, click **Edit Item Order**. To move multiple items at the same time, <SHIFT>+<CLICK> or <CTRL>+<CLICK> to select a group of items, then drag and drop the items to the desired position.

Adding and Modifying Resources

To access the different tabs (as shown in [Figure 2.4](#)), click on the tab for the desired resource type. To create a new resource entry, double-click in the first empty row and complete the *Edit [Resource Type] Resource Item #* helper form, as shown in [Figure 2.5](#). To modify a resource, click on the desired row in the parameter grid and click **Edit**.

Figure 2.5—Edit Register Input Resource helper form

Tip Using your mouse, hover over a “Parameter Settings” field. The type of content that belongs in the field will be displayed in the “Parameter Hint” field.

Registers

A **register** is a place in the code where a float value is stored. Some register types allow values to be passed between the program and the Scanner 3100. In the script, you can assign values to registers using the `Value` property or by using the name of the register, since `Value` is the default property of registers.

Each register object type has its own set of parameters, properties, and methods ([Table 2.3](#)). Parameters are defined when creating a register object. Use the parameter grid tools to set the parameters. Properties and methods are used by the script author to access a register. The properties contain all runtime data within a register, and methods are used to perform register-specific operations. For a complete list of parameters, properties, and methods, refer to the Scanner Logic Programmer Manual.

TABLE 2.3—REGISTER DESCRIPTIONS

Register Type	Usage	Qty	Description
Configuration	Input	32	Editable on web interface Configuration Registers page. Requires Configuration Editor access level.

Register Type	Usage	Qty	Description
Maintenance	Input	32	Editable on web interface Maintenance Registers page. Requires Calibration Tech access level.
Holding	Output	64	Viewable on web interface Holding Registers page. Allows you to publish register to Scanner systems (archive, display, input source, etc.).
Accumulation	Output	16	Periodic, 64-bit precision data viewable on web interface Accumulation Registers page. Allows you to publish register to Scanner systems (archive, display, input source, etc.).
Working	Internal	64	Scanner Logic programming language global variables.

Configuration and Maintenance Registers

The Scanner 3100 maintains Configuration and Maintenance register input values in nonvolatile memory because they are user-configuration values. The values will persist throughout power cycling of the device or restarting the Scanner Logic IDE after entering the `abortState` or the `failState`.

Although they are functionally the same, the Configuration and Maintenance registers differ in the user account privileges required to modify them on the web interface.

Holding Registers

The Scanner 3100 uses Holding registers to make output values available to systems other than the logic controller (i.e. archives, display, flow runs, web interface). These output values are not stored in nonvolatile memory, but are accessible by the Scanner host environment. Because the output values are not stored in nonvolatile memory, the Holding registers will lose their values if the program is restarted.

Accumulation Registers

The Scanner 3100 records incremental sums using Accumulation registers. The Scanner 3100 maintains current period and previous period totals (daily, interval, and triggered periods) for the values in Accumulation registers and stores these results in nonvolatile memory.

Working Registers

There are no user-defined variables in the Scanner Logic programming language. Use Working registers, which serve as global variables, to store intermediate calculation results or temporary values. They retain their values as program execution moves between states or in and out of subroutines.

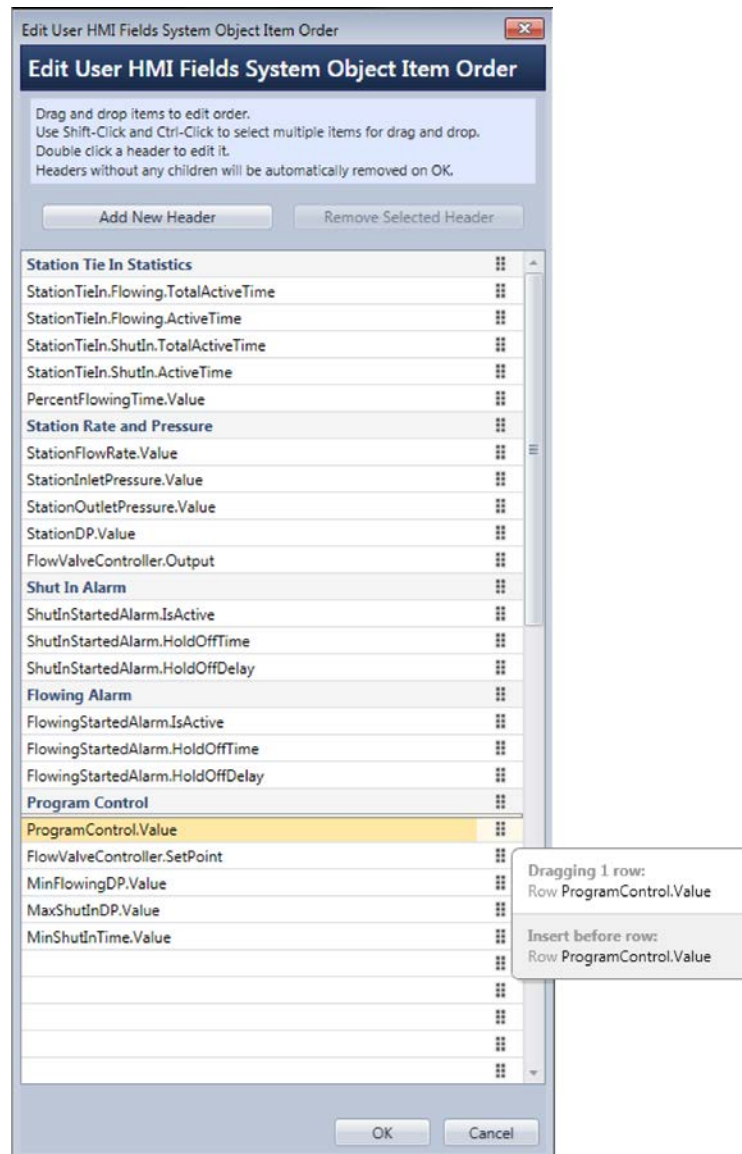
User HMI Fields

One of the most powerful aspects of the Scanner Logic IDE is the ability to consolidate the program information and settings needed by the system operator onto a custom human to machine interface (HMI) page on the Logic Controller HMI Fields page of the web interface. The content of the HMI Fields page involves configuring a list of up to 64 fields which can be organized by group headings. Each field is linked to an object property that has been declared within the program. This link allows

the web interface to display the contents of a property to the operator and potentially allow for the external modification of it.

If an object property linked to a HMI field that has read/write property, the webmodify option is available when creating the HMI Field. If set, webmodify will allow a web interface user to change the run-time contents of an object property within the executing Scanner Logic Script. Care must be exercised by the script author when allowing modification privileges to properties that are actively being managed by the executing script. Such a scenario could create control conflicts and result in a program that does not function as intended. Generally, the author should only set the webmodify option for properties that are not modified within the script.

The Edit Item Order button on the User HMI Fields parameter grid allows users to reorder fields and insert or delete headers.



To add a new header, click **Add New Header**. New headers appear at the top of the grid and can be dragged and dropped to the correct location.

Note Headers cannot be nested. Headers without children are automatically removed when **OK** is clicked.

To remove a header, select the header to be deleted and click **Remove Selected Header**. The header's children will remain in the grid, but will not be organized under the header.

If the executing script contains User HMI Field definitions, the page displaying the HMI fields will be the first page loaded when selecting Control>Scanner Logic Controller on the web interface.

Tip A custom HMI page can consolidate all the information and settings for the system operator onto one landing page of the Scanner 3100 web interface.

Results Section

The Results section of the IDE (*Figure 2.1*) displays information in three tabs.

Find Results

Find Results displays the results of any Find/Replace queries run on the script.

Error List

The Error List displays parser (real-time), compiler (on “Build” or “Start Debugging”), and runtime (during debugging) errors detected.

Breakpoints

The Breakpoints tab allows the user to view and manage the breakpoints set by the Editor using “Toggle Breakpoint,” “Enable All,” “Disable All,” or “Delete All” commands.

MENU STRUCTURE

The IDE menu structure has eight menus. Standard menu items, such as **File>Open** are intuitive and will not be addressed in this manual. The table below describes menu items specific to the IDE.

TABLE 2.4—MENU ITEM DESCRIPTIONS

Menu	Menu Selection	Description
File	Recent Documents	Select from the most-recently opened files.
Edit	Find and Replace	Make consistent changes to a text block in the file.
	Incremental Search	Search script increments from top-to-bottom.
	Reverse Incremental Search	Search script increments from bottom-to-top.
	Mark Line Modifications	Mark changes in lines of script.
	Show Whitespace	View the script with lines inserted between blocks.
	Bookmarks	Manage bookmarked areas of the file.
	Advanced	Perform common context-sensitive edit actions on document.
View	“Document Outline” to “Resources”	Toggle the visibility of the indicated layout item on and off in the interface.
	“Find Results” and “Error List”	Toggle the visibility of the indicated layout item on and off in the interface.

Menu	Menu Selection	Description
View (Cont'd)	Debug	Toggle “Breakpoints,” “Script Execution Status,” “Output,” and “Watch Tree” options on and off in the interface.
	Toolbars	Select the toolbars that will appear beneath the menu. See Toolbars below for more detail.
	Restore Default Layout	Return the interface to its original layout.
Project	Build	Compile the script file into a binary program file (SLBIN).
	Append Timestamp to SLBIN	Add a timestamp to the current script file.
Scanner	Change Target Platform	Change the firmware version for which the script will be compiled.
	Select Device Connection	Select the Scanner 3100 device(s) to which you wish to connect.
	Open SLOGIC	Open an SLOGIC file from a device connection.
	Erase SLBIN	Erase an SLBIN file.
Debug	Refer to Debug Commands	Select tools to use for debugging the script.

Toolbars

From the **View>Toolbars** menu, you can toggle on/off the File, Edit, Text Edit, Bookmarks, and Print toolbars.

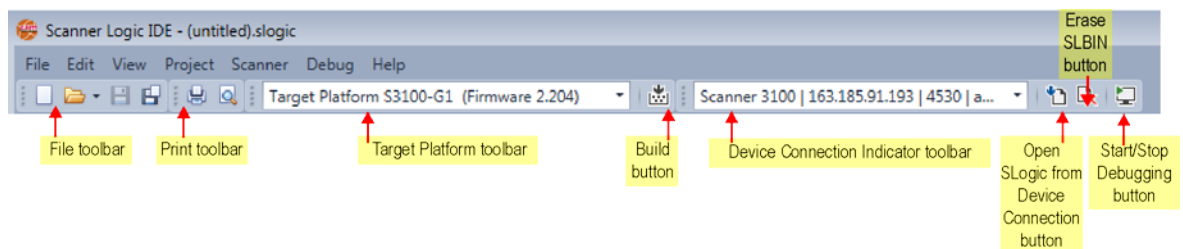


Figure 2.6—Toolbars

As shown in [Figure 2.6](#), the *Target Platform* and *Device Connection Indicator* toolbars are shown by default with the **Build** button between the indicators. The **Open SLogic**, **Erase SLBIN**, and **Start/Stop Debugging** buttons are located to the right of *Device Connection Indicator* toolbar.

Tip Hover over a toolbar button to view the button's function.

DOCUMENT OUTLINE

The Document Outline section of the interface shows an outline of the script document. To skip to a specific place in the script document, click on the appropriate outline item (see [Figure 2.1](#) for location of the Document Outline).

STATUS BAR

The status bar displays the following information:

- IDE version
- Target Platform version
- Debug program state indicator (only visible during Debug mode)
- Scanner program state (identified by color)
 - Blue = Program not running
 - Green = Program executing normally
 - Orange = Program in the Abort state (at breakpoint or halted)
 - Red = Program in Fail state (executing or halted)
- Script editor mode - insert mode (INS) or overview mode (OVR)

THE EDITOR

The Editor is a tool for viewing and changing a Scanner Logic Script file. The entire program source code is contained within a single text-based file and is hosted within the Editor. All other displays within the IDE (including all grid contents and the Document Outline contents) are driven from the textual content within the Editor. Any changes made to the grid data by use of the Edit (“helper”) forms will cause a textual change within the Editor.

The Editor is aware of the context of declaration groups and regions within the Scanner Logic Script file and applies editing properties based on this context. Regions of text can be collapsed and expanded. Text managed by the IDE will be blocked out as read-only with a grey background, and double-clicking within these sections brings linked grids to the foreground of the IDE focus. The specific behavior applied to each of the script regions by the Editor are described later in this section.

The Editor continuously scans and parses the user’s script file when keystrokes are detected, highlighting syntax errors as the user types and enabling tools that aid in the auto-completion of object names and Scanner Logic expressions.

The File actions within the IDE act upon the contents of the Editor.

Starting A New Script File

Selecting **File>New** will generate a new Scanner Logic Script file within the Editor. The source code for a Scanner Logic Script program is contained in a single text-based file having a file name ending with the SLOGIC extension. The new file will contain all the necessary structures and declarations for a minimum Scanner Logic Script program ([Figure 2.7](#)). All the elements within the new template file are required by the compiler to perform a successful compilation.

Once the new file has been generated in the Editor, the user can begin customizing the script file by adding resources and registers using the IDE grids as well as typing within the “Program Code,” “Subroutines,” and “System Declarations” regions.

The newly created file will have the default unassigned file path (“(untitled).slogic”) in editor. A file cannot be saved with the default name, so only the “Save As” option will be available for saving the file (selecting **Save** will trigger **Save As**).

```
// =====  
// Scanner Logic Script Program  
//  
// Program Name:  
// Program Version: 1.0  
// Author:  
// Date: 11/14/2017  
// Purpose:  
//  
// =====  
  
program  
{  
  
    #region Program Information  
  
        proginfo  
        {  
            ProgramName: "";  
            ProgramAuthor: "";  
            ProgramOwner: "";  
            ProgramVersion: 1.0;  
            ProgramCreationDate: "11/14/2017";  
            Access_OnlineSource: "allusers";  
            Access_OnlineControls: "allusers";  
            Access_WriteHMI: "allusers";  
            ProgramDescription: "";  
        }  
  
        #endregion  
  
        // )/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)  
        // Program Execution - User Tasks and States  
        // )/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)  
  
        task Task1  
        {  
            initial state State1  
            {  
  
                onEnter  
                {  
                }  
  
                onLoop  
                {  
                }  
  
                onExit  
                {  
                }  
  
            } // end state State1  
  
        } // end Task1  
  
        #region System Declarations  
  
        // )/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)  
        // Program Execution - System States  
        // )/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)/)  
  
        // -->-->-->-->-->-->-->-->-->-->-->-->-->-->-->-->  
        // Fail State
```


The SLOGIC file format is not target dependent and can be compiled later to different target platforms.

Opening a Script File

Select **File>Open** to open an SLOGIC file and load it into the Editor.

Once loaded, the Editor will scan and parse the contents of the file to populate the data within the Document Outline, the Program Information, as well as the Resource and Register grids. Any syntax or declaration errors discovered in the program source code will be underlined and will appear in the Error List.

Some elements of a program are declared within regions that allow for the collapsing and expanding of the text. When a program is first opened, all regions default to the collapsed state.

File Auto Recovery

In the event of an unexpected closure of the Scanner Logic IDE, an auto recovery file is created from the current contents of the Editor.

The file is stored within the host computer system as “%TEMP%\~AutoRecover.slogic” where “%TEMP%” is the system-provided location for temporary files.

On the startup of the IDE, a check is performed for the presence of the auto recovery file on the host system. If detected, the user is prompted to either open the file or discard it. If the user opens the file, it is like opening a New file template, and will have the default unassigned file path [“(untitled).slogic”] in editor. As with new files, only the “Save As” option will be available for saving the file.

Program Structure

The content of the Scanner Logic program is organized into five regions. Each of these regions is detected by the Editor and user-assisting properties are applied to their text. Most of these regions are collapsible within the Editor, consolidating entire sections of code into a single line to improve visibility of other code. Some regions maintained by the IDE are read-only and appear with light grey text background. Generally, the read-only sections can be double-clicked to launch the contextual grid or dialog used to edit them.

The following sections make up a program and are provided within in the new file template. For a complete description Scanner Logic program elements, see the Scanner Logic Programmer Manual.

Program Information Region

The “Program Information” region contains the `proginfo` object declaration. This object holds the information for identifying the program from within the Scanner web interface.

The region is collapsible within the Editor; however, it is not editable and will be presented with a gray background. The `proginfo` is edited by double-clicking the section within the Editor or selecting **Edit** on the “Program Information” grid. In this way, the IDE manages the contents of the `proginfo` validating user entries within the *Edit Program Information* dialog.

Program Declarations Region

The “Program Declarations” region contains `resources` and `registers` object declaration groups. These declaration groups contain items for Scanner inputs and outputs, as well as user-declared registers that will be used within the program.

This region is collapsible within the Editor. The object type declarations in this section are not constructed by the user. Instead, they are managed by the IDE and are read-only in the Editor. If no object type declarations exist, this section can be omitted from a program. Because the new file template has no pre-declared resources or register objects, this section is omitted until user-defined parameters are declared.

The user can add resource and register objects to this section by selecting a row within a resource or register grid and clicking **Edit**. Each of the resource and register objects have a dedicated *Edit Item* dialog for creating and validating user entries. Once an object declaration has been added to the region, it can be edited by double-clicking the generated text in the Editor or by reselecting the row from the object grid.

See the Scanner Logic Programmer Manual for a complete list of resource and register objects.

Program Code Region

The “Program Code” region contains the task declarations; each task declaration contains state declarations. User code resides in the `onEnter`, `onLoop`, and `onExit` sections of state objects.

This region is not collapsible within the Editor. The “Program Code” region is not managed by the IDE, and the user is free to declare and fill tasks and states to implement programmable control solutions.

System Declaration Region

The “System Declaration” region contains `abortState` and `failState` declarations. These special state objects contain the user code that will execute in the event of a user abort signal or a fatal program error.

This region is collapsible within the Editor. The “System Declaration” region is not managed by the IDE. The user is free to fill the `abortState` and `failState` state objects with custom statements and expressions to direct the programmable control application into a safe operating state.

Subroutine Region

The “Subroutine” region contains the global subroutine declarations. Subroutines are useful for consolidating commonly-used code in one location that can be invoked as required.

This region is collapsible within the editor. The “Subroutine” region is not managed by the IDE. The user is free to declare and fill subroutines to aid the implementation of programmable control solutions.

Editor Tools and Features

Code Navigation Selectors

The Editor hosts two dropdown selections (see [Figure 2.8](#)) which update depending upon where the cursor is positioned within the program. The left dropdown menu contains names of objects or object groups, and the right dropdown menu contains names of sub-items of the selected item in the left menu.

The selected item of each dropdown menu changes to match the object that the cursor is located in. These symbol selectors also act as navigation tools that advance the cursor to the place in the script when an item is selected from the dropdown lists.

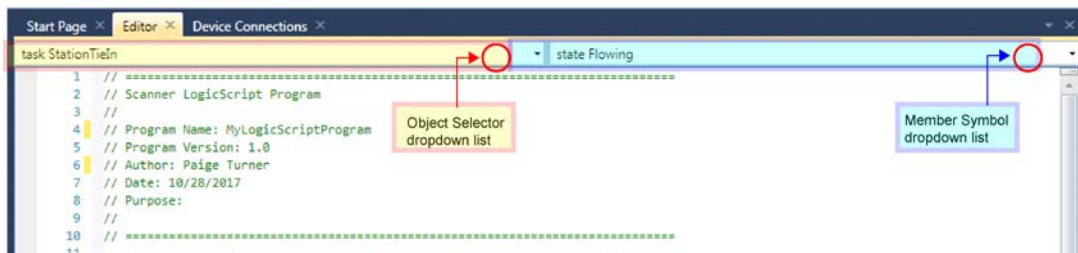


Figure 2.8—Editor code navigation selectors

Margin Indicators

The Editor's left margin conveys information about the status of modifications to the user in Edit mode and in Debug mode.

While in Edit Mode, a colored marker appears to the right of the line number when a line is modified (Figure 2.9). A green mark indicates that a line was changed and the modification has been applied to the file during the previous save action. A yellow mark indicates that changes have been made to the line, but have not been saved.

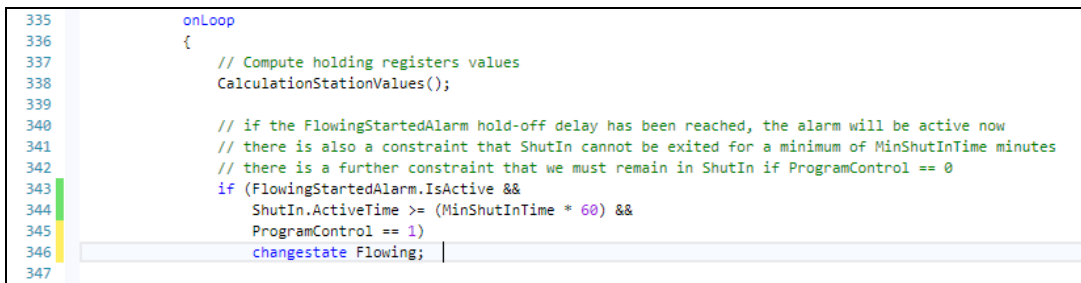


Figure 2.9—Edit mode modification indications

While in Debug mode, the far-left margin contains colored markers for tracking the debug session (Figure 2.10). The user can also insert and remove breakpoints in the margins in Debug mode.

The current executing line is indicated with a yellow arrow. If the program is actively running, the arrow will relocate to the current execution line every second. If the program is halted, performing a step execution will immediately relocate the yellow arrow to the new line.

If a breakpoint has been inserted on a program line, the editor will place a red circle in the margin and highlight the statement or expression where the debugging will stop.

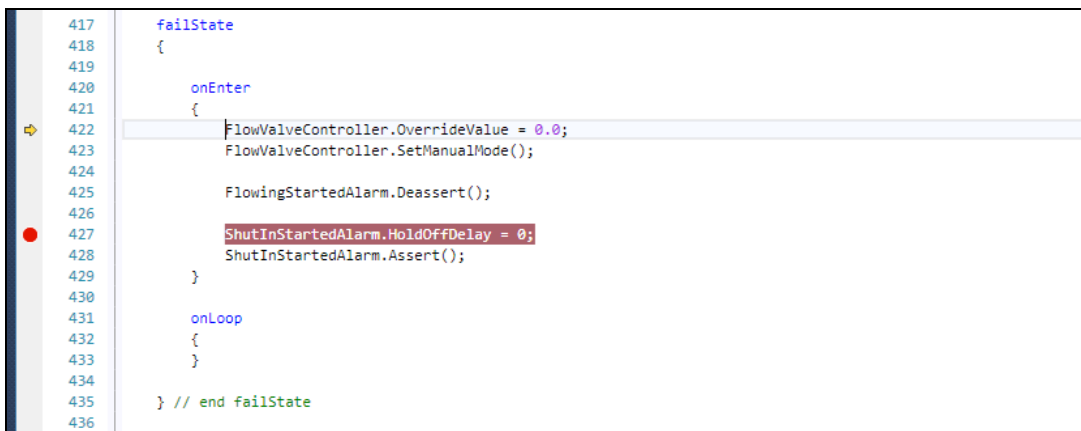
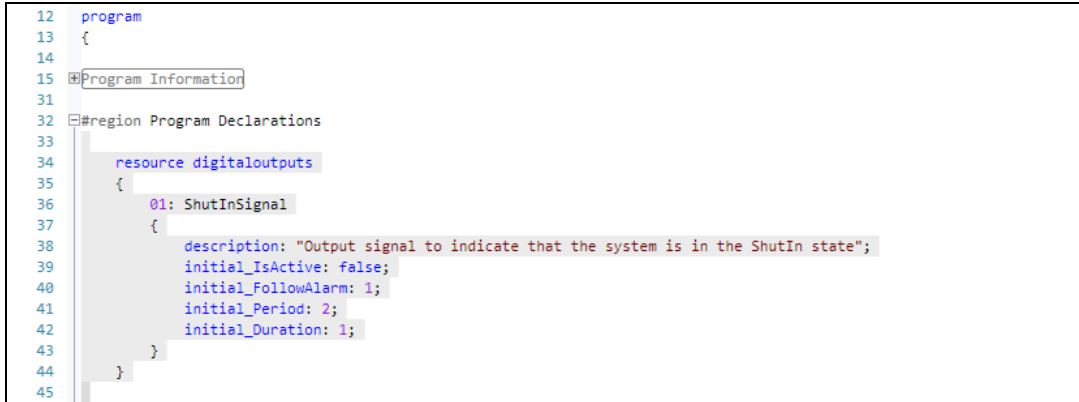


Figure 2.10—Debug mode execution and breakpoint indicators

To insert or remove a breakpoint, left-click within the debug margin or the line in which the action is desired. For more details, see [Breakpoints](#).

The Editor “Collapse/Expand” region appears to the right of the line numbers ([Figure 2.11](#)). Collapsible regions are indicated with a minus sign and a box around the region name. Expandable regions are indicated with a plus sign and the expanded area will appear on a grey background.



```

12 program
13 {
14
15 #Program Information
31
32 #region Program Declarations
33
34 resource digitaloutputs
35 {
36     01: ShutInSignal
37     {
38         description: "Output signal to indicate that the system is in the ShutIn state";
39         initial_IsActive: false;
40         initial_FollowAlarm: 1;
41         initial_Period: 2;
42         initial_Duration: 1;
43     }
44 }
45

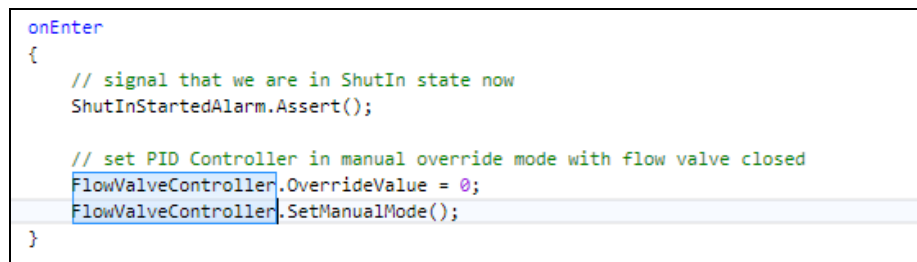
```

Figure 2.11—Editor collapsible region controls

Edit Functions

The Editor supports standard text editor functions, such as **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, **Delete**, **Find**, **Replace**, and **Select All**.

To select the text to be edited, you can use your mouse or keyboard. The Editor also supports a text block selection tool. To select text blocks using this tool, simply press and hold the <ALT> key and click within the editor.



```

onEnter
{
    // signal that we are in ShutIn state now
    ShutInStartedAlarm.Assert();

    // set PID Controller in manual override mode with flow valve closed
    FlowValveController.OverrideValue = 0;
    FlowValveController.SetManualMode();
}

```

Figure 2.12—Editor text block selection

Syntax Parser

The Editor scans and parses the program file as you type, allowing for real-time error reporting and for the identification of new regions, objects, statements, and expressions. Any errors detected within the program body are underlined in red. The Error List will also display the errors found.

```

onEnter
{
    // signal that we are in ShutIn state now
    ShutInStartedAlarm.Assert();

    // set PID Controller in manual override mode with flow valve closed
    FlowValveController.OverrideValue = 0;
    FlowValveController.SetManualMode();
}

```

Figure 2.13—Editor syntax error underlining

The Syntax Parser allows quick access to an object's definition from the object name within an expression or statement. Right-click on the object name and select **Go To Definition** to jump the cursor to the Editor definition.

Tip With the cursor placed on an object name, press F12 to view its definition.

Quick Information Tips

To view detailed information about an object, hover the mouse over the object name. The information will be presented in a *Quick Info Tip* window. User declaration information, including the user-provided name and description, and a description of the general function of the object is provided.

```

onEnter
{
    // signal that we are in ShutIn state now
    ShutInStartedAlarm.Assert();

    // set PID Controller in manual override mode with flow valve closed
    FlowValveController.OverrideValue = 0;
    FlowValveController.SetManualMode();
}
onLoop
{
    // compute noizing registers values
    CalculationStationValues();
}

```

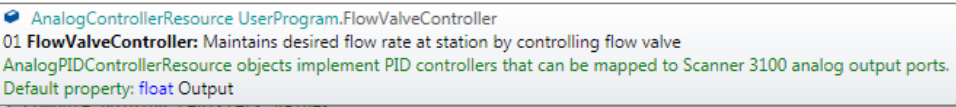

FlowValveController: Maintains desired flow rate at station by controlling flow valve
 AnalogPIDControllerResource objects implement PID controllers that can be mapped to Scanner 3100 analog output ports.
 Default property: float Output

Figure 2.14—Editor Quick Info Tips for objects

Hover over the associated property or method member of an object to display the information about the owner of the member and a description of the member or property purpose.

```

onEnter
{
    // signal that we are in ShutIn state now
    ShutInStartedAlarm.Assert();

    // set PID Controller in manual override mode with flow valve closed
    FlowValveController.OverrideValue = 0;
    FlowValveController.SetManualMode();
}
onLoop
{

```

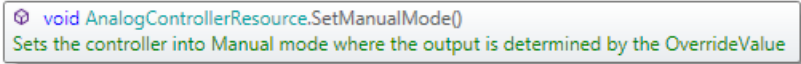

void AnalogControllerResource.SetManualMode()
 Sets the controller into Manual mode where the output is determined by the OverrideValue

Figure 2.15—Editor Quick Info Tips for object members

Auto-Completion Lists

In the Editor, an auto-completion feature assists you in choosing keywords and identifiers quickly, and can also automatically complete partially typed identifiers. Simply press <CTRL>+<SPACE> to

activate the auto-completion tool. Other actions that trigger the Auto Completion list to open are the following:

- Typing text after a math operator symbol
- Typing text after an assignment operator
- Typing text after an open parenthesis

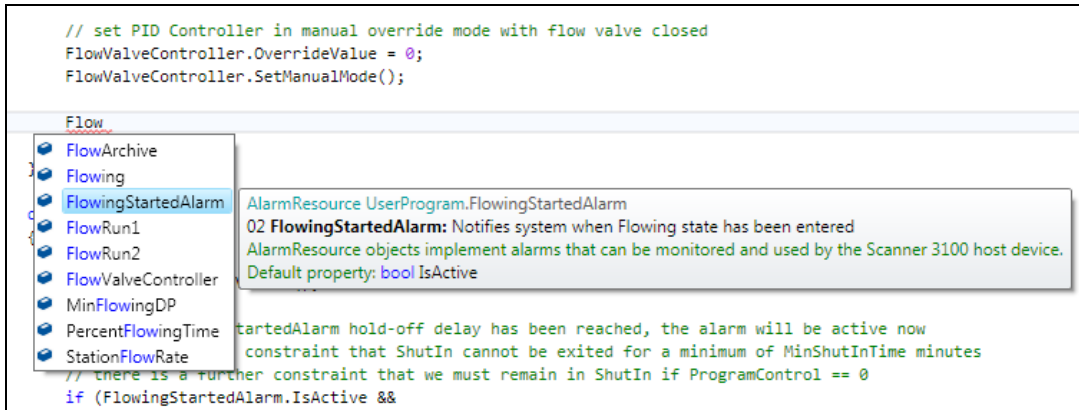


Figure 2.16—Editor auto-completion lists

Use the <UP ARROW> and <DOWN ARROW> to scroll through and highlight items from the auto-completion list. Whether contained within items or at the start of items, the typed text will incrementally search and filter the completion list without case sensitivity. As each item in the list is highlighted, Quick Info Text will appear to aid the user in selecting an appropriate item. Double-click the desired selection to automatically replace the text. Other actions that will trigger replacement of text include:

- Press <ENTER> to select highlighted item
- Press <SPACE> to select highlighted item
- Press any of the following characters: { } [] () . , ; + - * / % & | ! = < > ? ' "

If there is only one possible match and an auto-completion list is not open, place the cursor inside a partially-typed word and press <CTRL>+<SPACE> to automatically complete the word without opening the list.

Member Lists

When an object's text identifier plus a "." (dot operator) has been typed into the Editor, the Member List will appear. This list will initially contain all property and method members of an object. Whether at the start of items or contained within items, typed text will incrementally search and filter the Member List, regardless of text case. As each member in the list is highlighted, the item's Quick Info Text will appear to help the user select the appropriate item. Double-click the desired selection to automatically insert the text. The following actions will also select the highlighted item and automatically insert the existing text:

- Double-click entry in list
- Press <ENTER> to select highlighted item
- Press <SPACE> to select highlighted item
- Press any of the following characters: { } [] () . , ; + - * / % & | ! = < > ? ' "

Within the list, an object's properties are indicated with a wrench glyph and its methods are indicated by a cube glyph.

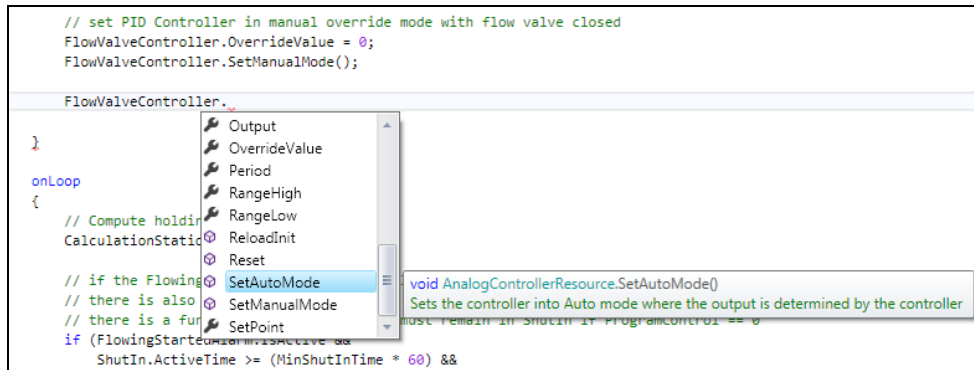


Figure 2.17—Member list

Vertical Split

Multiple sections of a program can be viewed within the Editor by using a split partition tool. When the Vertical Split partition is hidden (default), the vertical split handle will appear as a small shaded box just above the top arrow of the Editor scroll bar, as shown in [Figure 2.18](#). Slide the vertical split line down to view two scrollable partitions.

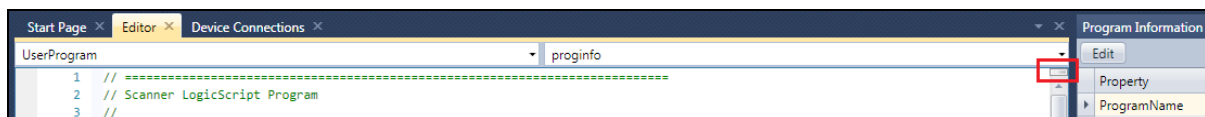


Figure 2.18—Editor hidden vertical split handle

By dragging this handle, you can create a custom view of two locations in the program, each of which are editable.

When the vertical split partition is in view ([Figure 2.19](#)), grab and drag the partition to adjust the view. To remove the vertical split view, return the partition to the top of the Editor.

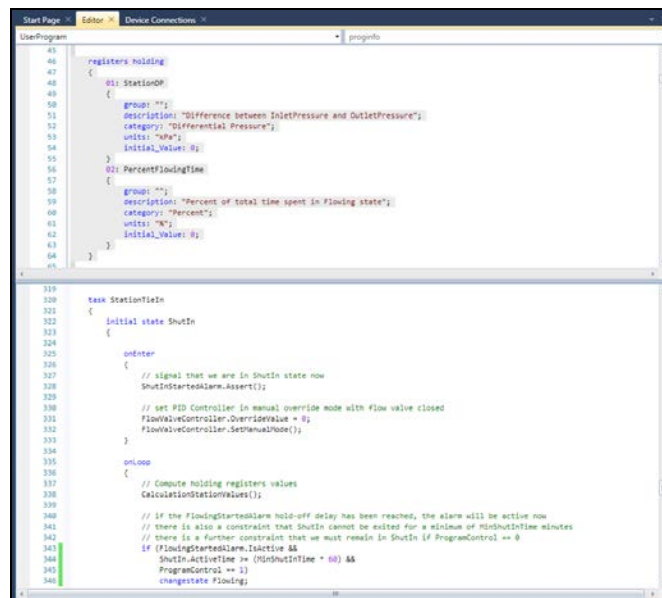


Figure 2.19—Editor vertical split view

SCANNER LOGIC SCRIPT COMPILER

The Scanner Logic Compiler is a part of the Scanner Logic IDE. The compiler transforms the user-created Scanner Logic script file (SLOGIC, the source code) into a form usable by the Scanner (SLBIN, the object code). This process involves preprocessing, lexical analysis, parsing, and semantic analysis (syntax-directed translation). The Scanner Logic program source code is essentially broken down by the compiler and reconstructed with simple low-level commands in binary form for the Scanner to execute.

The compiled object code (SLBIN file) is packaged with system information and stored in a file format which can be installed on the Scanner. An SLBIN file must be created for use by a specific Scanner target platform.

Target Platform

The compiler target platform is defined by the combination of the Scanner model and the firmware version of the device for which you are compiling the SLBIN file. A Scanner will only validate and execute an SLBIN created specifically for its target platform.

The *Device Connections* dialog can be used to discover the required target platform for a connected Scanner. If a device can be found after clicking **Refresh All**, the device model and firmware version will be displayed under the device information columns. See [Section 3](#) for instructions on how to create a device connection.

The IDE will include a list of supported target platforms. If your installation does not support your required target platform, an IDE software update is required. For IDE software updates, go to <http://www.cameron.slb.com/flowcomputers>, select Scanner 3100 Flow Computer and locate the “Software” section in the right column. To install, right-click on the desired software, choose **Save Link As...** and select the desired location for the installation file. Clicking on the software name will download the installer to C:\\Users\\Username\\Downloads.

Note Currently, Scanner 3100 is the only model supported by the Scanner Logic IDE.

Changing the Target Platform

The target platform must be selected before compilation. To change the target platform, choose **Scanner>Change Target Platform**. From the *Change Target Platform* dialog ([Figure 2.20](#)), select the device model and firmware version. Click **OK** to load the new target platform to the IDE.

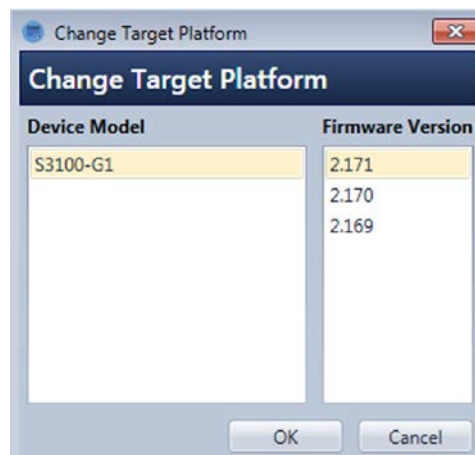


Figure 2.20—Change Target Platform dialog

The Target Platform Selection tool can be used to select a platform from a list of recently-selected target platforms, as shown in [Figure 2.21](#).

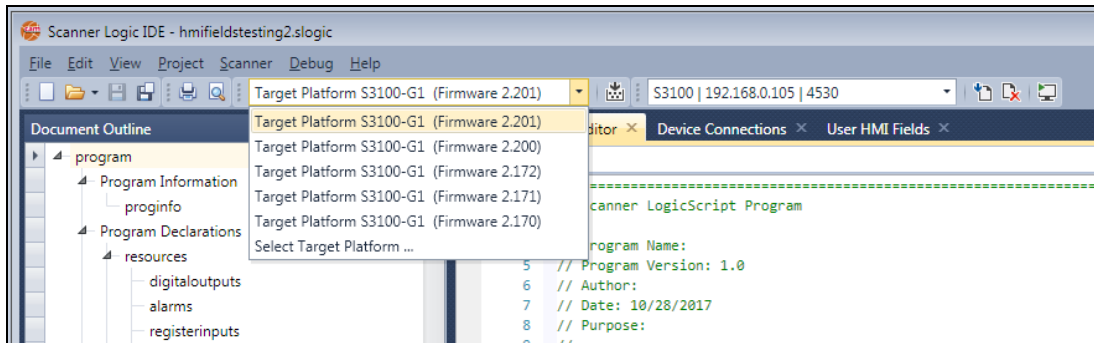


Figure 2.21—Target Platform Selection tool

Target Platform Properties

When a target platform is selected in the IDE, several properties of the IDE environment are also changed. For each target platform on a user system, the following properties are persistent and will be restored when a target platform is selected.

- **Edit Windows Layout.** Any customization of the position of the windows, grids, and tools made while in the Edit mode will be stored.
- **Debug Windows Layout.** Any customization of the position of the windows, grids, and debugging tools made while in the Debug mode will be stored.
- **Language Parser.** A target platform may contain expanded Scanner Logic capabilities and features.
- **Device Capacity.** A new Scanner target platform may have expanded resource or register objects lists available. This may include increased number of existing objects available or added object types.

If a target platform is selected for the first time on a Scanner Logic IDE installation, windows layouts and options will be at their defaults.

Compiling a Scanner Logic Program

To compile a script loaded into the Editor, select **Project>Build** on the menu bar.

Building a program primarily consists of compiling the Scanner Logic script (SLOGIC, the source code) and creating the SLBIN output file (object) for the selected target platform.

Program Build Process

The IDE will start the following sequence of actions to build a program:

1. A save of any unsaved changes to the Scanner Logic Script open in the Editor.
2. A scan for any present parsing errors. Any syntax errors still present in the script file (e.g. typos, syntax errors, etc.) will terminate this process. User will be notified that the build was unsuccessful ([Figure 2.22](#)) and the Error List grid will be updated with parser errors.

Compiler is invoked to produce an SLBIN file. Errors can occur during this compilation process and are described later in this section. If a compiler error is detected, the build is terminated and the Error List grid will be updated with the compiler errors.

- The SLBIN file is stored in the folder that holds the script (SLOGIC) file. The output file is automatically named with the following format.

<Name_of_slogic_file>_<DeviceModel>_<Firmware x1000>.slbin.

If any “.” characters exist in the user SLOGIC file name, they will be replaced with the “_” character.

If the file already exists in the target working directory, it will be overwritten. If this is not desirable, enable the option to append a time stamp to the output file name. This option is enabled from the menu **Project>Append TimeStamp to SLBIN**. A time stamp with the form “_YYYYMMDD_HHMMSS” will be added to the name produced above.

- A build message displaying the complete file path and name of the produced file will appear ([Figure 2.23](#)).

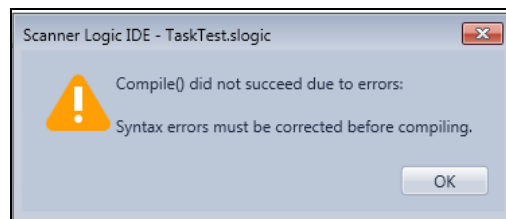


Figure 2.22—Unsuccessful Build message

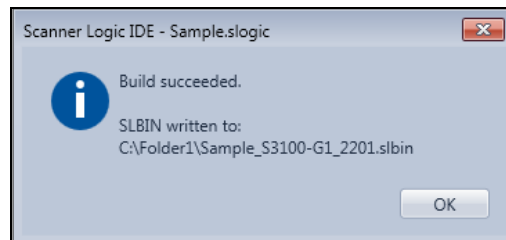


Figure 2.23—Successful Build message

The SLBIN File Format

The SLBIN file format is a packed collection of many binary parts. The Scanner flow computer will evaluate a SLBIN file for both integrity and authenticity before executing of the Scanner Logic program.

The following binary parts make up the SLBIN file:

- **Program Description.** All program information including the description provided by the program author. This information is available in the Scanner 3100 web interface at **Control>Scanner Logic Controller>Program Info**.
- **Object Tables.** All details of the user’s resource and register objects usage, including all display formatting information used by the web interface.
- **Execution Tables.** All low-level object code for tasks, states, and subroutines executed by the Scanner.
- **Source Code.** A compressed copy of the original Scanner Logic file (SLOGIC) that enables the downloading of the source code from a Scanner that is already executing a SLBIN.

The source code can be viewed on the Scanner 3100 web Interface by a user with sufficient access permissions. A link to open a new webpage containing the user source code is located at **Control>Scanner Logic Controller>Status** and is labeled “Program Source Code.”

- **Program Report.** A detailed listing of all resources used by a Scanner Logic program. For some Scanner interfaces, such as a Modbus serial protocol interface, the user names for objects are not available. For these device interfaces, the Scanner Logic resources are referenced by an index number. All these indexes are available in the Program Report.

The Program Report can be viewed by the user on the web interface. To open a new webpage containing the Program Report, select **Control>Scanner Logic Controller>Status** and is labeled “Program Reference Report.”

Logic Script Error Types

Parser Errors

As you type program code, the Scanner Logic language parser runs continuously in the background and attempts to recognize coding structures, keywords, operations, and identifiers in your code. When the parser cannot interpret the code because of syntax errors, incomplete or erroneous code structures, or typographical errors, it reports parser errors to the Error List panel. The errors include a message describing the problem and show the associated line number and column number of the error location. Double-clicking on an error will move the text cursor to the corresponding location of the error. At this location, there will be one or more words underlined in red indicating the source of the error. Hover the cursor over the underlined text to view the error message. The same error message is displayed in the Error List.

[Appendix C](#) contains a list of parser error messages that you may encounter. These errors are generally the result of coding mistakes. Correct the mistakes indicated, and the error messages will disappear when the code is reparsed. Refer to the Scanner Logic Programmer Manual if necessary.

Compiler Errors

Compiler errors may be detected when you attempt to compile the program. This error category relates to the content and structure of the binary output file (SLBIN) that is generated by the compile process. There are limits to the size of internal sections of the file, as well as restrictions intended to preserve the integrity of the metrological functions of the Scanner 3100. The Scanner Logic program is not allowed to exceed safe boundaries on size, stack memory usage, and CPU time slice usage. The goal is that the primary functions of the device will not be interrupted or corrupted, no matter what happens within the logic controller functional module.

When a compiler error occurs, no binary output file is generated. The errors must be resolved before the script file can be compiled successfully. The following compiler errors may be encountered.

TABLE 2.5—COMPILER ERRORS

Error	Description
Unable to compile script. Reduce code complexity by splitting statements with large numbers of operations into separate statements.	This error can occur when mathematical statements of extraordinary length are written with numerous terms, functions, or nested operations. Resolve the error by splitting the large statements into multiple smaller statements and combine the results together.

Error	Description
Maximum line count of 65536 exceeded.	Programs with many blank lines or many comment lines may become extremely long. Line numbers are embedded within the compiled binary code to allow the integrated debugger to correlate execution point with locations within the source code. For efficiency, the line number is limited to a 16-bit number. Resolve the problem by deleting blank lines, or consolidating multiple lines of comments or code into fewer lines.
Missing ProgInfoDeclaration node.	The compiler uses the information in the proginfo declaration block to embed into the binary output file. This error occurs if the block is missing. This situation could only occur if the program file has been edited outside of the IDE, and the proginfo block has been corrupted or deleted. Refer to the Scanner Logic Programmer Manual for the correct format of this block. Resolve the error by reversing the editing of the file, or retrieving a back-up copy of the program file.
Missing TaskDeclarationSection node or TaskDeclaration nodes.	This error occurs if all the tasks have been deleted from the file. At least one task must be declared, containing at least one state. Resolve the error by reversing the deletion of the task code.
UserSelections INC object size (xxx) exceeds maximum limit of 64KB.	The binary slbin file contains various sections. The UserSelections section provides information that the web interface uses to construct the Logic Controller screens. The space allocated for this information is limited, and contains the user-entered descriptions for all objects, including resource objects, registers objects, and HMI field objects. There are also entries for each of the declared tasks, states, and subroutines. The information from the Program Information block is also embedded in this section, which includes the Program Description field. If this object should ever exceed the maximum space allocated for it, you can attempt to resolve the error by reducing the length of the Program Description and user-entered item descriptions.

Error	Description
SourceFileZip INC object size (xxx) exceeds maximum limit of 256KB.	The content of the original slogic source file is included within the binary slbin file as a compressed object, so that the program can be extracted by the IDE from a Scanner 3100. If the compressed source code should ever exceed the maximum space allocated for it, resolve the error by reducing the size of the source code, including the user-entered descriptions and the comments within the file.
SourceListingHTML INC object size (xxx) exceeds maximum limit of 256KB.	An HTML rendered copy of the contents of the original slogic source file is included within the binary slbin file, with fonts, spacing, and coloring preserved to match the appearance of the code within the IDE editor window. It can be viewed via the web interface if access permissions have been set to allow it. If the HTML version of the source code listing should ever exceed the maximum space allocated for it, resolve the error by reducing the size of the source code, including the user-entered descriptions and the comments within the file.
ProgramReportHTML INC object size (xxx) exceeds maximum limit of 256KB.	The Program Report is an HTML file that is available to be viewed from the Logic Controller pages in the web interface. It lists the objects used in the program along with their index numbers. It is a useful resource when reading Logic Controller related data from the Scanner 3100 via Modbus, to determine which numbered registers to read to obtain the desired values. If this report should ever exceed the maximum space allocated for it, resolve the error by reducing the length of user-entered descriptions. In extreme cases, you may need to reduce the number of objects used in the program.

Error	Description
<p>The code in an execution path starting in State1 exceeds maximum allowed execution time (detected: 28.564, maximum: 20.0). Distribute your code over additional states to reduce execution demand per second.</p>	<p>The script execution engine in the Scanner 3100 executes the Scanner Logic Script program code in a once per second cycle. Typically, the execution cycle will begin at the top of an onLoop block of a state, and will proceed until either the bottom of the onLoop block is reached or a transition to another state occurs. If the latter case occurs, the onExit block of the current state is executed, then the onEnter block of the new state is executed, and then the execution cycle ends. At the next one second cycle, the execution begins at the top of the onLoop block of the new state. Depending upon how the program is written and how many different states there are, there can be many different code execution paths that could occur in any given script execution cycle.</p> <p>The time that it takes to execute the code in any cycle must fit within a small slice of one second to preserve the integrity of the Scanner 3100 real time operations. The amount of time that any code path can take is analyzed by the compiler and if any of the paths takes longer than the maximum execution cycle time, a compiler error is generated. In other words, too much code is being put into the onEnter/onLoop/onExit blocks of one or more states. Since a code path may include a transition to another state, the total amount of code starting from the onLoop block of the first state, including the onExit block of that state, and ending in the onEnter block of the new state, including the code in any subroutines called along the way, needs to be considered. Resolve this error by changing the design of your program to reduce the amount of code to execute, or by splitting the code among more states, effectively spreading the execution time for the code over more execution cycles.</p>

Error	Description
<p>The code in an execution path starting in State1 exceeds maximum allowed device stack depth (detected: 1002, maximum: 992). Avoid overly complex statements.</p>	<p>The stack is a data structure used by the script execution engine in the Scanner 3100 to keep track of intermediate operation results as it runs the program code. Stack space is a fixed resource, to prevent programs from compromising the primary operations of the Scanner 3100 by using up excessive amounts of memory. Complicated mathematical statements can have many intermediate results that are eventually combined to produce a single value for the statement. Each one of these intermediate results use up stack space. The compiler analyzes the program code and predicts the stack space usage of all program statements, and generates a compiler error if the stack space requirements exceed the maximum allowed. Resolve this error by splitting overly long and complicated mathematical statements into separate smaller calculation statements.</p>
<p>The code in an execution path starting in State1 exceeds maximum number of UserEventRecord.CreateEventRecord() calls (detected: 24, maximum: 20).</p>	<p>Creating an event record with the CreateEventRecord() method of a UserEventRecord system object requires a certain amount of system resources in the Scanner 3100. To protect the device from being overburdened by the total usage of these resources within one script execution cycle, there is a restriction on the maximum number of CreateEventRecord() calls that can occur within any code execution path. A code execution path is a possible sequence of program code that begins at the top of the onLoop block of a state and continues until either the end of that onLoop block if no state transitions occur. If there is a state transition, the code execution path flows through the onExit block of the state, into the onEnter block of the new state and stops at the end of the onEnter block. Code paths include any subroutines called along the way. The compiler analyzes the number of CreateEventRecord() calls within all possible code paths, and a compiler error is generated for any code paths that exceed the maximum number of such calls. Resolve the error by reducing the number of CreateEventRecord() calls that occur within one script execution cycle. You may be able to resolve the problem by adding more states in a chain to spread out the CreateEventRecord() calls into separate consecutive execution cycles.</p>

Error	Description
SLBIN file size (xxx) exceeds maximum limit of 512KB.	The maximum size for a compiled binary output file is limited, to restrict the usage of memory resources in the Scanner 3100. The binary SLBIN file includes program information, object parameter declarations, compiled source code, metadata for web interface, a compressed copy of the original source code, an HTML rendered copy of the source code for web display, and a program report for cross-referencing object indexes. Normal program sizes fall well within the maximum limit. Typical SLBIN file sizes are 30KB to 60KB. If the SLBIN file size should ever exceed the maximum allowed size, attempt to resolve the error by reducing the amount of comment text or object description text in the program. If this is not enough to reduce the file size, you may need to refactor the program to reduce the number of objects (resource, register, HMI field, task, state, subroutine) being used.

Runtime Errors

While executing the SLBIN file, the Scanner flow computer analyzes all low-level commands for correct and legal usage before they are executed. Any errors discovered in the compiled SLBIN during execution are called runtime errors. If a runtime error occurs, the Scanner web interface will report this serious and unexpected error. Cameron support should be contacted and provided with details and the source code (if possible) so that the program can be resolved with an update to the compiler. [Appendix D](#) contains a numeric list and the details of all possible runtime errors.

This page is left blank intentionally.

SECTION 3—MANAGING DEVICE CONNECTIONS

Connecting to a Scanner 3100 is a required step for uploading programs, downloading programs, and starting debug sessions with the Scanner.

Device connections contain all the information needed to contact and communicate with a locally or remotely installed Scanner, including user account information. To perform actions on a Scanner using the IDE, the user must have Configuration level or greater security access to the Scanner 3100.

The IDE must connect with a Scanner over a physical Ethernet connection. When connecting to remotely-installed Scanners, the remote network administrator must ensure that the proper port forwarding rules are in place to make the device reachable from outside the network.

Creating a New Connection with IP Address and Port Known

If you know the IP address and port address of the Scanner, create a device connection as follows:

1. Click the “Device Connections” tab in the center section of the IDE.
2. Click **New** from the Device Connections toolbar.
3. Enter the following in the *New Device Connection* dialog ([Figure 3.1](#)):
 - a. Connection Name—A user-specified name for the connection.
 - b. IP Address—The IP address of the device to which you are trying to connect. For a local device, this is the IP address of the device on the local network. For a remotely installed device, this is the gateway IP address to the remote network.
 - c. Port Address—The TCP port address of the device to which you are trying to connect. The Scanner’s logic debug port is always Port 4530. For a remotely installed device, enter the port number being internally forwarded to the remote IP address on the remote subnet. A network administrator may need to set up a port forwarding rule in the remote network router to all the TCP protocol packets to pass through.
 - d. User Name—The user name for the device to which you are trying to connect. The provided user account configured with Configuration Access or greater.
 - e. Password and Verify Password—The password for the device to which you are trying to connect.
4. Click **Verify TCP** to determine if the TCP/IP connection can be found at the provided IP address and port number. See [Figure 3.2a](#) and [Figure 3.2b](#) for the possible responses. If the device was not found, check your settings and/or network configuration, and try again.
5. Click **OK** to exit the Verify TCP dialog.
6. Click **OK** to apply the New Device Connection settings. If the information you entered is correct, the device will appear under the “Status: Found” heading. If not, the device will appear under “Status: Not Found,” which means either (1) the device does not exist on your network or (2) the device information was entered incorrectly. See [Figure 3.3](#) for an example of the “Status: Not Found” result. Note that the device information is “Unknown.”
7. To try again, click in the field to the left of the connection name and click **Edit**, verify the device’s IP address and Port address, and repeat steps 2 through 4. If the device remains not found, contact your IT Administrator for assistance.

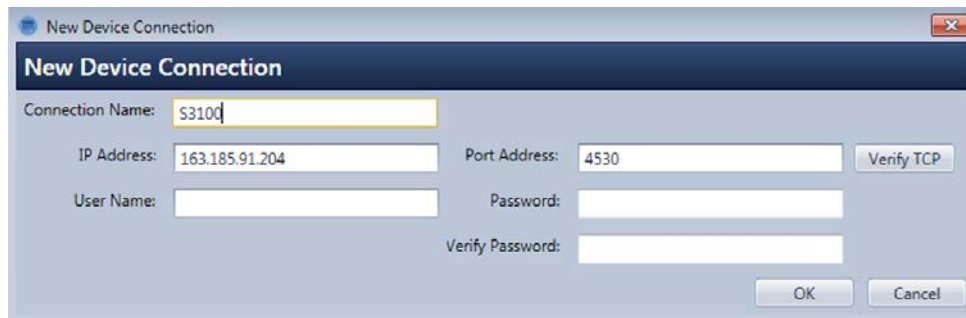


Figure 3.1—New Device Connection dialog

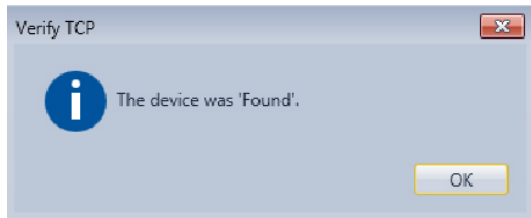


Figure 3.2a—Device Found

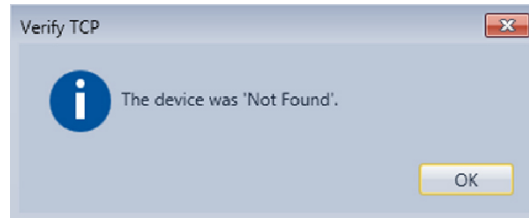


Figure 3.2b—Device Not Found

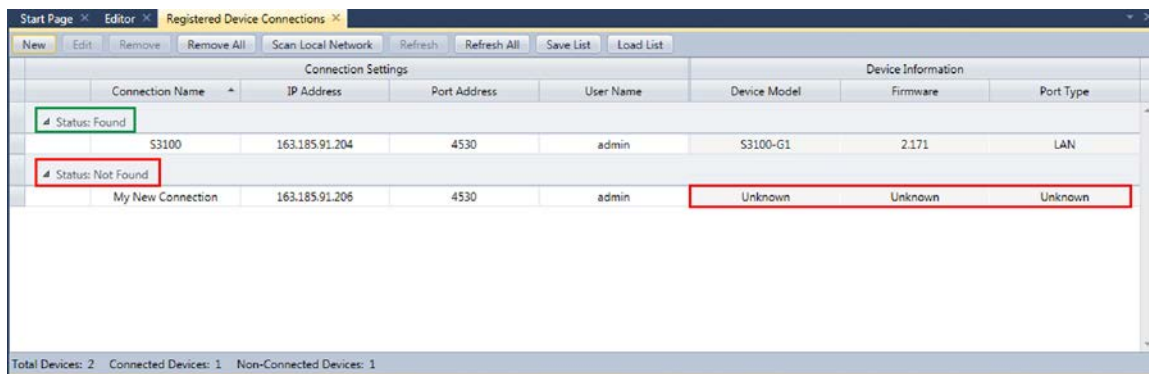


Figure 3.3—New Device Connection, Status: Not Found

Creating a New Connection with Unknown IP Address

To create a device connection to a Scanner 3100 installed on the same local network subnet as the computer hosting the IDE:

1. Click the “Device Connections” tab in the center section of the IDE.
2. Click **Scan Network** from the *Device Connections* toolbar.
3. From the *Scan Network* dialog (Figure 3.4), click in the box to the left of the device to which you want to connect and click **Add**. Repeat this step for each device to which you want to connect.
4. In the *New Device Connection* dialog (Figure 3.5), enter the following information.
 - a. Connection Name—User-specified name for the connection.
 - b. User Name—The user name for the device to which you are trying to connect. The provided user account must be configured with Configuration Access or greater.
 - c. Password and Verify Password—The password for the device to which you are trying to connect.
5. Click **OK** to connect to the device, which should now appear on the “Device Connections” tab in the Main screen, as shown in Figure 3.6.

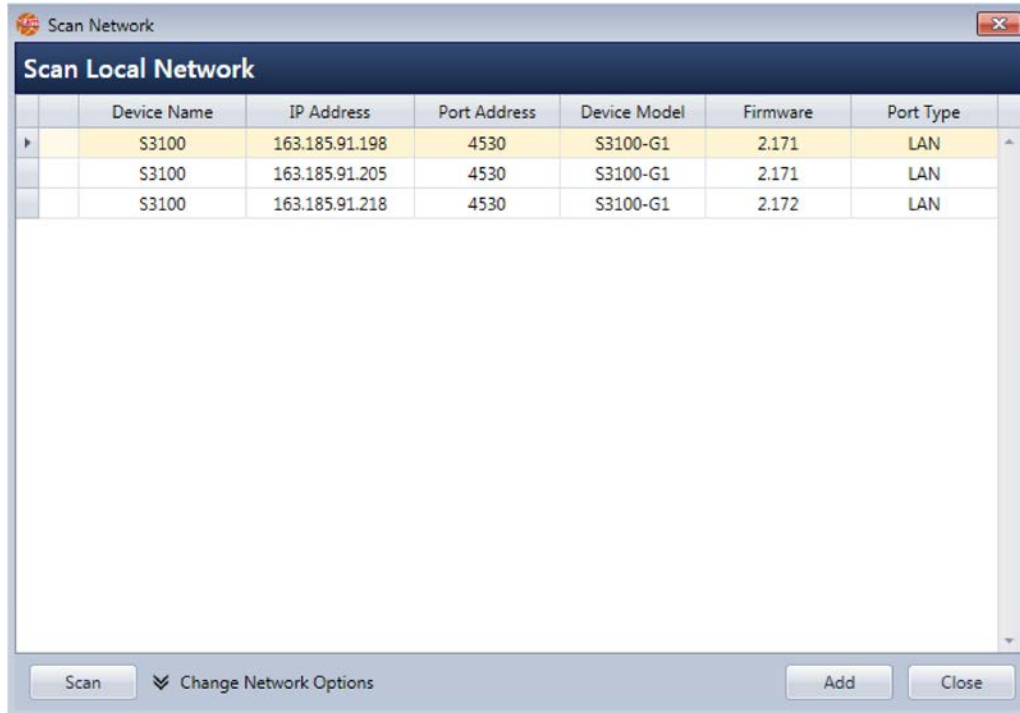


Figure 3.4—Scan Local Device dialog

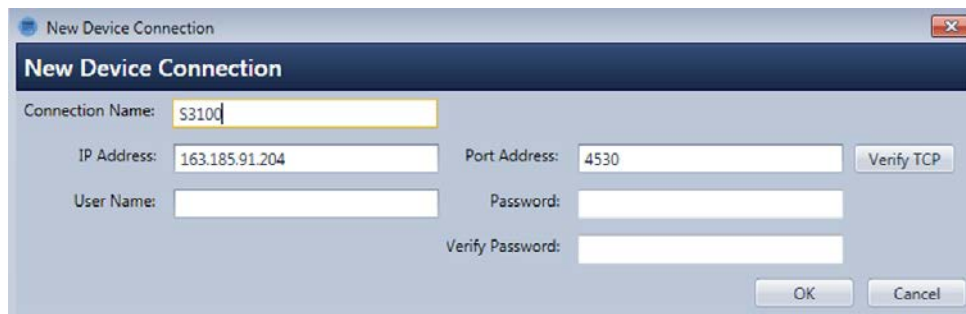


Figure 3.5—New Device Connection dialog

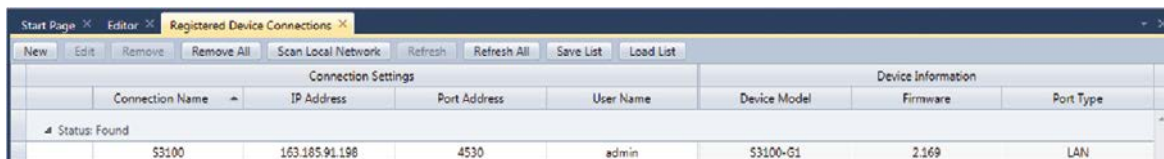


Figure 3.6—Device Connections tab (device connected)

Removing Device Connection(s)

To remove a connection to a Scanner device,

1. With the Device Connections tab selected and Scanner Connections in view, highlight the connection to be removed.
2. Click **Remove** on the Device Connections toolbar.
3. Click **OK** to verify that you want to delete the connection ([Figure 3.7](#)).



Figure 3.7—Remove Device Connection verification dialog (single device)

To remove all device connections,

1. Click **Remove All** on the Device Connections toolbar.
2. Click **OK** to verify that you want to delete connections to all devices (Figure 3.8).

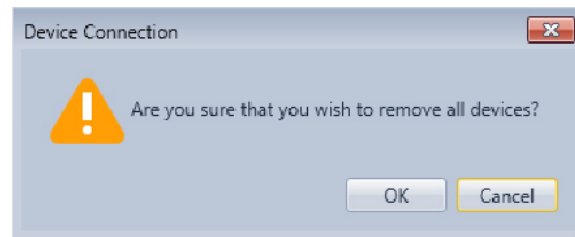


Figure 3.8—Remove Device Connection verification dialog (all devices)

INSTALLING A SCANNER LOGIC PROGRAM ON A SCANNER

Before uploading/downloading a program to/from a Scanner, you must select the Scanner connection with which to communicate. To select a device connection, click on the down arrow in the “Device Selection Indicator” toolbar. From the *Select Device Connection* dialog (Figure 3.9), select the device to which you want to connect.

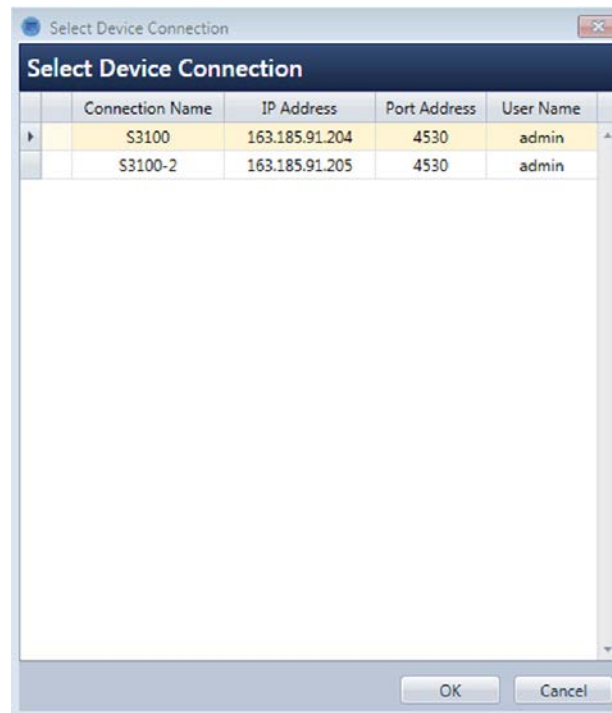


Figure 3.9—Select Device Connection dialog

Uploading a Program to a Scanner via the IDE

Note Uploading a program via the IDE always requires you to compile an SLOGIC script to an SLBIN file.

To upload an SLBIN program file to the selected Scanner device via the IDE,

1. Open an SLOGIC file in the editor (**File>Open**).
2. From your IDE screen, choose **Debug>Start Debugging**. If no parsing or compiler errors occur, the program is installed.
3. Choose **Debug>Stop Debugging** to disconnect the IDE. The Scanner will execute the program once it is release from the Debug mode.
4. From the web interface, verify the program status. See the Web Interface User Manual for more information.
5. From the web interface, select **Controllers>Scanner Logic Controller>Status** to view the results of the debug.

Uploading a Program to a Scanner via the Web Interface

To upload an SLBIN program file to the selected Scanner device via the web interface,

1. Open the web browser of your web-enabled computer, tablet, or phone, and log into the selected Scanner.
2. Select **Administration>General>Installed Files**.
3. In the *Scanner Logic Script Binary File* section, locate the “Install Scanner Logic Script File” field.
4. Click **Browse** and select the SLBIN file for the device and firmware to which you are connected (see [Figure 3.10](#)).
5. Click **OK** in the **Select File** dialog.
6. Click **Submit** in the web interface and click **OK** when the *Confirm* dialog ([Figure 3.11](#)) appears.
7. When successfully uploaded, the “SLBIN File Status” field will be populated ([Figure 3.12](#)). If an error occurred during the HTTP upload process or if the SLBIN file was not compiled to the correct target platform, the SLBIN File Status field will be marked as “Failed validation”.

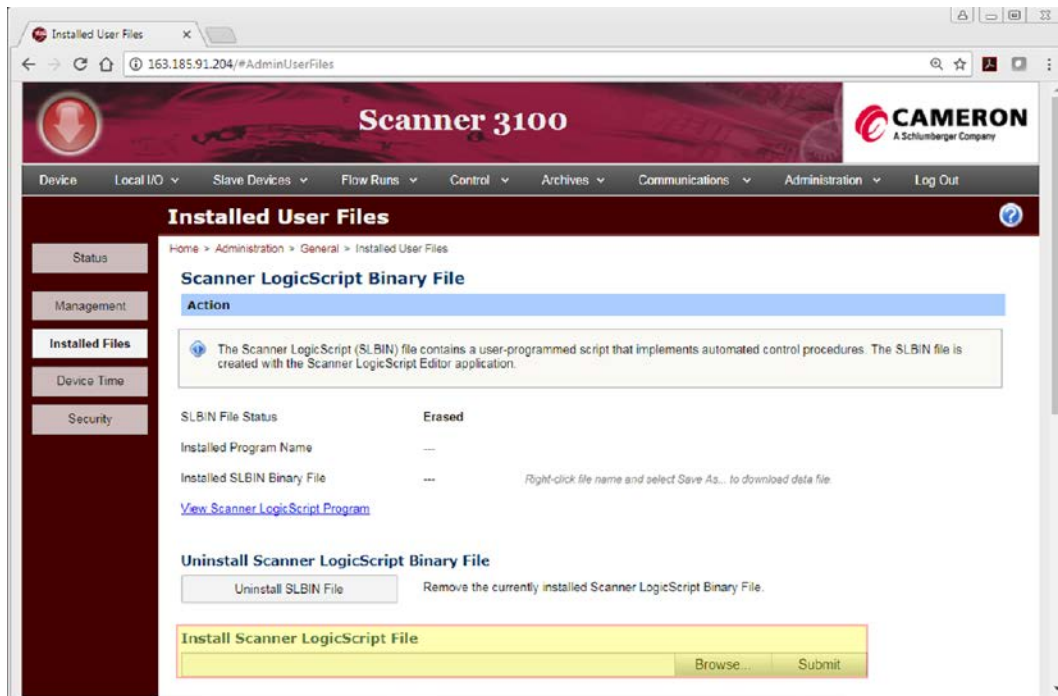


Figure 3.10—Installed User Files web interface page

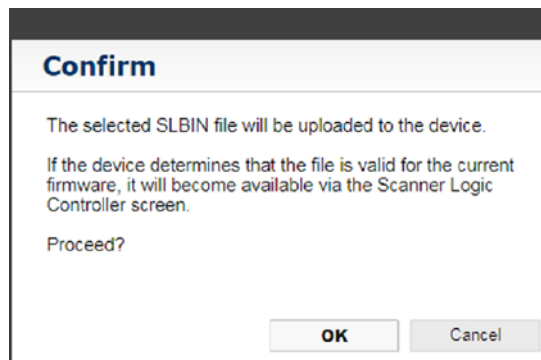


Figure 3.11—Confirm dialog



Figure 3.12—SLBIN file successfully uploaded

Downloading a Program from a Scanner via the IDE

To download the SLOGIC program source file from a selected Scanner device via the IDE,

1. Select the device from which you want to download the file.
2. Choose **Scanner>Open SLOGIC from Scanner Connection**.
3. Click **OK** when the “Current document will be replaced” warning appears, as shown in [Figure 3.13](#).



Figure 3.13—“Current document will be replaced” warning

3. The IDE will connect to the Scanner on the selected device connection and retrieve the active SLOGIC program file. The downloaded Scanner Logic program source code will appear in the Editor. If desired, the script can be saved as an SLOGIC file on disk.

Downloading a Program from a Scanner via the Web Interface

You can also download the SLBIN file directly from the device via the web interface if the access level specified by the `Access_OnlineSource` parameter in the Program Information tab allows sufficient permissions.

To download an SLBIN program file from the selected Scanner device via the web interface,

1. Open the web browser of a web-enabled computer, tablet, or phone, and log into the selected Scanner.
2. Navigate to the **Administration>General>Installed User Files** web page.
3. In the *Scanner Logic Script Binary File* section, locate the “Installed SLBIN Binary File” field ([Figure 3.14](#)).
4. Right-click on the SLBIN file name and select **Save [Link] As...** from the popup menu.
5. Browse to the location where the file will be saved.
6. Rename the file to accurately reflect the file contents.
7. Click **Save**.

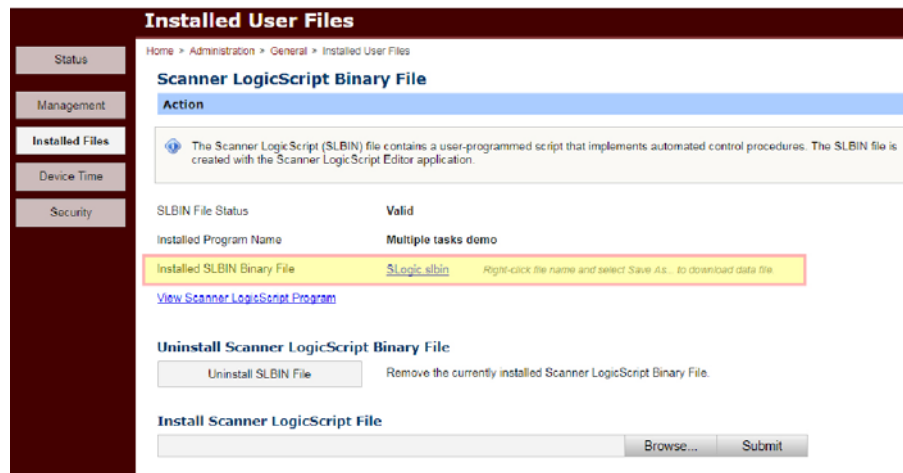


Figure 3.14—“Installed SLBIN Binary File” field

Uninstalling a Program from the Scanner via the IDE

To uninstall an SLBIN file from the Scanner via the IDE,

1. Choose **Scanner>Erase SLBIN**.
2. Click **OK** when the confirmation dialog (*Figure 3.15*) appears.

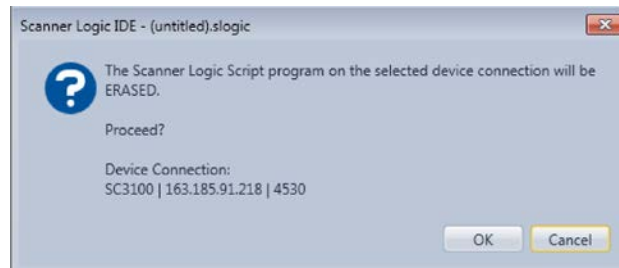


Figure 3.15—“Program on selected device connection will be ERASED” dialog

Uninstalling a Program from the Scanner via the Web Interface

To uninstall an SLBIN file from a Scanner via the web interface,

1. Open the web browser of a web-enabled computer, tablet, or phone, and log into the selected Scanner.
2. Select **Administration>General>Installed User Files**.
3. In the *Scanner Logic Script Binary File* section, locate the “Uninstall Scanner Logic Script Binary File” header (*Figure 3.16*).
4. Click **Uninstall SLBIN File**.
5. Click **OK** when the *Confirm* dialog appears.

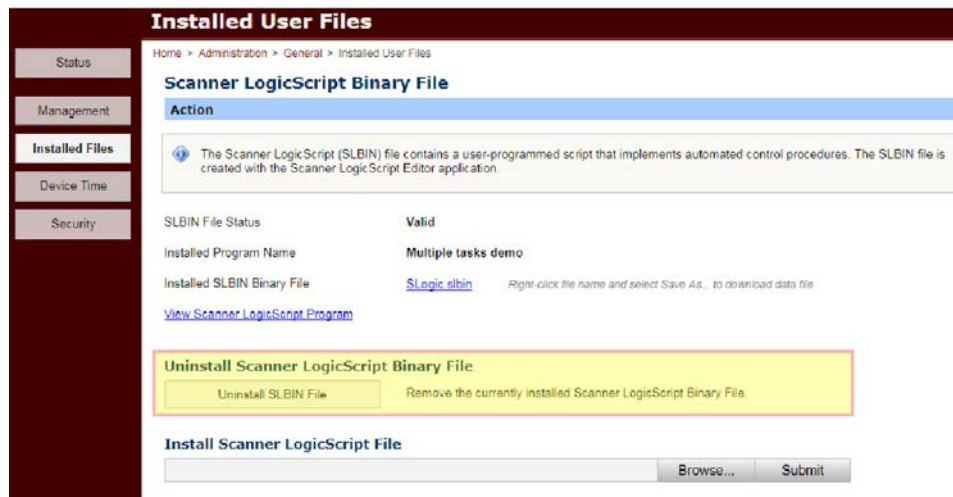


Figure 3.16—Uninstall Scanner Logic Script Binary File

This page is left blank intentionally.

SECTION 4—USING THE PROGRAM

SCRIPT TERMINOLOGY

Each script file is comprised of the following items in [Table 4.1](#).

TABLE 4.1—SCRIPT FILE ITEMS

Term	Definition
Task	A task is a collection of states. In Scanner Logic IDE, each script can contain up to four tasks, which can run concurrently. Each task has a user-assigned name and contains at least one state. One state must be specified as the initial state of the task.
State	A state includes groups of statements that are invoked at different times according to the state machine model. The Logic Controller system in the device runs the onEnter, onLoop, and onExit statement blocks when necessary to execute the state machine. Each state has a user-assigned name. A script can contain up to 96 states, distributed among one to four tasks.
Fail State	The Fail State is entered when a system error causes the program to be unable to run. Programming errors or system configuration errors (for example, mismatch in register inputs or digital I/O) are the most probable cause of a Fail State. There is only one Fail State in a script file.
Abort State	The Abort State is entered when an emergency stop is invoked via the web interface or a special function digital input. There is only one Abort State in a script file.
Subroutine	Subroutines are collections of statements invoked by a user-assigned name, thereby allowing the user to easily reuse the code throughout the program. Subroutines are in the region at the bottom of the script. They can be called from any state in any task. Each script can have up to 100 subroutines.

TUTORIAL: CREATING A SIMPLE PROGRAM

This section provides an easy-to-follow tutorial of the start-to-finish development of a program. The size and scope of this example are limited. This tutorial includes a discussion of the Scanner Logic IDE and parts of the Scanner 3100 web interface. However, this sample program will not explore the IDE or the web interface in depth. For detailed information about using the web interface, see the Scanner 3100 Web Interface Manual.

Problem

We will examine a sample problem that is a simplistic representation of a real-world application. In our example, we have a main gas flow line with tributary runs feeding in to the main pipe via metering stations. Our Scanner 3100 is connected to sensors and a flow control valve is installed at one of the stations. In this example, the subsidiary flow enters the station at its inlet at a higher pressure than the main flow line, creating a pressure differential which forces flow into the main line at the outlet of

the station. However, the inlet pressure of the flow is not always stable and at times becomes relatively low compared to the outlet pressure.

We wish to control the flow rate of the gas through the station to keep it constant by actively controlling the degree of opening of the variable flow control valve. Additionally, when the inlet flow pressure becomes too low, fluctuations in the pressure at either end of the station may result in a reversing of the flow, and we want to prevent that by closing the valve when the difference between the inlet and outlet pressures falls below a certain threshold. We will ignore the parts of the physical system that might be present to handle the case of the inlet pressure becoming too high.

We will construct a program to control whether the station's flow is connected (tied in) to the main flow or closed off (shut in) from it. The program will monitor the differential pressure between the station inlet and the station outlet, and decide when to open and close the flow control valve according to the conditions that we have defined. It will report to the system with alarms when it begins the tie in and shut in phases, and it will maintain and report various calculations. Also, it will activate a digital output whenever the flow is shut in. The program will set up a PID controller to actively adjust the opening size of the flow control valve to maintain our desired flow rate when the station is tied in.

Assumptions

- When the subsidiary flow pressure is too low, a smaller differential pressure exists between the subsidiary flow and the main flow.
- When a low differential pressure is detected, the system will be notified via an alarm. However, the condition must persist for a specified time before the alarm becomes active.
- If the low subsidiary line pressure state persists, the valve will be closed to shut in the subsidiary flow until the pressure level returns to an acceptable level.
- When the pressure differential returns to higher levels for a sustained time, the valve will be reopened and control of the subsidiary flow rate will resume.
- If the subsidiary flow goes offline, the valve will remain closed for a preset period to allow the pressure to build up, after which the valve will be reopened.
- The situation where the subsidiary flowline pressure might become too high is not considered for this example.

Scanner 3100 Inputs/Outputs Setup

For the program to interact with the real world, we will need access to various inputs and outputs (I/O). We will set these up in the IDE so that the program code can use them. However, the program does not directly take control of the Scanner 3100 hardware. We must configure Scanner device inputs to align with how the program expects to use them, and we must assign outputs to follow the values produced by the program. This model preserves the integrity of the Scanner configuration, and prevents unexpected disruption to the device setup when a program is uploaded.

We will assume that the Scanner 3100 is appropriately installed and that the following hardware configuration exists:

TABLE 4.2—HARDWARE DEPLOYMENT

Hardware	Location
Scanner 3100 – set up to compute flow using integrated MVT; static pressure readings from MVT will be used in the program as well	Upstream from flow valve
Flow control valve, wired to Analog Output 1	Downstream from Scanner 3100

Hardware	Location
Static pressure transducer, wired to Analog Input 1	Downstream from flow valve
External hardware to receive a digital output signal	n/a

The Scanner Logic program will not directly modify the Scanner 3100 user configuration. Resource Validation Errors are reported if the device configuration does not match the configuration required by the program. For the purposes of this example, we will not perform the required Scanner I/O configurations. See the Web Interface User Manual for more information about configuring Scanner I/Os.

Program Design

Scanner Logic programs are expressed as state machine models. It is possible to represent physical processes using state machines, and there is often more than one state machine design that can fit the same scenario.

The state machine is operated by the Logic Script execution engine once per second. In an execution cycle, the code in the onLoop body of the current state in each task is run once. If a transition to a new state occurs, onExit code will run to perform user-coded finalization actions in the state being exited. The last step of the execution cycle when a transition happens is the onEnter code will run to perform user-coded initialization actions in the new state being. The next execution cycle begins in the onLoop body of the new state. Typically, the code in the onLoop body is concerned with calculating values and with checking various conditions to determine whether to remain in the current state or to transition to a different state.

Define States and Transition Conditions

The first step in designing a state machine model is to identify the situations in the process that persist steadily for periods of time. In our example, these steady states occur when the subsidiary flow is tied in and when the flow is shut in. We can call these states the `Flowing` state and the `ShutIn` state. States are defined within tasks. Up to four tasks can be defined, but we will only need one, which we will name `StationTieIn`.

Next, identify what conditions would cause a transition from one state to another. When the process is in the `Flowing` state, we would like the station inlet pressure to be a certain amount higher than the outlet pressure. We will remain in the `Flowing` state while this is the case. When the difference between the inlet and outlet pressures falls below this amount persistently, the program should change from the `Flowing` state to the `ShutIn` state.

Conversely, when the system is in the `ShutIn` state, a differential pressure level above a specified threshold lasting for a specified period would trigger a transition back to the `Flowing` state. There is an additional constraint on this condition in that the transition back to `Flowing` cannot occur until a minimum period has elapsed while in the `ShutIn` state.

We will make the initial state of the program be the `ShutIn` state. We will also add a program control variable that we can configure via the web interface to force the program to enter the `ShutIn` state even when the differential pressure is at flowing levels.

Define Input and Output Resources

Scanner Logic provides an `AnalogPIDControllerResource` object that we can use to control the flow valve to maintain our desired flow rate while in the `Flowing` state. It has control variables, including `Kp`, `Ki`, and `Kd`, that affect how it behaves in adjusting the valve opening when the flow rate

deviates from its set point. The hardware Analog Output port of Scanner 3100 will be configured to follow the output of programmable logic controller. We will set the PID Controller into automatic operation mode to seek the target flow rate when we enter the Flowing state. Conversely, we will put the PID Controller into manual override mode and force the valve closed when we enter the ShutIn state.

We will use RegisterInputResource objects to access two static pressure inputs from the Scanner 3100, and calculate the differential pressure between the two inputs in code. The upstream or inlet static pressure will come from the integrated MVT in the Scanner 3100. The downstream or outlet static pressure will be obtained via a Scanner 3100 Analog Input connected to an external pressure sensor.

A third RegisterInputResource object can be used to bring in the flow rate coming through the flow control valve, which is being calculated by the Scanner 3100.

We will use an AlarmResource object to signal the Scanner 3100 of the point at which the station differential pressure falls below our defined threshold. We can take advantage of the alarm hold-off feature to provide some hysteresis before transitioning to the ShutIn state. This feature allows us to ensure that the differential pressure condition is persistent for a certain amount of time before we transition to the ShutIn state. With a hold-off period configured, we can *assert* the alarm immediately when the pressure difference falls below the ShutIn threshold, but the alarm output will not become *active* immediately.

If the pressure difference rises above the ShutIn threshold again within the hold-off period, then we will *deassert* the alarm, which cancels the hold-off delay. Since the alarm has not become active yet, the system remains in the Flowing state. However, if the pressure difference stays below the threshold for the entire hold-off period, the hold-off delay will complete, and the alarm output will then become active. We will monitor the “IsActive” property of the alarm to trigger the transition from Flowing to ShutIn, instead of using the pressure difference as the transition condition directly.

We will use a second AlarmResource object in a similar way to “de-bounce” the transition from the ShutIn state back to the Flowing state, so that transient pressure increases do not cause frequent transitions.

A DigitalOutputResource object will be used to produce an output signal in the Scanner 3100 that will indicate that the ShutIn state is active. Since we will have an AlarmResource set up to be active whenever we are in the ShutIn state, we can use the “FollowAlarm” property of the DigitalOutputResource to make the state of the Digital Output signal match the “IsActive” property of the AlarmResource automatically.

We want to remain in the ShutIn state for a stipulated minimum time before we transition back to the Flowing state. We could use one of the general-purpose TimerResource objects to keep track of the ShutIn time. However, we will take advantage of the built-in “ActiveTime” property of the state objects instead. The “ActiveTime” property holds the amount of time that the current state has been active since it was last entered. We will include a check on the “ActiveTime” in the conditions to transition from ShutIn to Flowing states.

TABLE 4.3—INPUT AND OUTPUT RESOURCES

Resource Type	Purpose	Name
Analog PID	Control the flow valve opening	FlowValveController
Register Input	Read in the static pressure at the station inlet	StationInletPressure
Register Input	Read in the static pressure at the station outlet	StationOutletPressure

Resource Type	Purpose	Name
Register Input	Read in the flow rate through the station	StationFlowRate
Alarm	Notify system of transition to Flowing state; provide hold-off to implement hysteresis	FlowingStartedAlarm
Alarm	Notify system of transition to ShutIn state; provide hold-off to implement hysteresis	ShutInStartedAlarm
Digital Output	Indicate to an external system that the ShutIn state is active	ShutInSignal

Define Input and Output Registers

There are certain variables in the system that we would like to specify and be able to adjust during the operation of the program. We will create **ConfigurationRegister** to hold these values. Configuration registers can be given identifier names, and they automatically appear on the *Logic Controller Configuration Registers* page in the web interface to allow users to modify values. These values are accessible by the program at run time and are used to affect how the program runs without having to re-compile the program.

TABLE 4.4— CONFIGURATION REGISTERS (USER INPUT VARIABLES)

Purpose	Name	Initial Value
Minimum differential pressure threshold before entering ShutIn	MinFlowingDP	200 kPa
Maximum differential pressure threshold before resuming Flowing	MaxShutInDP	225 kPa
Required minimum time to remain in ShutIn state	MinShutInTime	15 min
Control whether program can run freely or be forced to enter and remain in the ShutIn state (0 = ShutIn, 1 = run)	ProgramControl	0

Also, we will want to keep track of certain values that are calculated during the operation of the program and make them available for use by the Scanner 3100 host device and for display on its web interface. We will use **HoldingRegister** objects to output these calculated values. The Holding Register values are available to be viewed on the *Logic Controller Holding Registers* page in the web interface, and they can be archived by the Scanner 3100 or used as input values for various device features.

TABLE 4.5—HOLDING REGISTERS (USER OUTPUT VARIABLES)

Purpose	Name	Units
Calculated station differential pressure between inlet and outlet pressures	StationDP	kPa
Calculated percentage of time spent in Flowing	PercentFlowingTime	%

Plan Updating of Calculated Values

We have designated Holding Registers to store values that we will calculate within the program. These calculations will be done at various times during the operation of the state machine. This is a good opportunity to use Logic Script **Subroutines**. We can place all the calculation statements into a subroutine named **CalculateStationValues** and call this subroutine at appropriate times. This makes program debugging and maintenance easier by keeping the calculation code together in one place.

We will call the subroutine at the start of the code within the onEnter and onLoop blocks of both the **Flowing** and **ShutIn** states, so that the calculated values are up to date if we reference the values in subsequent code statements.

Plan State Entry and Exit Actions

There are certain actions that we would like to ensure take place whenever we enter or exit the **Flowing** and **ShutIn** states. In general, it is a good practice to place code that controls actions or effects that are dependent on what state the system is in into the onEnter and onExit code blocks. These blocks will always run when a state transition occurs. Right after a `changestate` statement is executed, the script engine runs the onExit block of the current state, and then runs the onEnter block of the state transitioned into. The execution cycle ends at the bottom of the onEnter block, and code starts running at the top of the onLoop block of the new state in the next execution cycle. The onExit block allows you to deactivate or remove an output

We need to manage changes for the **FlowingStartedAlarm** and **ShutInStartedAlarm**, so that they will be up to date with the transitions between states in the system. Note that we will need to initially assert the **ShutInStartedAlarm** in the onEnter code of the **ShutIn** state. Since the **ShutIn** state is designated as the **initial** state of the task, its onEnter code is the first code that executes when the program is run.

We also want to update calculated values and update the operating mode of **FlowValveController**.

TABLE 4.6—ENTRY AND EXIT ACTIONS

Item	Actions
ShutIn state onEnter	<ul style="list-style-type: none"> Update calculated values by calling CalculateStationValues() Signal that we have entered ShutIn state by asserting ShutInStartedAlarm Set FlowValveController to manual override mode and close valve
ShutIn state onExit	<ul style="list-style-type: none"> Signal that we have left ShutIn state by de-asserting ShutInStartedAlarm
Flowing state onEnter	<ul style="list-style-type: none"> Update calculated values by calling CalculateStationValues() Set FlowValveController to automatic mode, actively seeking target flow rate
Flowing state onExit	<ul style="list-style-type: none"> Signal that we have left Flowing state by de-asserting FlowingStartedAlarm

Define Interface to Program Variables

Scanner Logic provides **UserHMIField** objects that allow selected properties of objects within the program to be consolidated into a single list of fields. This list is available to the Scanner 3100 and can be accessed via a contiguous Modbus read or write, or can be accessed via the *Logic Controller HMI Fields* page in the web interface. This allows users to have convenient access to view and

modify program-specific variables. If the selected property is read/write, the attached User HMI Field object has the option to restrict access to read-only. To help organize the fields on the *Logic Controller HMI Fields* web page, headers can be defined for User HMI Fields.

We will select some object properties to access via User HMI Fields. These will include the Value properties of the Configuration Registers and Holding Registers designated above. User changes to input variables via User HMI Field objects are synchronized to their attached object properties automatically by Scanner Logic at the start of each execution cycle. Additionally, changes in output values from attached object properties are synchronized to the User HMI Fields at the end of each execution cycle.

We will create User HMI Fields for input variables that correspond to properties of selected **AlarmResource**, **AnalogPIDControllerResource**, and **ConfigurationRegister** objects, and we will set them to allow modification via the web interface or Modbus. The initial values for these input variables will be specified by the “Initial_...” declaration parameter corresponding to the selected property of the referenced objects (for example, Initial_Setpoint, Initial_HoldOffDelay, Initial_Value, etc.).

We will also create User HMI Field objects for output variables that are read from object properties of selected **AlarmResource**, **RegisterInputResource**, **AnalogPIDControllerResource**, **State**, and **HoldingRegister** objects. These values kept up to date by the program at run time, and since they are output variables, they are not editable in the web interface or Modbus.

TABLE 4.7— USER HMI FIELDS

Purpose	Property Name	Allow Modify
<u>Station Tie In Statistics</u>		
Total time spent in Flowing state (seconds)	StationTieIn.Flowing.TotalActiveTime	No
Amount of time since Flowing state entered (seconds)	StationTieIn.Flowing.ActiveTime	No
Total time spent in ShutIn state (seconds)	StationTieIn.ShutIn.TotalActiveTime	No
Amount of time since ShutIn state entered (seconds)	StationTieIn.ShutIn.ActiveTime	No
Percent of total time spent in Flowing state	PercentFlowingTime.Value	No
<u>Station Rate and Pressure</u>		
Flow rate through station	StationFlowRate.Value	No
Static pressure at station inlet	StationInletPressure.Value	No
Static pressure at station outlet	StationOutletPressure.Value	No
Difference between InletPressure and OutletPressure	StationDP.Value	No
Normalized output of FlowValveController	FlowValveController.Output	No
<u>Shut In Alarm</u>		
Notifies system that ShutIn state has been entered (1 = Active)	ShutInStartedAlarm.IsActive	No
ShutIn alarm active count in hold-off (seconds)	ShutInStartedAlarm.HoldOffTime	No

Purpose	Property Name	Allow Modify
Configured ShutIn alarm hold-off delay (seconds)	ShutInStartedAlarm.HoldOffDelay	Yes
Flowing Alarm		
Notifies system that Flowing state has been entered (1 = Active)	FlowingStartedAlarm.IsActive	No
Flowing alarm active count in hold-off (seconds)	FlowingStartedAlarm.HoldOffTime	No
Configured Flowing alarm hold-off delay (seconds)	FlowingStartedAlarm.HoldOffDelay	Yes
Program Control		
0 = hold program in ShutIn state, 1 = run program normally	ProgramControl.Value	Yes
The desired flow rate for the station, controlled by the flow valve	FlowValveController.Setpoint	Yes
Minimum differential pressure threshold for Flowing state, below which we transition to ShutIn state	MinFlowingDP.Value	Yes
Maximum differential pressure threshold for ShutIn state, above which we transition to Flowing state	MaxShutInDP.Value	Yes
Required minimum time to remain in ShutIn state (minutes)	MinShutInTime.Value	Yes

Coding the Program

At this point, we have planned the overall structure of our state machine program and we have defined the resources that we will need, as well as the inputs and outputs of the program. We will now proceed to declaring the required resources and registers in the IDE, and then entering the program code.

Adding Program Information

1. Start the Scanner Logic IDE program.
2. Click the **File>New** menu item. A basic empty template for a program will appear in the Editor window.
3. Locate the "Program Information" tab.
4. Click **Edit**, fill in the information shown in [Table 4.8](#), and click **OK**.

TABLE 4.8—PROGRAM INFORMATION

Item	Value
Program Name	IDE Getting Started Example
Program Author	John Smith
Program Owner	OilCo Ltd.
Program Version	<i>Leave as is.</i>
Program Creation Date	<i>Leave as is.</i>
Access_OnlineSource	Select “allusers.”
Access_OnlineControls	Select “allusers.”
Access_WriteHMI	Select “allusers.”
Description	This is a tutorial program for writing a program with the Scanner Logic Script language in the Scanner Logic IDE.

Adding Register Input Resources

1. Add Static Pressure sources by locating the “Register Inputs” tab, clicking on row 01, and clicking **Edit**.
2. Fill in the information according to [Table 4.9](#) and click **OK**.
3. Repeat the process for rows 02 to 03 using the settings [Table 4.10](#) to [Table 4.11](#).

TABLE 4.9—REGISTER INPUT RESOURCE ITEM 01 (STATION INLET PRESSURE)

Item	Value
Name	StationInletPressure
Description	Line pressure at station inlet, upstream from the flow control valve (integrated MVT)
Tagname	Select: ‘Stat Press: Holding: Inst Reading’
Tagcode	Automatically set: ‘m32_FC_IN_2_Holding_InstReading’
Category	Automatically set: ‘Static Pressure (gauge)’
Units	Select: ‘kPa(g)’

TABLE 4.10—REGISTER INPUT RESOURCE ITEM 02 (STATION OUTLET PRESSURE)

Item	Value
Name	StationOutletPressure
Description	Line pressure at station outlet, downstream from the flow control valve (pressure transducer)
Tagname	Select: ‘Analog 1: Holding: Inst Reading’
Tagcode	Automatically set: ‘m32_FC_IN_5_Holding_InstReading’

Item	Value
Category	Select: 'Static Pressure (gauge)'
Units	Select: 'kPa(g)'

TABLE 4.11—REGISTER INPUT RESOURCE ITEM 03 (STATION FLOW RATE)

Item	Value
Name	StationFlowRate
Description	Flow rate through station, as limited by flow control valve
Tagname	Select: 'FR1: HAccum: Gas Volume Flow Rate'
Tagcode	Automatically set: 'm32_FC_FR_1_HoldingAccum_GasVolumeFlowRate'
Category	Select: 'Gas Volume'
Units	Select: 'm3'
Rate	Click the checkbox, select: '/sec'

Adding Analog PID Controller

1. Locate the "Analog PID" tab, click on row 01, and click **Edit**.
2. Fill in the information in [Table 4.12](#) and click **OK**. Additional fields are on the Primary Controller tab page.

Note that the properties of PID Controllers used in the program are available to view and optionally modify on the *Logic Controller PID Controllers* page in the web interface. The *Webcontrolflags* parameter controls which groups of values can be modified on this page.

TABLE 4.12—ANALOG PID CONTROLLER RESOURCE ITEM 01 (FLOW VALVE CONTROLLER)

Item	Value
Name	FlowValveController
Description	Maintains desired flow rate at station by controlling flow valve
PidType	Select: 'simplepid'
Initial_IsAutoMode	False
Webcontrolflags	Check all checkboxes
ProcessVar	Select: 'FlowRate'
PidAction	Select: 'direct'
Initial_Period	1
Initial_RangeHigh	10.0
Initial_RangeLow	0.0
Initial_SetPoint	7.5
Initial_SetPointTolerance	0.0

Item	Value
Initial_SetPointDeadBand	1.0
Initial_OverrideValue	0.0
Initial_FailValue	0.0
Initial_Kp	1.0
Initial_Ki	0.0
Initial_Kd	0.0

Adding Alarm Resources

1. Locate the “Alarms” tab, click on row 01, and click **Edit**.
2. Fill in the information in [Table 4.13](#) and click **OK**.
3. Click on row 02 and click **Edit**.
4. Fill in the information in [Table 4.14](#) and click **OK**.

TABLE 4.13—ALARM RESOURCE ITEM 01 (SHUTIN STARTED ALARM)

Item	Value
Name	ShutInStartedAlarm
Description	Notifies system that ShutIn state has been entered
Initial_IsAsserted	true
Initial_HoldOffDelay	30

TABLE 4.14—ALARM RESOURCE ITEM 02 (FLOWING STARTED ALARM)

Item	Value
Name	FlowingStartedAlarm
Description	Notifies system when Flowing state has been entered
Initial_IsAsserted	false
Initial_HoldOffDelay	30

Adding Digital Output Resource

1. Locate the “Digital Outputs” tab, click on row 01, and click **Edit**.
2. Fill in the information in [Table 4.15](#) and click **OK**.

TABLE 4.15—DIGITAL OUTPUT RESOURCE ITEM 01 (SHUTIN SIGNAL)

Item	Value
Name	ShutInSignal
Group	<i>Leave blank</i>
Description	Output signal to indicate that the system is in the ShutIn state
Initial_IsAsserted	false

Item	Value
Initial_FollowAlarm	Select: '01:ShutInStartedAlarm'
Initial_Period	2
Initial_Duration	1

Adding Configuration Registers

1. Locate the "Configuration" tab, click on row 01, and click Edit.
2. Fill in the information in [Table 4.16](#) and click **OK**.
3. Repeat the process for rows 02 to 04 using the settings in [Table 4.17](#) to [Table 4.19](#).

TABLE 4.16—CONFIGURATION REGISTER ITEM 01 (PROGRAM CONTROL)

Item	Value
Name	ProgramControl
Group	<i>Leave blank</i>
Description	0 = hold program in ShutIn state, 1 = run program normally
Category	Select: 'No Units'
Initial_Value	0

TABLE 4.17—CONFIGURATION REGISTER ITEM 02 (MIN FLOWING DP)

Item	Value
Name	MinFlowingDP
Group	<i>Leave blank</i>
Description	Minimum differential pressure threshold for Flowing state, below which we transition to ShutIn state
Category	Select: 'Differential Pressure'
Units	Select: 'kPa'
Initial_Value	200

TABLE 4.18—CONFIGURATION REGISTER ITEM 03 (MAX SHUTIN DP)

Item	Value
Name	MaxShutInDP
Group	<i>Leave blank</i>
Description	Maximum differential pressure threshold for ShutIn state, above which we transition to Flowing state
Category	Select: 'Differential Pressure'
Units	Select: 'kPa'
Initial_Value	225

TABLE 4.19—CONFIGURATION REGISTER ITEM 04 (MIN SHUTIN TIME)

Item	Value
Name	MinShutInTime
Description	Required minimum time to remain in ShutIn state (minutes)
Category	Select: 'No Units'
Initial_Value	15

Adding Holding Registers

1. Locate the “Holding” tab, click on row 01, and click **Edit**.
2. Fill in the information in [Table 4.20](#) and click **OK**.
3. Click on row 02 and click **Edit**.
4. Fill in the information in [Table 4.21](#) and click **OK**.

TABLE 4.20—HOLDING REGISTER ITEM 01 (STATION DP)

Item	Value
Name	StationDP
Group	<i>Leave blank</i>
Description	Difference between InletPressure and OutletPressure
Category	Select: 'Differential Pressure'
Units	Select: 'kPa'
Initial_Value	0.0

TABLE 4.21—HOLDING REGISTER ITEM 02 (PERCENT FLOWING TIME)

Item	Value
Name	PercentFlowingTime
Group	<i>Leave blank</i>
Description	Percent of total time spent in Flowing state
Category	Select: 'Percent'
Units	Select: '%'
Initial_Value	0.0

Adding User Program Code in States and Subroutines

A new program template contains one task by default, named **Task1**, containing an empty state definition. User written code goes in between the sets of braces for onEnter, onLoop, and onExit. Each task can have multiple states defined inside it, with one of them marked as the **initial** state. When a program starts running, script execution begins at the onEnter block of the initial state of each task.

Script execution cycles occur once per second. Program execution during an execution cycle proceeds until either the code in an onEnter block or an onLoop block is completed. In either of these cases, the program will continue at the start of the onLoop block in the next execution cycle. If a transition to another state does not occur, the program will remain in the onLoop block of the current state.

When a transition is triggered, the program will immediately leave the current onLoop block, and run the onExit code of the current state, run the onEnter code of the new state, and then wait for the next execution cycle. At that point, the program will continue at the start of the onLoop block of the new state. Please refer to the Scanner Logic Script Programmer Manual for more information regarding tasks and states.

Follow the steps below and type in code when directed. We will be moving between different areas of the program code and adding content in increments. The complete program is listed in an appendix of this document.

Follow the steps below and type in code when directed. We will be moving between different areas of the program code and adding content in increments. The complete program is listed in [Appendix B](#) of this document.

1. Find the program code section in the Editor window. See [Figure 4.1](#) for an example.

```
//
//)))))))))
// Program Execution - User Tasks and States
//
//((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
task Task1
{
  initial state State1
  {
    onEnter
    {
    }

    onLoop
    {
    }

    onExit
    {
    }

  } // end state State1
} // end Task1
```

Figure 4.1—Program code region default content

2. Change the identifier Task1 to StationTieIn.
3. Change the identifier State1 to ShutIn. The initial keyword indicates that program execution for the StationTieIn task will begin in this state.
4. Select the lines for the entire ShutIn state, from “initial state Shutin” to the closing brace. Copy the state code and paste it below the ShutIn state, ensuring that you are not pasting inside the closing brace of the ShutIn state, and that you are pasting within the closing brace of the task.

5. Notice that there are parser errors showing in the *Error List* panel now. The text that is causing the syntax errors is underlined in red. If you double click on an error row in the *Error List*, the cursor will move to the location of that error.
6. Delete the keyword `initial` from the pasted state code, and change the name of the state to `Flowing`.
7. Navigate to the bottom of the Editor window and click the plus icon to the left of a collapsed region called **Subroutines** to expand the code lines within.
8. Type the following lines of code (as shown in [Figure 4.2](#)) within the **Subroutines** region, above the “#endregion” line revealed by expanding the region in the previous step.

```
void subroutine CalculateStationValues()
{
}
```

Figure 4.2—Subroutine to add

9. Type the following lines of code inside the braces of the subroutine added in the last step (as shown in [Figure 4.3](#)). This code will update the calculated Holding Register values for `StationDP` and `PercentFlowingTime`.

```
StationDP = StationInletPressure - StationOutletPressure;

PercentFlowingTime = StationTieIn.Flowing.TotalActiveTime /
(StationTieIn.Flowing.TotalActiveTime + StationTieIn.ShutIn.TotalActiveTime);
```

Figure 4.3—Subroutine code to add

Notice how the auto-completion feature opens a drop-down list of possible identifiers and keywords available at various times as you type. The list automatically filters as you type, and as a shortcut, you can cursor down to the word that you want and press `Enter` or double-click the word to place it into your code. If the list closes and you want to reopen it, press `<CTRL>+<SPACE>`. Also, notice that you can hover your cursor over identifiers and a quick info tip appears that provides some useful information about the identifier.

10. We want the calculated values to be up to date at the start of each execution cycle, before we potentially reference the values in subsequent code. The very first execution cycle of the program starts in the `onEnter` block of the `initial` state and stops at the end of the `onEnter`. Every execution cycle thereafter starts in the `onLoop` block of the current state, and either end at the bottom of the same `onLoop` block or end at the bottom of the `onEnter` block of a different state. Enter the code in [Figure 4.4](#) as the first line in the `onEnter` block of the `ShutIn` state, and as the first line in the `onLoop` blocks of both the `ShutIn` and `Flowing` states.

```
CalculateStationValues();
```

Figure 4.4—Call `CalculateStationValues()` subroutine

11. Next, we will define the conditions that will cause transitions between states. The transition from the `Flowing` state to the `ShutIn` state occurs when the `ShutInStartedAlarm` becomes active, which in turn occurs after its hold-off delay is completed. Alternatively, if the

ProgramControl register value is 0, we must transition directly to ShutIn. Add the code from [Figure 4.5](#) in the onLoop block of the Flowing state, after the line containing CalculateStationValues() added in the previous step.

```
if (ShutInStartedAlarm.IsActive ||
    ProgramControl == 0)
    changestate ShutIn;
```

Figure 4.5—Transition to ShutIn state

12. For the ShutInStartedAlarm to become active, it must be asserted to start the hold-off delay, and the alarm must remain asserted for the entire duration of the delay. The alarm is asserted when the StationDP reaches or falls below the threshold of MinFlowingDP, and is de-asserted when StationDP rises above MinFlowingDP again before the hold-off delay has elapsed. Add the code in [Figure 4.6](#) in the onLoop block of the Flowing state, after the code entered from [Figure 4.5](#) in the step above.

```
if (StationDP <= MinFlowingDP)
    ShutInStartedAlarm.Assert();
else
    ShutInStartedAlarm.Deassert();
```

Figure 4.6—Update ShutInStartedAlarm

Notice that we are using just the names of the objects in this code, rather than accessing the Value properties. Many object types have default properties that can be omitted in the code, and the parser will automatically know to use the default property. In the usage above, the default property of the objects is Value. You can view this information in the quick info tips that appear when you hover the mouse cursor over each of the relevant identifiers. For example, the first line of code above is interpreted as shown in [Figure 4.7](#) below.

```
if (StationDP.Value <= MinFlowingDP.Value)
```

Figure 4.7—Default properties

13. Now we will write the code that transitions from the ShutIn state to the Flowing state. When the FlowingStartedAlarm becomes active after its hold-off delay is completed, we are ready to transition to the Flowing state. However, we also need to ensure that MinShutInTime minutes have elapsed before we can leave the ShutIn state. Furthermore, if the ProgramControl register value is “0,” we must remain in ShutIn and can only transition if ProgramControl is “1.” Add the code in [Figure 4.8](#) in the onLoop block of the ShutIn state, after the line containing CalculateStationValues() added in a previous step.

```
if (FlowingStartedAlarm.IsActive &&
    ShutIn.ActiveTime >= (MinShutInTime * 60) &&
    ProgramControl == 1)
    changestate Flowing;
```

Figure 4.8—Transition to Flowing state

14. For the `FlowingStartedAlarm` to become active, it must be asserted to start the hold-off delay, and the alarm must remain asserted for the entire duration of the delay. The alarm is asserted when the `StationDP` reaches or exceeds the threshold of `MaxShutInDP`, and is deasserted when `StationDP` falls below `MinFlowingDP` again before the hold-off delay has elapsed. Add the code in [Figure 4.9](#) in the `onLoop` block of the `ShutIn` state after the code entered from [Figure 4.8](#) above.

```
if (StationDP >= MaxShutInDP)
    FlowingStartedAlarm.Assert();
else
    FlowingStartedAlarm.Deassert();
```

Figure 4.9—Update `FlowingStartedAlarm`

15. Next, we will add code to the `onExit` and `onEnter` blocks of the two states. When we transition out of the `ShutIn` state, we want to de-assert `ShutInStartedAlarm`, so that it can be used in the `Flowing` state for transition checking in the code that was entered from [Figure 4.5](#) and [Figure 4.6](#). Type the code in [Figure 4.10](#) in the `onExit` block of the `ShutIn` state.

```
ShutInStartedAlarm.Deassert();
```

Figure 4.10—De-assert `ShutInStartedAlarm`

16. When we transition out of the `Flowing` state, we want to de-assert `FlowingStartedAlarm` so it can be used in the `ShutIn` state for transition checking in the code that was entered from [Figure 4.8](#) and [Figure 4.9](#). Type the code in [Figure 4.11](#) in the `onExit` block of the `Flowing` state.

```
FlowingStartedAlarm.Deassert();
```

Figure 4.11—De-assert `FlowingStartedAlarm`

17. When we enter the `ShutIn` state, we want to ensure that the `ShutInStartedAlarm` is asserted, no matter how the state was arrived at. We also need to set the `FlowValveController` into manual override mode and force the valve to close. Enter the code in [Figure 4.12](#) in the `onEnter` block of the `ShutIn` state, after the `CalculateStationValues()` line.

```
ShutInStartedAlarm.Assert();

FlowValveController.OverrideValue = 0.0;
FlowValveController.SetManualMode();
```

Figure 4.12—Set `FlowValveController` in manual mode

18. Similarly, when we enter the `Flowing` state, we want to ensure that the `ShutInStartedAlarm` is asserted, no matter how the state was arrived at. Also, we will set the `FlowValveController` into automatic operation mode and allow it to actively control the flow valve to achieve the desired setpoint value. The controller `SetPoint` property is connected to a User HMI Field, so it can be adjusted by the user over the web interface or Modbus. Enter the code in [Figure 4.13](#) in the `onEnter` block of the `Flowing` state.

```
FlowingStartedAlarm.Assert();
FlowValveController.SetAutoMode();
```

Figure 4.13—Set FlowValveController in auto mode

- Find the “System Declarations” collapsible region and click the plus icon to the left of the region heading to open it. You will see a failstate declaration and an abortstate declaration. The program can enter the fail state if Scanner 3100 I/O configuration that the script depends upon has been changed or if some severe program error occurs. Also, the program can be aborted via the web interface Logic Controller Program Control screen or via a digital input signal configured for the purpose. These actions will cause the program to enter the abort state. Entry into either of these special states will trigger the Scanner 3100 to create an event log for the occurrence.

The program will loop indefinitely in the onLoop blocks of the failstate and abortstate once entered. The changestate keyword is not allowed in these states, so they cannot be exited. The only way to resume program operation is to restart the program, which can be done via the Program Control screen in the web interface, after resolving any error conditions that might have triggered a failstate entry.

You can add user code to execute in the onEntry or onLoop blocks of these two special system states. This is useful for returning the system to a safe or predictable condition. We will cause the flow valve to close, de-assert the FlowingStartedAlarm, and force the ShutInStartedAlarm to become active. Add the code in [Figure 4.14](#) into the onEnter blocks of failState and abortState.

```
FlowValveController.OverrideValue = 0.0;
FlowValveController.SetManualMode();

FlowingStartedAlarm.Deassert();

ShutInStartedAlarm.HoldOffDelay = 0;
ShutInStartedAlarm.Assert();
```

Figure 4.14—Code to add to failState and abortState

We are now done entering the code for the example program. If there are errors showing in the Error List at this point, go over these steps again or compare your program with the complete program listed in Appendix B to verify that no typographical mistakes have been made.

Adding User HMI Fields

- Locate the “User HMI Fields” tab, click on row 01, and click **Edit**.
- Fill in the information according to [Table 4.22](#) and click **OK**.
- Repeat the process for rows 02 to 21 using the settings in [Table 4.23](#) through [Table 4.42](#).

TABLE 4.22—USER HMI FIELD ITEM 01 (STATIONTIEIN.FLOWING.TOTALACTIVETIME)

Item	Value
PropertyName	StationTieIn.Flowing.TotalActiveTime

Item	Value
Header	Station Tie In Statistics
Description	Total time spent in Flowing state (seconds)
Webmodify	False

TABLE 4.23—USER HMI FIELD ITEM 02 (STATIONTIEIN.FLOWING.ACTIVETIME)

Item	Value
PropertyName	StationTieIn.Flowing.ActiveTime
Header	<i>Leave blank</i>
Description	Amount of time since Flowing state entered (seconds)
Webmodify	False

TABLE 4.24—USER HMI FIELD ITEM 03 (STATIONTIEIN.SHUTIN.TOTALACTIVETIME)

Item	Value
PropertyName	StationTieIn.ShutIn.TotalActiveTime
Header	<i>Leave blank</i>
Description	Total time spent in ShutIn state (seconds)
Webmodify	False

TABLE 4.25—USER HMI FIELD ITEM 04 (STATIONTIEIN.SHUTIN.ACTIVETIME)

Item	Value
PropertyName	StationTieIn.ShutIn.ActiveTime
Header	<i>Leave blank</i>
Description	Amount of time since ShutIn state entered (seconds)
Webmodify	False

TABLE 4.26—USER HMI FIELD ITEM 05 (PERCENTFLOWINGTIME.VALUE)

Item	Value
PropertyName	PercentFlowingTime.Value
Header	<i>Leave blank</i>
Description	Percent of total time spent in Flowing state
Webmodify	False

TABLE 4.27—USER HMI FIELD ITEM 06 (STATIONFLOWRATE.VALUE)

Item	Value
PropertyName	StationFlowRate.Value
Header	Station Rate and Pressure
Description	Flow rate through station
Webmodify	False

TABLE 4.28—USER HMI FIELD ITEM 07 (STATIONINLETPRESSURE.VALUE)

Item	Value
PropertyName	StationInletPressure.Value
Header	<i>Leave blank</i>
Description	Static pressure at station inlet
Webmodify	False

TABLE 4.29—USER HMI FIELD ITEM 08 (STATIONOUTLETPRESSURE.VALUE)

Item	Value
PropertyName	StationOutletPressure.Value
Header	<i>Leave blank</i>
Description	Static pressure at station outlet
Webmodify	False

TABLE 4.30—USER HMI FIELD ITEM 09 (STATIONDP.VALUE)

Item	Value
PropertyName	StationDP.Value
Header	<i>Leave blank</i>
Description	Difference between InletPressure and OutletPressure
Webmodify	False

TABLE 4.31—USER HMI FIELD ITEM 10 (FLOWVALVECONTROLLER.OUTPUT)

Item	Value
PropertyName	FlowValveController.Output
Header	<i>Leave blank</i>
Description	Normalized output of FlowValveController
Webmodify	False

TABLE 4.32—USER HMI FIELD ITEM 11 (SHUTINSTARTEDALARM.ISACTIVE)

Item	Value
PropertyName	ShutInStartedAlarm.IsActive
Header	Shut In Alarm
Description	Notifies system that ShutIn state has been entered (1 = Active)
Webmodify	False

TABLE 4.33—USER HMI FIELD ITEM 12 (SHUTINSTARTEDALARM.HOLDOFFTIME)

Item	Value
PropertyName	ShutInStartedAlarm.HoldOffTime
Header	<i>Leave blank</i>
Description	ShutIn alarm active count in hold-off (seconds)
Webmodify	False

TABLE 4.34—USER HMI FIELD ITEM 13 (SHUTINSTARTEDALARM.HOLDOFFDELAY)

Item	Value
PropertyName	ShutInStartedAlarm.HoldOffDelay
Header	<i>Leave blank</i>
Description	Configured ShutIn alarm hold-off delay (seconds)
Webmodify	True

TABLE 4.35—USER HMI FIELD ITEM 14 (SHUTINSTARTEDALARM.ISACTIVE)

Item	Value
PropertyName	FlowingStartedAlarm.IsActive
Header	Flowing Alarm
Description	Notifies system that Flowing state has been entered (1 = Active)
Webmodify	False

TABLE 4.36—USER HMI FIELD ITEM 15 (FLOWINGSTARTEDALARM.HOLDOFFTIME)

Item	Value
PropertyName	FlowingStartedAlarm.HoldOffTime
Header	<i>Leave blank</i>
Description	Flowing alarm active count in hold-off (seconds)
Webmodify	False

TABLE 4.37—USER HMI FIELD ITEM 16 (FLOWINGSTARTEDALARM.HOLDOFFDELAY)

Item	Value
PropertyName	FlowingStartedAlarm.HoldOffDelay
Header	<i>Leave blank</i>
Description	Configured Flowing alarm hold-off delay (seconds)
Webmodify	True

TABLE 4.38—USER HMI FIELD ITEM 17 (PROGRAMCONTROL.VALUE)

Item	Value
PropertyName	ProgramControl.Value
Header	Program Control
Description	0 = hold program in ShutIn state, 1 = run program normally
Webmodify	True

TABLE 4.39—USER HMI FIELD ITEM 18 (FLOWVALVECONTROLLER.SETPOINT)

Item	Value
PropertyName	FlowValveController.SetPoint
Header	<i>Leave blank</i>
Description	The desired flow rate for the station, controlled by the flow valve
Webmodify	True

TABLE 4.40—USER HMI FIELD ITEM 19 (MINFLOWINGDP.VALUE)

Item	Value
PropertyName	MinFlowingDP.Value
Header	<i>Leave blank</i>
Description	Minimum differential pressure threshold for Flowing state, below which we transition to ShutIn state
Webmodify	True

TABLE 4.41—USER HMI FIELD ITEM 20 (MAXSHUTINDP.VALUE)

Item	Value
PropertyName	MaxShutInDP.Value
Header	<i>Leave blank</i>
Description	Maximum differential pressure threshold for ShutIn state, above which we transition to Flowing state

Item	Value
Webmodify	True

TABLE 4.42—USER HMI FIELD ITEM 21 (MINSHUTINTIME.VALUE)

Item	Value
PropertyName	MinShutInTime.Value
Header	<i>Leave blank</i>
Description	Required minimum time to remain in ShutIn state (minutes)
Webmodify	True

This page is left blank intentionally.

SECTION 5—DEBUGGING SCRIPTS

The purpose of Debug Mode is to give the user a detailed view of the runtime execution of a Scanner Logic program. The IDE has tools that allow a user to control the runtime execution of a program so that the following observations can be made:

- Task switch execution events
- When a task has a state change
- When a program line is reached
- The results of a single line execution
- The contents of register and resource properties at any point in the execution
- The line-by-line execution flow of a program

This section will describe how to establish a debug session and the tools used to control and observe the program execution.

STARTING A DEBUG SESSION

Starting a debug session with a Scanner requires an error-free Scanner Logic program, the selection of a compiler target platform, and the selection of a Scanner 3100 device connection.

Reviewing Device Connections

Select the Device Connections grid ([Figure 5.1](#)) to view all Scanner device connections that have been created within the IDE. If no connections exist, see [Section 3](#) for instructions about creating a new connection.

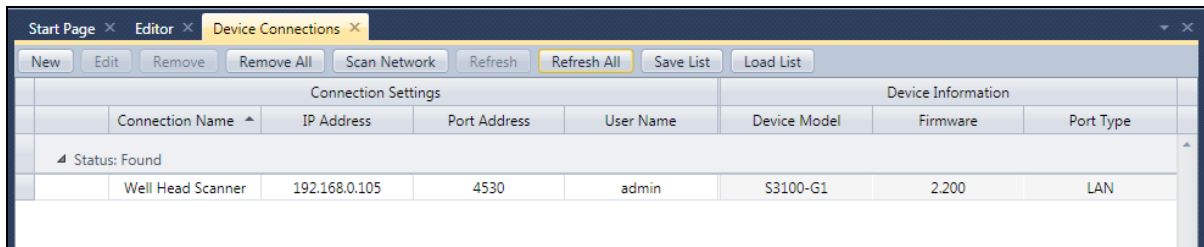


Figure 5.1—Reviewing active device connections.

The Scanner for which you wish to start a debug session must be accessible over a device connection.

Click **Refresh All** to test the current state of the desired device connection. The IDE will attempt to communicate over all the created device connections and retrieve the device information from all Scanners that are found.

If the desired device connection is found, note the device model and firmware version to complete the Debug Session start up.

An attempt to start a debug session with a lost device is permitted. The IDE will make the attempt to locate the desired connection and notify the user if the Scanner cannot be found.

Selecting Target Platform and Device Connection

Starting the debug session involves compiling the Scanner Logic program open in the Editor and uploading the resulting SLBIN file to the Scanner over the selected device connection.

To ensure the correct SLBIN is created from the compilation process, the target platform must be selected. “Target Platform” refers to the Scanner device’s model and firmware version.

1. To select a target platform, choose **Scanner>Change Target Platform** or click in the Target Platform selector (*Figure 5.2*) and select the desired target platform.

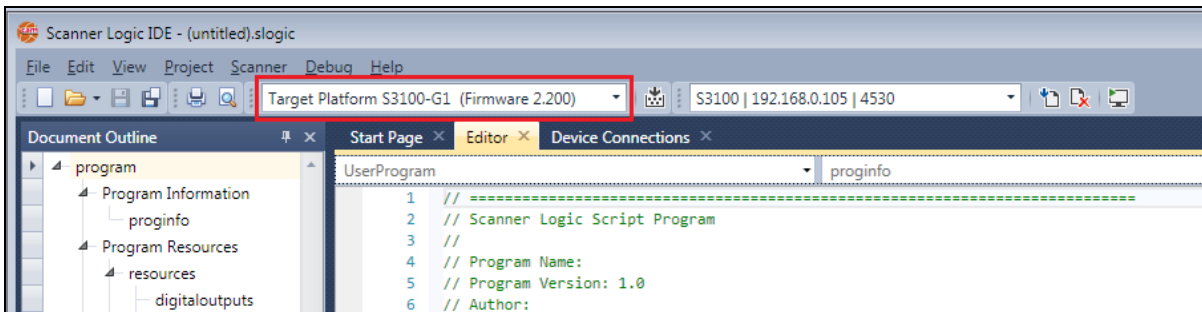


Figure 5.2—Target Platform selector

The target platform selected must match the device model and firmware version of the Scanner at the desired device connection. An SLBIN compiled to an incompatible Target Platform will be rejected by the Scanner.

2. Choose the desired device connection by selecting **Scanner>Select Device Connections** or by clicking in the Device Connection selector (*Figure 5.3*) and selecting the desired connection.

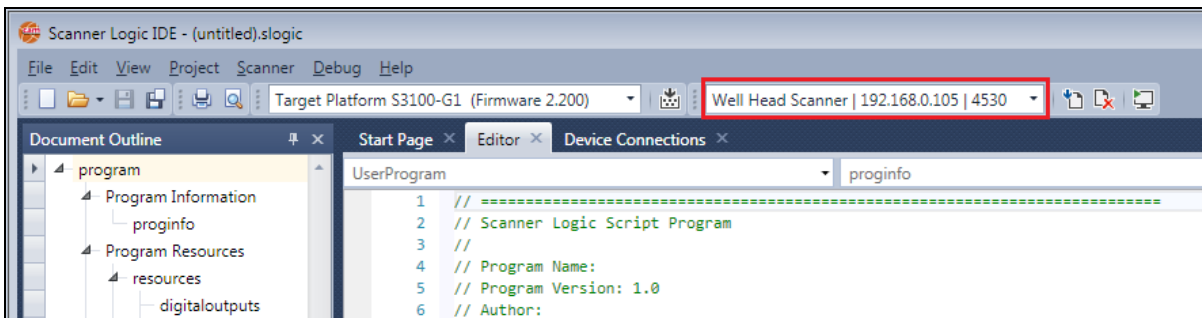


Figure 5.3—Device Connection selector

Debug Session Start Sequence

Select **Debug>Start Debugging** on the menu bar of the IDE to start a debug session.

Tip From the Editor, press F8 to start a debug session.

To begin the debug session, the IDE automatically performs the actions below in sequence:

- A save command of any previously unsaved changes will be sent to the file open in the Editor.
- A compiler build command to create the SLBIN file will be sent. Any errors still present in the program (e.g. typos, syntax errors, etc.) will halt this process. Any reported compiler errors (e.g. binary is too large to load to the Scanner, or is too demanding to run) will terminate this process.
- The TCP/IP connection will be established via the selected device connection. If the connection cannot be established, the process will be terminated.
- The IDE will request that any installed SLBIN file be erased. If the Scanner does not have an installed SLBIN, the user will not be prompted. Otherwise, the user must grant permission to perform the erasure.

- The IDE will request a debug session in the Scanner. While the debug session is active, the Scanner communicates Scanner Logic runtime data to the IDE after each execution cycle (i.e. every second). This data is like a snapshot of the logic system used to populate the debug panels and indicators. The debug session makes it possible to stop the logic program execution cycle. The debug session will last until it completes or until 20 minutes of inactivity has been detected.
- The IDE will preconfigure a halt condition (like the “Break All” command) so the new SLBIN will be halted at the beginning of its first execution cycle.
- The IDE will upload the new SLBIN file to the Scanner to validate the integrity of the file and to verify that it was built to the required target platform. The IDE will also confirm the authenticity and integrity of the transfer. If any check fails, the process will be terminated.
- The Scanner will begin executing the uploaded program and halt when it reaches the first user-created state execution.
- The IDE will transition into Debug mode. While in Debug mode, the Editor document will be set to read-only. The Debug mode has its own window layout ([Figure 5.4](#)) that can be modified by the user.

Debug Mode Windows Layout

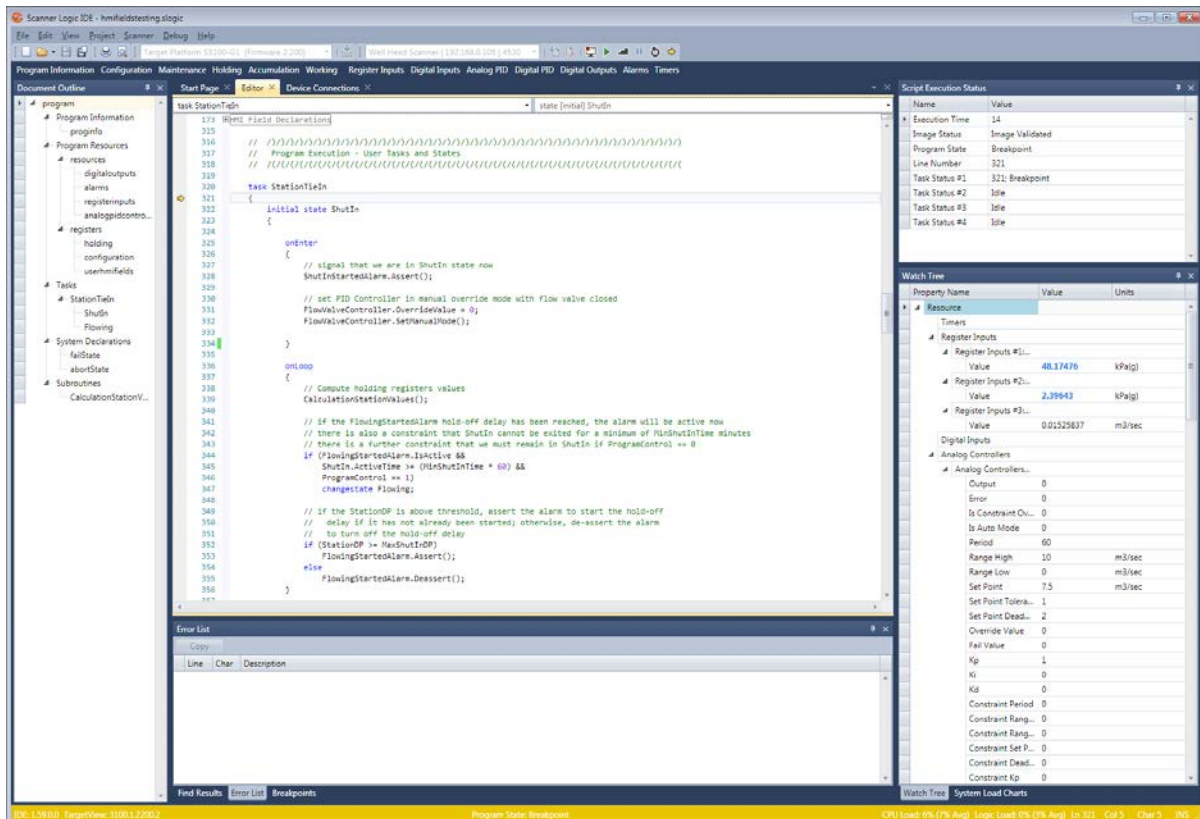


Figure 5.4—Debug mode windows default layout

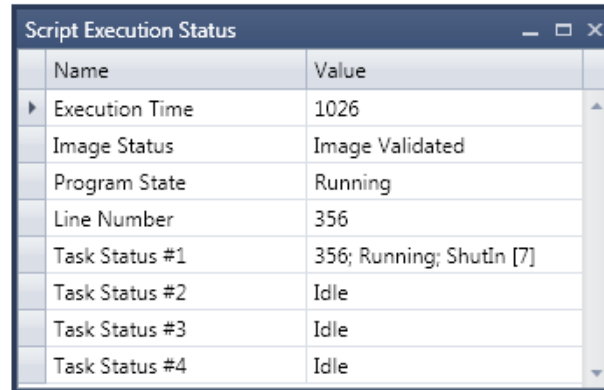
The Debug Mode windows layout is a customizable layout loaded when the IDE successfully enters debug mode. Like the Edit Mode windows layout, this layout is persistent for a given target platform selection.

Each of the Debug Mode windows presents a view of the state of an executing Scanner Logic program, as well as the runtime data produced by the program. The contents of each window are updated when debug data is received from the connected Scanner.

While in debug mode, the Editor is read-only and the program cannot be modified. All Resource and Register grids are also available to view but cannot be edited. By default, these grids are parked along the top of the Debug Mode windows layout and set to auto-collapse.

Script Execution Status

The *Script Execution Status* window shows detailed information about the program's current execution state.



Name	Value
Execution Time	1026
Image Status	Image Validated
Program State	Running
Line Number	356
Task Status #1	356; Running; ShutIn [7]
Task Status #2	Idle
Task Status #3	Idle
Task Status #4	Idle

Figure 5.5—Script Execution Status window

The following rows in the grid provide insight into the execution state of the program.

- Execution Time—Seconds since program start or restart
- Image Status—The validation status of the SLBIN file in the Scanner.

Note This will always be reported as “Image Validated” when debugging since a valid SLBIN is required to begin, and debugging will abort if the state ever changes.

- Program State—Possible states are Running, Breakpoint, Fail, Fail Breakpoint, Abort, and Abort Breakpoint.
- Line Number—The current executing line number in the program. This line can belong to any task. This is also indicated in the Editor breakpoint margin by a yellow arrow.
- Task Status #N (where N = [1,2,3,4])—The current task information if the program is currently executing code within the indicated Task block. Composed of four elements:
 - <TaskLineNumber>; <TaskExecutionState>; <TaskCurrentStateName> [<TaskCurrentStateID>]
 - TaskLineNumber—The current line number executed inside that task.
 - TaskExecutionState—The current Execution State of that task (“Idle”, “Running”, or “Breakpoint”)
 - TaskCurrentStateName—The name of the current or last known state block executed in the task.
 - TaskCurrentStateID—The ID of the current or last known state block executed in the task.

Watch Tree

The Debug Watch Tree window contains a collapsible tree view of all the runtime data from the relevant Scanner Logic objects reported by the connected Scanner. The relevant objects are all user-declared objects from the program (Resource, Register, Task, and State objects), as well the system objects (listed under the “System” node in Watch Tree). The declared user object names are included to help identify an object used by the programmer for a specific purpose.

Property Name	Value	Units
Alarms		
Alarms #1: ShutInStartedAlarm		
Is Active	1	
Is Asserted	1	
Hold Off Delay	30	
Hold Off Time	30	
Active Time	43904	
Inactive Time	0	
Alarms #2: FlowingStartedAlarm		
Is Active	0	
Is Asserted	0	
Hold Off Delay	30	
Hold Off Time	0	
Active Time	0	
Inactive Time	44849	
Register		
Configuration		
Configuration #1: ProgramControl		
Value	0	
Configuration #2: MinFlowingDP		
Value	200	kPa
Configuration #3: MaxShutInDP		
Value	225	kPa
Configuration #4: MinShutInTime		
Value	15	
Maintenance		
Holding		
Holding #1: StationDP		
Value	45.26842	kPa
Holding #2: PercentFlowingTime		
Value	0	%
Accumulation		

Figure 5.6—Debug Watch Tree window

The grid items with attached values within the tree are the property members of each object. The units of the property values are those selected in the declarations of the objects within the Scanner Logic program. For a complete description of the Scanner Logic object properties, refer to the Scanner Logic Programmer Manual.

The Watch Tree supports runtime value change indication. If the contents of a cell in the value column have changed between updates, the value will be emphasized in bold blue font. When the next runtime data update is received (in one second), all emphasis will reset or persist based on any change in the updated value.

Depending on the Scanner Logic program size being executed and the number of objects declared within it, the Watch Tree can contain a considerable amount of data. To limit the view to property values of interest, use the collapse property. Right-click within the Watch Tree to access the “Collapse Children,” “Expand Children,” “Collapse All,” and “Expand All” commands.

System Load Charts

The System Load charts provide another view of the execution status of the connected Scanner. The two scrolling charts display a brief history of the CPU usages of the metrological functions and the Scanner Logic execution load over the last 30 seconds of received debug data. Each bar indicates the Scanner’s processor load as measured over the previous 1-second calculation period.

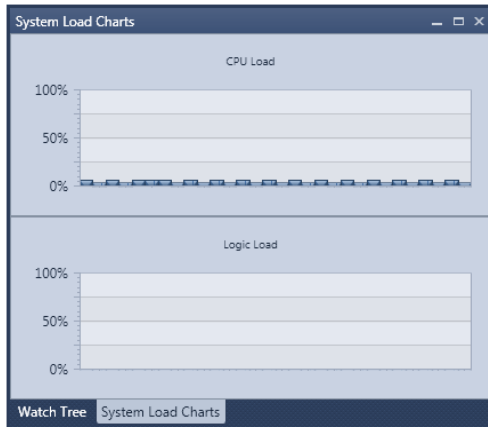


Figure 5.7—System Load charts

Considerable CPU resources are allocated for the Scanner Logic execution and only Scanner Logic programs containing states with large execution blocks will register in the Logic Load chart.

If a program has large execution blocks within states causing the Scanner to approach the 100% Logic Load limit, refactor the program states. Splitting up large states into multiple smaller states can make the Logic Script more readable and reduce the Logic Load.

Debug Status Bar

While in Debug mode, the status bar at the bottom of the IDE indicates the debug state. Because it is always available for viewing, the user should become familiar within its operation.



Figure 5.8—Debug status bar

The status bar's background color will change to indicate the Program State. [Table 5.1](#) shows the possible program states.

TABLE 5.1—POSSIBLE PROGRAM STATES

Program State	Color	Status
Running	Green	Program is executing normally.
Breakpoint	Orange	Program execution is halted: <ul style="list-style-type: none"> At a user-defined breakpoint (indicated in Editor Margin and Breakpoint tab) After executing the operation at the point the “Break All” command is received by the Scanner.
Fail	Red	Program is executing code in the failState block.
Fail Breakpoint	Red	Program execution is halted in failState block.
Abort	Orange	Program is executing code in the abortState block.
Abort Breakpoint	Orange	Program execution is halted in the abortState block.

Stepping through Program Code

The user can identify the state of program execution using the Script Execution Status window and Debug status bar. Both tools will indicate program execution status and the current execution line number and are described in the Debug Mode Windows Layout section. Within the Editor, a yellow arrow indicator (➡) is placed in the margin on the current execution line.

Program execution is controlled using debug commands and breakpoints. The debug commands are direct controls over the starting and stopping of program execution while the breakpoint allows the user to indirectly halt execution when the Scanner reaches a marked line number.

Debug Commands

Run

The “Run” command will start or resume the program execution. When transmitted to the Scanner via the device connection, the Scanner will begin a “free run” of the program from the current execution line. The Scanner will continue to run the program until halted. Halting can occur because the program reaches a breakpoint or the “Break All” command is executed by the user.

While the program status is running, the current line indication and line number will be the last line executed within a Scanner Logic execution cycle. As the execution changes between the user-defined states within the program, the current line can be expected to jump.

While the program status is running, the Watch Tree will also be updated with the live runtime data once per second. The Watch Tree indicates which values have changed from the previous update by emboldening a property value in a blue font.

The “Run” command is only available when the program is halted. It can be performed by selecting **Debug>Run** on the menu bar.

Step

The “Step” command will execute a single program line at the current halted program line number. When transmitted to the Scanner over the device connection, the Scanner will begin program execution but will then halt when the beginning of a new program line is detected.

When the “Step” command is executed, the program status will remain as “breakpoint.” With each successive “Step” command both the current line and the Watch Tree will be updated. In Scanner Logic, a statement may be expressed over multiple line numbers. When stepping through a Scanner Logic statement that is expressed on multiple lines, the current line number may return to the first line of the statement or expression after it is fully executed.

The “Step” command is only available when the program is halted. It can be performed by selecting **Debug>Step** on the menu bar.

Break All

The “Break All” command will halt execution. When transmitted to the Scanner over the device connection, the Scanner will complete the currently executing line and then halt execution.

The “Break All” command is only available when the program is running. The command can be performed by selecting **Debug>Break All** on the menu bar.

Restart

The “Restart” command will restart the program execution from the beginning. When transmitted to the Scanner over the device connection, the Scanner will cease the current execution and restart the program as though a device power on reset has occurred. Any non-volatile object property will be restored (such as the value of Configuration and Maintenance registers) and all other object properties will be set to their initial value parameters in the object declaration. The program will be halted on the first line within the first declared task.

The “Restart” command is only available when the program is running. It can be performed by selecting **Debug>Restart** on the menu bar.

Go To Current Execution Line

The “Go To Current Execution Line” command will refocus the Editor to the location of the current execution line and move the cursor there. This is a useful debug command when working with larger script. The “Go To Current Execution Line” command does not affect the program execution and is not transmitted to the Scanner.

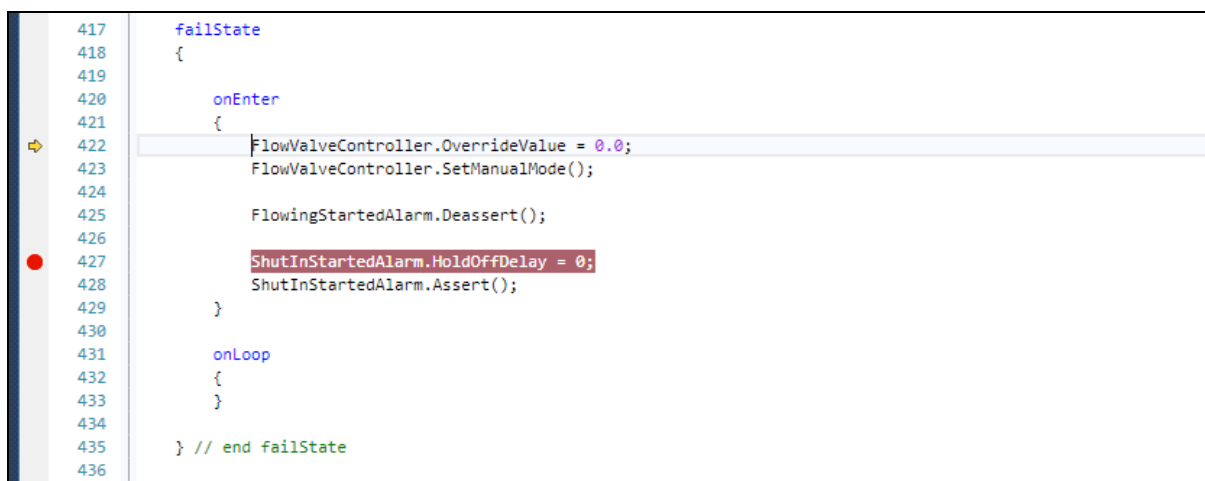
If the Editor is displaying a vertical split, the focus and cursor position will only be changed in the lower view.

The “Go To Current Execution Line” command is only available when the program is halted. It can be performed by selecting **Debug>Go To Current Execution Line** on the menu bar.

Breakpoints

Breakpoints allow the user to indirectly halt program execution when the Scanner reaches a marked line number. Breakpoints can be used to halt and view the program status as it changes states or just before a line of interest is executed. The Scanner Logic IDE allows up to 16 breakpoints to be defined at any one time.

Breakpoints are viewable within the margin of the Editor. If a breakpoint is inserted on a program line, the Editor will display a red circle indicator in the margin and highlight the statement or expression marking the point of halted execution.



```

417     failState
418     {
419
420         onEnter
421         {
422             FlowValveController.OverrideValue = 0.0;
423             FlowValveController.SetManualMode();
424
425             FlowingStartedAlarm.Deassert();
426
427             ShutInStartedAlarm.HoldOffDelay = 0;
428             ShutInStartedAlarm.Assert();
429         }
430
431         onLoop
432         {
433         }
434
435     } // end failState
436

```

Figure 5.8—Breakpoints in the Editor

The left-most portion of the Editor margin can be used to insert and remove breakpoints on a line. Left-clicking the mouse within the debug margin will toggle a breakpoint on the line.

Breakpoints cannot be added on blank or comment lines. In general, you can place a breakpoint on a line that contains a semicolon (i.e. a “statement”) or contains code in parenthesis (i.e. “expressions”). If a single line contains multiple statements or expressions, users may have to separate that code such that the statement they desire to place a breakpoint on is on its own line.

Breakpoints can also be managed using the Breakpoint grid. Within the grid, each breakpoint is listed in the order it was added. At the current cursor location within the Editor, a breakpoint can be added and removed with the **Toggle Breakpoint** button. Breakpoints can also be disabled within this grid. Disabled breakpoints are retained by the IDE, but do not cause the Scanner to halt execution.

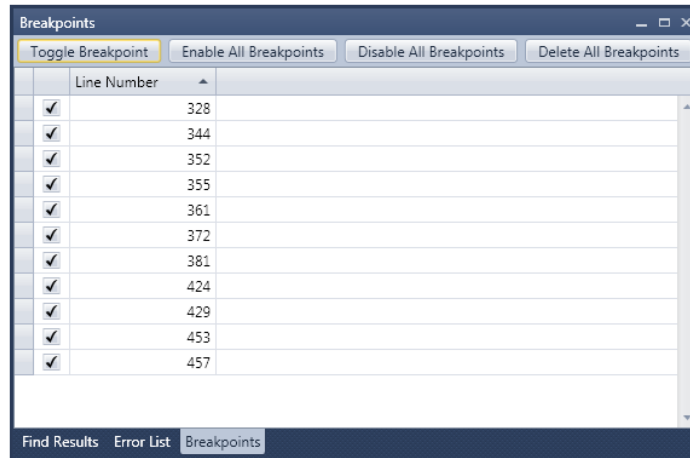


Figure 5.9—Debug Breakpoints

Stop Debugging

The user can stop the debug session with a Scanner and exit Debug mode with the “Stop Debugging” command. When transmitted to the Scanner over the device connection, the Scanner will close the debug session with the IDE, stop the transmission of run time data updates, and disconnect from the IDE. The program execution status on the Scanner will be changed to “Running and execution will freely proceed from the last line executed in the debug mode. With the debug session closed, the Scanner will ignore all breakpoints.

The IDE saves the current debug window layout and closes it, then loads the Edit Mode window layout.

The “Stop Debugging” command is available during all program execution states. It can be performed by selecting **Debug>Stop Debugging** on the menu bar.

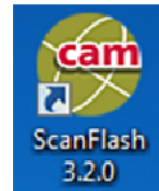
This page is left blank intentionally.

APPENDIX A—OTHER PROGRAMS

DOWNLOADING AND INSTALLING SCANFLASH

ScanFlash* is an optional utility that can be used to simultaneously upload firmware updates and configuration files to the SCANNER 3100 flow computer. To download the utility,

1. Go to the SCANNER 3100 website at <http://www.cameron.slb.com/flowcomputers>.
2. Select **Scanner Model 3100 Flow Computer**.
3. Locate the ScanFlash utility under the “Software” heading to the right of the page.
4. Right-click, choose **SAVE LINK AS...**, and select the desired storage location. By default, the file will be saved to C:\USERNAME\Downloads.
5. Browse to the Installation file and double-click it to open.
6. Select **Setup.exe** and run the installation program. By default, the files will be stored to C:\Cameron\ScanFlash.



This page is left blank intentionally.

APPENDIX B—SAMPLE PROGRAM SOLUTION

```
// =====
// Scanner LogicScript Program
//
// Program Name:
// Program Version: 1.0
// Author:
// Date: 10/28/2017
// Purpose:
//
// =====

program
{
#region Program Information

    proginfo
    {
        ProgramName: "IDE Getting Started Example";
        ProgramAuthor: "John Smith";
        ProgramOwner: "OilCo Ltd";
        ProgramVersion: 1.000;
        ProgramCreationDate: "10/28/2017";
        Access_OnlineSource: "allusers";
        Access_OnlineControls: "allusers";
        Access_WriteHMI: "allusers";
        ProgramDescription: "This is a tutorial program for writing a program with the Scanner Logic
Script language in the Scanner Logic IDE.";
    }

#endregion

#region Program Declarations

resource digitaloutputs
{
    01: ShutInSignal
    {
        description: "Output signal to indicate that the system is in the ShutIn state";
        initial_IsActive: false;
        initial_FollowAlarm: 1;
        initial_Period: 2;
        initial_Duration: 1;
    }
}

registers holding
{
    01: StationDP
    {
        group: "";
        description: "Difference between InletPressure and OutletPressure";
        category: "Differential Pressure";
        units: "kPa";
        initial_Value: 0;
    }
    02: PercentFlowingTime
    {
        group: "";
        description: "Percent of total time spent in Flowing state";
        category: "Percent";
        units: "%";
        initial_Value: 0;
    }
}
}
```

```

registers configuration
{
  01: ProgramControl
  {
    group: "";
    description: "0 = hold program in ShutIn state, 1 = run program normally";
    category: "No Units";
    units: "";
    initial_Value: 0;
  }
  02: MinFlowingDP
  {
    group: "";
    description: "Minimum differential pressure threshold for Flowing state, below which we
transition to ShutIn state";
    category: "Differential Pressure";
    units: "kPa";
    initial_Value: 200;
  }
  03: MaxShutInDP
  {
    group: "";
    description: "Maximum differential pressure threshold for ShutIn state, above which we
transition to Flowing state";
    category: "Differential Pressure";
    units: "kPa";
    initial_Value: 225;
  }
  04: MinShutInTime
  {
    group: "";
    description: "Required minimum time to remain in ShutIn state (minutes)";
    category: "No Units";
    units: "";
    initial_Value: 15;
  }
}

resource alarms
{
  01: ShutInStartedAlarm
  {
    description: "Notifies system that ShutIn state has been entered";
    initial_IsAsserted: true;
    initial_HoldOffDelay: 30;
  }
  02: FlowingStartedAlarm
  {
    description: "Notifies system when Flowing state has been entered";
    initial_IsAsserted: false;
    initial_HoldOffDelay: 30;
  }
}

resource analogpidcontrollers
{
  01: FlowValveController
  {
    description: "Maintains desired flow rate at station by controlling flow valve";
    webcontrolflags: 0x003F;
    processvar: StationFlowRate;
    pidtype: "simplepid";
    pidaction: "direct";
    initial_IsAutoMode: false;
    initial_Period: 1;
    initial_RangeHigh: 10;
    initial_RangeLow: 0;
    initial_SetPoint: 7.5;
    initial_SetPointTolerance: 1;
    initial_SetPointDeadBand: 2;
    initial_OverrideValue: 0;
  }
}

```

```

        initial_FailValue: 0;
        initial_Kp: 1;
        initial_Ki: 0;
        initial_Kd: 0;
    }
}

resource registerinputs
{
    01: StationInletPressure
    {
        description: "Line pressure at station inlet, upstream from the flow control valve
(integrated MVT)";
        tagname: "Stat Press: Holding: Inst Reading";
        tagcode: "m32_FC_IN_2_Holding_InstReading";
        category: "Static Pressure (gauge)";
        units: "kPa(g)";
    }
    02: StationOutletPressure
    {
        description: "Line pressure at station outlet, downstream from the flow control valve
(pressure transducer)";
        tagname: "Analog 1: Holding: Inst Reading";
        tagcode: "m32_FC_IN_5_Holding_InstReading";
        category: "Static Pressure (gauge)";
        units: "kPa(g)";
    }
    03: StationFlowRate
    {
        description: "Flow rate through station, as limited by flow control valve";
        tagname: "FR1: HAccum: Gas Volume Flow Rate";
        tagcode: "m32_FC_FR_1_HoldingAccum_GasVolumeFlowRate";
        category: "Gas Volume";
        units: "m3";
        rate: "/sec";
    }
}

#endregion

#region HMI Field Declarations

hmifields user
{
    01: UserHMI_01
    {
        propertyname: "StationTieIn.Flowing.TotalActiveTime";
        header: "Station Tie In Statistics";
        description: "Total time spent in Flowing state (seconds)";
        webmodify: false;
    }
    02: UserHMI_02
    {
        propertyname: "StationTieIn.Flowing.ActiveTime";
        description: "Amount of time since Flowing state entered (seconds)";
        webmodify: false;
    }
    03: UserHMI_03
    {
        propertyname: "StationTieIn.ShutIn.TotalActiveTime";
        description: "Total time spent in ShutIn state (seconds)";
        webmodify: false;
    }
    04: UserHMI_04
    {
        propertyname: "StationTieIn.ShutIn.ActiveTime";
        description: "Amount of time since ShutIn state entered (seconds)";
        webmodify: false;
    }
    05: UserHMI_05
    {

```

```
    propertyname: "PercentFlowingTime.Value";
    description: "Percent of total time spent in Flowing state";
    webmodify: false;
}
06: UserHMI_06
{
    propertyname: "StationFlowRate.Value";
    header: "Station Rate and Pressure";
    description: "Flow rate through station";
    webmodify: false;
}
07: UserHMI_07
{
    propertyname: "StationInletPressure.Value";
    description: "Static pressure at station inlet";
    webmodify: false;
}
08: UserHMI_08
{
    propertyname: "StationOutletPressure.Value";
    description: "Static pressure at station outlet";
    webmodify: false;
}
09: UserHMI_09
{
    propertyname: "StationDP.Value";
    description: "Difference between InletPressure and OutletPressure";
    webmodify: false;
}
10: UserHMI_10
{
    propertyname: "FlowValveController.Output";
    description: "Normalized output of FlowValveController";
    webmodify: false;
}
11: UserHMI_11
{
    propertyname: "ShutInStartedAlarm.IsActive";
    header: "Shut In Alarm";
    description: "Notifies system that ShutIn state has been entered (1 = Active)";
    webmodify: false;
}
12: UserHMI_12
{
    propertyname: "ShutInStartedAlarm.HoldOffTime";
    header: "";
    description: "ShutIn alarm active count in hold-off (seconds)";
    webmodify: false;
}
13: UserHMI_13
{
    propertyname: "ShutInStartedAlarm.HoldOffDelay";
    header: "";
    description: "Configured ShutIn alarm hold-off delay (seconds)";
    webmodify: true;
}
14: UserHMI_14
{
    propertyname: "FlowingStartedAlarm.IsActive";
    header: "Flowing Alarm";
    description: "Notifies system that Flowing state has been entered (1 = Active)";
    webmodify: false;
}
15: UserHMI_15
{
    propertyname: "FlowingStartedAlarm.HoldOffTime";
    header: "";
    description: "Flowing alarm active count in hold-off (seconds)";
```



```

    // Compute holding registers values
    CalculateStationValues();

    // if the FlowingStartedAlarm hold-off delay has been reached, the alarm will be
    // active now
    // there is also a constraint that ShutIn cannot be exited for a minimum of
    // MinShutInTime minutes
    // there is a further constraint that we must remain in ShutIn if ProgramControl == 0
    if (FlowingStartedAlarm.IsActive &&
        ShutIn.ActiveTime >= (MinShutInTime * 60) &&
        ProgramControl == 1)
        changestate Flowing;

    // if the StationDP is above threshold, assert the alarm to start the hold-off
    // delay if it has not already been started; otherwise, de-assert the alarm
    // to turn off the hold-off delay
    if (StationDP >= MaxShutInDP)
        FlowingStartedAlarm.Assert();
    else
        FlowingStartedAlarm.Deassert();
}

onExit
{
    // signal that we are out of ShutIn state now
    // (ShutInSignal digital output follows this alarm)
    ShutInStartedAlarm.Deassert();
}

} // end state ShutIn

state Flowing
{
    onEnter
    {
        // signal that we are in Flowing state now
        FlowingStartedAlarm.Assert();

        // put PID controller into auto mode and control flow valve to seek setpoint
        FlowValveController.SetAutoMode();
    }

    onLoop
    {
        // Compute holding registers values
        CalculateStationValues();

        // if the ShutInStartedAlarm hold-off delay has been reached,
        // the alarm will be active now.

        if (ShutInStartedAlarm.IsActive ||
            ProgramControl == 0)
            changestate ShutIn;

        // if the StationDP is below threshold, assert the alarm to start the hold-off
        // delay if it has not already been started; otherwise, de-assert the alarm
        // to turn off the hold-off delay
        if (StationDP <= MinFlowingDP)
            ShutInStartedAlarm.Assert();
        else
            ShutInStartedAlarm.Deassert();
    }

    onExit
    {
        // signal that we are out of Flowing state now
        FlowingStartedAlarm.Deassert();
    }

} // end state Flowing

```



```
void subroutine CalculateStationValues()
{
    // Compute the Station Differential Pressure
    StationDP = StationInletPressure - StationOutletPressure;

    // Compute the Percent Flowing Time.
    PercentFlowingTime = StationTieIn.Flowing.TotalActiveTime /
    (StationTieIn.Flowing.TotalActiveTime + StationTieIn.ShutIn.TotalActiveTime);
}

#endregion

} // end program
```

APPENDIX C—PARSER ERROR MESSAGES

This appendix contains a list of parser error messages that you may encounter. These errors are generally the result of accidental coding mistakes. Correct the mistakes indicated, and the error messages will disappear when the code is reparsed. Refer to the Programmers Manual if necessary.

"Unexpected token: 'xxx'"	'Timer1' does not contain a definition for 'StartTime'
'Activate' is a method, which is not valid in this context. Did you intend to invoke the method?	'CalculateTotals' is a subroutine, which is not valid in the given context.
The name 'LinePressure' does not exist in the current context.	'CalculateFactor' is a subroutine, which is not valid in the given context. Did you intend to invoke the subroutine?
Integer constant is outside the range of type 'uint'.	Floating point constant is outside the range of type 'float'.
Target 'task' must be an object to access member properties or methods.	Non-invocable member 'ActiveTime' cannot be used like a method.
'RestartExecution' method of task 'MainTask' cannot be invoked from within the 'MainTask' task.	'RestartExecution' method of task 'MainTask' cannot be invoked in the current context.
Subroutine name expected.	Subroutines cannot be called recursively.
Subroutine calls cannot be made within other subroutines.	Non-invocable expression 'DoCalculations()' cannot be used like a method.
Method 'Sin' takes 1 argument.	Method 'Power' takes 2 arguments.
Method 'Activate' takes 0 arguments.	There is no argument given that corresponds to the required formal parameter 'Exponent' of 'Power(float,float)'
Argument 1: Cannot implicitly convert type 'bool' to 'float'.	The left-hand side of an assignment must be a property of an object.
Property 'Output' cannot be assigned to -- it is read only.	Default property 'IsActive' of 'OverflowAlarm' cannot be assigned to -- it is read only.
Cannot convert type 'float' to object type 'RealTime'. Did you intend to assign to a property of the object?	Cannot assign a negative value to a property of type 'uint'.
Cannot assign a value of type 'float' to a property of type 'uint'. Try an explicit cast to convert the value type.	Unknown parameter 'categories'.
Cannot assign a value of type 'string' to a parameter of type 'float'. Parameter values must be of the correct type.	The name 'FlowAlarm' is not a RegisterInputResource.
Operator '+' cannot be applied to operands of type 'float' and 'string'.	Invalid operation. Division by constant zero.
Cannot convert type 'string' to 'float'.	Type casts to type 'string' are not supported.

The type 'int32' could not be found.	Operator '-' cannot be applied to operand of type 'bool'.
The operand of an increment or decrement operator must be a variable or property.	The operand of an increment or decrement operator cannot be a constant.
Cannot implicitly convert type 'uint' to 'bool'. Are you missing a type cast?	The 'changestate' statement cannot be used in this context. It can only be used within an onLoop block.
The 'changestate' statement cannot be used inside the Abort state.	The 'return' statement cannot be used in this context.
The 'continue' statement cannot be used in this context.	The 'resource alarms' declaration already exists in this scope.
"The index '99' is outside of the valid range of 01 to 64 for 'registers holding'.	The task 'ExtraTask' exceeds the maximum number of tasks permitted (4).
The state 'AnotherState' exceeds the maximum number of states permitted (96).	The name 'Flow' is already defined in the program scope.
Missing 'onExit' block in the state scope.	Missing 'abortState' declaration in the program scope.
Missing 'failState' declaration in the program scope.	There is already an 'abortState' declared in this program.
Missing 'onEnter' block in the abortState scope.	There is already an 'failState' declared in this program.
Missing 'onLoop' block in the failState scope.	The 'onExit' block is not valid in this scope.
The subroutine 'ExtraSub' exceeds the maximum number of subroutines permitted (100).	Parameter 'RealTime.Day' cannot be assigned to - it is read only.
The parameter named 'initial_Value' has already been assigned in this scope.	The index '01' has already been used in this scope.
The name 'Alarm2' is already defined in the program scope.	Unknown value for 'category' parameter: 'Freq'.
Unknown value for 'category' parameter: 'Current'. Note that the Accumulation register type has fewer allowed categories values.	Missing 'category' parameter
Unknown value for 'units' parameter: 'cubicft'. See documentation for valid units of category 'Gas Volume'.	Unknown value for 'rate' parameter: '/week'.
Unknown value for 'pidtype' parameter: 'test'. See documentation for valid parameter values.	Unknown value for 'pidaction' parameter: 'abc'. See documentation for valid parameter values.
Unknown value for 'con_pidaction' parameter: 'def'. See documentation for valid parameter values.	The 'propertyname' parameter could not be resolved to an object property: 'MyInput.Value'. Has the intended object been renamed?

When 'propertyname' parameter references a read-only object property, 'webmodify' parameter value cannot be 'true'.	An 'initial' state has already been declared in task 'Main'.
Missing 'initial' state declaration in task 'MonitorFlow'.	The name 'State1' is already defined in the current task scope.
Subroutine must have a return type.	Only assignment, invocation, increment, and decrement expressions can be used as a statement.
The 'proginfo' block has already been declared in this scope.	The Program Information section is out of sequence.
The Program Declarations section is out of sequence.	The HMI Field Declarations section is out of sequence.
The Task Declarations section is out of sequence.	Task declaration is missing state declarations.
The 'onEnter' section has already been declared in the state scope.	The 'onEnter' section is out of sequence.
The 'onLoop' section has already been declared in the state scope.	The 'onLoop' section is out of sequence.
The 'onExit' section has already been declared in the state scope.	Only the 'void' return type is supported in this release.

This page is left blank intentionally.

APPENDIX D—RUNTIME ERROR CODES

Code	Name	Description
SL_E000	OK	No error
SL_E001	Init State Execution Error	Execution error encountered within Initialization State. See Execution Result.
SL_E002	Fail State Execution Error	Execution error encountered within Fail State. See Execution Result.
SL_E003	Abort State Execution Error	Execution error encountered within Abort State. See Execution Result.
SL_E004	State On Enter Execution Error	Execution error encountered within OnEnter of User State. See Execution Result.
SL_E005	State On Loop Execution Error	Execution error encountered within OnLoop of User State. See Execution Result.
SL_E006	State On Exit Execution Error	Execution error encountered within OnExit of User State. See Execution Result.
SL_E007	Subroutine Execution Error	Execution error encountered within OnExec of User Subroutine. See Execution Result.
SL_E008	Register Input Category Not Matched	
SL_E009	Resource Mode Warning	
SL_E010	Init State Load Failed Because Image Invalid	Attempted to load Initialization State from an invalid SLBIN.
SL_E011	Init State Not Programmed	Illegal SLBIN has a Task that does not contain an initialization state.
SL_E012	Init State Corrupt	Initialization State unable to load because of corrupted state table.
SL_E013	Init State Not Ready	Internal error caused by attempting to execute Initialization State before it is loaded.
SL_E014	Init State Did Not Enter User State	Illegal Initialization State does not contain Enter command.
SL_E015	Init State Unexpected Error	Unexpected internal state indicating an execution error while loading Initialization State.
SL_E016	Failure To Continue Init State	Attempt to continue execution of Initialization State failed.
SL_E017	Illegal Goto From On Enter	Illegal Goto executed within OnEnter of a State.
SL_E018	Illegal Goto From On Exit	Illegal Goto executed within OnExit of a State.
SL_E019	State Load Failed Because Image Invalid	Attempted to load State from an invalid SLBIN.
SL_E020	State Not Programmed	Attempted to load a State index which is not included in current SLBIN.
SL_E021	Attempt To Load Invalid State Number	Attempted to load a State index which is out of range.
SL_E022	State Table Corrupt	SLBIN contains a corrupted State Table.
SL_E023	State Unexpected Error	Unexpected internal state indicating an execution error while loading State.
SL_E024	State On Enter Not Loaded	Internal error caused by attempting to execute OnEnter of a State before it is loaded.
SL_E025	State On Loop Not Loaded	Internal error caused by attempting to execute OnLoop of a State before it is loaded.

Code	Name	Description
SL_E026	State On Exit Not Loaded	Internal error caused by attempting to execute OnExit of a State before it is loaded.
SL_E027	State On Enter Not Ready	Internal error caused by attempting to execute OnEnter of a State with an unknown load error.
SL_E028	State On Loop Not Ready	Internal error caused by attempting to execute OnLoop of a State with an unknown load error.
SL_E029	State On Exit Not Ready	Internal error caused by attempting to execute OnExit of a State with an unknown load error.
SL_E030	Subroutine Load Failed Because Image Invalid	Attempted to load Subroutine from an invalid SLBIN.
SL_E031	Attempt To Load Invalid Subroutine Number	Attempted to load a Subroutine index which is out of range.
SL_E032	Called Inactive Subroutine	Attempted to load a Subroutine index which is not included in current SLBIN.
SL_E033	Subroutine Table Corrupt	SLBIN contains a corrupted Subroutine Table.
SL_E034	Subroutine Not Loaded	Error when attempting to resume from break point within subroutine.
SL_E035	Subroutine Not Ready	Internal error caused by attempting to execute OnExec of a Subroutine with an unknown load error.
SL_E036	Illegal Goto From Subroutine	Illegal Goto executed within OnExec of a Subroutine.
SL_E037	Illegal Call From Subroutine	Illegal Call executed within OnExec of a Subroutine.
SL_E038	Subroutine Unexpected Error	Unexpected internal state indicating an execution error while loading Subroutine.
SL_E039	Unexpected Stack Over Flow	Unexpected stack over flow when loading program counter.
SL_E040	Unexpected Stack Under Flow	Unexpected stack under flow when loading program counter.
SL_E041	Unexpected Stack Re Entry	Unexpected stack initialization from subroutine call while loading program counter.
SL_E042	Unexpected Stack Exit	Unexpected stack initialization from subroutine call return while loading program counter.
SL_E043	Unexpected Stack Depth	Unexpected stack depth while loading program counter.
SL_E044	Instruction Block Illegal Values	Illegal Instruction Block values are not self-consistent.
SL_E045	Instruction Block Byte Size Not Word Aligned	Error in instruction block record. Byte count of execution block not word aligned.
SL_E046	Instruction Cache Index Invalid	Error in instruction block record. Byte count of execution block valid size.
SL_E047	Instruction Cache Illegal Image Index	Error in instruction block record. Execution block outside of valid SLBIN range.
SL_E048	Unexpected Instruction Block Error	Unexpected error in instruction block record load.
SL_E049	Instruction Block Record Corrupt	Retrieved instruction block failed record CRC.
SL_E050	Instruction Block Data Corrupt	Retrieved instruction block failed data block CRC.
SL_E051	Stack State End Error	Non-zero stack depth error encountered on End OpCode completing state execution.
SL_E052	Stack State Change Error	Non-zero stack depth error encountered on Change State OpCode completing state execution.

Code	Name	Description
SL_E053	Stack Subroutine Re Entry Error	Illegal attempt to initialize stack for subroutine entry. Stack already serving subroutine.
SL_E054	Stack Subroutine Exit Error	Illegal attempt to exit stack subroutine service. Stack not serving subroutine.
SL_E055	Stack Subroutine Exit Depth Error	Incorrect stack depth when leaving subroutine.
SL_E056	Program Counter Left Legal Space	Illegal attempt of Program Counter to leave designated execution block.
SL_E057	Instruction Word Invalid	Encountered instruction word that failed validation.
SL_E058	Op Code Unknown Error	Encountered unknown OpCode in a valid instruction word.
SL_E059	Unexpected State Execution Error	Unexpected error during state execution.
SL_E060	Unexpected Subroutine Execution Error	Unexpected error during subroutine execution.
SL_E061	Invalid Op Code Error	Invalid OpCode error encountered.
SL_E062	Op Code Drop Stack Under Flow	Stack under flow occurred while executing Drop OpCode.
SL_E063	Op Code Dup Stack Under Flow	Stack under flow occurred while executing Dup OpCode.
SL_E064	Op Code Dup Stack Over Flow	Stack over flow occurred while executing Dup OpCode.
SL_E065	Op Code Push Float Error Loading Operand	Error loading float operand while executing Push_Float OpCode.
SL_E066	Op Code Push Float Invalid Float Operand	Invalid Float Operand while executing Push_Float OpCode.
SL_E067	Op Code Push Float Stack Over Flow	Stack over flow while executing Push_Float OpCode.
SL_E068	Op Code Push Integer Error Loading Operand	Error loading Integer operand while executing Push_Integer OpCode.
SL_E069	Op Code Push Integer Invalid Integer Operand	Invalid Integer Operand while executing Push_Integer OpCode.
SL_E070	Op Code Push Integer Stack Over Flow	Stack over flow while executing Push_Integer OpCode.
SL_E071	Op Code Push Specifier Error Loading Operand	Error loading Specifier operand while executing Push_Specifier OpCode.
SL_E072	Op Code Push Specifier Invalid Specifier Operand	Invalid Specifier Operand while executing Push_Specifier OpCode.
SL_E073	Op Code Push Specifier Stack Over Flow	Stack over flow while executing Push_Specifier OpCode.
SL_E074	Op Code Push Address Error Loading Operand	Error loading Address operand while executing Push_Address OpCode.
SL_E075	Op Code Push Address Invalid Address Operand	Invalid Address Operand while executing Push_Address OpCode.
SL_E076	Op Code Push Address Stack Over Flow	Stack over flow while executing Push_Address OpCode.
SL_E077	Op Code Store Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for Store OpCode.

Code	Name	Description
SL_E078	Op Code Store Argument AInvalid Specifier	Argument A on stack found to be non-valid specifier while executing Store OpCode.
SL_E079	Op Code Store Argument BStack Under Flow	Stack under flow occurred while retrieving literal Argument B for Store OpCode.
SL_E080	Op Code Store Argument BInvalid Literal	Argument B on stack found to be non-valid literal word while executing Store OpCode.
SL_E081	Op Code Store Invalid RScore	Attempt to store to invalid Resource Specifier code
SL_E082	Op Code Store Specifier Not AProperty	Invalid attempt to store a value to a resource specifier that is not a property.
SL_E083	Op Code Store Specifier Not Writable	Invalid attempt to store to a resource specifier that is read only.
SL_E084	Op Code Store Invalid Asset Index	Internal error while executing Store OpCode returned invalid LM Asset index.
SL_E085	Op Code Store Internal Store Error	Internal error occurred while writing property during the execution of the Store OpCode.
SL_E086	Op Code Store Incompatible Data Type	Internal error while executing Store OpCode returned unexpected error.
SL_E087	Op Code Store Unexpected Error Result	Internal error while executing Store OpCode returned unexpected error.
SL_E088	Op Code Recall Argument AStack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for Recall OpCode.
SL_E089	Op Code Recall Argument AInvalid Specifier	Argument A on stack found to be non-valid specifier while executing Recall OpCode.
SL_E090	Op Code Recall Invalid RScore	Attempt to Recall to invalid Resource Specifier code while executing Recall OpCode.
SL_E091	Op Code Recall Specifier Not AProperty	Invalid attempt to Recall a value to a resource specifier that is not a property.
SL_E092	Op Code Recall Invalid Asset Index	Internal error while executing Recall OpCode returned invalid LM Asset index.
SL_E093	Op Code Recall Internal Recall Error	Internal error occurred while writing property during the execution of the Recall OpCode.
SL_E094	Op Code Recall Incompatible Data Type	Internal error while executing Recall OpCode returned unexpected error.
SL_E095	Op Code Recall Unexpected Error Result	Internal error while executing Recall OpCode returned unexpected error.
SL_E096	Op Code Recall Stack Over Flow	Stack Over Flow error occurred while executing Recall OpCode.
SL_E097	Op Code Recall Unexpected Stack Error	Unexpected Stack error occurred while executing Recall OpCode.
SL_E098	Op Code Invoke Method Argument AStack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for Invoke Method OpCode.
SL_E099	Op Code Invoke Method Argument AInvalid Specifier	Argument A on stack found to be non-valid specifier while executing Invoke Method OpCode.
SL_E100	Op Code Invoke Method Invalid RScore	Attempt to invoke an invalid Resource Specifier code while executing Invoke Method OpCode.

Code	Name	Description
SL_E101	Op Code Invoke Method Specifier Not AMethod	Invalid attempt to invoke a value to a resource specifier that is not a method.
SL_E102	Op Code Invoke Method Invalid Asset Index	Internal error while executing Invoke Method OpCode returned invalid LM Asset index.
SL_E103	Op Code Invoke Method Internal Method Error	Internal error occurred while executing method during the execution of the Invoke Method OpCode.
SL_E104	Op Code Invoke Method Incompatible Data Type	Internal error while executing Invoke Method OpCode returned unexpected error.
SL_E105	Op Code Invoke Method Unexpected Error Result	Internal error while executing Invoke Method OpCode returned unexpected error.
SL_E106	Op Code Invoke Method Stack Over Flow	Stack Over Flow error occurred while executing Invoke Method OpCode.
SL_E107	Op Code Invoke Method Unexpected Stack Error	Unexpected Stack error occurred while executing Invoke Method OpCode.
SL_E108	Op Code Change State Argument AStack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for Change State OpCode.
SL_E109	Op Code Change State Argument AInvalid Specifier	Argument A on stack found to be non-valid specifier while executing Change State OpCode.
SL_E110	Op Code Change State Invalid RScore	Attempt to change to an invalid Resource Specifier code while executing Change State OpCode.
SL_E111	Op Code Change State Specifier Not AState	Invalid attempt change to a resource specifier that is not a state.
SL_E112	Op Code Change State Invalid State Index	Internal error while executing Change State OpCode which contained an invalid State Index.
SL_E113	Op Code Changes State Unexpected Error Result	Internal error while executing Change State OpCode returned unexpected error.
SL_E114	Op Code Call Argument AStack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for Call OpCode.
SL_E115	Op Code Call Argument AInvalid Specifier	Argument A on stack found to be non-valid specifier while executing Call OpCode.
SL_E116	Op Code Call Invalid RScore	Attempt to call an invalid Resource Specifier code while executing Call OpCode.
SL_E117	Op Code Call Specifier Not ASubroutine	Invalid attempt call a resource specifier that is not a Subrutine.
SL_E118	Op Code Call Invalid Subroutine Index	Internal error while executing Call OpCode which contained an invalid Subroutine Index.
SL_E119	Op Code Call Unexpected Error Result	Internal error while executing Call OpCode returned unexpected error.
SL_E120	Op Code Jump Argument AStack Under Flow	Stack under flow occurred while retrieving Address Argument A for Jump OpCode.
SL_E121	Op Code Jump Argument AInvalid Address	Argument A on stack found to be non-valid Address while executing Jump OpCode.
SL_E122	Op Code Jump Illegal Jump	Jump attempted illegal PC offset which would result in leaving instruction block.

Code	Name	Description
SL_E123	Op Code Branch If False Argument AStack Under Flow	Stack under flow occurred while retrieving Address Argument A for BranchIfFalse OpCode.
SL_E124	Op Code Branch If False Argument AInvalid Address	Argument A on stack found to be non-valid Address while executing BranchIfFalse OpCode.
SL_E125	Op Code Branch If False Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for BranchIfFalse OpCode.
SL_E126	Op Code Branch If False Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing BranchIfFalse OpCode.
SL_E127	Op Code Branch If False Illegal Else Jump	BranchIfFalse attempted illegal Else PC offset which would result in leaving instruction block.
SL_E128	Op Code If Then Else Argument AStack Under Flow	Stack under flow occurred while retrieving Address Argument A for IfThenElse OpCode.
SL_E129	Op Code If Then Else Argument AInvalid Address	Argument A on stack found to be non-valid Address while executing IfThenElse OpCode.
SL_E130	Op Code If Then Else Argument BStack Under Flow	Stack under flow occurred while retrieving Address Argument B for IfThenElse OpCode.
SL_E131	Op Code If Then Else Argument BInvalid Address	Argument B on stack found to be non-valid Address while executing IfThenElse OpCode.
SL_E132	Op Code If Then Else Argument CStack Under Flow	Stack under flow occurred while retrieving Literal Argument C for IfThenElse OpCode.
SL_E133	Op Code If Then Else Argument CInvalid Literal	Argument C on stack found to be non-valid Literal while executing IfThenElse OpCode.
SL_E134	Op Code If Then Else Illegal Then Jump	IfThenElse attempted illegal Then PC offset which would result in leaving instruction block.
SL_E135	Op Code If Then Else Illegal Else Jump	IfThenElse attempted illegal Else PC offset which would result in leaving instruction block.
SL_E136	Op Code Logical AND Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Logical AND OpCode.
SL_E137	Op Code Logical AND Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Logical AND OpCode.
SL_E138	Op Code Logical AND Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Logical AND OpCode.
SL_E139	Op Code Logical AND Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Logical AND OpCode.
SL_E140	Op Code Logical AND Stack Over Flow	Stack over flow occurred while pushing result of Logical AND OpCode.
SL_E141	Op Code Logical OR Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Logical OR OpCode.
SL_E142	Op Code Logical OR Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Logical OR OpCode.
SL_E143	Op Code Logical OR Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Logical OR OpCode.
SL_E144	Op Code Logical OR Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Logical OR OpCode.

Code	Name	Description
SL_E145	Op Code Logical OR Stack Over Flow	Stack over flow occurred while pushing result of Logical OR OpCode.
SL_E146	Op Code Equality Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Equality OpCode.
SL_E147	Op Code Equality Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Equality OpCode.
SL_E148	Op Code Equality Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Equality OpCode.
SL_E149	Op Code Equality Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Equality OpCode.
SL_E150	Op Code Equality Stack Over Flow	Stack over flow occurred while pushing result of Equality OpCode.
SL_E151	Op Code Inequality Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Inequality OpCode.
SL_E152	Op Code Inequality Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Inequality OpCode.
SL_E153	Op Code Inequality Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Inequality OpCode.
SL_E154	Op Code Inequality Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Inequality OpCode.
SL_E155	Op Code Inequality Stack Over Flow	Stack over flow occurred while pushing result of Inequality OpCode.
SL_E156	Op Code Greater Than Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Greater Than OpCode.
SL_E157	Op Code Greater Than Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Greater Than OpCode.
SL_E158	Op Code Greater Than Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Greater Than OpCode.
SL_E159	Op Code Greater Than Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Greater Than OpCode.
SL_E160	Op Code Greater Than Stack Over Flow	Stack over flow occurred while pushing result of Greater Than OpCode.
SL_E161	Op Code Less Than Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Less Than OpCode.
SL_E162	Op Code Less Than Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Less Than OpCode.
SL_E163	Op Code Less Than Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Less Than OpCode.
SL_E164	Op Code Less Than Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Less Than OpCode.
SL_E165	Op Code Less Than Stack Over Flow	Stack over flow occurred while pushing result of Less Than OpCode.
SL_E166	Op Code Greater Than Equals Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Greater Than Equals OpCode.

Code	Name	Description
SL_E167	Op Code Greater Than Equals Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Greater Than Equals OpCode.
SL_E168	Op Code Greater Than Equals Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Greater Than Equals OpCode.
SL_E169	Op Code Greater Than Equals Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Greater Than Equals OpCode.
SL_E170	Op Code Greater Than Equals Stack Over Flow	Stack over flow occurred while pushing result of Greater Than Equals OpCode.
SL_E171	Op Code Less Than Equals Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Less Than Equals OpCode.
SL_E172	Op Code Less Than Equals Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Less Than Equals OpCode.
SL_E173	Op Code Less Than Equals Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Less Than Equals OpCode.
SL_E174	Op Code Less Than Equals Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Less Than Equals OpCode.
SL_E175	Op Code Less Than Equals Stack Over Flow	Stack over flow occurred while pushing result of Less Than Equals OpCode.
SL_E176	Op Code Add Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Add OpCode.
SL_E177	Op Code Add Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Add OpCode.
SL_E178	Op Code Add Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Add OpCode.
SL_E179	Op Code Add Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Add OpCode.
SL_E180	Op Code Add Stack Over Flow	Stack over flow occurred while pushing result of Add OpCode.
SL_E181	Op Code Subtract Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Subtract OpCode.
SL_E182	Op Code Subtract Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Subtract OpCode.
SL_E183	Op Code Subtract Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Subtract OpCode.
SL_E184	Op Code Subtract Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Subtract OpCode.
SL_E185	Op Code Subtract Stack Over Flow	Stack over flow occurred while pushing result of Subtract OpCode.
SL_E186	Op Code Multiply Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Multiply OpCode.
SL_E187	Op Code Multiply Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Multiply OpCode.
SL_E188	Op Code Multiply Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Multiply OpCode.
SL_E189	Op Code Multiply Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Multiply OpCode.

Code	Name	Description
SL_E190	Op Code Multiply Stack Over Flow	Stack over flow occurred while pushing result of Multiply OpCode.
SL_E191	Op Code Divide Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Divide OpCode.
SL_E192	Op Code Divide Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Divide OpCode.
SL_E193	Op Code Divide Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Divide OpCode.
SL_E194	Op Code Divide Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Divide OpCode.
SL_E195	Op Code Divide Stack Over Flow	Stack over flow occurred while pushing result of Divide OpCode.
SL_E196	Op Code Modulo Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Modulo OpCode.
SL_E197	Op Code Modulo Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Modulo OpCode.
SL_E198	Op Code Modulo Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Modulo OpCode.
SL_E199	Op Code Modulo Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Modulo OpCode.
SL_E200	Op Code Modulo Stack Over Flow	Stack over flow occurred while pushing result of Modulo OpCode.
SL_E201	Op Code Exponentiation Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Exponentiation OpCode.
SL_E202	Op Code Exponentiation Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Exponentiation OpCode.
SL_E203	Op Code Exponentiation Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for Exponentiation OpCode.
SL_E204	Op Code Exponentiation Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing Exponentiation OpCode.
SL_E205	Op Code Exponentiation Stack Over Flow	Stack over flow occurred while pushing result of Exponentiation OpCode.
SL_E206	Op Code Bitwise AND Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for BitwiseAND OpCode.
SL_E207	Op Code Bitwise AND Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing BitwiseAND OpCode.
SL_E208	Op Code Bitwise AND Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for BitwiseAND OpCode.
SL_E209	Op Code Bitwise AND Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing BitwiseAND OpCode.
SL_E210	Op Code Bitwise AND Stack Over Flow	Stack over flow occurred while pushing result of BitwiseAND OpCode.
SL_E211	Op Code Bitwise OR Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for BitwiseOR OpCode.
SL_E212	Op Code Bitwise OR Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing BitwiseOR OpCode.

Code	Name	Description
SL_E213	Op Code Bitwise OR Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for BitwiseOR OpCode.
SL_E214	Op Code Bitwise OR Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing BitwiseOR OpCode.
SL_E215	Op Code Bitwise OR Stack Over Flow	Stack over flow occurred while pushing result of BitwiseOR OpCode.
SL_E216	Op Code Bitwise XOR Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for BitwiseXOR OpCode.
SL_E217	Op Code Bitwise XOR Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing BitwiseXOR OpCode.
SL_E218	Op Code Bitwise XOR Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for BitwiseXOR OpCode.
SL_E219	Op Code Bitwise XOR Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing BitwiseXOR OpCode.
SL_E220	Op Code Bitwise XOR Stack Over Flow	Stack over flow occurred while pushing result of BitwiseXOR OpCode.
SL_E221	Op Code Shift Left Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for ShiftLeft OpCode.
SL_E222	Op Code Shift Left Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing ShiftLeft OpCode.
SL_E223	Op Code Shift Left Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for ShiftLeft OpCode.
SL_E224	Op Code Shift Left Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing ShiftLeft OpCode.
SL_E225	Op Code Shift Left Stack Over Flow	Stack over flow occurred while pushing result of ShiftLeft OpCode.
SL_E226	Op Code Shift Right Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for ShiftRight OpCode.
SL_E227	Op Code Shift Right Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing ShiftRight OpCode.
SL_E228	Op Code Shift Right Argument BStack Under Flow	Stack under flow occurred while retrieving Literal Argument B for ShiftRight OpCode.
SL_E229	Op Code Shift Right Argument BInvalid Literal	Argument B on stack found to be non-valid Literal while executing ShiftRight OpCode.
SL_E230	Op Code Shift Right Stack Over Flow	Stack over flow occurred while pushing result of ShiftRight OpCode.
SL_E231	Op Code Logical Negation Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Logical Negation OpCode.
SL_E232	Op Code Logical Negation Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Logical Negation OpCode.
SL_E233	Op Code Logical Negation Stack Over Flow	Stack over flow occurred while pushing result of Logical Negation OpCode.
SL_E234	Op Code Bitwise Negation Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Bitwise Negation OpCode.

Code	Name	Description
SL_E235	Op Code Bitwise Negation Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Bitwise Negation OpCode.
SL_E236	Op Code Bitwise Negation Stack Over Flow	Stack over flow occurred while pushing result of Bitwise Negation OpCode.
SL_E237	Op Code Negation Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Negation OpCode.
SL_E238	Op Code Negation Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Negation OpCode.
SL_E239	Op Code Negation Stack Over Flow	Stack over flow occurred while pushing result of Negation OpCode.
SL_E240	Op Code Abs Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Abs OpCode.
SL_E241	Op Code Abs Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Abs OpCode.
SL_E242	Op Code Abs Stack Over Flow	Stack over flow occurred while pushing result of Abs OpCode.
SL_E243	Op Code Ceil Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Ceil OpCode.
SL_E244	Op Code Ceil Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Ceil OpCode.
SL_E245	Op Code Ceil Stack Over Flow	Stack over flow occurred while pushing result of Ceil OpCode.
SL_E246	Op Code Floor Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Floor OpCode.
SL_E247	Op Code Floor Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Floor OpCode.
SL_E248	Op Code Floor Stack Over Flow	Stack over flow occurred while pushing result of Floor OpCode.
SL_E249	Op Code Square Root Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for SquareRoot OpCode.
SL_E250	Op Code Square Root Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing SquareRoot OpCode.
SL_E251	Op Code Square Root Stack Over Flow	Stack over flow occurred while pushing result of SquareRoot OpCode.
SL_E252	Op Code Log Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Log OpCode.
SL_E253	Op Code Log Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Log OpCode.
SL_E254	Op Code Log Stack Over Flow	Stack over flow occurred while pushing result of Log OpCode.
SL_E255	Op Code Log 10 Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Log10 OpCode.
SL_E256	Op Code Log 10 Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Log10 OpCode.
SL_E257	Op Code Log 10 Stack Over Flow	Stack over flow occurred while pushing result of Log10 OpCode.
SL_E258	Op Code Exp Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Exp OpCode.
SL_E259	Op Code Exp Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Exp OpCode.

Code	Name	Description
SL_E260	Op Code Exp Stack Over Flow	Stack over flow occurred while pushing result of Exp OpCode.
SL_E261	Op Code Sine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Sine OpCode.
SL_E262	Op Code Sine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Sine OpCode.
SL_E263	Op Code Sine Stack Over Flow	Stack over flow occurred while pushing result of Sine OpCode.
SL_E264	Op Code Cosine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Cosine OpCode.
SL_E265	Op Code Cosine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Cosine OpCode.
SL_E266	Op Code Cosine Stack Over Flow	Stack over flow occurred while pushing result of Cosine OpCode.
SL_E267	Op Code Tangent Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Tangent OpCode.
SL_E268	Op Code Tangent Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Tangent OpCode.
SL_E269	Op Code Tangent Stack Over Flow	Stack over flow occurred while pushing result of Tangent OpCode.
SL_E270	Op Code Arcsine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Arcsine OpCode.
SL_E271	Op Code Arcsine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Arcsine OpCode.
SL_E272	Op Code Arcsine Stack Over Flow	Stack over flow occurred while pushing result of Arcsine OpCode.
SL_E273	Op Code Arccosine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Arccosine OpCode.
SL_E274	Op Code Arccosine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Arccosine OpCode.
SL_E275	Op Code Arccosine Stack Over Flow	Stack over flow occurred while pushing result of Arccosine OpCode.
SL_E276	Op Code Arctangent Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for Arctangent OpCode.
SL_E277	Op Code Arctangent Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing Arctangent OpCode.
SL_E278	Op Code Arctangent Stack Over Flow	Stack over flow occurred while pushing result of Arctangent OpCode.
SL_E279	Op Code Hyperbolic Sine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for HyperbolicSine OpCode.
SL_E280	Op Code Hyperbolic Sine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing HyperbolicSine OpCode.
SL_E281	Op Code Hyperbolic Sine Stack Over Flow	Stack over flow occurred while pushing result of HyperbolicSine OpCode.
SL_E282	Op Code Hyperbolic Cosine Argument AStack Under Flow	Stack under flow occurred while retrieving Literal Argument A for HyperbolicCosine OpCode.
SL_E283	Op Code Hyperbolic Cosine Argument AInvalid Literal	Argument A on stack found to be non-valid Literal while executing HyperbolicCosine OpCode.

Code	Name	Description
SL_E284	Op Code Hyperbolic Cosine Stack Over Flow	Stack over flow occurred while pushing result of HyperbolicCosine OpCode.
SL_E285	Op Code Hyperbolic Tangent Argument A Stack Under Flow	Stack under flow occurred while retrieving Literal Argument A for HyperbolicTangent OpCode.
SL_E286	Op Code Hyperbolic Tangent Argument A Invalid Literal	Argument A on stack found to be non-valid Literal while executing HyperbolicTangent OpCode.
SL_E287	Op Code Hyperbolic Tangent Stack Over Flow	Stack over flow occurred while pushing result of HyperbolicTangent OpCode.
SL_E288	Op Code Pre Inc Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for PreInc OpCode.
SL_E289	Op Code Pre Inc Argument A Invalid Specifier	Argument A on stack found to be non-valid Specifier while executing PreInc OpCode.
SL_E290	Op Code Pre Inc Stack Over Flow	Stack over flow occurred while pushing result of PreInc OpCode.
SL_E291	Op Code Post Inc Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for PostInc OpCode.
SL_E292	Op Code Post Inc Argument A Invalid Specifier	Argument A on stack found to be non-valid Specifier while executing PostInc OpCode.
SL_E293	Op Code Post Inc Stack Over Flow	Stack over flow occurred while pushing result of PostInc OpCode.
SL_E294	Op Code Pre Dec Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for PreDec OpCode.
SL_E295	Op Code Pre Dec Argument A Invalid Specifier	Argument A on stack found to be non-valid Specifier while executing PreDec OpCode.
SL_E296	Op Code Pre Dec Stack Over Flow	Stack over flow occurred while pushing result of PreDec OpCode.
SL_E297	Op Code Post Dec Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for PostDec OpCode.
SL_E298	Op Code Post Dec Argument A Invalid Specifier	Argument A on stack found to be non-valid Specifier while executing PostDec OpCode.
SL_E299	Op Code Post Dec Stack Over Flow	Stack over flow occurred while pushing result of PostDec OpCode.
SL_E300	Op Code Rand Stack Over Flow	Stack over flow occurred while pushing result of Rand OpCode.
SL_E301	Op Code Push Constant Error Loading Operand	Error loading Integer operand while executing Push_Constant OpCode.
SL_E302	Op Code Push Constant Invalid Constant Index Operand	Invalid Integer Constant Index Operand while executing Push_Constant OpCode.
SL_E303	Op Code Push Constant Stack Over Flow	Stack over flow occurred while pushing result of Push_Constant OpCode.
SL_E304	Op Code Store Dup Argument A Stack Under Flow	Stack under flow occurred while retrieving Specifier Argument A for StoreDup OpCode.
SL_E305	Op Code Store Dup Argument A Invalid Specifier	Argument A on stack found to be non-valid specifier while executing StoreDup OpCode.
SL_E306	Op Code Store Dup Argument B Stack Under Flow	Stack under flow occurred while retrieving literal Argument B for StoreDup OpCode.

Code	Name	Description
SL_E307	Op Code Store Dup Argument BInvalid Literal	Argument B on stack found to be non-valid literal word while executing StoreDup OpCode.
SL_E308	Op Code Store Dup Invalid RScore	Attempt to StoreDup to invalid Resource Specifier code
SL_E309	Op Code Store Dup Specifier Not AProperty	Invalid attempt to StoreDup a value to a resource specifier that is not a property.
SL_E310	Op Code Store Dup Specifier Not Writable	Invalid attempt to StoreDup to a resource specifier that is read only.
SL_E311	Op Code Store Dup Invalid Asset Index	Internal error while executing StoreDup OpCode returned invalid LM Asset index.
SL_E312	Op Code Store Dup Internal Store Dup Error	Internal error occurred while writing property during the execution of the StoreDup OpCode.
SL_E313	Op Code Store Dup Incompatible Data Type	Internal error while executing StoreDup OpCode returned unexpected error.
SL_E314	Op Code Store Dup Unexpected Error Result	Internal error while executing StoreDup OpCode returned unexpected error.
SL_E315	Op Code Store Dup Stack Over Flow	Stack over flow occurred while pushing duplicate argument within StoreDup OpCode.
SL_E316	Op Code Swap Argument AStack Under Flow	Stack under flow occurred while retrieving Argument A for Swap OpCode.
SL_E317	Op Code Swap Argument BStack Under Flow	Stack under flow occurred while retrieving Argument B for Swap OpCode.
SL_E318	Op Code Swap Argument BStack Over Flow	Stack over flow occurred while pushing Argument B within Swap OpCode.
SL_E319	Op Code Swap Argument AStack Over Flow	Stack over flow occurred while pushing Argument A within Swap OpCode.
SL_E320	Op Code Cast BOOL Argument AStack Under Flow	Stack under flow occurred while retrieving Argument A for Cast Bool OpCode.
SL_E321	Op Code Cast BOOL Incompatible Data Type	Attempting to cast Incompatible Data Type for Cast Bool OpCode.
SL_E322	Op Code Cast BOOL Argument AStack Over Flow	Stack over flow occurred while pushing Argument A within Cast Bool OpCode.
SL_E323	Op Code Cast UINT Argument AStack Under Flow	Stack under flow occurred while retrieving Argument A for Cast UINT OpCode.
SL_E324	Op Code Cast UINT Incompatible Data Type	Attempting to cast Incompatible Data Type for Cast UINT OpCode.
SL_E325	Op Code Cast UINT Argument AStack Over Flow	Stack over flow occurred while pushing Argument A within Cast UINT OpCode.

WARRANTY - LIMITATION OF LIABILITY: Seller warrants only title to the products, software, supplies and materials and that, except as to software, the same are free from defects in workmanship and materials for a period of one (1) year from the date of delivery. Seller does not warranty that software is free from error or that software will run in an uninterrupted fashion. Seller provides all software "as is." THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, OF MERCHANTABILITY, FITNESS, OR OTHERWISE WHICH EXTEND BEYOND THOSE STATED IN THE IMMEDIATELY PRECEDING SENTENCE. Seller's liability and Buyer's exclusive remedy in any case of action (whether in contract, tort, breach of warranty or otherwise) arising out of the sale or use of any products, software, supplies, or materials is expressly limited to the replacement of such products, software, supplies, or materials on their return to Seller or, at Seller's option, to the allowance to the customer of credit for the cost of such items. In no event shall Seller be liable for special, incidental, indirect, punitive or consequential damages. Seller does not warrant in any way products, software, supplies and materials not manufactured by Seller, and such will be sold only with the warranties that are given by the manufacturer thereof. Seller will pass only through to its purchaser of such items the warranty granted to it by the manufacturer.

sensiaglobal.com

Add intelligent action to your oil & gas solutions

© Sensia LLC 2021. All rights reserved.

* Mark of Sensia. Other company, product, and service names are the properties of their respective owners.

