Scanner Logic Programmer Manual

Language Reference

Manual No. 50331442, Rev. 01

*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 50331442, Rev. 01
January 2018

Contents

Part I—Introductory Information 19

1 Introduction 19
2 Who Should Read This Document? 20
3 Organization of This Document 21
4 Conventions Used in This Guide 22
5 Parameter Value Type Codes 23
6 See Also 25

Part II—Logic Script Language Elements 27

7 Introduction 27
8 Comments 28
8.1 General Description 28
9 Literal Values 29
9.1 General Description 29
9.2 Numeric Literals 29
9.3 Boolean Literal 29
9.4 String Literal 29
10 Identifiers 31
10.1 General Description 31
11 Keywords 32
11.1 General Description 32
11.2 Keyword Categories 32
12 Data Types 37
12.1 General Description 37
12.2 Primitive Data Types 37
12.3 Type Conversion and Type Casting 38
12.4 Run Time Numeric Bounds Handling 40
12.5 Object Data Types 40
13 Constants 43
13.1 General Description 43
13.2 Remarks 43
14 Variables 44
14.1 General Description 44
14.2 Remarks 44
15 Operators 45

15.1 General Description 45
15.2 Operator Categories 45
15.3 Operator Precedence....50
15.4 Associativity....51
15.5 Adding Parentheses 53
16 Expressions....54
16.1 General Description 54
16.2 Expression Values....54
16.3 Operator precedence and associativity 55
16.4 Literals and simple names 55
16.5 Assignment expressions....55
16.6 Invocation expressions 56
16.7 Remarks 56
17 Statements 57
17.1 General Description 57
17.2 Types of Statements 58
17.3 Assignment Statements....58
17.4 Increment and Decrement Statements 59
17.5 Invocation Statements....59
17.6 Selection Statements 60
17.7 Jump Statements....61
17.8 Embedded Statements 63
17.9 Nested Statement Blocks....64
17.10 Parameter Assignment Statements 64
18 Preprocessor Directives....65
18.1 Region and Endregion Directives 65
18.2 Pragma directive 66
Part III—Logic Script Objects....67
19 Program Object 67
19.1 General Description 67
19.2 Program Information Region....68
19.3 Program Declarations Region....68
19.4 System Declarations Region 69
19.5 Program Code Region 69
19.6 Subroutines Region 69
20 Program Information Object 71
20.1 General Description 71

20.2 Declaring proginfo 71
20.3 Declaration Parameters 71
21 Logic Script Resource Objects 73
21.1 General Description 73
21.2 Resource Object Types 74
21.3 Usage Notes 74
22 Register Input Resource Object 76
22.1 General Description 76
22.2 Required S3100 Device Configuration 76
22.3 Declaring RegisterInputResource Objects 76
22.4 Declaration Parameters 78
22.5 Properties 78
22.6 Methods 78
22.7 Usage 79
23 Digital Input Resource Object 80
23.1 General Description 80
23.2 Required S3100 Device Configuration 80
23.3 Declaring DigitalInputResource Objects 80
23.4 Declaration Parameters 81
23.5 Properties 81
23.6 Usage 82
24 Analog PID Controller Resource Object 83
24.1 General Description 83
24.2 Required S3100 Device Configuration 83
24.3 Declaring AnalogPIDControllerResource Objects 83
24.4 Declaration Parameters 85
24.5 Properties 87
24.6 Methods 88
24.7 Usage 88
24.7.1 Simple PID 88
24.7.2 Constraint Override PID 90
25 Digital PID Controller Resource Object 93
25.1 General Description 93
25.2 Required S3100 Device Configuration 93
25.3 Declaring DigitalPIDControllerResource Objects 93
25.4 Declaration Parameters 95
25.5 Properties 97

25.6 Methods 98
25.7 Usage 98
25.7.1 Simple PID 98
25.7.2 Constraint Override PID 100
26 Digital Output Resource Object 103
26.1 General Description 103
26.2 Required S3100 Device Configuration 103
26.3 Declaring DigitalOutputResource Objects 103
26.4 Declaration Parameters 104
26.5 Properties 105
26.6 Methods 106
26.7 Usage 107
27 Alarm Resource Object 108
27.1 General Description 108
27.2 Required S3100 Device Configuration 108
27.3 Declaring AlarmResource Objects 108
27.4 Declaration Parameters 109
27.5 Properties 110
27.6 Methods 110
27.7 Usage 111
28 Timer Resource Object 112
28.1 General Description 112
28.2 Required S3100 Device Configuration 112
28.3 Declaring TimerResource Objects 112
28.4 Declaration Parameters 113
28.5 Properties 113
28.6 Methods 114
28.7 Usage 114
29 Logic Script Register Objects 116
29.1 General Description 116
29.2 Register Object Types 117
30 Configuration Register Object 119
30.1 General Description 119
30.2 Declaring ConfigurationRegister Objects 119
30.3 Declaration Parameters 121
30.4 Properties 121
30.5 Methods 122

30.6 Usage 122
31 Maintenance Register Object 124
31.1 General Description 124
31.2 Declaring MaintenanceRegister Objects....124
31.3 Declaration Parameters 126
31.4 Properties 126
31.5 Methods 127
31.6 Usage 127
32 Holding Register Object 129
32.1 General Description 129
32.2 Declaring HoldingRegister Objects....130
32.3 Declaration Parameters 131
32.4 Properties 131
32.5 Methods 132
32.6 Usage 132
33 Accumulation Register Object 134
33.1 General Description 134
33.2 Declaring AccumulationRegister Objects....135
33.3 Declaration Parameters 136
33.4 Properties 137
33.5 Methods 137
33.6 Usage 138
34 Working Register Object 140
34.1 General Description 140
34.2 Declaring working registers 140
34.3 Declaration Parameters 141
34.4 Properties 141
34.5 Methods 141
34.6 Usage 141
35 Task Object 144
35.1 General Description 144
35.2 Declaring Tasks 144
35.3 Properties 145
35.4 Methods 145
35.5 Usage 145
36 State Object 147
36.1 General description 147

36.2 Declaring States 147
36.3 Properties 147
36.4 Methods 148
36.5 Modifiers 148
36.6 Usage 148
37 System State Objects 150
38 Abort State Object 151
38.1 General description 151
38.2 Declaring abortState 151
38.3 Entering abortState 151
38.4 Usage 152
39 Fail State Object 154
39.1 General Description 154
39.2 Declaring failState 154
39.3 Entering failState 154
39.4 Usage 155
40 Subroutines 157
40.1 General Description 157
40.2 Declaring Subroutines 157
40.3 Usage 157
41 System Objects 159
41.1 General Description 159
41.2 System Object Types 159
42 Real Time System Object 160
42.1 General Description 160
42.2 Properties 160
42.3 Usage 160
43 Flow Run System Object 162
43.1 General Description 162
43.2 Required S3100 Device Configuration 162
43.3 Properties 162
43.4 Methods 163
43.5 Usage 163
44 Flow Archive System Object 165
44.1 General Description 165
44.2 Required S3100 Device Configuration 165

44.3 Methods 165
44.4 Usage 166
45 Triggered Archive System Object 167
45.1 General Description 167
45.2 Required S3100 Device Configuration 167
45.3 Properties 167
45.4 Methods 168
45.5 Usage 168
46 User Event Record System Object 170
46.1 General Description 170
46.2 Properties 170
46.3 Methods 170
46.4 Usage 170
47 Printed Ticket System Object 172
47.1 General Description 172
47.2 Methods 172
48 Display System Object 173
48.1 General Description 173
48.2 Required S3100 Device Configuration 173
48.3 Properties 173
48.4 Methods 174
48.5 Usage 174
49 Math Object 176
49.1 General Description 176
49.2 Math Constants 176
49.3 Math Library Functions 176
49.3.1 Math Trigonometric Functions 178
50 User HMI Field Object 179
50.1 General Description 179
50.2 Declaring HmiFields Objects 180
50.3 Declaration Parameters 181
50.4 Properties 181
50.5 Usage 181
Part IV—Appendix A 183
51 Scanner 3100 Unit Categories 183
51.1 General Description 183

51.2 Measurement Categories 183
51.3 Unit Types 185
51.3.1 None 185
51.3.2 Volume 185
51.3.3 Static Pressure (Absolute) 186
51.3.4 Static Pressure (Gauge) 186
51.3.5 fferential Pressure 187
51.3.6 Temperature 187
51.3.7 Mass 188
51.3.8 Energy 188
51.3.9 Voltage 189
51.3.10 Molar Mass 189
51.3.11 Length 189
51.3.12 Frequency 189
51.3.13 Resistance 190
51.3.14 Current 190
51.3.15 Time 190
51.3.16 Percent 190
51.3.17 Pulse 191
51.3.18 Viscosity 191
51.3.19 Mole 191
51.3.20 Relative Density 191
51.3.21 Fraction 192
51.3.22 System Ticks 192
51.3.23 Real Date 192
51.3.24 Real Time 192
51.3.25 Unity 192
51.3.26 Power 193
51.3.27 Charge 193
51.4 Rate Scalar 193

TABLES

Table 3.4-1. Syntax conventions....22
Table 3.4-1. Value type codes....23
Table 3.4-1. Other documents....25
Table 11.2-1. Program Structure Keywords....32
Table 11.2-2. Program Flow Control Keywords....33
Table 11.2-3. Data Type Keywords....33
Table 11.2-4. Literal Value Keywords....33
Table 11.2-5. Program Information Declaration Keywords....34
Table 11.2-6. Resource and Registers Declaration Keywords....34
Table 11.2-7. Resource and Registers Parameter Keywords....35
Table 11.2-8. Preprocessor Directive Keywords....36
Table 12.2-1. Primitive Data Types used in Scanner Logic Script....37
Table 12.3-1. Supported Type Conversions....39
Table 12.5-1. Object Types used in Scanner Logic Script....40
Table 13.1-1. Math Constants....43
Table 15.2-1. Primary Operators....45
Table 15.2-2. Unary Operators....46
Table 15.2-3. Arithmetic Operators....47
Table 15.2-4. Relational Operators....47
Table 15.2-5. Equality Operators....48
Table 15.2-6. Conditional AND Operator....49
Table 15.2-7. Conditional OR Operator....49
Table 15.2-8. Assignment and Compound Assignment Operators....49
Table 15.3-1. Operator precedence from highest to lowest....50
Table 15.4-1. Order of evaluation of operators and operands....52
Table 15.5-1. Order of evaluation of operators and operands....53
Table 17.2-1. Types of statements....58
Table 17.7-1. List of program control keywords....61
Table 20.3-1. Program Information object declaration parameters....71
Table 21.2-1. Types of resource objects....74
Table 22.2-1. `RegisterInputResource` object required S3100 device configuration....76
Table 22.4-1. `RegisterInputResource` object declaration parameters....78
Table 22.5-1. `RegisterInputResource` object properties....78
Table 22.6-1. `RegisterInputResource` object methods....78
Table 23.2-1. `DigitalInputResource` object required S3100 device configuration....80
Table 23.4-1. `DigitalInputResource` object declaration parameters....81
Table 23.5-1. `DigitalInputResource` object properties....81
Table 24.2-1. `AnalogPIDControllerResource` object required S3100 device configuration 83
Table 24.4-1. `AnalogPIDControllerResource` object declaration parameters....85

Table 24.5-1. `AnalogPIDControllerResource` object properties....87
Table 24.6-1. `AnalogPIDControllerResource` object methods....88
Table 25.2-1. DigitalPIDControllerResource object required S3100 device configuration....93
Table 25.4-1. `DigitalPIDControllerResource` object declaration parameters....95
Table 25.5-1. `DigitalPIDControllerResource` object properties....97
Table 25.6-1. `DigitalPIDControllerResource` object methods....98
Table 26.2-1. DigitalOutputResource object required S3100 device configuration....103
Table 26.4-1. `DigitalOutputResource` object declaration parameters....104
Table 26.5-1. `DigitalOutputResource` object properties....105
Table 26.6-1. `DigitalOutputResource` object methods....106
Table 27.4-1. `AlarmResource` object declaration parameters....109
Table 27.5-1. `AlarmResource` object properties....110
Table 27.6-1. `AlarmResource` object methods....110
Table 28.4-1. `TimerResource` object declaration parameters....113
Table 28.5-1. `TimerResource` object properties....113
Table 28.6-1. `TimerResource` object methods....114
Table 29.2-1. Types of register objects....117
Table 30.3-1. `ConfigurationRegister` object declaration parameters....121
Table 30.4-1. `ConfigurationRegister` object properties....121
Table 30.5-1. `ConfigurationRegister` object methods....122
Table 31.3-1. `MaintenanceRegister` object declaration parameters....126
Table 31.4-1. `MaintenanceRegister` object properties....126
Table 31.5-1. `MaintenanceRegister` object methods....127
Table 32.3-1. `HoldingRegister` object declaration parameters....131
Table 32.4-1. `HoldingRegister` object properties....131
Table 32.5-1. `HoldingRegister` object methods....132
Table 33.1-1. `AccumulationRegister` object published values....135
Table 33.3-1. `AccumulationRegister` object declaration parameters....136
Table 33.4-1. `AccumulationRegister` object properties....137
Table 33.5-1. `AccumulationRegister` object methods....137
Table 34.3-1. `WorkingRegister` object declaration parameters....141
Table 34.4-1. `WorkingRegister` object properties....141
Table 34.5-1. `WorkingRegister` object methods....141
Table 35.3-1. `Task` object properties....145
Table 35.4-1. `Task` object methods....145
Table 36.3-1. `State` object properties....147
Table 36.4-1. `State` object methods....148
Table 36.5-1. `State` object modifiers....148
Table 38.3-1. Triggering the `Abort State` in a S3100 device....152
Table 39.3-1. Resource Validation Errors....155
Table 41.2-1. Types of system objects....159

Table 42.2-1. `System_RealTime` object properties....160
Table 43.2-1. `System_FlowRun` object required S3100 device configuration....162
Table 43.3-1. `System_FlowRun` object properties....162
Table 43.4-1. `System_FlowRun` object methods....163
Table 44.2-1. `System_FlowArchive` object required S3100 device configuration....165
Table 44.3-1. `System_FlowArchive` object methods....165
Table 45.2-1. `System_Display` object required S3100 device configuration....167
Table 45.3-1. `System_TriggeredArchive` object properties....167
Table 45.4-1. `System_TriggeredArchive` object methods....168
Table 46.2-1. `System_UserEventRecord` object properties....170
Table 46.3-1. `System_UserEventRecord` object methods....170
Table 47.2-1. `System_PrintedTicket` object methods....172
Table 48.2-1. `System_Display` object required S3100 device configuration....173
Table 48.3-1. `System_Display` object properties....173
Table 48.4-1. `System_Display` object methods....174
Table 49.2-1. `System_Math` Object Constants....176
Table 49.3-1. `System_Math` Object Library Functions....177
Table 49.3-2. `System_Math` Object Trigonometric Functions....178
Table 50.3-1. `HmiFields` object declaration parameters....181
Table 50.4-1. `HmiFields` object properties....181
Table 51.2-1. Measurement categories....183
Table 51.3-1. Units for Unit Type None (Index 0)....185
Table 51.3-2. Units for Unit Type Volume (Index 1)....185
Table 51.3-3. Units for Unit Type Static Pressure (Index 2)....186
Table 51.3-4. Units for Unit Type Static Pressure (Gauge) (Index 3)....186
Table 51.3-5. Units for Unit Type Differential Pressure (Index 4)....187
Table 51.3-6. Units for Unit Type Temperature (Index 5)....187
Table 51.3-7. Units for Unit Type Mass (Index 6)....188
Table 51.3-8. Units for Unit Type Energy (Index 7)....188
Table 51.3-9. Units for Unit Type Voltage (Index 8)....189
Table 51.3-10. Units for Unit Type Molar Mass (Index 9)....189
Table 51.3-11. Units for Unit Type Length (Index 10)....189
Table 51.3-12. Units for Unit Type Frequency (Index 11)....189
Table 51.3-13. Units for Unit Type Resistance (Index 12)....190
Table 51.3-14. Units for Unit Type Current (Index 13)....190
Table 51.3-15. Units for Unit Type Time (Index 14)....190
Table 51.3-16. Units for Unit Type Percent (Index 15)....190
Table 51.3-17. Units for Unit Type Pulse (Index 16)....191
Table 51.3-18. Units for Unit Type Viscosity (Index 17)....191
Table 51.3-19. Units for Unit Type Mole (Index 18)....191
Table 51.3-20. Units for Unit Type Relative Density (Index 19)....191

Table 51.3-21. Units for Unit Type Fraction (Index 20).....192
Table 51.3-22. Units for Unit Type System Ticks (Index 21).....192
Table 51.3-23. Units for Unit Type Real Date (Index 22).....192
Table 51.3-24. Units for Unit Type Real Time (Index 23).....192
Table 51.3-25. Units for Unit Type Unity (Index 24).....192
Table 51.3-26. Units for Unit Type Power (Index 25).....193
Table 51.3-27. Units for Unit Type Charge (Index 26).....193
Table 51.4-1. Units for Rate Scalar.....193

EXAMPLES

Example 8.1-1. Block comment 28
Example 8.1-2. Single line comments 28
Example 8.1-3. Nested comments 28
Example 9.2-1. Numeric literals 29
Example 9.4-1. String literal 29
Example 9.4-2. Invalid string literal 30
Example 10.1-1. Valid identifier characters 31
Example 10.1-2. Valid identifiers 31
Example 10.1-3. Not valid identifiers 31
Example 11.1-1. Characters used in keywords 32
Example 12.3-1. Implicit type conversion examples 38
Example 12.3-2. Explicit type conversion (type casting) examples 39
Example 15.2-1. Unary operator example 45
Example 15.2-2. Binary operators example 45
Example 15.3-1. Operator precedence – multiplication has higher precedence than addition 50
Example 15.4-1. Assignment operator is right-associative 52
Example 16.1-1. Examples of expressions 54
Example 16.4-1. Literal expression examples 55
Example 16.5-1. Assignment expression examples 56
Example 16.6-1. Invocation expression examples 56
Example 17.1-1. Some examples of statements 57
Example 17.3-1. Example assignment statement – resource object property 59
Example 17.3-2. Example assignment statement – register object property 59
Example 17.3-3. Example assignment statement – system object property 59
Example 17.4-1. Example increment/decrement statements 59
Example 17.5-1. Example invocation statements – subroutine calls 60
Example 17.5-2. Example invocation statements – object method calls 60
Example 17.6-1. Selection statements 61
Example 17.7-1. Example jump statements 62
Example 17.8-1. Embedded statements – single-line and statement block 63
Example 17.9-1. Nested statement blocks 64
Example 17.10-1. Example parameter assignment statements 64
Example 18.1-1. Example region directives 65
Example 19.1-1. Overall structure of the `program` object 67
Example 20.2-1. A `proginfo` declaration with parameter assignment statements 71
Example 21.1-1. A `registerinputs` resource declaration group with two `RegisterInputResource` objects 73
Example 21.3-1. Default Properties of Resource Objects 75

Example 22.3-1. `RegisterInputResource` object declaration example 77
Example 22.7-1 `RegisterInputResource` object usage 79
Example 23.3-1. `DigitalInputResource` object declaration example 81
Example 23.6-1. `DigitalInputResource` object usage 82
Example 24.3-1. `AnalogPIDControllerResource` object declaration example 84
Example 24.7-1. `AnalogPIDControllerResource` object usage as a simple PID 89
Example 24.7-2. `AnalogPIDControllerResource` object usage as a PID with a constraint override controller 90
Example 25.3-1. `DigitalPIDControllerResource` object declaration as a simple PID 94
Example 25.3-2. `DigitalPIDControllerResource` object declaration as a PID with a constraint override controller 94
Example 25.7-1. `DigitalPIDControllerResource` object usage as a simple PID 98
Example 25.7-2. `DigitalPIDControllerResource` object usage as a PID with a constraint override controller 100
Example 26.3-1. `DigitalOutputResource` object declaration example 104
Example 26.7-1. `DigitalOutputResource` object usage example 107
Example 27.3-1. `AlarmResource` object declaration example 109
Example 27.7-1. `AlarmResource` object usage example 111
Example 28.3-1. `TimerResource` object declaration example 112
Example 28.7-1. `TimerResource` object usage example 114
Example 29.1-1. Using Logic Script Register objects 116
Example 30.2-1. `ConfigurationRegister` object declaration example 120
Example 30.6-1. `ConfigurationRegister` object usage example 122
Example 31.2-1. `MaintenanceRegister` object declaration example 125
Example 31.6-1. `MaintenanceRegister` object usage example 127
Example 32.2-1. `HoldingRegister` object declaration example 130
Example 32.6-1. `HoldingRegister` object usage example 132
Example 33.2-1. `AccumulationRegister` object declaration example 136
Example 33.6-1. `AccumulationRegister` object usage example 138
Example 34.2-1. `WorkingRegister` object declaration example 140
Example 34.6-1. `WorkingRegister` object usage example 142
Example 35.2-1. `Task` object declaration example 144
Example 35.5-1. `Task` object usage example 145
Example 36.2-1. `State` object declaration example 147
Example 36.6-1. `State` object usage example 149
Example 36.6-1. System Declarations region 150
Example 38.2-1. `Abort State` object declaration example 151
Example 38.4-1. `Abort State` object usage example 152
Example 39.2-1. `Fail State` object declaration example 154
Example 39.4-1. `Fail State` object usage example 155
Example 40.2-1. `Subroutine` declaration example 157

Example 40.3-1. `Subroutine` usage example....158
Example 42.3-1. `System_RealTime` object usage example....160
Example 43.5-1. `System_FlowRun` object usage example....163
Example 44.4-1. `System_FlowArchive` object usage example....166
Example 45.5-1. `System_TriggeredArchive` object usage example....168
Example 46.4-1. `System_UserEventRecord` object usage example....170
Example 48.5-1. `System_Display` object usage example....174
Example 50.2-1. `HmiFields` object declaration example....180

This page is left blank intentionally.

Part I—Introductory Information

1 Introduction

Scanner* Logic Script is a programmable logic controller (PLC) language created by Cameron Valves & Measurement for use with the Scanner 3100 flow computer. It allows users to produce programmable output values and control signals based on the desired control algorithm. The Scanner 3100 can be configured to subscribe to these output values and control signals to produce hardware output effects. You can create programs—sets of written instructions within a state machine model—to automate control tasks using all available hardware and software parameters of the device.

Scanner Logic Script was designed to be simple to understand and use, yet powerful enough to implement a wide variety of complex control applications. Another design goal was for the Scanner Logic Script execution engine to be conservative in its use of Scanner 3100 processor time and system memory. Programming language features are deliberately limited in the Scanner Logic Script language to achieve the goals of simplicity and reduced system resource usage.

Note: Cameron also provides a complimentary Scanner Logic Integrated Development Environment (IDE) application. Scanner Logic Script allows users to create, edit and compile logic programs, upload binaries to Scanner 3100 targets and perform live debugging operations in a graphical environment. For more information, see Scanner Logic IDE Documentation.

2 Who Should Read This Document?

You should use this document if you write or modify Scanner Logic Script programs, or if you configure Scanner 3100 devices and need to know how logic programs should work within the device.

This document is intended to be a reference manual for programmers who have some prior experience in writing programs or in coding for logic controllers. Scanner Logic Script Language Reference Manual assumes that you are familiar with concepts of programmable logic controllers, and that you are familiar with structured high-level procedural languages (e.g. C/C++/C# type languages) and programming with objects. It is assumed that you know common programming terminology. You should also have some familiarity with the Scanner 3100 product and with its web interface.

This document is not organized to be an instructional guide or primer in writing programs in general nor does it step the reader through constructing, running, debugging, and using Scanner Logic programs.

In general, basic concepts of programming are not explained in this manual. However, concepts that are new or that operate differently in Logic Script than in other languages are explained.

3 Organization of This Document

This guide describes the elements of the Scanner Logic Script language in a series of chapters grouped into parts.

Part I – Introductory Information

The chapters in *Part I—Introductory Information* describe the contents of this document and its organization, plus information about conventions for text font usage in this document and codes for allowable value types.

Part II –Logic Script Language Elements

The chapters in *Part II—Logic Script Language Elements* introduce the basic components of the Scanner Logic Script language. These are the elements that compose the instructions and logic of a program.

Part III – Logic Script Objects

The chapters in *Part III—Logic Script Objects* provide reference information for each of the object types available for us in the Scanner Logic Script Language. Program, Task, and State objects provide the structure and overall flow of the program. Resource and register objects provide means of exchanging input and output with the Scanner 3100 host device. Other objects provide additional information or functionality. The majority of writing a program in Scanner Logic Script is made up of interacting with these objects.

Part IV – Appendix A

Part IV—Appendix A describes the measurement unit categories available in the Scanner 3100. These category and unit selections are used in declaring parameters of various resources and registers.

4 Conventions Used in This Guide

Glossary terms are shown in **boldface** where they are defined.
The following conventions are used in syntax description.

Table 8.1-1. Syntax conventions

Item	Description
`Language element`	Plain computer font indicates an element that you type exactly as shown. If there are special symbols (for example, + or &), you also type them exactly as shown.
`placeholder`	Italic text indicates a placeholder that you replace with an appropriate value.
`[optional]`	Brackets indicate that the enclosed language element or elements are optional.
`(a group)`	Parentheses group elements together. However, the parentheses shown in Function Syntax (Positional Parameters) are part of the syntax.
`a | b | c`	Vertical bars separate elements in a group from which you must choose a single element. The elements are often grouped within parentheses or brackets.

5 Parameter Value Type Codes

Throughout this manual, there will be examples showing the syntax for declaring various types of objects. These declarations may have parameters that need to be declared and initialized with desired values. The allowable values for these parameters are either given as a data type or are represented with value type codes in the reference tables.

The following parameter value type codes are used in this document.

Table 8.1-1. Value type codes

Type Code	Description
<str32>	A string that is maximum 32 characters long. Example: "A string"
<str80>	A string that is maximum 80 characters long. Example: "A string"
<str256>	A string that is maximum 256 characters long. Example: "A string"
<str1024>	A string that is maximum 1024 characters long. Example: "A string"
<date>	A date encoded in the format DD/MM/YYYY as a string. Example: "05/04/2017"
<option>	One of a set of string values that will be specified in the reference information. Example: "nousers"|"adminusers"|"configusers"| "maintusers"|"allusers"
<tagname>	Register Tag Name A descriptive name for the Scanner 3100 register selected from a list of available registers as a string. Example: "Analog 1: Holding: Inst Reading"
<tagcode>	Register Tag Code A structured identifier for the selected Scanner 3100 register as a string. Example: "m32_FC_IN_5_Holding_InstReading"
<category>	Measurement Category A Scanner 3100 measurement category as a string. See *Chapter 51* for more information. Example: "Static Pressure (gauge)"

Type Code	Description
`<unit>`	**Measurement Unit** A Scanner 3100 measurement unit corresponding to the selected measurement category as a string. See *Chapter 51* for more information. Example: `"Pa(g)"`
`<rate>`	**Rate Scalar Unit** A rate scalar unit that is selected from a list of units as a string, if the value is a rate. Example: `"/sec"`
`<reginput>`	**Register Input** The name of one of the `registerinputs` resource objects that have been declared in the program. Example: `AnalogInput1`
`<propname>`	**Property Name** The name of a property specifying both the object name of the user declared object or system object and the property name of the object. Example: `AnalogInput1.Value`

6 See Also

This document is intended to be solely a language reference manual for Scanner Logic Script.

Additional information to support working with Scanner Logic Script can be obtained in these other documents:

Table 8.1-1. Other documents

Type Code	Description
Scanner Logic IDE User Manual	Information on how to use the Scanner Logic Script Integrated Development Environment (IDE) software for code creation, compilation, uploading and real-time debugging.
Scanner 3100 Web Interface User Manual	Information on configuring the Scanner 3100 Flow Computer using its web interface, including how to view information about an installed Scanner Logic program.
Scanner 3100 EFM Hardware User Manual	Detailed information on connecting and configuring the Scanner 3100 Flow Computer inputs and outputs.

This page is left blank intentionally.

Part II—Logic Script Language Elements

7 Introduction

Scanner Logic Script is a high-level, structured, procedural language. It is a domain-specific language designed to allow building logic control programs and is not an all-purpose programming language. The programming language elements available are intentionally limited to being a subset of features normally available in general purpose languages. This helps make Scanner Logic Script easier to learn and use, and ensures that programs cannot be written that may affect the integrity of the primary function of the Scanner 3100 as a metrological device.

8 Comments

8.1 General Description

A **comment** is text that is ignored by Scanner Logic Script when a program is executed. You can use comments to describe what is happening in the program or make other kinds of notes. There are two kinds of comments: block comments and end-of-line comments.

A **block comment** begins with the characters `/*` and ends with the characters `*/`. Block comments must be placed between other statements. That means they can be placed on the same line at the beginning or end of a statement, but cannot be embedded within a simple (one-line) statement.

Example 8.1-1. Block comment

```
/*
This is a block comment
*/
```

A **single line comment** begins with the characters // (two forward slashes) and ends with the end of the line. The comment can exist by itself on a line, or come after program code on a line.

Example 8.1-2. Single line comments

```
//single line comments extend to the end of the line

HoldingReg1.Value = 2; // this is another comment
```

You can nest comments—that is, comments can contain other comments—as in this example.

Example 8.1-3. Nested comments

```
/* Here are some
      //nested comments
*/
```

9 Literal Values

9.1 General Description

A **literal** is a fixed value that evaluates to itself. It is interpreted just as it is written.

9.2 Numeric Literals

A **numeric literal** is a sequence of digits that can include other characters, such as a unary minus sign, dot (decimal point), or "E" (in exponential notation).

In the current version of Scanner Logic Script parser interprets number literals as 32-bit floating point numbers by default. Numbers without decimals are compiled as floating-point numbers into the program code output file. The following are some numeric literals:

Example 9.2-1. Numeric literals

```
-32767

3.1415

1.602E-19
```

9.3 Boolean Literal

Boolean literals are the keywords `true` and `false`.

9.4 String Literal

A **string literal** consists of a series of characters enclosed in a pair of double quote marks, as in the following example:

Example 9.4-1. String literal

```
“A basic string.”
```

Any additional double quote marks used within the enclosing double quotes are invalid. Otherwise, all letter, number, and symbol characters may be used within the enclosing pair of double quote marks. Single quote marks are not supported. Escape sequence characters are not supported.

For example, the following string is invalid:

Example 9.4-2. Invalid string literal

```
“Additional “quotes” are invalid.”
```

Strings are not a data type that can be modified or assigned within the program code of a Scanner Logic program. They are used as parameter values within the Program Information and the Program Resource Declaration regions of a script program.

String literals are limited in their maximum lengths. The number of characters that may be used in a string value depends on the parameter that is being assigned. In the reference tables, value type codes like `<str32>` are used to indicate how long a string can be.

Some parameters that require a string value can only accept certain specific strings. In these cases, the value type code `<options>` is used in the reference tables, and the allowable values are listed.

10 Identifiers

10.1 General Description

An **identifier** is a name used to identify an item in the Scanner Logic Script program. The item could be an object in the program, a property or method of an object, or subroutine.

An identifier must begin with a letter and can be up to 32 characters long. Identifiers can contain any of these characters:

Example 10.1-1. Valid identifier characters

```
ABCDEFGHIJKLOMNOPQRSTUVWXYZabcdefghijklomnopqrstuvwxyz0123456789_
```

Identifiers are case sensitive. For example, the identifiers `myvariable` and `MyVariable` are not equivalent.

The following are examples of valid identifiers:

Example 10.1-2. Valid identifiers

```
Pressure

State01

Flow_Accumulation
```

The following are not valid identifiers:

Example 10.1-3. Not valid identifiers

```
C-

on&ready

411

pass^fail

1stState
```

11 Keywords

11.1 General Description

A **keyword** is a reserved word in the Scanner Logic Script language. Keywords consist of alphanumeric and special characters:

Example 11.1-1. Characters used in keywords

```
ABCDEFGHIJKLOMNOPQRSTUVWXYZabcdefghijklomnopqrstuvwxyz0123456789_#
```

Keywords are globally defined. You should not attempt to reuse keywords in your programs for resource, registers, or other object names.

11.2 Keyword Categories

This chapter lists Scanner Logic Script keywords in categories and provides a brief description for each. The chapter numbers in the Ref. column indicate where to find more information regarding the keyword.

Table 11.2-1. Program Structure Keywords

Keyword	Description	Ref.
`program`	Denotes entire program scope	*Ch. 19*
`proginfo`	Block containing program information	*Ch. 20*
`resource`	Keyword used for declaring resource objects	*Ch. 21*
`registers`	Keyword used for declaring register objects	*Ch. 29*
`task`	Task declaration keyword	*Ch. 35*
`state`	State declaration keyword	*Ch. 36*
`initial`	Modifier that denotes declared state as the initial state for its task	*Ch. 36*
`logged`	Modifier for onEnter and onExit blocks that flags program to log event when they are entered	*Ch. 36*
`onEnter`	Declares onEnter code block for state	*Ch. 36*
`onLoop`	Declares onLoop code block for state	*Ch. 36*
`onExit`	Declares onExit code block for state	*Ch. 36*
`abortState`	System state entered when an abort command is invoked	*Ch. 38*
`failState`	System state entered when runtime error is encountered by the program	*Ch. 39*

Keyword	Description	Ref.
`subroutine`	Subroutine declaration keyword	*Ch. 40*
`hmifields`	Keyword used for declaring hmifields declaration keyword	*Ch. 50*

Table 11.2-2. Program Flow Control Keywords

Keyword	Description	Ref.
`if`	Denotes code block only executed when attached logical expression resolves to `true`	*Ch. 17*
`else`	Denotes code block only executed when logical expression attached to preceding `if` statement resolves to `false`	*Ch. 17*
`continue`	Jumps out of current execution block (`onEnter, onLoop, onExit`) to its closing brace	*Ch. 17*
`return`	Exits out of current `subroutine` block and execution returns to the code following the location of its invocation	*Ch. 17*
`changestate`	Triggers a `state` transition	*Ch. 17*

Table 11.2-3. Data Type Keywords

Keyword	Description	Ref.
`bool`	Boolean type: `true` or `false`	*Table 12.2-1*
`float`	32-bit floating point type	*Table 12.2-1*
`uint`	32-bit unsigned integer type	*Table 12.2-1*
`void`	void value type (for methods that have no return value)	*Table 12.2-1*

Table 11.2-4. Literal Value Keywords

Keyword	Description
`true`	Boolean literal
`false`	Boolean literal

Table 11.2-5. Program Information Declaration Keywords

Keyword	Description	Ref.
ProgramName	The name of the program	*Ch. 20*
ProgramAuthor	The name of the program author	*Ch. 20*
ProgramOwner	The name of the company that owns the program	*Ch. 20*
ProgramVersion	The program version as a floating-point number	*Ch. 20*
ProgramCreationDate	The creation date of the program as a string	*Ch. 20*
OnlineSourceAccess	Declares minimum user security level required to access script source file in web interface	*Ch. 20*
ProgramDescription	Description of program's purpose	*Ch. 20*

Table 11.2-6. Resource and Registers Declaration Keywords

Keyword	Description	Ref.
alarms	Used in declaring alarm objects	*Ch. 27*
analogpidcontrollers	Used in declaring analog PID controller objects	*Ch. 24*
digitalpidcontrollers	Used in declaring digital PID controller objects	*Ch. 25*
digitalinputs	Used in declaring digital input objects	*Ch. 23*
digitaloutputs	Used in declaring digital output objects	*Ch. 26*
registerinputs	Used in declaring register input objects	*Ch. 22*
timers	Used in declaring timer objects	*Ch. 28*
accumulation	Used in declaring accumulation type registers	*Ch. 33*
configuration	Used in declaring configuration type registers	*Ch. 30*
holding	Used in declaring holding type registers	*Ch. 32*
maintenance	Used in declaring maintenance type registers	*Ch. 31*
working	Used in declaring working type registers	*Ch. 34*
user	Used in declaring hmi field objects	*Ch. 50*

Table 11.2-7. Resource and Registers Parameter Keywords

Keyword	Description	Ref.
`category`	Declaration parameter for category of units	*Ch. 21-34*
`description`	Declaration parameter for user string describing object	*Ch. 21-34*
`initial_*`	Declaration parameter that sets the initial value for a property of a resource or registers object. There are many of this kind of parameter, and they are listed in the reference tables for each object type.	*Ch. 21-34*
`pidaction`	Declaration parameter for direct or reverse PID behavior	*Ch. 21-34*
`pidtype`	Declaration parameter for simple PID controller or constraint override PID controller	*Ch. 21-34*
`con_pidaction`	Declaration parameter for direct or reverse PID behavior for constraint override controller	*Ch. 21-34*
`con_processvar`	Declaration parameter for user-selected registerinputs resource to use for constraint controller process variable	*Ch. 21-34*
`processvar`	Declaration parameter for user-selected registerinputs resource to use for process variable	*Ch. 21-34*
`propertyname`	Declaration parameter for user-selected object property to use for hmi field	*Ch. 50*
`rate`	Declaration parameter for rate scalar unit	*Ch. 21-34*
`tagname`	Declaration parameter descriptive string describing the input tag name	*Ch. 21-34*
`tagcode`	Declaration parameter for complete Scanner 3100 register tag descriptor that is used as a source of input	*Ch. 21-34*
`units`	Declaration parameter for unit	*Ch. 21-34*
`webcontrolflags`	Declaration parameter for user control over what properties of a analogpidcontrollers or digitalpidcontrollers can be modified on the web interface	*Ch. 21-34*
`webmodify`	Declaration parameter for specifying if a hmi field can be modified from the web interface.	*Ch. 50*

Table 11.2-8. Preprocessor Directive Keywords

Keyword	Description	Ref.
`#region`	Denotes beginning of collapsible code section	*Ch. 18*
`#endregion`	Denotes ending of collapsible code section	*Ch. 18*
`#pragma`	Denotes a special compiler directive	*Ch. 18*

12 Data Types

12.1 General Description

All values and objects in a program have a data type. The supported data types can be categorized as primitive data types and object data types. Scanner Logic Script supports a reduced set of data types to reduce complexity in the Logic Script implementation and to make writing script programs simpler for users.

12.2 Primitive Data Types

Primitive data types are built-in value types. The following table lists the primitive data types that are used in Scanner Logic Script.

Table 12.2-1. Primitive Data Types used in Scanner Logic Script

Data Type	Represents	Range
`bool`	Boolean value	`true` or `false`
`float`	32-bit floating point type	-3.402823 x 10^{38} to + 3.402823 x 10^{38}
`uint`	32-bit unsigned integer type	0 to 4,294,967,295
`string`	A string of characters	Empty string (“”) to 1024-character strings
`void`	When used as a return type, specifies that the method or subroutine does not return a value	n/a

In the current version of Scanner Logic Script, all number literals are interpreted as floating point numbers even if they have no fractional or decimal portion. Integer number literals are automatically converted to **`float`** values when they are used in expressions and statements. This has ramifications for numeric precision, as 32-bit floating point numbers can only have about 6 significant digits of precision.

In the current version of Scanner Logic Script, strings are only used in providing values for parameter assignment statements in the **`proginfo`**, **`resource`**, and **`registers`** objects. They are not accessed nor manipulated in user program code.

Parameters used in parameter assignment statements for **`proginfo`**, **`resource`**, **`registers`**, and system objects can be **`bool`**, **`float`**, **`uint`**, or **`string`** type. Literal values of the correct type must be used when assigning values to parameters. Arithmetic expressions, type casts, etc. are not supported in parameter assignment statements.

Properties of objects have `bool`, `float`, or `uint` type. Currently, there are no `string` properties implemented.

Arguments are values passed into methods or subroutines at their invocation. In the current version of Scanner Logic Script, only methods of the Math object accept arguments, which are `float` type. Currently, subroutines do not accept arguments.

Return values from methods and subroutines have a specified type. In the current version of Scanner Logic Script, only methods of the Math object return values from method calls, which are `float` type. Currently, subroutines do not return values and must be declared with the `void` return type.

12.3 Type Conversion and Type Casting

Type conversion occurs when a value of one type is used in a situation that requires another type. In some cases, a value of one type can be implicitly converted to another type, for example, from a `uint` value to a `float` value. No special syntax is required. Values of type `uint` can be directly assigned to `float` properties, and the `uint` value will be implicitly converted to a floating-point value by the compiler. When a `uint` value is converted to a `float`, the value is preserved, albeit with potentially some loss of precision beyond about 6 significant digits.

Example 12.3-1. Implicit type conversion examples

```
// assigning a uint 1 to a float type property implicitly converts the value to 1.0
HoldingReg1.Value = 1;

// assigning a uint type property to a float property implicitly converts to float
HoldingReg2.Value = Timer1.Time;
```

In most other cases however, the parser will issue an error warning of type mismatch when you try to use a value of the wrong type in a place where certain types are required. This is because information may be lost in the conversion.

The most common situation of this sort is when assigning a value to a property. If the literal value, or property value, or expression value that you are trying to store to the property does not resolve to type that is compatible with the target property, the value must be type cast to the appropriate type.

Type casting forces a value of one type to be interpreted as a different type, possibly changing the value. For example, if you wish to assign a `float` value to a `uint` property, it must be explicitly cast to `uint` type, and any fractional part of the `float` value will be truncated and lost.

Example 12.3-2. Explicit type conversion (type casting) examples

```
// assigning a float type property to a uint property requires a type cast to uint
Timer1.Time = (uint)HoldingReg1.Value;

// assigning a float 12.34 to a uint type property requires a type cast to uint
Timer1.Time = (uint)12.34;
```

Table 12.3-1 below lists the type conversions supported by Scanner Logic Script and what happens to the values when they are converted.

Table 12.3-1. Supported Type Conversions

From	To	Requirement
`uint`	`uint`	No conversion needed.
`uint`	`float`	Implicit conversion. Precision may be lost if the `uint` value has more than 6 significant digits.
`uint`	`bool`	`(bool)` type cast required. 0 becomes `false`, all other values become `true`.
`float`	`uint`	`(uint)` type cast required. Any fractional part will be truncated, and the result is bounded to be within 0 to 4,294,967,295.
`float`	`float`	No conversion needed.
`float`	`bool`	`(bool)` type cast required. 0.0 becomes `false`, all other values become `true`.
`bool`	`uint`	`(uint)` type cast required. `true` becomes 1, `false` becomes 0
`bool`	`float`	`(float)` type cast required. `true` becomes 1.0, `false` becomes 0.0
`bool`	`bool`	No conversion needed.

12.4 Run Time Numeric Bounds Handling

Since there is limited ability to report run time numeric errors to users, the current implementation of Scanner Logic Script is designed not to halt the program when numeric exceptions occur.

When an operation on an unsigned integer value produces an underflow below 0, the result is bounded to 0. Similarly, when an overflow over 4,294,967,295 of an unsigned integer occurs, the result is bounded to 4,294,967,295. When a floating-point value is divided by 0, the floating-point value NaN results instead of a divide by zero exception occurring. Take these behaviors into account when designing your program.

12.5 Object Data Types

Object data types refer to objects in the program rather than simple values like numbers or strings. There are various types of objects accessible in Scanner Logic programs. They are pre-defined system objects like `System_RealTime`, or they are objects created by the parser to reflect the declarations made in program code for things like `RegisterInputResource` or `Task` objects. There are no user-defined types. You cannot declare or derive your own classes.

When you declare `resource` and `registers` items, Scanner Logic Script creates variables for those types of objects using the name that you specify. When you declare a `task`, Scanner Logic Script creates the object for the `task`, and the name that you declare for the `task` is a variable of the `Task` type (note that the lowercase word `task` is the keyword for declaring `Task` objects, and the uppercase word `Task` is the name of the object type). Similarly, each state that is declared is an instance of the `State` type, and its name is the variable of the `State` type.

Table 12.5-1. Object Types used in Scanner Logic Script

Object Type	Represents	Ref.
`AlarmResource`	Alarms resource object type. Instances declared in `resource alarms` declaration group.	*Ch. 27*
`RegisterInputResource`	Register Inputs resource object type. Instances declared in `resource registerinputs` declaration group.	*Ch. 22*
`DigitalInputResource`	Digital Input resource object type. Instances declared in `resource digitalinputs` declaration group.	*Ch. 23*

Object Type	Represents	Ref.
`AnalogPIDControllerResource`	Analog PID Controller resource object type. Instances declared in **`resource analogpidcontrollers`** declaration group.	*Ch. 24*
`DigitalPIDControllerResource`	Digital PID Controller resource object type. Instances declared in **`resource digitalpidcontrollers`** declaration group.	*Ch. 25*
`DigitalOutputResource`	Digital Output resource object type. Instances declared in **`resource digitaloutputs`** declaration group.	*Ch. 23*
`TimerResource`	Timer resource object type. Instances declared by user in **`resource timers`** declaration group.	*Ch. 28*
`ConfigurationRegister`	Configuration register object type. Instances declared by user in **`registers configuration`** declaration group.	*Ch. 30*
`MaintenanceRegister`	Maintenance register object type. Instances declared by user in **`registers maintenance`** declaration group.	*Ch. 31*
`HoldingRegister`	Holding register object type. Instances declared by user in **`registers holding`** declaration group.	*Ch. 32*
`AccumulationRegister`	Accumulation register object type. Instances declared by user in **`registers accumulation`** declaration group.	*Ch. 33*
`WorkingRegister`	Working register object type. Instances declared by user in **`registers working`** declaration group.	*Ch. 34*
`Task`	Task object type. Instances declared by user in program code region.	*Ch. 35*
`State`	State object type. Instances declared by user inside **`task`** declaration bodies.	*Ch. 36*
`System_RealTime`	Real Time object type. Instances declared by Scanner Logic Script.	*Ch. 42*
`System_FlowRun`	Flow Run object type. Instances declared by Scanner Logic Script.	*Ch. 43*
`System_FlowArchive`	Flow Archive object type. Instances declared by Scanner Logic Script.	*Ch. 44*
`System_TriggeredArchive`	Triggered Archive object type. Instances declared by Scanner Logic Script.	*Ch. 45*

Object Type	Represents	Ref.
`System_UserEventRecord`	User Event Record object type. Instances declared by Scanner Logic Script.	*Ch. 46*
`System_PrintedTicket`	Printed Ticket object type. Instances declared by Scanner Logic Script.	*Ch. 47*
`System_Display`	Display object type. Instances declared by Scanner Logic Script.	*Ch. 48*
`System_Math`	Math object type. Instances declared by Scanner Logic Script.	*Ch. 49*
`HMI Fields`	User HMI object type. Instances declared by Scanner Logic IDE.	*Ch. 50*

13 Constants

13.1 General Description

A **constant** is a predefined identifier that resolves to a fixed literal value. Scanner Logic Script defines the following mathematical constants. More information about these constant values can be found in *Chapter 49* about the `System_Math` object.

Table 13.1-1. Math Constants

Constant	Data Type	Description
`Math.E`	`float`	Euler's Number
`Math.LN2`	`float`	Natural logarithm of 2
`Math.LN10`	`float`	Natural logarithm of 10
`Math.LOG2E`	`float`	Base-2 logarithm of e
`Math.LOG10E`	`float`	Base-10 logarithm of e
`Math.PI`	`float`	Pi, the ratio of a circle's circumference to its diameter

13.2 Remarks

User-defined constants are not supported in the current version of Scanner Logic Script. If you wish to have a value represented by an identifier in your code, you can use a `WorkingRegister` object and pre-set the `Value` property using the `initial_Value` parameter in the object declaration (see *Chapter 34*).

14 Variables

14.1 General Description

A **variable** is a name that is given to a data value or an object that Scanner Logic programs can manipulate. Variable names follow the rules for Identifiers in Chapter 10.

The names that are assigned to the various objects in the Program Declarations Region (see *Chapters 21 to 34*) are in effect user-declared object type variables. The names given to tasks and states in the Program Code Region (see *Chapters 35 to 36*) are also in effect user-declared object type variables.

The system objects (see *Chapter 41 to 49*) are accessed by using system-defined object type variable names.

14.2 Remarks

In the current version of Scanner Logic Script, there are no user-declared local variables or global variables for primitive data types like `bool`, `float`, or `uint`. This limitation significantly simplifies memory management and stack management issues in the program host environment. This reduced complexity allows the Scanner's logic controller functionality to be compartmentalized and decoupled from the device's primary measurement functions, thereby protecting its measurement integrity.

The `WorkingRegister` objects function as global variables. They have a `Value` property in which intermediate value, calculation results, etc. can be stored (see *Chapter 34*).

15 Operators

15.1 General Description

An **operator** is a program element that specifies an operation to perform in an expression or statement. An **operation** is the evaluation of an expression that contains an operator to produce a value from one or more operands. An **operand** is a value or an expression that is provided as input to an operator.

15.2 Operator Categories

Scanner Logic Script provides different categories of operators. Operators that operate on two values are called **binary operators**, while operators that operate on a single value are known as **unary operators**.

The following statement contains a single unary operator and a single operand. The increment operator (**++**) modifies the operand (**HoldingReg1.Value**).

Example 15.2-1. Unary operator example

```
HoldingReg1.Value++;
```

The following statement contains two binary operators, each with two operands. The assignment operator (**=**) has the uint property **HoldingReg1.Value** and the expression 2 + 3 as operands. The expression 2 + 3 itself consists of the addition operator and two operands, 2 and 3.

Example 15.2-2. Binary operators example

```
HoldingReg1.Value = 2 + 3;
```

Certain operators are called **primary operators** because they are evaluated first before any other operators in an expression, that is, they have the highest precedence. Operators that are within the same category below share the same precedence level. The categories are listed in order of precedence.

Table 15.2-1. Primary Operators

Operator	Description	Example
.	**Member Access** Property and method members of objects are accessed via the dot operator.	x.y

Operator	Description	Example
`f(x)`	**Invocation** Methods of objects or subroutines of the program are invoked with the parentheses following the method name. There may be zero or more parameters within the parentheses.	`x.y()` `x()`
`x++`	**Postfix Increment** Returns the value of x and then updates the storage location for x with x + 1.	`x++`
`x--`	**Postfix Decrement** Returns the value of x and then updates the storage location for x with x - 1.	`x++`

Table 15.2-2. Unary Operators

Operator	Description	Example
`+`	**Identity** As a unary operator, + has no effect and is removed on compile.	`+x`
`-`	**Numeric Negation** Only integers can be subtracted from dates. Scanner Logic Script interprets such an integer as a number of seconds.	`-x`
`!`	**Logical Negation**. A unary logical operator that results in true if the operand to its right is false, and false if the operand is true.	`!x`
`++x`	**Prefix Increment** Returns the value of x after updating the storage location of x with x + 1.	`++x`
`--x`	**Prefix Decrement** Returns the value of x after updating the storage location of x with x - 1.	`--x`
`(T)x`	**Type Casting** Returns the value of x converted to a value of the type specified by the type `T` in the parentheses. The only valid types for type casting are `uint`, `float`, and `bool`. Type casting is most commonly used in using or writing to properties that have a type that needs to be matched. See *Type Conversion and Type Casting* for more information about type casting.	`(float)x`

Table 15.2-3. Arithmetic Operators

Operator	Description	Example
*	**Multiplication** A binary arithmetic operator that multiplies the number to its left and the number to its right.	`x * y`
/	**Division** A binary arithmetic operator that divides the number to its left by the number to its right.	`x / y`
%	**Modulus** A binary arithmetic operator that divides the number to its left by the number to its right and returns the remainder as its result. This remainder is not restricted to being an integer and can be a floating-point value.	`x % y`
+	**Addition** A binary arithmetic operator that adds the number or date to its left and the number or date to its right. Only integers can be added to dates. Scanner Logic Script interprets such an integer as a number of seconds.	`x + y`
-	**Subtraction** The binary operator subtracts the number to its right from the number or date to its left.	`x - y`

Table 15.2-4. Relational Operators

Operator	Description	Example
>	**Greater than** A binary comparison operator that results in `true` if the value of the left-hand operand is greater than the value of the right-hand operand. Both operands must evaluate to values of the same class. If they don't, Scanner Logic Script attempts to coerce the right-hand operand to the class of the left-hand operand.	`x > y`

Operator	Description	Example
<	**Less than** A binary comparison operator that results in `true` if the value of the left-hand operand is less than the value of the right-hand operand. Both operands must evaluate to values of the same class. If they don't, Scanner Logic Script attempts to coerce the right-hand operand to the class of the operand to the left.	`x < y`
>=	**Greater than or equal to** A binary comparison operator that results in `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand. Both operands must evaluate to values of the same class. If they don't, Scanner Logic Script attempts to coerce the right-hand operand to the class of the operand to the left.	`x >= y`
<=	**Less than or equal to** A binary comparison operator that results in `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand. Both operands must evaluate to values of the same class. If they don't, Scanner Logic Script attempts to coerce the right-hand operand to the class of the operand to the left.	`x <= y`

Table 15.2-5. Equality Operators

Operator	Description	Example
==	**Equality** A binary comparison operator that results in `true` if its two operands have the same values. The operands can be of any literal type.	`x == y`
!=	**Inequality** A binary comparison operator that results in `true` if its two operands have different values. The operands can be of any class.	`x != y`

Table 15.2-6. Conditional AND Operator

Operator	Description	Example
`&&`	**Conditional AND** A binary logical operator that combines two Boolean values. The result is `true` only if both operands evaluate to `true`.	`x && y`

Table 15.2-7. Conditional OR Operator

Operator	Description	Example
`||`	**Conditional OR** A binary logical operator that combines two Boolean values. The result is `true` if either operand evaluates to `true`.	`x || y`

Table 15.2-8. Assignment and Compound Assignment Operators

Operator	Description	Example	Equivalent To
`=`	**Assignment** Store the value of the expression on the right side into the operand on the left side.	`x = y`	`x = y`
`+=`	**Addition Assignment** Add the value of the right operand to the value of the left operand, store the result in the left operand, and return the new value.	`x += y`	`x = x + y`
`-=`	**Subtraction Assignment** Subtract the value of the right operand from the value of the left operand, store the result in the left operand, and return the new value.	`x -= y`	`x = x - y`
`*=`	**Multiplication Assignment** Multiply the value of the right operand by the value of the left operand, store the result in the left operand, and return the new value.	`x *= y`	`x = x * y`
`/=`	**Division Assignment** Divide the value of the left operand by the value of the right operand, store the result in the left operand, and return the new value.	`x /= y`	`x = x / y`

%=	Modulus Assignment Divide the value of the left operand to the value of the right operand, store the remainder in the left operand, and return the new value.	x %= y	x = x % y

15.3 Operator Precedence

Each operator has a defined precedence. In an expression that contains multiple operators that have different precedence levels, the precedence of the operators determines the order in which the operators are evaluated. When evaluating expressions, Scanner Logic Script evaluates the operators with higher precedence before lower precedence operators. Parentheses can be used to cause expressions containing lower-precedence operators to be evaluated before higher-precedence operators.

In the following expression, for example, the operations are not simply performed from left to right—the multiplication operation `2 * 5` is evaluated first, because the multiplication operator has higher precedence than the addition operator.

Example 15.3-1. Operator precedence – multiplication has higher precedence than addition

```
12 + 2 * 5

12 + (2 * 5) // equivalent expression

// result: 22
```

Table 15.3-1 below shows the order in which Scanner Logic Script operators are evaluated.

Table 15.3-1. Operator precedence from highest to lowest

Order	Operators	Associativity	Notes
1	`(...)`	Innermost to outermost	Parentheses around an expression are not operators themselves. Instead, they cause the enclosed expression to be evaluated and resolved to a single value first before processing normal precedence order.
2	`x.y`	Left to right	Member access
2	`f(x)`	Left to right	Method or subroutine invocation

Order	Operators	Associativity	Notes
3	`x++` `x--`	None	Since the operand of a postfix increment or postfix decrement operation is a variable or a property, and the result of the operation is a `uint` or `float`, these operators cannot be used associatively.
4	`+x` `-x`	Innermost to outermost	Numeric identity and negation
5	`!`	Innermost to outermost	Logical negation
6	`++x` `--x`	None	Since the operand of a prefix increment or prefix decrement operation is a variable or a property, and the result of the operation is a `uint` or `float`, there cannot be more than one of these operators in a row.
7	`(T)x`	Innermost to outermost	Explicitly convert x to type `T`

8	`*`
	`/`
	`%`
		Left to right	Multiplicative operators

9	`+` `-`	Left to right	Additive operators

10	`>`
	`>=`
	`<`
	`<=`
		None	Since the result of relational operators is bool, and the operands of these operators cannot be bool, there cannot be more than one of these operator in a row

11	`==` `!=`	Left to right	Equality operators
12	`&&`	Left to right	Conditional AND
13	`||`	Left to right	Conditional OR

14	`=`
	`+=`
	`-=`
	`*=`
	`/=`
	`%=`
		Right to left	Assignment operations are processed from right to left with equal precedence, regardless of the kind of arithmetic operation in compound assignment operators.

15.4 Associativity

When two or more operators that have the same precedence are present in an expression, they are evaluated based on **associativity**. Left-associative operators are evaluated in order from left to right. For example, x * y / z is evaluated as (x * y) / z. Right-associative operators are evaluated in order from right to left. For example, the assignment operator is right associative.

Example 15.4-1. Assignment operator is right-associative

```
a = b = c;

a = (b = c); // equivalent expression

(a = b) = c; // ERROR! NOT equivalent expression
```

The Associativity column in Table 15.3-1 shows the associativity type of the operators. The word "None" in the Associativity column indicates that you cannot have multiple consecutive occurrences of the operation in an expression. For example, the expression `3 > 2 > 1` is not legal because the associativity for the greater than operator is "None."

To evaluate expressions with multiple unary operators of the same order, Scanner Logic Script applies the operator closest to the operand first, then applies the next closest operator, and so on. For example, the expression `!!!true` is equivalent to `!(!(!true))`.

Whether the operators in an expression are left associative or right associative, the operands of each expression are evaluated from left to right first, before applying the operation. The following examples illustrate the order of evaluation of operators and operands.

Table 15.4-1. Order of evaluation of operators and operands

Statement	Order of Evaluation
`a = b`	`a, b, =`
`a = b + c`	`a, b, c, +, =`
`a = b + c * d`	`a, b, c, d, *, +, =`
`a = b * c + d`	`a, b, c, *, d, +, =`
`a = b - c + d`	`a, b, c, -, d, +, =`
`a += b -= c`	`a, b, c, -=, +=`

15.5 Adding Parentheses

You can alter the order in which Scanner Logic Script performs operations by grouping sub-expressions that you want evaluated first within parentheses. This causes the inner sub-expression to be evaluated to a single result value, which is then used as an operand in the outer expression that contains the parenthesized sub-expression. If there are sub-expressions parenthesized within other sets of parentheses, then the parenthesized expressions are evaluated from innermost to outermost.

Table 15.5-1. Order of evaluation of operators and operands

Statement	Order of Evaluation
`a = (b + c) * d`	`a, b, c, +, d, *, =`
`a = b - (c + d)`	`a, b, c, d, +, -, =`
`a = (b + c) * (d - e)`	`a, b, c, +, d, e, -, *, =`

16 Expressions

16.1 General Description

An **expression** is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, or method. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, or type.

Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls. Expressions can range from simple to very complex.

Example 16.1-1. Examples of expressions

```
Math.Exp(5) + Math.PI

15 > RealTime.Month

Math.Sin(Math.Cos(ConfigurationReg1.Value))

!false

((x < 10) && (x > 5)) || ((x > 20) && (x < 25))
```

Expressions are used in programs to represent or derive values. The simplest kinds of expressions, called literal expressions, are representations of values in programs. More complex expressions typically combine literals, variables, operators, and object specifiers.

16.2 Expression Values

In most of the contexts in which expressions are used, for example in statements or method parameters, the expression is expected to evaluate to some value. If x and y are integers, the expression x + y evaluates to a numeric value. The expression `Math.Cos(x)` evaluates to a `float` because that is the return type of the method.

However, although a type name (for example, `bool`, `float`, `uint`) is classified as an expression, it does not evaluate to a value and therefore can never be the result of any expression. You cannot pass a type name to a method parameter, or use it in a new expression, or assign it to a variable.

Every value has an associated type. For example, if x and y are both variables of type `float`, the value of the expression x + y is also typed as `float`. If the value is assigned to a variable of a different type, or if x and y are different types, the rules of type conversion are applied. For more information about how such conversions work, see *Table 12.3-1*.

16.3 Operator precedence and associativity

The way in which an expression is evaluated is governed by the rules of associativity and operator precedence. For more information, see *Chapter 15*.

Most expressions, except assignment expressions and method invocation expressions, must be embedded in a statement. For more information, see *Chapter 17*.

16.4 Literals and simple names

The simplest types of expressions are literals. A literal is a constant value that has no name. For example, in the following code example, the values `5`, `1.2`, and `true` are literal values: For more information on literals, see *Chapter 29*.

Example 16.4-1. Literal expression examples

```
HoldingReg1.Value = 5;

if (true)
   HoldingReg2.Value = 1.2;
```

16.5 Assignment expressions

Assignment expressions evaluate the sub-expression on the right side of the assignment operator to produce a single value, which is stored into the variable or object property that is on the left side of the assignment. The reason that assignments can be considered expressions is that the value assigned to the left operand is available to participate in other expressions. The type of the operand on the left of the assignment operator is the type of the entire expression, and the entire expression evaluates to the right value. It is possible to use an assignment expression anywhere an expression value is required.

Example 16.5-1. Assignment expression examples

```
HoldingReg1.Value = Math.Cos(5);

WorkingReg1.Value = Math.Pow(ConfigReg1.Value, 2) + 1;

WorkingReg3.Value = WorkingReg2.Value = WorkingReg1 * 1.1;
```

16.6 Invocation expressions

In the following code example, the calls to `Cos`, `Sin`, and `Start` are invocation expressions.

Example 16.6-1. Invocation expression examples

```
HoldingReg1.Value = Math.Cos(5);

HoldingReg1.Value = Math.Cos(Math.Sin(2 * Math.PI));

Timer1.Start();

// this statement produces an error because the Start() method of the TimerResource
// object has a return type of void, i.e. it has no return value
HoldingReg1.Value = Timer1.Start();
```

A method invocation requires the name of the method, followed by parentheses and any method parameters. Method invocations evaluate to the return value of the method, if the method returns a value. Methods that return void cannot be used in place of a value in an expression.

16.7 Remarks

Whenever a variable or object property is identified from an expression, the value of that item is used as the value of the expression. An expression can be placed anywhere in Scanner Logic Script program code where a value or object is required, as long as the expression ultimately evaluates to the required type.

Certain types of expressions can be used alone as a statement, forming **expression statements**. Assignment expressions, prefix/postfix increment/decrement expressions, and invocation expressions cause actions to occur in the program in addition to evaluating to a result value.

17 Statements

17.1 General Description

The actions that a Scanner Logic Script program takes are expressed in **statements**. A statement is a series of elements that follows a Scanner Logic Script syntax. Statements can include keywords, resources, registers, operators, constants, expressions, and so on.

Every Scanner Logic Script program consists of statements. When a program runs, it executes the statements within blocks of program code in order and carries out their instructions.

Common actions include declaring various program objects, assigning values, calling methods or subroutines, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in start and end braces ({}) and can contain nested blocks. Statement blocks do not require a semicolon at their end. The following code shows two examples of single-line statements, and a multi-line statement block:

Example 17.1-1. Some examples of statements

```
HoldingReg2.Value = Math.Exp(5) + Math.PI;


changestate NewState;


if (Alarm1.IsAsserted == true)
{
   HoldingReg1.Value++;
   Timer1.Start();

   CalculationsSubroutine();
}
```

17.2 Types of Statements

The following categories of statements are supported in Scanner Logic Script.

Table 17.2-1. Types of statements

Category	Description
Expression statements	Expression statements that calculate a value must store the value in a variable. Assignment statements Increment and decrement statements Invocation statements

Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions.
	`if`
	`else`

Jump statements	Jump statements transfer control to another section of code.
	`changestate`
	`continue`
	`return`

Empty statement	The empty statement consists of a single semicolon. It can be used in places where a statement is required but no action needs to be performed.

In the current version of Scanner Logic Script, there is no support for iteration type statements (for loops, while loops, etc.). This is an intentional limitation to reduce the complexity of the script execution engine and to avoid the need to protect the device from excessive or infinite looping, which would jeopardize the integrity of the primary measurement functionality of the Scanner 3100.

However, note that the operation of the state machine model that is the basis of Scanner Logic programs, with its execution cycle mechanism, in effect provides a controlled looping construct (see *Chapter 36*).

17.3 Assignment Statements

Assignment statements are a kind of expression statement. Assignment expressions evaluate to a single value, with the additional action that the value is stored into a variable or object property. When an assignment expression is used alone in a statement, the assignment expression value is discarded after it has been stored in the left operand.

Example 17.3-1. Example assignment statement – resource object property

```
// assign a value to a property of a resource object
   PIDController.Ki = 1.23;
```

Example 17.3-2. Example assignment statement – register object property

```
// assign a value to a property of a register object
   HoldingReg1.Value = 7.14 * TemperatureReading.Value;


// equivalent to the statement above, since Value is the default property of
// HoldingRegister objects and of RegisterInputResource objects
   HoldingReg1 = 7.14 * TemperatureReading;
```

Example 17.3-3. Example assignment statement – system object property

```
// assign a value to a system object property
   Display.SetValue = 2;
```

17.4 Increment and Decrement Statements

Prefix increment, postfix increment, prefix decrement, and postfix decrement expressions can be used by themselves in expression statements. They can be used alone in statements because they perform the action of storing the expression value back into the operand in addition to resolving to a value. When increment/decrement expressions are used in expression statements, their resulting expression value is discarded.

Example 17.4-1. Example increment/decrement statements

```
   CounterWorkingReg.Value++;

   --WorkingResult;

   WorkingResult--;
```

17.5 Invocation Statements

Invocation statements are a kind of expression statement. Invocation expressions can potentially evaluate to a single return value, or they might not return any value. Invocation expressions can be used alone in statements because they have the potential to perform various actions in the program within the code of the method or subroutine that is being invoked, in addition to potentially resolving to a value. When an invocation expression that does resolve to a value is used alone in a statement, the return value is ignored.

Example 17.5-1. Example invocation statements – subroutine calls

```
state SomeState
{
   onEnter
   { ... }

   onLoop
   {
      Subroutine1();

      Subroutine2();
   }

   onExit
   { ... }
}

...

void subroutine Subroutine1()
{
   ...
}

void subroutine Subroutine2()
{
   ...
}
```

Example 17.5-2. Example invocation statements – object method calls

```
// resource object method call
   Alarm1.Assert();

// task object method call
   Task1.RestartExecution();

// system object method call
   FlowRun1.EnableAccumulation();

// math object method call
   WorkingResult.Value = Math.Tan(1.1);
```

17.6 Selection Statements

A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is true or not.

Example 17.6-1. Selection statements

```
if (Pressure.Value > 100)
   FlowRun1.EnableAccumulation();


if (Pressure.Value > 100)
   FlowRun1.EnableAccumulation();
else
   FlowRun1.DisableAccumulation();


if (Pressure.Value > 100)
{
   FlowRun1.EnableAccumulation();
   HoldingReg1.Value = Pressure.Value;
   Alarm1.Deassert();
}
else
{
   FlowRun1.DisableAccumulation();
   Alarm1.Assert();
}
```

17.7 Jump Statements

Branching is performed using jump statements, which cause an immediate transfer of the program control. The following keywords are used in jump statements

Table 17.7-1. List of program control keywords

Keyword	Description
`changestate`	Immediately triggers a transition to the target **`state`**, executes **`onExit`** block of current state then **`onEnter`** block of target **`state`**. May only be invoked in the **`onLoop`** block of a state object.
`continue`	Immediately jumps program to the end of the current code block and continues to end of execution cycle. May only be used within the **`onEnter`**, **`onLoop`**, and **`onExit`** blocks of a state object.
`return`	End execution of current subroutine. May only be used within a subroutine. Can be omitted if the subroutine does not return a value and the program logic flow does not need to exit subroutine early.

Scanner Logic programs are modeled on a state machine concept. Program flow remains looping inside a state until conditions that trigger a transition to another state are detected. The **`changestate`** statement is the mechanism by which programs transition between states. This statement consists of the keyword **`changestate`**, followed by the name of a **`state`** within the same **`task`**, followed by a semicolon.

The **continue** statement is used to finish a script execution cycle before the normal end of the code in the current execution block (**onEnter**, **onLoop**, **onExit**) of a **state** object. It consists of the keyword **continue** followed by a semicolon. It causes program flow to skip over all the subsequent code statements in the execution block and go to the end brace of the block. If a **continue** statement is used within an **onEnter** block, execution will continue into the **onLoop** block as usual. If a **continue** statement is used within an **onLoop** block, the execution cycle ends and program flow resumes at the top of the **onLoop** block at the next execution cycle. If a **continue** statement is used in an **onExit** block, program flow enters the **onEnter** block of the target **state** object as usual.

The **return** statement is used within **subroutine** bodies to cause the program execution flow to exit the **subroutine** at that point and return to executing the code from the point immediately after where the **subroutine** was called. If the subroutine returns no value, then the **return** statement consists of just the keyword **return** followed by a semicolon. If the **subroutine** has a return type other than **void**, the **return** statement must also include a return expression evaluating to a single value of the return type placed between the keyword and the semicolon. The **return** statement can be omitted for subroutines with no return value if there is no need for program flow to exit the **subroutine** before the end of the code statements in its body.

Example 17.7-1. Example jump statements

```
state Monitoring
{
   onEnter
   {
      if (ContactAlarm.IsAsserted)
         continue;

      CheckContact();
   }

   onLoop
   {
      if (Temperature < 0)
         changestate Heating;
   }

   onExit
   {
      ...
   }
}

state Heating
{
```

```
   ...
}

...

void subroutine CheckContact()
{
   if (Contact.IsActive)
   {
      ContactAlarm.Assert();
      return;
   }

   // some other actions
   ...

   return;
}
```

17.8 Embedded Statements

Some statements, including **`if`** and **`if...else`**, always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by start and end braces in a statement block. Even single-line embedded statements can be enclosed in **`{}`** braces, as shown in the following example.

Example 17.8-1. Embedded statements – single-line and statement block

```
// single-line embedded statement
if (Pressure.Value > 100)
   FlowRun1.EnableAccumulation();


// single-line embedded statement placed in a statement block
if (Pressure.Value > 100)
{
   FlowRun1.EnableAccumulation();
}


// multiple statements in a statement block; the block takes the place of the
// single-line embedded statement
if (Pressure.Value > 100)
{
   FlowRun1.EnableAccumulation();
   HoldingReg1.Value = Pressure.Value;
   DigitalOut1.Activate();
}
```

17.9 Nested Statement Blocks

Statement blocks can be nested, as shown in the following code:

Example 17.9-1. Nested statement blocks

```
if (Pressure.Value > 100)
{
   if (Temperature.Value > 10)
   {
      FlowRun1.EnableAccumulation();
      HoldingReg1.Value = Pressure.Value;
      DigitalOut1.Activate();
   }
   else
   {
      FlowRun1.DisableAccumulation();
      DigitalOut1.Deactivate();
   }
}
```

17.10 Parameter Assignment Statements

There is another kind of assignment statement that is only used in the declaration of the **proginfo** object and of the **resource** and **registers** objects. Parameter assignment statements have a parameter name, a colon instead of an equals sign as the assignment operator, a literal value, and semicolon. This type of statement syntax is only valid within the bodies of the object declarations. Each of the chapters in 0 list the available declaration parameters for each object type.

Example 17.10-1. Example parameter assignment statements

```
proginfo
{
   ProgramName: "Brewer Method";
   ...
   ProgramVersion: 1.0;
   ...
}


resource registerinputs
{
   01: SurfacePressure
   {
      description: "The pressure at the surface of the well";
      ...
      category: "Static Pressure (gauge)";
      units: "psig";
   }
}
```

18 Preprocessor Directives

A **preprocessor directive** is composed of the # symbol and an identifier. The directive may be followed by directive text on the same line that is used by the preprocessor directive.

18.1 Region and Endregion Directives

Scanner Logic Script uses the directives **`#region`** and **`#endregion`** to set apart certain sections of the program code for special treatment in the Scanner Logic Script IDE. The **`#region`** directive is used to mark the text as being collapsible in the IDE editor window. Entire sections of text can be demarcated in this way to aid in organizing the script document or to improve readability by removing visual clutter from around the important parts of the program.

Example 18.1-1. Example region directives

```
program
{

#region Program Declarations

   resource alarms
   {
   ...
   }

#endregion

   task Main
   {

#region A set of related states
      state State1
      { ... }

      state State2
      { ... }

      state State3
      { ... }
#endregion

   ...

   }
```

18.2 Pragma directive

The `#pragma` directive is used to issue a special command to the compiler. These commands are defined by the Scanner Logic Script language development team for special purposes. An example of such a command is `#pragma nometrics`, which disables the function of the compiler that verifies that characteristics of the binary program code like path execution times and maximum device stack usage are not exceeded.

Part III—Logic Script Objects

19 Program Object

19.1 General Description

The Scanner Logic Script program is contained within the Program object, which is held in a single text-based file with a file name ending with the "`SLOGIC`" extension. The Program object begins with the **`program`** keyword, and the body of the Program object is enclosed within a pair of open and close braces.

The content of the Program body is organized into 5 regions. Most of these regions are "collapsible" in the code editor (IDE), so that entire sections of code can be consolidated into a single display line to improve visibility of more important areas of code.

The regions of the program should be maintained in the order shown in Example 19.1-1.

Example 19.1-1. Overall structure of the **`program`** object

```
program
{

#region Program Information

   proginfo
   {
   ...
   }

#endregion

#region Program Declarations

   resource alarms
   {
   ...
   }

   ...

   registers configuration
   {
   ...
   }

   ...
```

```
#endregion

   task Main
   {
   ...
   }

#region System State Declarations

   abortState
   {
   ...
   }

   failState
   {
   ...
   }

#endregion

#region Subroutines

   void subroutine Sub()
   {
   ...
   }

#endregion
}
```

19.2 Program Information Region

By convention, the Program Information region should be enclosed between **#region…#endregion** preprocessor directives, with "Program Information" as the text for the **#region** directive, albeit the compiler does not enforce this structure.

This region contains the **proginfo** object declaration. This object holds the identifying information about the program that is accessible from within the Scanner 3100 web interface.

See *Chapter 20* for more information about the **proginfo** object.

19.3 Program Declarations Region

By convention, the Program Declarations region should be enclosed between **#region…#endregion** preprocessor directives, with "Program Declarations" as the text for the **#region** directive, albeit the compiler does not enforce this structure.

This region contains **resource** and **registers** object declaration groups. These declaration groups contain declarations items for Scanner 3100 inputs and outputs that will be used within the program.

See *Chapters 21 to 34* for more information about **resource** and **registers** objects.

19.4 System Declarations Region

By convention, the System Declarations region should be enclosed between **#region…#endregion** preprocessor directives, with "System Declarations" as the text for the **#region** directive, albeit the compiler does not enforce this structure.

This region contains **abortState** and **failState** declarations. These special state objects contain the user code that will execute in the case of a user abort signal or a fatal program error.

See *Chapters 37 to 39* for more information about **abortState** and **failState**.

19.5 Program Code Region

The Program Code region, where the logic and control instructions are declared, is not enclosed between **#region…#endregion** preprocessor directives. It simply lies between the System Declarations region and the Subroutines region. By default, when you start a new program in the IDE, this area of the program file is not set to collapsible. However, you are free to add your own **#region…#endregion** directives to create your own custom collapsible code regions.

This region contains the **task** declarations, each of which contain **state** declarations. User code resides in the **onEnter, onLoop, onExit** sections of **state** objects. During program execution, each active **task** runs in parallel with the other tasks, and has one current **state**. User code can change the current **state** of each **task** when specified conditions are fulfilled.

See *Chapters 35 to 36* for more information about **task** and **state** objects.

19.6 Subroutines Region

By convention, the Subroutines region should be enclosed between **#region…#endregion** preprocessor directives, with "Subroutines" as the text for the **#region** directive, albeit the compiler does not enforce this structure.

This region contains the global `subroutine` declarations. Subroutines are useful for consolidating commonly used code in one location that can be invoked from code within any `state`.

See *Chapter 40* for more information about `subroutines`.

20 Program Information Object

20.1 General Description

The Program Information region provides information that is used to identify the script program and determine its source code availability level for web interface users. This information will appear in the Program Information screen of the web interface. There are no properties or methods of the **proginfo** object that are accessible by user code at run time.

20.2 Declaring proginfo

The Program Information declaration begins with the **proginfo** keyword, and contains a list of parameter assignment statements within a pair of open and close braces. All parameters are required, and there will be a parser error if any of the information is missing.

Example 20.2-1. A **proginfo** declaration with parameter assignment statements

```
proginfo
{
   ProgramName: "Brewer Method";
   ProgramAuthor: "A. Programmer";
   ProgramOwner: "Cameron Valves & Measurement";
   ProgramVersion: 1.0;
   ProgramCreationDate: "04/15/2016";
   Access_OnlineSource: "allusers";
   Access_OnlineControls: "allusers";
   Access_WriteHMI: "allusers";
   ProgramDescription: "Implements the Brewer Method example for dewatering";
}
```

20.3 Declaration Parameters

Table 20.3-1. Program Information object declaration parameters

Parameter	Type Code	Description
ProgramName	<str32>	Title of the script program.
ProgramAuthor	<str32>	Name of the script developer.
ProgramOwner	<str32>	Name of the script owner.
ProgramVersion	float	User defined version number of the script program.
ProgramCreationDate	<date>	Date on which the script program was compiled.

Parameter	Type Code	Description
`Access_OnlineSource`	`<option>`	Whether source code is viewable via web interface, and minimum access permission level required to view if viewable. Allowable values: `"nousers" | "adminusers" | "configusers" | "maintusers" | "allusers"`
`Access_OnlineControls`	`<option>`	The minimum access permission level required to activate web interface controls for restarting and aborting execution of the program. Allowable values: `"nousers" | "adminusers" | "configusers" | "maintusers" | "allusers"`
`Access_WriteHMI`	`<option>`	The minimum access permission level required to perform writes on a user defined HMI web interface page. Allowable values: `"nousers" | "adminusers" | "configusers" | "maintusers" | "allusers"`
`ProgramDescription`	`<str1024>`	Comment describing purpose or usage of program.

21 Logic Script Resource Objects

21.1 General Description

A **resource** object, or resource item, in the Scanner Logic Script environment provides an interface to inputs, outputs, and device registers of the Scanner. Resource object declarations of the same type are grouped together within a resource declaration group.

Resource items have an index number within their resource declaration group, and have user-assigned identifier names. A resource item declaration includes several parameter assignment statements between a pair of open and close braces.

Example 21.1-1. A **registerinputs** resource declaration group with two **RegisterInputResource** objects

```
resource registerinputs
{
   01: CasingPressure
   {
      description: "Casing pressure reading";
      tagname: "StatPres_InstReading";
      tagcode: "m32_FC_IN_2_Holding_InstReading";
      category: "Static Pressure (absolute)";
      units: "psia";
   }
   02: TubingPressure
   {
      description: "Tubing pressure reading";
      tagname: "Analog2_InstReading";
      tagcode: "m32_FC_IN_6_Holding_InstReading";
      category: "Static Pressure (absolute)";
      units: "psia";
   }
}
```

21.2 Resource Object Types

The types of resource objects available in Scanner Logic Script are:

Table 21.2-1. Types of resource objects

Resource Type	Usage	Qty	Description
`RegisterInputResource`	`input`	32	Allows program to read input values from the Scanner 3100 device.
`DigitalInputResource`	`input`	6	Maps to the digital input ports of the Scanner 3100 to allow reading the state of the ports.
`AnalogPIDControllerResource`	`output`	2	Provides a PID controller object whose output can be mapped to a Scanner 3100 analog output port.
`DigitalPIDControllerResource`	`output`	1	Provides a PID controller object whose output can be mapped to a Scanner 3100 digital valve controller output.
`DigitalOutputResource`	`output`	6	Provides a digital output proxy object whose output can be mapped to a Scanner 3100 digital output port
`AlarmResource`	`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.
`TimerResource`	`utility`	8	Provides an object that can keep track of the time in seconds between an execution of the start and stop methods.

21.3 Usage Notes

Declaration parameters initialize internal values in the resource objects at compile time.

When declaring a resource item, some of the properties of the object may be initialized using parameters beginning with an **`initial_`** prefix and ending with the property name. The **`ReloadInit()`** method of the resource objects reloads the initial values into the properties during program execution.

Resource objects also have properties and methods that can be accessed by user code at run time. **Properties** are named data values of an object that can be read or written by user code. Some properties are read-only and cannot be modified by user code. **Methods** are named actions of an object that can be invoked by user code. Methods can potentially accept input arguments and return values.

All resource objects have a **default property**. In user code, the default property can be omitted and the name of a register can be used by itself to resolve to its default property. This can help make script programs easier to read and understand.

Example 21.3-1. Default Properties of Resource Objects

```
// these statements are equivalent; the default property of TimerResource is Time
if (Timer1.Time > 60)
    changestate NextState;

if (Timer1 > 60)
    changestate NextState;

// these statements are equivalent; the default property of RegisterInputResource
// is Value
HoldingReg2.Value = RegisterInput1.Value + 10;

HoldingReg2.Value = RegisterInput1 + 10;
```

22 Register Input Resource Object

22.1 General Description

`RegisterInputResource` objects are input resources. Each `RegisterInputResource` object is an interface to a physical Scanner input or any other register from the Scanner host environment.

Up to 32 `RegisterInputResource` objects may be declared for use in a Scanner Logic Script program.

22.2 Required S3100 Device Configuration

A `RegisterInputResource` object has an associated measurement category, since the Scanner registers that are imported via these objects all have a designated measurement category. In most cases, the category will be automatically set by IDE assistants when the register tagname is selected. In some cases, where the measurement category of a Scanner register is dynamic (e.g. Analog Inputs), the programmer must indicate the expected category. The script programmer can specify the units (via `category`, `unit`, and `rate` parameters) of the value that is presented to the program by the script execution engine.

The measurement category of each input register that corresponds to a declared `RegisterInputResource` object will be verified at the beginning of the program and at the start of each script execution cycle. If the category specified does not match the category of the selected Scanner register, the Scanner Logic Script will encounter a run time error and the `failState` state will be invoked.

Table 22.2-1. `RegisterInputResource` object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Analog Input	Transducer Type	Must match category of `RegisterInputResource` object
Pulse Input	Accumulation Type	Must match category of `RegisterInputResource` object

22.3 Declaring RegisterInputResource Objects

The `RegisterInputResource` declaration group begins with the keywords `resource registerinputs`, and contains a resource item declaration for each `RegisterInputResource` item to be used in the program within a pair of open and

close braces. Each resource item declaration consists of a unique resource index number between `01` and `32` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no **`RegisterInputResource`** items are required, the entire resource declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 22.3-1. **`RegisterInputResource`** object declaration example

```
resource registerinputs
{
   01: SurfacePressure
   {
      description: "The pressure at the surface of the well";
      tagname: "Analog1_InstReading";
      tagcode: "m32_FC_IN_5_Holding_InstReading";
      category: "Static Pressure (gauge)";
      units: "psig";
   }
   02: ActualFlowRate
   {
      description: "The measured flow rate";
      tagname: "FlowRun1_LiquidOilVolumeFlowRate";
      tagcode: "m32_FC_FR_1_HoldingAccum_LiquidOilVolumeFlowRate";
      category: "Liquid Volume";
      units: "m3";
      rate: "/sec";
   }

   ...

   32: TankTemperature
   {
      description: "Temperature that is monitored for activating the tank heater";
      tagname: "Analog2_InstReading";
      tagcode: "m32_FC_IN_6_Holding_InstReading";
      category: "Temperature";
      units: "degF";
   }
}
```

22.4 Declaration Parameters

Table 22.4-1. **RegisterInputResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the RegisterInputResource object
tagname	<tagname>	The descriptive string describing the register tag name.
tagcode	<tagcode>	The complete Scanner 3100 register tag descriptor that is used as the source of the input
category	<category>	The unit category of the S3100 input source. The category must be selected if the input source has a dynamic category, otherwise the category will be set automatically. *See Chapter 51 for more information.*
units	<unit>	Numerator of the measurement unit desired for the value imported from the Scanner 3100 register, and denominator if required for the specified measurement units category. *See Chapter 51 for more information.*
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. *See Chapter 51 for more information.*

22.5 Properties

Table 22.5-1. **RegisterInputResource** object properties

Property	Data Type	Access	Description
Value [default]	float	RO	Holds the value of the register inputs object, converted to the measurement units specified in the declaration; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

22.6 Methods

All methods listed below do not accept parameters nor return values.

Table 22.6-1. **RegisterInputResource** object methods

Method	Return Type	Description
Clear()	void	Sets **Value** to 0.0 until next update.

22.7 Usage

Example File: **UsingRegisterInputs.slogic**

Here, a **RegisterInputResource** is used to access the live reading from analog input 1, when a low flow rate condition is met a **changestate** is triggered.

Example 22.7-1 **RegisterInputResource** object usage

```
resource registerinputs
{
   01: TubingFlowRate
   {
      description: "Tubing Flow Rate in cubic meters per second live reading";
      tagname: "Analog1_InstReading";
      tagcode: "m32_FC_IN_5_Holding_InstReading";
      category: "Liquid Volume";
      units: "m3";
      rate: "/sec";
   }
}

...

task Task1
{
   initial state MonitorFlowRate
   {
      onEnter { }

      onLoop
      {
         if (TubingFlowRate < 1.50) changestate LowFlowRate;
      }

      onExit { }
   }

   state LowFlowRate
   {
      ...
   }
```

23 Digital Input Resource Object

23.1 General Description

`DigitalInputResource` objects are input resources that are interfaces to physical Scanner hardware digital inputs. If declared, each `DigitalInputResource` corresponds directly to the physical DIO port with the matching index number in the Scanner host environment.

Up to 6 `DigitalInputResource` objects may be declared for use in a Scanner Logic Script program.

23.2 Required S3100 Device Configuration

The Scanner 3100 has 6 physical DIO ports that may be set to map to the declared resource objects. The Digital I/O Mode of each DIO port that corresponds to a declared `DigitalInputResource` will be verified at the beginning of the program and at the start of each script execution cycle. If the Digital I/O Mode specified is not "Input Mode", the Scanner Logic Script will encounter a run time error and the `failState` state will be invoked. Physical DIO ports are configured by default as "Input Mode".

Note: `DigitalInputResource` and `DigitalOutputResource` objects both map to the same set of physical DIO ports and are mutually exclusive functions. A `DigitalInputResource` and `DigitalOutputResource` defined at the same index will generate a run time error and the `failState` state will be invoked.

Table 23.2-1. `DigitalInputResource` object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Digital In/Out	Digital I/O Mode	Input Mode

23.3 Declaring DigitalInputResource Objects

The `DigitalInputResource` declaration group begins with the keywords `resource digitalinputs`, and contains a resource item declaration for each `DigitalInputResource` to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number between `01` and `06` and a user defined identifier separated by a colon, and contains a list of parameters assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration

group. If no **DigitalInputResource** items are required, the entire declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 23.3-1. **DigitalInputResource** object declaration example

```
resource digitalinputs
{
   01: HighPressureSwitchMyDigitalInputName1
   {
      description: "High pressure switch";
   }
   02: StartWellTestMyDigitalInputName2
   {
      description: "External button for activating well test mode";
   }

   ...

   06: TankFullMyDigitalInputName6
   {
      description: "Tank limit signal";
   }
}
```

23.4 Declaration Parameters

Table 23.4-1. **DigitalInputResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the purpose of the **DigitalInputResource** object.

23.5 Properties

Table 23.5-1. **DigitalInputResource** object properties

Property	Data Type	Access	Description
IsActive [default]	bool	RO	Indicates if the digital input is currently in the asserted state
ActiveTime	uint	RO	Number of consecutive seconds **IsActive** has been true
InactiveTime	uint	RO	Number of consecutive seconds **IsActive** has been false

23.6 Usage

Example File: **UsingDigitalInputs.slogic**

In this example, a **DigitalInputResource** item is declared which activates when it receives a signal from a high-pressure switch. Upon activation, a **changestate** is triggered.

Example 23.6-1. **DigitalInputResource** object usage

```
resource digitalinputs
{
   01: HighPressureSwitch
   {
      description: "High pressure switch";
   }
}

...

task Task1
{
   state MonitorPressureSwitch
   {
      onEnter {}

      onLoop
      {
         if (HighPressureSwitch.IsActive) { changestate HandleHighPressure; }
      }

      onExit {}
   }

   state HandleHighPressure
   {
      ...
   }
}
```

24 Analog PID Controller Resource Object

24.1 General Description

`AnalogPIDControllerResource` objects are output resources. If declared, each `AnalogPIDControllerResource` corresponds directly to the physical Analog Output port with matching index number in the Scanner.

Up to 2 `AnalogPIDControllerResource` objects may be declared for use in a Scanner Logic Script program.

24.2 Required S3100 Device Configuration

`AnalogPIDControllerResource` items are objects that exist within the Scanner Logic Script program environment. They are not the same as the PID controllers that are implemented by the Analog Outputs in the Scanner host environment, and their outputs are not directly connected to any actual Scanner hardware outputs. Scanner outputs need to be configured to follow Scanner Logic Script `AnalogPIDControllerResource` outputs explicitly to manifest real-world effects.

The Scanner 3100 has 2 physical Analog Output ports that may be assigned to follow the declared resource objects. The Analog Output Mode of each Analog Output that corresponds to a declared `AnalogPIDControllerResource` item will be verified at the beginning of the program and at the start of each script execution cycle. If the Analog Output Mode specified is not "Track Scanner Logic Controller", the Scanner Logic Script will encounter a run time error and the `failState` state will be invoked. Physical Analog Output ports are configured by default as "Disabled".

Table 24.2-1. `AnalogPIDControllerResource` object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Analog Output	Analog Output Mode	Track Scanner Logic Controller

24.3 Declaring AnalogPIDControllerResource Objects

The `AnalogPIDControllerResource` declaration group begins with the keywords `resource analogpidcontrollers`, and contains a resource item declaration for each `AnalogPIDControllerResource` object to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number between `01` and `02` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no **`AnalogPIDControllerResource`** items are required, the entire resource declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 24.3-1. **`AnalogPIDControllerResource`** object declaration example

```
resource analogpidcontrollers
{
   01: SimpleAnalogPIDMyAnalogPIDControllerName1
   {
      Description : " A simple analog PID for controlling Gas Flow Rate (m3/hr)";
      Webcontrolflags : 0x0000;
      Processvar : FlowRun1_GasVolumeFlowRate;
      Pidtype : "simplepid";
      Pidaction : "reverse";
      initial_IsAutoMode : true;

      // If pidtype is set as “simple”
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
   }

   02: AnalogPID_ConstraintOverrideMyAnalogPIDControllerName2
   {
      description : "An analog PID with a constraint override ...";
      webcontrolflags : 0x0000;
      processvar : FlowRun2_GasVolumeFlowRate;
      pidtype : "constraintovrpid";
      pidaction : "reverse";
      initial_IsAutoMode : : true;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
```

```
      initial_Kd : 0.15;

      // If pidtype is set as “constraintovr”
      con_processvar : AnalogIn_StaticPressure;
      con_pidaction : "direct";
      initial_ConstraintPeriod : 4;
      initial_ConstraintRangeHigh : 50;
      initial_ConstraintRangeLow : 10;
      initial_ConstraintSetPoint : 20;
      initial_ConstraintDeadBand : 2;
      initial_ConstraintKp : 0.95;
      initial_ConstraintKi : 0.7;
      initial_ConstraintKd : 0.05;
   }
}
```

24.4 Declaration Parameters

Table 24.4-1. **AnalogPIDControllerResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the **AnalogPIDControllerResource** object
webcontrolflags	uint	Bit encoded flags which control the permissions to modify properties on the web interface
processvar	RegisterInput Resource	Selected **RegisterInputResource** object for process variable
pidtype	<option>	PID controller type Allowable values: “simplepid” | “constraintovrpid”
pidaction	<option>	PID action Allowable values: “direct” | “reverse”
con_processvar	RegisterInput Resource	Selected **RegisterInputResource** object for process variable for constraint controller
con_pidaction	<option>	PID action for constraint controller Allowable values: “direct” | “reverse”
initial_IsAutoMode	bool	Initial state of the controller
initial_Period	uint	Initial execution period of the controller
initial_RangeHigh	float	Initial process Variable high range
initial_RangeLow	float	Initial process Variable low range
initial_SetPoint	float	Initial point value set used in Auto mode

Parameter	Type Code	Description
initial_SetPointTolerance	float	Initial set point tolerance value used in Auto mode. The controller will stop adjusting if Error < SetPointTolerance
initial_SetPointDeadBand	float	Initial set point dead band value used in Auto mode. The controller will begin adjusting if Error >= SetPointDeadBand
initial_OverrideValue	float	The initial Override Value (Fraction) used in Manual Mode
initial_FailValue	float	The initial Fail Value (Fraction) used if processvar is in a fail state
initial_Kp	float	Initial proportional gain factor
initial_Ki	float	Initial integral gain factor
initial_Kd	float	Initial differential gain factor
initial_ConstraintPeriod	uint	Initial execution period of constraint override controller
initial_ConstraintRangeHigh	float	Initial constraint Variable high range
initial_ConstraintRangeLow	float	Initial constraint Variable low range
initial_ConstraintSetPoint	float	Initial point value set in Constraint Override Mode
initial_ConstraintDeadBand	float	Initial Dead Band level for Constraint Override Set Point
initial_ConstraintKp	float	Initial constraint controller proportional gain
initial_ConstraintKi	float	Initial constraint controller integral gain
initial_ConstraintKd	float	Initial constraint controller differential gain

24.5 Properties

Table 24.5-1. **AnalogPIDControllerResource** object properties

Property	Data Type	Access	Description
Output [default]	float	RO	The normalized output of the controller
Error	float	RO	The current normalized error between the Set Point and the process variable
IsAutoMode *	bool	RO	Indicates true if the controller is in auto
Period *	uint	RO	Active execution period of the controller
RangeHigh *	float	RO	Process Variable high range
RangeLow *	float	RO	Process Variable low range
SetPoint *	float	R/W	Set Point value used in Auto mode
SetPointTolerance *	float	R/W	Set point tolerance value used in Auto mode. The controller will stop adjusting if Error < SetPointTolerance
SetPointDeadBand *	float	R/W	Set point dead band value used in Auto mode. The controller will begin adjusting if Error >= SetPointDeadBand
OverrideValue *	float	R/W	Override Value (Fraction) used in Manual Mode
FailValue *	float	R/W	Fail Value (Fraction) used if ProcessVar is in a fail state
Kp *	float	R/W	Proportional gain factor
Ki *	float	R/W	Integral gain factor
Kd *	float	R/W	Differential gain factor
ConstraintPeriod *	uint	RO	Actual execution period of constraint override controller
ConstraintRangeHigh *	float	RO	Constraint Variable high range
ConstraintRangeLow *	float	RO	Constraint Variable low range
ConstraintSetPoint *	float	R/W	Set Point value in Constraint Override Mode
ConstraintDeadBand *	float	R/W	Dead Band level for Constraint Override Set Point
ConstraintKp *	float	R/W	Constraint controller proportional gain

Property	Data Type	Access	Description
`ConstraintKi *`	`float`	`R/W`	Constraint controller integral gain
`ConstraintKd *`	`float`	`R/W`	Constraint controller differential gain
`IsConstraintOverride`	`bool`	`RO`	Indicates true if the controller is in constraint override

* These properties can be initialized with **`initial_`*** parameters.

24.6 Methods

All methods listed below do not accept parameters nor return values.

Table 24.6-1. **`AnalogPIDControllerResource`** object methods

Method	Return Type	Description
`Reset()`	`void`	Resets the PID controller algorithm
`SetAutoMode()`	`void`	Sets the controller into Auto mode where the output is determined by the controller
`SetManualMode()`	`void`	Sets the controller into Manual mode where the output is determined by the OverrideValue
`ReloadInit()`	`void`	Reloads the initial values of properties specified in the declaration

24.7 Usage

Example File: **`UsingAnalogPID.slogic`**

The sample script demonstrates how to setup and use **`AnalogPIDControllerResource`** objects as Simple and Constraint Override PID controllers.

24.7.1 Simple PID

The SimplePID_Task sets up a simple PID for controlling Gas Flow Rate on Flow Run 1. The task starts in the ManualMode state where the flow value is shut. In the AutoMode state, the PID controller is set to maintain the flow rate at its setpoint. A 5 second input on PIDModeSwitch will change between the two modes.

Example 24.7-1. **`AnalogPIDControllerResource`** object usage as a simple PID

```
resource digitalinputs
{
   01: SimplePIDModeSwitch { ... }
   ...
}

resource registerinputs
{
   01: FlowRun1_GasVolumeFlowRate { ... }
   ...
}

resource analogpidcontrollers
{
   01: SimpleAnalogPID
   {
      description : "A simple analog PID for controlling Gas Flow Rate (m3/hr)";
      webcontrolflags : 0x0000;
      processvar : FlowRun1_GasVolumeFlowRate;
      pidtype : "simplepid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
   }
   ...
}

task SimplePID_Task
{
   initial state ManualMode
   {
      onEnter
      {
         SimpleAnalogPID.OverrideValue = 0.0;
         SimpleAnalogPID.SetManualMode();
      }

      onLoop
      {
         if (SimplePIDModeSwitch.ActiveTime > 5) changestate AutoMode;
      }

      onExit { }
   }
```

```
   state AutoMode
   {
      onEnter
      {
         SimpleAnalogPID.SetPoint = 125;
         SimpleAnalogPID.SetAutoMode();
      }

      onLoop
      {
         if (SimplePIDModeSwitch.ActiveTime > 5) changestate ManualMode;
      }

      onExit { }
   }
}
```

24.7.2 Constraint Override PID

The PressureOverridePID_Task sets up a PID for controlling Gas Flow Rate on Flow Run 2 with a constraint override controller. The task starts in the ManualMode state where the flow value is shut. In the AutoMode state, the PID controller is set to maintain the flow rate at its setpoint. Both states will transition to a HighPressure state where additional handling can be performed if the controller goes into constraint override. A 5-second input on ConstraintovrpidModeSwitch will change between the two modes.

Example 24.7-2. **`AnalogPIDControllerResource`** object usage as a PID with a constraint override controller

```
resource digitalinputs
{
   ...
   02: ConstraintovrpidModeSwitch { ... }
}

resource registerinputs
{
   ...
   02: FlowRun2_GasVolumeFlowRate { ... }
   03: AnalogIn_StaticPressure { ... }
}

resource analogpidcontrollers
{
   ...
   02: AnalogPID_PressureOverride
   {
      description : "An analog PID with a constraint override ...";
      webcontrolflags : 0x0000;
      processvar : FlowRun2_GasVolumeFlowRate;
      pidtype : "constraintovrpid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
```

```
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
      con_processvar : AnalogIn_StaticPressure;
      con_pidaction : "reverse";
      initial_ConstraintPeriod : 4;
      initial_ConstraintRangeHigh : 50;
      initial_ConstraintRangeLow : 10;
      initial_ConstraintSetPoint : 20;
      initial_ConstraintDeadBand : 2;
      initial_ConstraintKp : 0.95;
      initial_ConstraintKi : 0.7;
      initial_ConstraintKd : 0.05;
   }
}

task PressureOverridePID_Task
{
   initial state ManualMode
   {
      onEnter
      {
         AnalogPID_PressureOverride.ConstraintSetPoint = 30;

         AnalogPID_PressureOverride.OverrideValue = 0.0;
         AnalogPID_PressureOverride.SetManualMode();
      }

      onLoop
      {
         if (AnalogPID_PressureOverride.IsConstraintOverride)
             changestate HighPressure;
         if (ConstraintovrpidModeSwitch.ActiveTime > 5) changestate AutoMode;
      }

      onExit { }
   }

   state AutoMode
   {
      onEnter
      {
         AnalogPID_PressureOverride.ConstraintSetPoint = 125;
         AnalogPID_PressureOverride.SetAutoMode();
      }

      onLoop
      {
         if (AnalogPID_PressureOverride.IsConstraintOverride)
             changestate HighPressure;
         if (ConstraintovrpidModeSwitch.ActiveTime > 5) changestate ManualMode;
      }

      onExit { }
   }
```

```
   state HighPressure
   {
      ...
   }
}
```

25 Digital PID Controller Resource Object

25.1 General Description

`DigitalPIDControllerResource` objects are output resources. If declared, the `DigitalPIDControllerResource` may be mapped to the Digital Valve Controller in the Scanner.

Up to 1 `DigitalPIDControllerResource` object may be declared for use in a Scanner Logic Script program.

25.2 Required S3100 Device Configuration

The `DigitalPIDControllerResource` object exists within the Scanner Logic Script program environment. It is not the same as the PID controller that is implemented in the Digital Valve Controller of the Scanner host environment, and its outputs are not directly connected to any actual Scanner hardware outputs. Scanner outputs need to be configured to follow Scanner Logic Script `DigitalPIDControllerResource` outputs explicitly in order to manifest real-world effects.

The Scanner 3100 has one Digital Valve Controller that may be assigned to follow the declared `DigitalPIDControllerResource` object. The Digital Valve Controller Mode of the Digital Valve Controller will be verified at the beginning of the program and at the start of each script execution cycle. If the Digital Valve Controller Mode specified is not “Track Scanner Logic Controller”, the Scanner Logic Script will encounter a run time error and the `failState` state will be invoked. Digital Valve Controller Mode is configured by default as “Disabled”.

Table 25.2-1. DigitalPIDControllerResource object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Digital Valve Controller	Digital Valve Controller Mode	Track Scanner Logic Controller

25.3 Declaring DigitalPIDControllerResource Objects

The `DigitalPIDControllerResource` declaration group begins with the keywords `resource digitalpidcontrollers`, and contains a resource item declaration for each `DigitalPIDControllerResource` to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number (in this case, just `01`) and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no **`DigitalPIDControllerResource`** items are required, the entire resource declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 25.3-1. **`DigitalPIDControllerResource`** object declaration as a simple PID

```
resource digitalpidcontrollers
{
   01: SimpleDigitalPID
   {
      description : "A simple digital PID for controlling Gas Flow Rate (m3/hr)";
      webcontrolflags : 0x0000;
      processvar : FlowRun1_GasVolumeFlowRate;
      pidtype : "simplepid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
   }
}
```

Example 25.3-2. **`DigitalPIDControllerResource`** object declaration as a PID with a constraint override controller

```
resource digitalpidcontrollers
{
   01: DigitalPID_PressureOverride
   {
      description : "A digital PID with a constraint override ...";
      webcontrolflags : 0x0000;
      processvar : FlowRun1_GasVolumeFlowRate;
      pidtype : "constraintovrpid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
```

```
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
      con_processvar : AnalogIn_StaticPressure;
      con_pidaction : "reverse";
      initial_ConstraintPeriod : 4;
      initial_ConstraintRangeHigh : 50;
      initial_ConstraintRangeLow : 10;
      initial_ConstraintSetPoint : 20;
      initial_ConstraintDeadBand : 2;
      initial_ConstraintKp : 0.95;
      initial_ConstraintKi : 0.7;
      initial_ConstraintKd : 0.05;
   }
}
```

25.4 Declaration Parameters

Table 25.4-1. **DigitalPIDControllerResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the **DigitalPIDControllerResource** object
webcontrolflags	uint	Bit encoded flags which control the permissions to modify properties on the web interface
processvar	RegisterInput Resource	Selected **RegisterInputResource** object for process variable
pidtype	<option>	PID controller type Allowable values: “simplepid” | “constraintovrpid”
pidaction	<option>	PID action Allowable values: “direct” | “reverse”
con_processvar	RegisterInput Resource	Selected **RegisterInputResource** object for constraint source
con_pidaction	<option>	PID action for constraint controller Allowable values: “direct” | “reverse”
initial_IsAutoMode	bool	Initial state of the controller, Indicates true if the controller is in auto
initial_Period	uint	Initial execution period of the controller
initial_RangeHigh	float	Initial process Variable high range
initial_RangeLow	float	Initial process Variable low range

Parameter	Type Code	Description
initial_SetPoint	float	Initial point value set used in Auto mode
initial_SetPointTolerance	float	Initial set point tolerance value used in Auto mode. The controller will stop adjusting if Error < SetPointTolerance
initial_SetPointDeadBand	float	Initial set point dead band value used in Auto mode. The controller will begin adjusting if Error >= SetPointDeadBand
initial_OverrideValue	float	The initial Override Value (Fraction) used in Manual Mode
initial_FailValue	float	The initial Fail Value (Fraction) used if ProcessVar is in a fail state
initial_Kp	float	Initial proportional gain factor
initial_Ki	float	Initial integral gain factor
initial_Kd	float	Initial differential gain factor
initial_ConstraintPeriod	uint	Initial execution period of constraint override controller
initial_ConstraintRangedHigh	float	Initial constraint Variable high range
initial_ConstraintRangeLow	float	Initial constraint Variable low range
initial_ConstraintSetPoint	float	Initial point value set in Constraint Override Mode
initial_ConstraintDeadBand	float	Initial Dead Band level for Constraint Override Set Point
initial_ConstraintKp	float	Initial constraint controller proportional gain
initial_ConstraintKi	float	Initial constraint controller integral gain
initial_ConstraintKd	float	Initial constraint controller differential gain

25.5 Properties

Table 25.5-1. **DigitalPIDControllerResource** object properties

Property	Data Type	Access	Description
Output [default]	float	RO	The normalized output of the controller
IsAutoMode *	bool	RO	Indicates true if the controller is in auto
Period *	uint	RO	Active execution period of the controller in seconds
RangeHigh *	float	RO	Process Variable high range
RangeLow *	float	RO	Process Variable low range
SetPoint *	float	R/W	Set Point value used in Auto mode
SetPointTolerance *	float	R/W	Set point tolerance value used in Auto mode. The controller will stop adjusting if Error < SetPointTolerance
SetPointDeadBand *	float	R/W	Set point dead band value used in Auto mode. The controller will begin adjusting if Error >= SetPointDeadBand
OverrideValue *	float	R/W	Override Value (Fraction) used in Manual Mode
FailValue *	float	R/W	Fail Value (Fraction) used if ProcessVar is in a fail state
Kp *	float	R/W	Proportional gain factor
Ki *	float	R/W	Integral gain factor
Kd *	float	R/W	Differential gain factor
ConstraintPeriod *	uint	RO	Actual execution period of constraint override controller
ConstraintRangedHigh *	float	RO	Constraint variable high range
ConstraintRangeLow *	float	RO	Constraint variable low range
ConstraintSetPoint *	float	R/W	Set point value in constraint override mode
ConstraintDeadBand *	float	R/W	Dead band level for constraint override set point
ConstraintKp *	float	R/W	Constraint controller proportional gain
ConstraintKi *	float	R/W	Constraint controller integral gain

Property	Data Type	Access	Description
ConstraintKd *	float	R/W	Constraint controller differential gain
IsConstraintOverride	bool	RO	Indicates true (1) if the controller is in constraint override

* These properties can be initialized with **initial_*** parameters.

25.6 Methods

All methods listed below do not accept parameters nor return values.

Table 25.6-1. **DigitalPIDControllerResource** object methods

Method	Return Type	Description
Reset()	void	Resets the PID controller algorithm
SetAutoMode()	void	Changes PID controller to auto mode
SetManualMode()	void	Changes PID controller to manual mode
ReloadInit()	void	Reloads the initial values of properties specified in the declaration

25.7 Usage

Two sample scripts demonstrate how to setup and use **DigitalPIDControllerResource** objects as Simple and Constraint Override PID controllers.

25.7.1 Simple PID

Example File: **UsingDigitalPIDs_simple.slogic**

The SimplePID_Task sets up a simple PID for controlling Gas Flow Rate on Flow Run 1. The task starts in the ManualMode state where the flow value is shut. In the AutoMode state, the PID controller is set to maintain the flow rate at its setpoint. A 5 second input on PIDModeSwitch will change between the two modes.

Example 25.7-1. **DigitalPIDControllerResource** object usage as a simple PID

```
resource digitalinputs
{
   01: SimplePIDModeSwitch { ... }
}

resource registerinputs
{
   01: FlowRun1_GasVolumeFlowRate { ... }
```

```
}

resource digitalpidcontrollers
{
   01: SimpleDigitalPID
   {
      description : "A simple digital PID for controlling Gas Flow Rate (m3/hr)";
      webcontrolflags : 0x0000;
      processvar : FlowRun1_GasVolumeFlowRate;
      pidtype : "simplepid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
   }
}

task SimplePID_Task
{
   initial state ManualMode
   {
      onEnter
      {
         SimpleDigitalPID.OverrideValue = 0.0;
         SimpleDigitalPID.SetManualMode();
      }

      onLoop
      {
         if (SimplePIDModeSwitch.ActiveTime > 5) changestate AutoMode;
      }

      onExit { }
   }

   state AutoMode
   {
      onEnter
      {
         SimpleDigitalPID.SetPoint = 125;
         SimpleDigitalPID.SetAutoMode();
      }

      onLoop
      {
         if (SimplePIDModeSwitch.ActiveTime > 5) changestate ManualMode;
```

```
      }

      onExit { }
   }
}
```

25.7.2 Constraint Override PID

Example File: `UsingDigitalPIDs_constraintovr.slogic`

The PressureOverridePID_Task sets up a PID for controlling Gas Flow Rate on Flow Run 1 with a constraint override controller. The task starts in the `ManualMode` state where the flow value is shut. In the `AutoMode` state, the PID controller is set to maintain the flow rate at its setpoint. Both states will transition to a HighPressure state where additional handling can be performed if the controller goes into constraint override. A 5-second input on ConstraintovrpidModeSwitch will change between the two modes.

Example 25.7-2. **`DigitalPIDControllerResource`** object usage as a PID with a constraint override controller

```
resource digitalinputs
{
   01: ConstraintovrpidModeSwitch { ... }
}

resource registerinputs
{
   01: FlowRun1_GasVolumeFlowRate { ... }
   02: AnalogIn_StaticPressure { ... }
}

resource digitalpidcontrollers
{
   01: DigitalPID_PressureOverride
   {
      description : "A digital PID with a constraint override ...";
      webcontrolflags : 0x0000;
      processvar : FlowRun1_GasVolumeFlowRate;
      pidtype : "constraintovrpid";
      pidaction : "reverse";
      initial_IsAutoMode : false;
      initial_Period : 1;
      initial_RangeHigh : 150;
      initial_RangeLow : 50;
      initial_SetPoint : 100;
      initial_SetPointTolerance : 1;
      initial_SetPointDeadBand : 1.2;
      initial_OverrideValue : 0.7;
      initial_FailValue : 0.5;
      initial_Kp : 0.9;
      initial_Ki : 0.7;
      initial_Kd : 0.15;
```

```
      con_processvar : AnalogIn_StaticPressure;
      con_pidaction : "reverse";
      initial_ConstraintPeriod : 4;
      initial_ConstraintRangeHigh : 50;
      initial_ConstraintRangeLow : 10;
      initial_ConstraintSetPoint : 20;
      initial_ConstraintDeadBand : 2;
      initial_ConstraintKp : 0.95;
      initial_ConstraintKi : 0.7;
      initial_ConstraintKd : 0.05;
   }
}

task PressureOverridePID_Task
{
   initial state ManualMode
   {
      onEnter
      {
         DigitalPID_PressureOverride.ConstraintSetPoint = 30;

         DigitalPID_PressureOverride.OverrideValue = 0.0;
         DigitalPID_PressureOverride.SetManualMode();
      }

      onLoop
      {
         if (DigitalPID_PressureOverride.IsConstraintOverride)
            changestate HighPressure;

         if (ConstraintovrpidModeSwitch.ActiveTime > 5) changestate AutoMode;
      }

      onExit { }
   }

   state AutoMode
   {
      onEnter
      {
         DigitalPID_PressureOverride.ConstraintSetPoint = 125;
         DigitalPID_PressureOverride.SetAutoMode();
      }

      onLoop
      {
         if (DigitalPID_PressureOverride.IsConstraintOverride)
            changestate HighPressure;

         if (ConstraintovrpidModeSwitch.ActiveTime > 5) changestate ManualMode;
      }

      onExit { }
   }
```

```
   state HighPressure
   {
      ...
   }
}
```

26 Digital Output Resource Object

26.1 General Description

`DigitalOutputResource` objects are output resources that can interface to Scanner hardware digital outputs. If declared, each `DigitalOutputResource` object corresponds directly to the physical DIO port with the matching index number in the Scanner. At the end of each script execution cycle, the script interpreter engine publishes the `DigitalOutputResource` output values.

Up to 6 `DigitalOutputResource` objects that may be declared for use in a Scanner Logic Script program.

26.2 Required S3100 Device Configuration

The Scanner 3100 has 6 physical DIO ports that may be assigned to the declared `DigitalOutputResource` objects. The Digital I/O Mode of each DIO port that corresponds to declared `DigitalOutputResource` objects will be verified at the beginning of the program and at the start of each script execution cycle. If the Digital I/O Mode specified is not "Track Scanner Logic Controller", the Scanner Logic Script will encounter a run time error and the `failState` state will be invoked. Physical DIO ports are configured by default as "Input Mode".

Note: `DigitalInputResource` and `DigitalOutputResource` items both map to the same set of physical DIO ports and are mutually exclusive functions. A `DigitalInputResource` object and `DigitalOutputResource` object defined at the same index will generate a run time error and the `failState` state will be invoked.

Table 26.2-1. DigitalOutputResource object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Digital In/Out	Digital I/O Mode	Track Scanner Logic Controller

26.3 Declaring DigitalOutputResource Objects

The `DigitalOutputResource` declaration group begins with the keywords `resource digitaloutputs`, and contains a resource item declaration for each `DigitalOutputResource` object to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number between `01` and `06` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no **`DigitalOutputResource`** items are required, the entire declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 26.3-1. **`DigitalOutputResource`** object declaration example

```
resource digitaloutputs
{
   01: FlowValve
   {
      description: "Flow Valve actuated when Output is on";
      initial_IsActive: false;
      initial_FollowAlarm: 0;
      initial_Period: 2;
      initial_Duration: 0;
   }
   02: Burner
   {
      description: "Switch that activates the burners";
      initial_IsActive: true;
      initial_FollowAlarm: 0;
      initial_Period: 50;
      initial_Duration: 1;
   }

   ...

   06: PressureReleaseValue
   {
      description: "Vents line on high pressure alarm";
      initial_IsActive: false;
      initial_FollowAlarm: 18;
      initial_Period: 2;
      initial_Duration: 0;
   }
}
```

26.4 Declaration Parameters

Table 26.4-1. **`DigitalOutputResource`** object declaration parameters

Parameter	Type Code	Description
`description`	<str256>	A user-provided string describing the purpose of the **`DigitalOutputResource`** object

Parameter	Type Code	Description
`initial_IsActive`	`bool`	In the initial state of the device, indicates if the digital output is currently in the asserted state. Only applies to outputs not programed to follow an alarm.
`initial_FollowAlarm`	`uint`	Initial Follow Alarm configuration setting. Control output state with Alarm IsActive state: 0=Manual, 1=Alarm1, ..., 32= Alarm32
`initial_Period`	`uint`	Initial Period configuration setting. Pulse out period in number of 10ms ticks
`initial_Duration`	`uint`	Initial Duration configuration setting. Pulse out active time in number of 10ms ticks

26.5 Properties

Table 26.5-1. **`DigitalOutputResource`** object properties

Property	Data Type	Access	Description
`IsActive *` `[default]`	`bool`	`RO`	Indicates if the digital output is currently in the asserted state. If pulsing mode activated with `AddPulseAccum()` method, output will indicate `IsActive` = 0.
`FollowAlarm *`	`uint`	`R/W`	Control output state with Alarm `IsAsserted` state: 0=Manual, 1=Alarm1, ..., 32= Alarm32
`Period *`	`uint`	`R/W`	Pulse out period in number of 10ms ticks
`Duration *`	`uint`	`R/W`	Pulse out active time in number of 10ms ticks. A zero pulse duration will produce a continuous pulse of width equal to the integer portion of the pulse accumulator time 10ms.
`ActiveTime`	`uint`	`RO`	Number of consecutive seconds `IsActive` has been true
`InactiveTime`	`uint`	`RO`	Number of consecutive seconds `IsActive` has been false
`PulseLoad`	`uint`	`R/W`	Incremental accumulation value to be added to accumulator with the `AddPulseAccumulation()` method. Fractional values can be loaded into the pulse accumulator.

* These properties can be initialized with **`initial_*`** parameters.

26.6 Methods

All methods listed below do not accept parameters nor return values.

Table 26.6-1. **DigitalOutputResource** object methods

Method	Return Type	Description
`Activate()`	`void`	Assert the **DigitalOutputResource** object to the Not-Normal state. Clears internal pulsing accumulator. Only applies to outputs not programmed to follow an alarm.
`Deactivate()`	`void`	Deassert the **DigitalOutputResource** object and sets the Normal state. Clears internal pulsing accumulator. Only applies to outputs not programmed to follow an alarm.
`ClearPulseAccum()`	`void`	Clears the internal pulsing accumulator and resets the output to the not active state. Only applies to outputs not programmed to follow an alarm.
`AddPulseAccum()`	`void`	Loads any positive accumulation value in the **PulseLoad** property into the internal pulse accumulator. Any integer portion of the result within the accumulator will be translated into pulses. Only applies to outputs not programmed to follow an alarm.
`ReloadInit()`	`void`	Reloads the init values for **IsActive**, **FollowAlarm**, **Period**, and **Duration**

26.7 Usage

Example File: `UsingDigitalOutputs.slogic`

A `DigitalOutputResource` object can be used to activate a flow valve when a signal from a high pressure switch is received.

Example 26.7-1. **`DigitalOutputResource`** object usage example

```
resource digitaloutputs
{
   01: FlowValve
   {
      description: "Flow Valve actuated when Output is on";
      initial_IsActive: false;
      initial_FollowAlarm: 0;
      initial_Period: 2;
      initial_Duration: 0;
   }
}

resource digitalinputs
{
   01: HighPressureSwitch { ... }
}

...

task Task1
{
   state ControlFlowValve
   {
      onEnter { FlowValve.Deactivate(); }

      onLoop
      {
         if (HighPressureSwitch.IsActive) { changestate HighPRessureState; }
      }

      onExit { FlowValve.Activate(); }
   }

   state HighPressureState
   {
      ...
   }
}
```

27 Alarm Resource Object

27.1 General Description

`AlarmResource` objects are output resources that can interface with the Scanner host environment.

`AlarmResource` objects exist within the Scanner Logic Script program environment. They are separate and independent from the Device Alarms implemented in the Scanner host environment.

At the end of each script execution cycle, the script interpreter engine publishes output of the `AlarmResource` objects. `AlarmResource` states are viewable in the Scanner Logic Script status page of the web interface. Additionally, Scanner Logic Script `AlarmResource` registers will be available to the Scanner host environment for logging, triggering Device Alarms, or activating Digital Outputs, etc. Setting an `AlarmResource` object active and inactive causes a record to be created in the Scanner alarm archive.

Up to 32 `AlarmResource` objects may be declared for use in a Scanner Logic Script program.

27.2 Required S3100 Device Configuration

There are no device configuration requirements for `AlarmResource` objects.

27.3 Declaring AlarmResource Objects

The `AlarmResource` declaration group begins with the keywords `resource alarms`, and contains a resource item declaration for each `AlarmResource` object to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number between `01` and `32` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no `AlarmResource` items are required, the entire declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 27.3-1. **AlarmResource** object declaration example

```
resource alarms
{
   01: GasSupplyAlarm
   {
      description: "Alarm to indicates gas supply is low";
      initial_HoldOffDelay: 0;
      initial_IsAsserted: false;
   }
   02: HighPressureAlarm
   {
      description: "Signaled for excessive pipe pressures";
      initial_HoldOffDelay: 0;
      initial_IsAsserted: false;
   }

   ...

   32: HeaterFailure
   {
      description: "Heater inactive for 10 or more minutes";
      initial_HoldOffDelay: 600;
      initial_IsAsserted: false;
   }
}
```

27.4 Declaration Parameters

Table 27.4-1. **AlarmResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the **AlarmResource** object
initial_IsAsserted	bool	The initial state of the alarm, set upon program start or after invoking the ReloadInit () method
initial_HoldOffDelay	uint	The initial value for the **HoldoffDelay**, set upon program start or after invoking the **ReloadInit()** method

27.5 Properties

Table 27.5-1. **AlarmResource** object properties

Property	Data Type	Access	Description
IsActive [default]	bool	RO	State of the alarm determined by IsAsserted && (HoldOffCount == HoldOffDelay)
IsAsserted *	bool	RO	State set by the Assert() and Deassert() methods
HoldOffDelay *	uint	R/W	The configured hold-off delay in seconds before the alarm becomes active
HoldOffTime	uint	RO	The current seconds timer of the hold-off delay
ActiveTime	uint	RO	Number of consecutive seconds IsActive has been true
InactiveTime	uint	RO	Number of consecutive seconds IsActive has been false

* These properties can be initialized with **initial_*** parameters.

27.6 Methods

All methods listed below do not accept parameters nor return values.

Table 27.6-1. **AlarmResource** object methods

Method	Return Type	Description
Assert()	void	Assert the alarms object. If the alarm is not already active, it will become active at the beginning of the next execution cycle or when the HoldOffDelay has elapsed.
Deassert()	void	Deassert the alarms object. If the alarm is not already not active, IsActive will become false at the beginning of the next execution cycle.
ReloadInit()	void	Reloads the init values for IsActive, FollowAlarm, Period, and Duration

27.7 Usage

Example File: `UsingAlarms.slogic`

In this example, the program waits in an **onLoop** block until a digital input connected to a high-pressure switch is activated. This triggers a **changestate** and asserts an alarm during the **onExit** execution.

Example 27.7-1. **AlarmResource** object usage example

```
resource alarms
{
   01: HighPressureAlarm
   {
      description: "Alarm to indicate high pressure threshold";
      initial_IsAsserted: false;
      initial_HoldOffDelay: 4;
   }
}

resource digitalinputs
{
   01: HighPressureSwitch { ... }
}

...

task Task1
{
   initial state ControlFlowValve
   {
      onEnter { HighPressureAlarm.Deassert(); }

      onLoop
      {
         if (HighPressureSwitch.IsActive) { changestate HighPressureState; }
      }

      onExit
      {
         HighPressureAlarm.Assert();
      }
   }

   state HighPressureState
   {
      ...
   }
}
```

28 Timer Resource Object

28.1 General Description

`TimerResource` objects are a system resource of the Scanner Logic Script program environment. `TimerResource` objects increment their count by 1 at the start of each program execution cycle if they are actively running.

Up to 8 `TimerResource` objects may be declared for use in a Scanner Logic Script program.

These user-defined timers are general purpose. Note that many of the resource objects have their own integrated timers that automatically measure and count useful periods (e.g. `DigitalInputResource.ActiveTime, State.ActiveTime, State.TotalEntryCount`).

28.2 Required S3100 Device Configuration

There are no device configuration requirements for `TimerResource` objects.

28.3 Declaring TimerResource Objects

The `TimerResource` declaration group begins with the keywords `resource timers`, and contains a resource item declaration for each `TimerResource` object to be used in the program within a pair of open and close braces. Each resource item declaration consists of a unique resource index number between `01` and `08` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused resource items may be omitted from the resource declaration group. If no `TimerResource` items are required, the entire resource declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 28.3-1. `TimerResource` object declaration example

```
resource timers
{
   01: TotalProductionTime
   {
      description: "Total time spent in production state";
      initial_Time: 0;
      initial_IsActive: false;
```

```
   }
   02: WellTestTimer
   {
      description: "Time spent in all Well Test states";
      initial_Time: 0;
      initial_IsActive: false;
   }

   ...

   08: SLOGICTotalOperationTime
   {
      description: "Alarm to indicate high pressure threshold";
      initial_Time: 0;
      initial_IsActive: true;
   }
}
```

28.4 Declaration Parameters

Table 28.4-1. **TimerResource** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the **TimerResource** object
initial_Time	uint	The initial number of seconds (the number of program execution cycles) since the timer was started
initial_IsActive	bool	The initial state of the timer where active is running. **IsActive** state is programmed by the **Start()** and **Stop()** methods after initialization.

28.5 Properties

Table 28.5-1. **TimerResource** object properties

Property	Data Type	Access	Description
Time [default]	uint	R/W	The number of seconds (the number of program execution cycles) since the timer was started
IsActive *	bool	RO	Indicates if the timer is active running or stopped (as programed by the Start and Stop methods).

* These properties can be initialized with **initial_*** parameters.

28.6 Methods

All methods listed below do not accept parameters nor return values.

Table 28.6-1. **`TimerResource`** object methods

Method	Return Type	Description
`Start()`	`void`	Start the timer running; the Time property will increment by 1 at the start of each execution cycle
`Stop()`	`void`	Stop the timer; the Time property will contain the number of seconds since the timer was started
`Reset()`	`void`	Reset the Time property to zero; the timer continues to run
`StopReset()`	`void`	Stop the timer and reset the Time property to zero
`ReloadInit()`	`void`	Reloads the init values for Time

28.7 Usage

Example File: **`UsingTimers.slogic`**

In this example, a **`TimerResource`** object waits for one minute before triggering a **`changestate`**. The timer is stopped **`onExit`**.

Example 28.7-1. **`TimerResource`** object usage example

```
resource timers
{
   01: Timer1
   {
      description: "One minute threshold";
      initial_Time: 0;
      initial_IsActive: false;
   }
}

...

task Task1
{
   state OneMinuteTimer
   {
      onEnter { Timer1.Start(); }

      onLoop
      {
         if (Timer1.Time >= 60) { changestate State2; }
      }

      onExit { Timer1.Stop(); }
```

```
    }


    state State2
    {
        ...
    }
}
```

29 Logic Script Register Objects

29.1 General Description

A **register** is a named container in which to store a **`Value`**. Some register types allow values to be passed between the script program and the Scanner host environment. A register's name, specified in the register item declaration, follows the rules described for identifiers (see *Chapter 10*). All registers have a **`Value`** property of type **`float`** that can be read and written. Assignment statements can assign values to registers either by using the **`Value`** property or by using the name of the register alone, since **`Value`** is the default property of registers.

Input values are obtained via **`ConfigurationRegister`** objects (see *Chapter 30*) and **`MaintenanceRegister`** objects (see *Chapter 31*). The Scanner 3100 maintains the values of the Configuration Registers and Maintenance Registers in nonvolatile memory, since they are user configuration values. Their values will persist throughout power cycling of the device or restarting the Scanner Logic Script program after entering the **`abortState`** or the **`failState`**.

Output values are stored in **`HoldingRegister`** objects (see *Chapter 32*) or **`AccumulationRegister`** objects (see *Chapter 33*). These output register values are accessible by the Scanner host environment. The values are not stored in nonvolatile memory like the Configuration Registers and Maintenance Registers, and will lose their values if the program is restarted.

The Accumulation Registers are like the Holding Registers, but they store totals of incremental accumulations that have been added to them rather than a single value at a time. The Scanner 3100 maintains current period and previous period totals for the values in Accumulation Registers.

There are no user-defined variables in Scanner Logic Script. The storage of intermediate calculation results or temporary values can be accomplished using **`WorkingRegister`** objects (see *Chapter 34*), which serve the purpose of global variables. They retain their values as program execution moves between states or in and out of subroutines.

Example 29.1-1. Using Logic Script Register objects

```
MyHoldRegName1 = MyConfigRegName1 + Math.Sin(MyWorkRegName1);

if (MyMaintRegName1 >= MyMaintRegName2)
{
   ...
}
```

Scanner Logic Script registers always hold values as a `float` type. The type of a register cannot be changed during script program execution. You cannot assign values to a register that do not match the type defined for the register `Value` property.

Each register must be assigned to a specific register number (01, 02, 03, …). This number is used for identifying the registers in Scanner systems that are not aware of the user defined name. For example, when accessing `configuration` registers via a serial protocol, they will have a default label within the protocol map as `m32_LM_CREG_1_Config_RN` (where N is the register number). Similarly, changes to the `configuration` registers are captured by the Scanner in the User Event Archive and will be identified in the `m32_LM_CREG_1_Config_RN` format.

When defining registers, register group labels can be created and assigned to each register. Register group labels allow the user to organize registers into functional groups to help guide the Web Interface operator when configuring and maintaining the active Scanner Logic Script program. On the Web Interface register pages, register group labels will be sorted alphabetically and displayed with all registers to which they have been assigned. Quick access links at the top of the Web Interface page will allow for convenient navigation to a register group.

The Web Interface provides a SCANNER LOGIC SCRIPT PROGRAM REPORT on the Logic Controller Status page. This report summarizes the resources defined by the Logic Script program and tabulates register numbers with user defined names and descriptions for reference.

29.2 Register Object Types

Table 29.2-1. Types of register objects

Register Type	Usage	Qty	Description
`ConfigurationRegister`	`Input`	32	Editable on web interface Configuration registers page. Security access level > Configuration Editor.
`MaintenanceRegister`	`Input`	32	Editable on web interface Maintenance registers page. Security access level > Calibration Tech.
`HoldingRegister`	`Output`	64	Viewable on web interface Holding registers page. Publish register to Scanner systems (archive, display, input source, etc).

Register Type	Usage	Qty	Description
`AccumulationRegister`	`Output`	16	Viewable on web interface Accumulation registers page. Publish register to Scanner systems (archive, display, input source, etc). Periodic data. 64 bit precision.
`WorkingRegister`	`Internal`	64	Logic Script global variables.

30 Configuration Register Object

30.1 General Description

`ConfigurationRegister` objects are used to allow user input into the Scanner Logic Script program at run time. These registers will appear in a Configuration Registers webpage in the Logic Controller section of the web interface to allow users to view and change the `ConfigurationRegister` values. The Configuration Registers webpage is accessible by users with Configuration Editor access and higher. User changes to `ConfigurationRegister` values are available to the program at the moment the changes are saved.

The register declaration for this register type allows the specification of a measurement units category, unit type, and rate. The selection of the measurement units category will determine the set of valid selections for unit type and rate scalar. The script execution engine will convert the values between the device system units and the specified units within the Scanner Logic Script program as required.

Up to 32 `ConfigurationRegister` objects may be declared for use in a Scanner Logic Script program.

Each `ConfigurationRegister` must be assigned to a specific register number (01 – 32). When accessing these registers through Scanner systems that are unaware of the user defined name (e.g. serial protocol, User Event Archive), these registers are identified using their defined number with the format `m32_LM_CREG_1_Config_RN` (where N is the register number).

Each `ConfigurationRegister` may also optionally be assigned a register group label. Register group labels allow the user to organize registers into functional groups to help guide the Web Interface operator when configuring and maintaining the active Scanner Logic Script program.

30.2 Declaring ConfigurationRegister Objects

The `ConfigurationRegister` declaration group begins with the keywords `registers configuration`, and contains a registers item declaration for each `ConfigurationRegister` to be used in the program within a pair of open and close braces. Each registers item declaration consists of a unique register index number between `01` and `32` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused registers items may be omitted from the registers declaration group. Omitting the group parameter will place the register into the "Ungrouped" register group label. If no **`ConfigurationRegister`** items are required, the entire declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 30.2-1. **`ConfigurationRegister`** object declaration example

```
registers configuration
{
   01: TubingConstant
   {
      group: “Station Parameters”;
      description: "A unitless constant used in calculating pressure test value";
      category: "No Units";
      units: "";
      initial_Value: 0;
   }
   02: GasSupplyMin
   {
      group: “Station Parameters”;
      description: "Minimum level for gas supply pressure before alarm is asserted";
      category: "Static Pressure (gauge)";
      units: "psig";
      initial_Value: 0;
   }

   ...

   32: LiquidFlowRateMinimum
   {
      group: “Alarm Settings”;
      description: "Threshold flow rate (m3/hr) for entering LowFlowState";
      category: "Liquid Volume";
      units: "m3";
      rate: "/hr";
      initial_Value: 25;
   }
}
```

30.3 Declaration Parameters

Table 30.3-1. **`ConfigurationRegister`** object declaration parameters

Parameter	Type Code	Description
`group`	`<str64>`	A user-provided string label that will be used to group related registers in the Web Interface.
`description`	`<str256>`	A user-provided string describing the configuration register object
`category`	`<category>`	The units category of the S3100 configuration register. See Measurement Categories section. See *Chapter 51* for more information.
`units`	`<unit>`	Numerator of the measurement unit desired for the configuration value, and denominator if required for the specified measurement units category. Must be specified if category is not None. See *Chapter 51* for more information.
`rate`	`<rate>`	Rate scalar unit desired for the configuration value. See *Chapter 51* for more information.
`initial_Value`	`float`	Initial value of the configuration register object

30.4 Properties

Table 30.4-1. **`ConfigurationRegister`** object properties

Property	Data Type	Access	Description
`Value * [default]`	`float`	`R/W`	Holds the value of the configuration register object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

* These properties can be initialized with **`initial_`*** parameters.

30.5 Methods

All methods listed below do not accept parameters nor return values.

Table 30.5-1. **ConfigurationRegister** object methods

Method	Return Type	Description
`ReloadInit()`	`void`	Reloads the initial value for Value

30.6 Usage

Configuration Registers are ideal for programing operational parameters that vary at each installation. Using these registers will often avoid having to alter the program, recompile, and upload the program to the Scanner.

Example File: **UsingConfigurationRegisters.slogic**

In this example, the program is waiting for a pressure drop to trigger a **changestate**. The pressure live reading is received from Analog Input 1 and the **configuration** register allows a user with an access level of Configuration Editor or greater to designate the desired threshold pressure.

Example 30.6-1. **ConfigurationRegister** object usage example

```
registers configuration
{
   01: PressureThreshold
   {
      group: “Station Parameters”;
      description: "This is a pressure threshold variable";
      category: "Static Pressure (gauge)";
      units: "psig";
      initial_Value: 100.0;
   }
}

resource registerinputs
{
   01: RealTimePressure { ... }
}

...

task Task1
{
   state WaitForPressureDrop
   {
      onEnter {}

      onLoop
      {
```

```
         if (RealTimePressure < PressureThreshold)
         {
            changestate HandlePressureDrop;
         }
      }

      onExit {}
   }

   state HandlePressureDrop
   {
      ...
   }
}
```

31 Maintenance Register Object

31.1 General Description

`MaintenanceRegister` objects are used to allow user input into the Scanner Logic program at run time. These registers will appear in a Maintenance Registers webpage in the Logic Controller section of the web interface to allow users to view and change the `MaintenanceRegister` values. The Maintenance Registers webpage is accessible by users with Calibration Tech access and higher. User changes to `MaintenanceRegister` values are available to the program at the moment the changes are saved.

The register declaration for this register type allows the specification of a measurement units category, unit type, and rate. The selection of the measurement units category will determine the set of valid selections for unit type and rate scalar. The script execution engine will convert the values between the device system units and the specified units within the Scanner Logic Script program as required.

Up to 32 `MaintenanceRegister` objects may be declared for use in a Scanner Logic Script program.

Each register must be assigned to a specific register number (01 – 32). When accessing these registers through Scanner systems that are unaware of the user defined name (e.g. serial protocol, User Event Archive), these registers are identified using their defined number with the format `m32_LM_MREG_1_Config_RN` (where N is the register number).

Each `MaintenanceRegister` may also optionally be assigned a register group label. Register group labels allow the user to organize registers into functional groups to help guide the Web Interface operator when configuring and maintaining the active Scanner Logic Script program.

31.2 Declaring MaintenanceRegister Objects

The `MaintenanceRegister` declaration group begins with the keywords `registers maintenance`, and contains a registers item declaration for each `MaintenanceRegister` to be used in the program within a pair of open and close braces. Each registers item declaration consists of a unique register index number between `01` and `32` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused registers items may be omitted from the registers declaration group. Omitting the group parameter will place the register into the "Ungrouped" register group label. If no **`MaintenanceRegister`** items are required, the entire declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 31.2-1. **`MaintenanceRegister`** object declaration example

```
registers maintenance
{
   01: ProductionHoldTime
   {
      group: "Station Operating Parameters";
      description: "Production hold time (minutes)";
      category: "No Units";
      units: "";
      initial_Value: 60;
   }
   02: PressureThreshold
   {
      group: "Station Operating Parameters";
      description: "Threshold for pressure test value for dewatering decision";
      category: "Static Pressure (gauge)";
      units: "psig";
      initial_Value: 100.0;
   }

   ...

   32: LiquidFlowRateMinimum
   {
      group: "Station Alarm Parameters";
      description: "Threshold flow rate (m3/hr) for entering LowFlowState";
      category: "Liquid Volume";
      units: "m3";
      rate: "/hr";
      initial_Value: 25;
   }
}
```

31.3 Declaration Parameters

Table 31.3-1. **MaintenanceRegister** object declaration parameters

Parameter	Type Code	Description
group	<str64>	A user-provided string label that will be used to group related registers in the Web Interface.
description	<str256>	A user-provided string describing the maintenance register object
category	<category>	The unit category of the S3100 maintenance register. See *Measurement Categories*. See *Chapter 51* for more information.
units	<unit>	Numerator of the measurement unit desired for the maintenance value, and denominator if required for the specified measurement units category. Must be specified if category is not None. See *Chapter 51* for more information.
rate	<rate>	Rate scalar unit desired for the maintenance value. See *Chapter 51* for more information.
initial_Value	float	Initial value of the maintenance register object

31.4 Properties

Table 31.4-1. **MaintenanceRegister** object properties

Property	Data Type	Access	Description
Value * [default]	float	R/W	Holds the value of the maintenance register object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

* These properties can be initialized with **initial_*** parameters.

31.5 Methods

Table 31.5-1. **MaintenanceRegister** object methods

Method	Return Type	Description
`ReloadInit()`	`void`	Reloads the init value for Value

31.6 Usage

Maintenance Registers are ideal for programing operational parameters that require adjustments over time as conditions of an installation change. Using these registers will often avoid having to alter the program, recompile, and upload the program to the Scanner.

Example File: **UsingMaintenanceRegisters.slogic**

In this example, a technician with Calibration Tech access level may set a desired production hold off timer and pressure threshold using two separate **MaintenanceRegister** items. A dewatering process is triggered when the **ProductionTimer** exceeds the user set **ProductionHoldTime** and the **CasingPressure** is above the **PressureThreshold**.

Example 31.6-1. **MaintenanceRegister** object usage example

```
registers maintenance
{
   01: ProductionHoldTime
   {
      group: “Station Operating Parameters”;
      description: "Production hold time (minutes)";
      category: "No Units";
      initial_Value: 60;
   }
   02: PressureThreshold
   {
      group: “Station Operating Parameters”;
      description: "This is a pressure threshold variable";
      category: "Static Pressure (gauge)";
      units: "psig";
      initial_Value: 100.0;
   }
}

resource registerinputs
{
   01: CasingPressure { ... }
}

resource timers
{
```

```
   01: ProductionTimer { ... }
}

...

task Task1
{
   state NormalProduction
   {
      onEnter { ProductionTimer.Start(); }

      onLoop
      {
         if (ProductionTimer / 60 > ProductionHoldTime)
         {
            if (CasingPressure > PressureThreshold)
            {
               changestate Dewatering;
            }
         }
      }

      onExit { ProductionTimer.StopReset(); }
   }

   state Dewatering
   {
      ...
   }
}
```

32 Holding Register Object

32.1 General Description

`HoldingRegister` objects are used to allow a Scanner Logic program to create values usable by Scanner systems at run time.

The register declaration for this register type allows the specification of a measurement units category, unit type, and rate. The selection for the measurement units category will determine the set of valid selections for unit type and rate scalar. The script execution engine will convert the values between the device system units and the specified units within the Scanner Logic Script program as required.

Holding Registers publish their values to the Scanner host environment at the end of each script execution cycle for use in archiving, displaying on the LCD, as inputs for Flow Runs and Calculators, and driving Analog and Digital outputs. The selection of a `HoldingRegister` object's measurement units category will determine how the `HoldingRegister` value can be used by the Scanner. For example, a Flow Run pressure source will only allow the selection of `holding` registers that are a category of Static Pressure (gauge) or Static Pressure (absolute).

These registers will appear in a Holding Registers webpage in the Logic Controller section of the web interface to allow users to view the `HoldingRegister` values. The Holding Registers webpage is accessible by all users with valid access.

Up to 64 `HoldingRegister` objects may be declared for use in a Scanner Logic Script program.

Each register must be assigned to a specific register number (01 – 64). When accessing these registers through Scanner systems that are unaware of the user defined name (e.g. serial protocol, User Event Archive), these registers are identified using their defined number with the format `m32_LM_HREG_1_Config_RN` (where N is the register number).

Each `HoldingRegister` may also optionally be assigned a register group label. Register group labels allow the user to organize registers into functional groups to help guide the Web Interface operator when configuring and maintaining the active Scanner Logic Script program.

32.2 Declaring HoldingRegister Objects

The **HoldingRegister** declaration group begins with the keywords **registers holding**, and contains a registers item declaration for each **holding** register to be used in the program within a pair of open and close braces. Each registers item declaration consists of a unique register index number between 01 and 64 and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : "", uint : 0, float : 0.0, bool : false). Unused registers items may be omitted from the registers declaration group. Omitting the group parameter will place the register into the "Ungrouped" register group label. If no **HoldingRegister** items are required, the entire registers declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 32.2-1. **HoldingRegister** object declaration example

```
registers holding
{
   01: ProductionHoldCount
   {
      group: "Station Operating Values";
      description: "Production hold time count (minutes)";
      category: "No Units";
      units: "";
      initial_Value: 60;
   }
   02: MaximumPressure
   {
      group: "Station Operating Values";
      description: "The maximum pressure observed.";
      category: "Static Pressure (gauge)";
      units: "psig";
      initial_Value: 100.0;
   }

   ...

   64: LiquidFlowRateMinimum
   {
      group: "Station Alarm Values";
      description: "Minimum flow rate (m3/hr) while in LowFlowState";
      category: "Liquid Volume";
      units: "m3";
      rate: "/hr";
      initial_Value: 25;
   }
}
```

32.3 Declaration Parameters

Table 32.3-1. **HoldingRegister** object declaration parameters

Parameter	Type Code	Description
group	<str64>	A user-provided string label that will be used to group related registers in the Web Interface.
description	<str256>	A user-provided string describing the holding register object
category	<category>	The unit category of the S3100 holding register. See *Section 51.2*. See *Chapter 51* for more information.
units	<unit>	Numerator of the measurement unit desired for the holding value, and denominator if required for the specified measurement units category. If category is anything other than "None, the value must be specified. See *Chapter 51* for more information.
rate	<rate>	Rate scalar unit desired for the holding value. See *Chapter 51* for more information.
initial_Value	float	Initial value of the holding register object.

32.4 Properties

Table 32.4-1. **HoldingRegister** object properties

Property	Data Type	Access	Description
Value * [default]	float	R/W	Holds the value of the holding register object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

* These properties can be initialized with **initial_*** parameters.

32.5 Methods

Table 32.5-1. **HoldingRegister** object methods

Method	Return Type	Description
`ReloadInit()`	`void`	Reloads the init value for Value

32.6 Usage

Example File: **UsingHoldingRegisters.slogic**

This sample script receives flow rate values from two separate **RegisterInputResource** items. The initial state calculates the **CasingFlowRate** and stores the result into a Holding Register. The calculated **CasingFlowRate** stored within the Holding Register is viewable on the Holding Registers webpage on the web interface by users with any valid access and is available for use by other Scanner systems outside of the Logic Script program. **CasingFlowRate** is available for display on the LCD, inclusion in archives, and use in any Scanner input requiring a Liquid Volume category.

Example 32.6-1. **HoldingRegister** object usage example

```
registers holding
{
   01: CasingFlowRate
   {
      group: “Well Site Operating Values”;
      description: "Calculated casing flow rate";
      category: "Liquid Volume";
      units: "m3";
      rate: "/sec";
      initial_Value: 0;
   }
}

resource registerinputs
{
   01: AcutalFlowRate { ... }
   02: TubingFlowRate { ... }
}

...

task Task1
{
   state CalculateCasing
   {
      onEnter {}

      onLoop
```

```
      {
         CasingFlowRate = ActualFlowRate – TubingFlowRate;
         changestate NormalProduction;
      }

      onExit {}
   }

   state NormalProduction
   {
      ...
   }
```

33 Accumulation Register Object

33.1 General Description

`AccumulationRegister` objects allow a Scanner Logic program to accumulate incremental values into registers that are usable by Scanner systems at run time. These registers implement a 64-bit internal accumulator and store sets of current period and previous period values.

The register declaration for this register type allows the specification of a measurement units category, unit type, and rate. The selection for the measurement units category will determine the set of valid selections for unit type and rate scalar. The script execution engine will convert the values between the device system units and the specified units within the Scanner Logic Script program as required.

Accumulation Registers publish a set of values to the Scanner host environment at the end of each script execution cycle to other Scanner systems for use in archiving, displaying on the LCD, as inputs for Flow Runs and Calculators, and driving Analog and Digital outputs. Each `AccumulationRegister` will publish a Daily Total, an Interval Total, and a Triggered Total value for both the current and previous period in addition to a Grand Total and an Incremental Change. The selection of an `AccumulationRegister` object's measurement units category will determine how the Accumulation register's published values can be used by the Scanner. For example, a Flow Run pulse input source will only allow the selection of Accumulation Register values that are a category of Uncorrected Gas Volume, Uncorrected Liquid Volume, or Mass.

These registers will appear in an Accumulation Registers webpage in the Logic Controller section of the web interface to allow users to view the `AccumulationRegister` values. The Accumulation Registers webpage is accessible by all users with valid access.

Up to 16 `AccumulationRegister` objects that may be declared for use in a Scanner Logic Script program.

Each register must be assigned to a specific register number (01 – 16). When accessing these registers through Scanner systems that are unaware of the user defined name (e.g. serial protocol, User Event Archive), these registers are identified using their defined number using the formatting in the following table where N is the register number.

Each **AccumulationRegister** may also optionally be assigned a register group label. Register group labels allow the user to organize registers into functional groups to help guide the Web Interface operator when configuring and maintaining the active Scanner Logic Script program.

Table 33.1-1. **AccumulationRegister** object published values

Published Value	Host Environment Availability	Tag Name
Grand Total	Archive Display Alarms Analog Out Digital Out Calculators	m32_LM_AREG_N_GrandTotal
Incremental Change	Calculators	m32_LM_AREG_N_IncrementalChange
Daily Total	Display Alarms Digital Out	m32_LM_AREG_N_DailyTotal
Interval Total	Display	m32_LM_AREG_N_IntervalTotal
Triggered Total	Display	m32_LM_AREG_N_TriggeredTotal
Previous Daily Total	Archive Display Calculators	m32_LM_AREG_N_PreviousDailyTotal
Previous Interval Total	Display	m32_LM_AREG_N_PreviousIntervalTotal
Previous Triggered Total	Display	m32_LM_AREG_N_PreviousTriggeredTotal

33.2 Declaring AccumulationRegister Objects

The **AccumulationRegister** declaration group begins with the keywords **registers accumulation**, and contains a registers item declaration for each **AccumulationRegister** to be used in the program within a pair of open and close braces. Each registers item declaration consists of a unique register index number between `01` and `16` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused registers items may be omitted from the registers declaration group. Omitting the group parameter will place the register into the "Ungrouped"

register group label. If no **AccumulationRegister** items are required, the entire registers declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 33.2-1. **AccumulationRegister** object declaration example

```
registers accumulation
{
   01: StationTotalMass
   {
      group: “Station Totals”;
      description: "Accumulated difference in Mass flow between FR1 and FR2";
      category: "Mass";
      units: "kg";
      initial_Value: 0;
   }
   02: SiteTotalEnergy
   {
      group: “Station Totals”;
      description: "Accumulated energy across all flow runs";
      category: "Energy";
      units: "J";
      initial_Value: 0;
   }

   ...

   16: PumpTestVolume
   {
      group: “Test Results Totals”;
      description: "Total fluid volume used during pump tests";
      category: "Liquid Volume";
      units: "m3";
      initial_Value: 0;
   }
}
```

33.3 Declaration Parameters

Table 33.3-1. **AccumulationRegister** object declaration parameters

Parameter	Type Code	Description
group	<str64>	A user-provided string label that will be used to group related registers in the Web Interface.
description	<str256>	A user-provided string describing the accumulation register object

Parameter	Type Code	Description
category	<category>	The unit category of the S3100 accumulation register. See *Measurement Categories*. See *Chapter 51* for more information.
units	<unit>	Numerator of the measurement unit desired for the accumulation value, and denominator if required for the specified measurement units category. If the category is anything but "None," a value must be specified. See *Chapter 51* for more information.
initial_Value	float	Initial value of the accumulation register object.

33.4 Properties

Table 33.4-1. **AccumulationRegister** object properties

Property	Data Type	Access	Description
Value * [default]	float	RO	Holds the value of the accumulation register object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property
LoadAccum	float	R/W	Value of the incremental accumulation added to accumulator with the **AddAccumulation()** method

* These properties can be initialized with **initial_*** parameters.

33.5 Methods

Table 33.5-1. **AccumulationRegister** object methods

Method	Return Type	Description
ClearAccumulation()	void	Clears the internal accumulator
AddAccumulation()	void	Loads the value of incremental accumulation in the LoadAccum property into the internal accumulator
ReloadInit()	void	Reloads the init value for Value

33.6 Usage

Example File: **UsingAccumulationRegisters.slogic**

This example uses an Accumulation Register (**StationTotal**) to store the difference in gas mass flow between FlowRun1 and FlowRun2. When the **StationTotal** exceeds 100 kg of accumulated difference in either direction, change to a new state and handle that condition. The **StationTotal** Accumulation Register is published for use in other Scanner systems, and may be setup for display on the LCD, inclusion in archiving, or used as an input source.

Example 33.6-1. **AccumulationRegister** object usage example

```
registers accumulation
{
   01: StationTotalMass
   {
      group: “Station Totals”;
      description: "Difference in Gas Mass between FlowRun1 and FlowRun2";
      category: "Mass";
      units: "kg";
      initial_Value: 0;
   }
}

resource registerinputs
{
   01: FlowRun1_GasMass_Incremental
   {
      tagname: "FR1: HAccum: Gas Mass Incremental Change";
      tagcode: "m32_FC_FR_1_HoldingAccum_GasMassIncrementalChange";
      category: "Mass";
      units: "kg";
   }
   02: FlowRun2_GasMass_Incremental
   {
      tagname: "FR2: HAccum: Gas Mass Incremental Change";
      tagcode: "m32_FC_FR_2_HoldingAccum_GasMassIncrementalChange";
      category: "Mass";
      units: "kg";
   }
}


task Task1
{
   initial state RecordStationTotals
   {
      onEnter { StationTotalMass.ClearAccumulation(); }

      onLoop
      {
         StationTotalMass.LoadAccum = FlowRun1_GasMass_Incremental –
```

```
                                    FlowRun2_GasMass_Incremental;

         StationTotalMass.AddAccumulation();

         if (StationTotalMass >= 100 || StationTotalMass <= -100)
            changestate HandleAccumulatedDifference;
      }

      onExit { }
   }

   state HandleAccumulatedDifference { ... }
}
```

34 Working Register Object

34.1 General Description

WorkingRegister objects are internal registers to a Scanner Logic program and provide a means to create temporary value storage which can be used to store intermediate calculation values or to pass values between areas of code within the program. Working Registers are accessible by code in all states and subroutines, and take the place of local variables.

These registers are not available to the Scanner host environment or the web interface.

Up to 64 **WorkingRegister** objects that may be declared for use in a Scanner Logic Script program.

34.2 Declaring working registers

The **WorkingRegister** declaration group begins with the keywords **registers working**, and contains a registers item declaration for each **WorkingRegister** to be used in the program within a pair of open and close braces. Each registers item declaration consists of a unique register index number between `01` and `64` and a user defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : `""`, uint : `0`, float : `0.0`, bool : `false`). Unused registers items may be omitted from the registers declaration group. If no **WorkingRegister** items are required, the entire registers declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 34.2-1. **WorkingRegister** object declaration example

```
registers working
{
   01: PressureTestValue
   {
      description: "Global variable storing the computed pressure test value (psia)";
      initial_Value: 0;
   }
   02: PumpTestIterations
   {
      description: "Countdown of pump tests left to run in batch";
      initial_Value: 15;
```

```
   }

   ...

   64: PressureTestStdDev
   {
      description: "Standard deviation for pressure test 10 sample rolling average.";
      initial_Value: 0;
   }
}
```

34.3 Declaration Parameters

Table 34.3-1. **WorkingRegister** object declaration parameters

Parameter	Type Code	Description
description	<str256>	A user-provided string describing the working register object
initial_Value	float	Initial value of the working register object

34.4 Properties

Table 34.4-1. **WorkingRegister** object properties

Property	Data Type	Access	Description
Value * [default]	float	R/W	Holds the value of the working register object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

* These properties can be initialized with **initial_*** parameters.

34.5 Methods

Table 34.5-1. **WorkingRegister** object methods

Method	Return Type	Description
ReloadInit()	void	Reloads the init value for Value

34.6 Usage

Working Registers allow users to create temporary or interim values that the Scanner Logic Script program can call upon for use in other statements. Used as a scratch pad register space, the Working Registers can provide simplification and efficiency to a program. When the result of a complex statement is required in multiple locations within a program, consider using Working Registers to store the result so that it may be reused. Since subroutines in the current version of Scanner Logic Script do not

support return values, Working Registers are useful for holding the result of a calculation performed in a subroutine for use by the code that called the subroutine.

Example File: `UsingWorkingRegisters.slogic`

In this example, the `TankFillVolume` is calculated in the `onEnter` code block and stored in a Working Register. This value can now be used in other parts of the program to derive further properties which are stored in Holding Registers for publishing.

Example 34.6-1. **`WorkingRegister`** object usage example

```
registers working
{
   01: TankFillVolume
   {
      description: "Computed tank volume filled";
      initial_Value: 0;
   }
}

resource registerinputs
{
   01: TankLevel { ... }
}

registers configuration
{
   01: TankRadius { ... }
   02: TankVolume { ... }
   03: FluidDensity {... }
}

registers holding
{
   01: TankFillFraction { ... }
   02: FluidWeight { ... }
}

...

task Task1
{
   state CalculateVesselStats
   {
      onEnter
      {
         TankFillVolume = Math.PI * Math.Pow(TankRadius,2) * TankLevel;
      }

      onLoop
      {
         TankFillFraction = TankFillVolume / TankVolume;
```

```
         FluidWeight = TankFillVolume * FluidDensity;

         changestate PublishStats;
      }

      onExit {}
   }

   state PublishStats
   {
      ...
   }
}
```

35 Task Object

35.1 General Description

The executable code of a Scanner Logic Script program is contained within the **state** objects of a **task** object.

When declaring a task, it must contain at least one complete state. An **initial** state must also be declared.

The state machines in each task appear to run in "parallel". In actuality, the code in the current state of each task runs in a round robin fashion at each execution cycle, which occur once per second.

A Scanner Logic Script **program** must declare one **task** and may declare up to 4 **tasks** in total.

35.2 Declaring Tasks

The **Task** object declaration begins with the keyword **task**, followed by a user defined identifier and contains one or more **State** object declarations within a pair of open and close braces. One of the states must be declared as the entry point of the task with the **initial** keyword.

Parser errors will be generated for improper or incomplete declarations.

Example 35.2-1. **Task** object declaration example

```
task MyTaskName1
{
   initial state State1
   {
      onEnter {}
      onLoop {}
      onExit {}
   }

   state State2
   {
      onEnter {}
      onLoop {}
      onExit {}
   }
}
```

35.3 Properties

Table 35.3-1. **Task** object properties

Property	Data Type	Access	Description
`TotalEntryCount`	`uint`	`RO`	Total number of times the state is entered since method
`CurrentState`	`uint`	`RO`	The current state index being executed by the task

35.4 Methods

Table 35.4-1. **Task** object methods

Method	Return Type	Description
`ResetTotals()`	`void`	Clears the `TotalActiveTime` and `TotalEntryCount` values
`RestartExecution()`	`void`	Re-initializes target task on next execution cycle, current execution cycle proceeds to completion

35.5 Usage

Example File: `UsingTasks.slogic`

The sample script contains two tasks each with multiple states. Entering debug mode and stepping through the program demonstrates the concurrent execution of the tasks' execution blocks.

Example 35.5-1. **Task** object usage example

```
task Task1
{
   initial state StateName1
   {
      onEnter { }
      onLoop  { changestate StateName2; }
      onExit  { }
   }

   state StateName2
   {
      onEnter { }
      onLoop  { changestate StateName3; }
      onExit  { }
   }

   state StateName3
   {
      onEnter { }
```

```
      onLoop  { changestate StateName1; }
      onExit  { }
   }
}

task Task2
{
   initial state StateName1
   {
      onEnter { }
      onLoop  { changestate StateName2; }
      onExit  { }
   }

   state StateName2
   {
      onEnter { }
      onLoop  { changestate StateName1; }
      onExit  { }
   }
}
```

36 State Object

36.1 General description

All declared states must contain an `onEnter`, `onLoop` and `onExit` block. One of the states within each task must be declared as the `initial` state.

36.2 Declaring States

The `State` object declaration begins with the keyword `state`, followed by a user defined identifier and contains one `onEnter`, one `onLoop` and one `onExit` block within a pair of open and close braces. A state may be declared as the entry point of its containing task with the `initial` modifier keyword. The `onEnter`, and `onExit` blocks may be configured to create event records by declaring them with the `logged` modifier keyword.

Parser errors will be generated for improper or incomplete declarations.

Example 36.2-1. **`State`** object declaration example

```
state State1
{
   onEnter logged
   {
   }

   onLoop
   {
   }

   onExit logged
   {
   }
}
```

36.3 Properties

Table 36.3-1. **`State`** object properties

Property	Data Type	Access	Description
`IsActive`	`bool`	`RO`	Indicates if target state is currently active
`ActiveTime`	`uint`	`RO`	Number of consecutive seconds `IsActive` has been true for target state
`TotalActiveTime`	`uint`	`RO`	Total number of seconds `IsActive` has been true since method `ResetTotalCounts` was called

Property	Data Type	Access	Description
`TotalEntryCount`	`uint`	`RO`	Total number of times the state is entered since method

36.4 Methods

Table 36.4-1. **`State`** object methods

Method	Return Type	Description
`ResetTotals()`	`void`	Clears the `TotalActiveTime` and `TotalEntryCount` values

36.5 Modifiers

Table 36.5-1. **`State`** object modifiers

Modifier	Description
`logged`	Marks the `onEnter` or `onExit` blocks to be logged by the event log
`initial`	Marks declared state as initial state for its task

36.6 Usage

Example File: `UsingStates.slogic`

This sample script shows the use of states as a means to organize program logic. In this scenario, the program remains in the `NormalConditions` state when normal conditions are present and transitions to the `LowTemp` state that controls a burner valve when the system temperature drops too low. The script returns to the `NormalConditions` state when the appropriate temperature has been reached. The conditions checks also ensure a minimum amount of time has been spent in each state before allowing the active state to change.

Example 36.6-1. **State** object usage example

```
resource registerinputs
{
   01: TempReading { ... }
}

resource digitaloutputs
{
   01: BurnerValve { ... }
}

task Task1
{
   initial state NormalConditions
   {
      onEnter { }

      onLoop
      {
         if (TempReading < 10 && NormalConditions.ActiveTime > 60)
            changestate LowTemp;
      }

      onExit { }
   }

   state LowTemp
   {
      // Entry into this state will create an event in the archive.
      onEnter logged { BurnerValve.Activate(); }

      onLoop
      {
         if (TempReading > 15 && LowTemp.ActiveTime > 10)
            changestate NormalConditions;
      }

      onExit { BurnerValve.Deactivate(); }
   }
}
```

37 System State Objects

There are two system states that are required to be declared within the `program` object, the `abortState` and the `failState`. These states come after the task declarations and before the subroutine declarations, within a set of `#region - #endregion` preprocessor directives which form a collapsible region named System Declarations.

Example 36.6-1. System Declarations region

```
#region System Declarations
...

abortState
{
   onEnter { }
   onLoop { }
}

failState
{
   onEnter { }
   onLoop { }
}

#endregion
```

38 Abort State Object

38.1 General description

The `abortState` is one of two required system state objects that must be declared within the `program` object. When an operator sends an abort signal, the Scanner Logic Script runs to the end of the current script execution cycle. At the start of the next execution cycle the program transitions to the `abortState` where it executes the `onEnter` block once before executing the `onLoop` block once a second until the program is restarted. Note that the `onExit` block of the state in which the abort is detected does not execute.

38.2 Declaring abortState

The `Abort State` object declaration begins with the keyword `abortstate` and contains one `onEnter` and one `onLoop` block within a pair of open and close braces. The `onEnter` block may be configured to create event records by declaring it with the `logged` modifier keyword.

Parser errors will be generated for improper or incomplete declarations, or if the Abort State object declaration is omitted.

Example 38.2-1. **`Abort State`** object declaration example

```
abortstate
{
   onEnter logged
   {
   }

   onLoop
   {
   }
}
```

38.3 Entering abortState

At any point during program execution, an abort signal may sent to trigger the transition to the `abortState`. There are several methods for signaling the abort of a Scanner Logic Script program:

Table 38.3-1. Triggering the **Abort State** in a S3100 device

Entry Point	Description
Web Interface	Navigate to #LCControl page on the web interface (Navigation Bar -> Control -> Scanner Logic Controller -> Program Control) and click the "Emergency Stop" button. Refer to the *Scanner 3100 Web Interface User Manual* for more details.
Hardware switch via Digital Input	Attach a hardware switch to a Digital I/O port. Configure the Digital I/O Mode as "Special Function Input Mode" and select the Special Function Selection "Abort Scanner Logic Script program". Refer to the *Scanner 3100 Web Interface User Manual* for more details.
Modbus Register Command	Send **800001** to Modbus Command Register **75**. Refer to the *Modbus Protocol Manual* for your device for more details.

38.4 Usage

Example File: **UsingabortState.slogic**

In this example script, after sending an abort command two **UserEventRecords** are created containing the day and month when the abort command was sent. Two **holding** registers are also used, one displays the tank level at the time of the abort command being sent, the other is continually updated in the **abortState onLoop** displaying a live reading of fluid level.

Example 38.4-1. **Abort State** object usage example

```
resource registerinputs
{
   01: TankLevel
   {
      description: "Percentage of tank filled with fluid";
      ...
   }
}

registers configuration
{
   01: TankMaxHeight { ... }
}

registers holding
{
   01: FluidLevel { ... }
   02: CurrentFluidLevel { ... }
}

... // Send abort command during runtime execution

abortState
{
   onEnter
```

```
   {
      UserEventRecord1.Value = RealTime.Day;
      UserEventRecord2.Value = RealTime.Month;
      UserEventRecord1.CreateEventRecord();
      UserEventRecord2.CreateEventRecord();

      FluidLevel = (TankLevel / 100) * TankMaxHeight;
   }

   onLoop
   {
      CurrentFluidLevel = (TankLevel / 100) * TankMaxHeight;
   }
}
```

39 Fail State Object

39.1 General Description

The `failState` is one of two required system state objects that must be declared within the `program` object. When a run time error occurs, Scanner Logic Script stops execution at the current location and automatically transitions to the `failState` where it executes the `onEnter` block once before executing the `onLoop` block once a seconds until the program is restarted.

39.2 Declaring failState

The `Fail State` object declaration begins with the keyword `failstate` and contains one `onEnter` and one `onLoop` block within a pair of open and close braces. The `onEnter` block may be configured to create event records by declaring it with the `logged` modifier keyword.

Parser errors will be generated for improper or incomplete declarations, or if the Fail State object declaration is omitted.

Example 39.2-1. **`Fail State`** object declaration example

```
failstate
{
   onEnter logged
   {
   }

   onLoop
   {
   }
}
```

39.3 Entering failState

During program execution, errors may occur due to resource misconfiguration, fail states in Scanner inputs, or problems caused by statements in the program itself.

The most commonly encountered errors are resource validation errors. These errors are due to associated Scanner resources entering an invalid or unexpected state during program execution.

Table 39.3-1. Resource Validation Errors

Logic Script Resource	<N>	Validation Error(s)
`FlowRun<N>`	1,2	Flow Run <N> Accumulation Control is not configured for SLogic control. Flow Run <N> Is disabled or is failing.
`FlowArchive`	N/A	Creation of Partial Archive Records is disabled and will prevent SLogic function.
`analogpidcontrollers`	1, 2	Analog Output <N> is not configured for SLogic control.
`digitalpidcontrollers`	N/A	Digital Valve 1 is not configured for SLogic control.
`digitaloutputs`	1, 2, …, 6	Digital Output <N> is not configured for SLogic control.
`digitalinputs`	1, 2, …, 6	Digital Input <N> is not enabled and is required by installed SLogic program.
`Display`	N/A	Local LCD Display mode is not configured for SLogic control.
`registerinputs`	1, 2, …, 32	Register Input <N>'s associated Scanner register is not a valid selection. Register Input <N> category does not match the category of the associated Scanner register. Register Input <N>'s associated Scanner register is Disabled or in Fail state.

39.4 Usage

Example File: **`UsingfailState.slogic`**

This example will cause a run time error and enter the **`failstate`**. The **`onEnter`** for the **`failState`** is **`logged`** and the program asserts an alarm to warn the user. It also deactivates a valve switch and starts a timer in order to record how long the program has been in failure mode.

Example 39.4-1. **`Fail State`** object usage example

```
resource digitaloutputs
{
   01: ValveControl { ... }
}

resource alarms
{
   01: Alarm1 { ... }
}

resource timers
{
   01: FailTimer { ... }
```

```
}

task Task1
{
   // Note: To observe a run time error, ensure that Digital I/O 1 "Digital I/O Mode"
      is not set to "Track Scanner Logic Controller".
   initial state State1
   {
      onEnter { }

      onLoop { }

      onExit { }
   }
}

failState
{
   onEnter logged
   {
      ValveControl.Deactivate();
      Alarm1.Assert();
      FailTimer.Start();
   }

   onLoop { }
}
```

40 Subroutines

40.1 General Description

A **subroutine** is a collection of statements that can be invoked by name. Subroutines are usable from any state in any task and have access to the global collection of **register** and **resource** objects. The **return** statement may be used at any point within a **subroutine** to stop its execution and return to the location where the **subroutine** was called to resume execution.

In the current version of Scanner Logic Script, a **subroutine** does not accept arguments or directly return values. The only return type allowed in a **subroutine** declaration is **void**, indicating that there is no return value. Nevertheless, a **subroutine** can access any of the resource or registers objects in the program scope. to use as arguments or indirectly provide as return values for the calling task.

A **subroutine** may not call itself or any other **subroutine**.

A Scanner Logic Script **program** may declare up to 100 **subroutines**.

40.2 Declaring Subroutines

The Subroutine declaration begins with the keywords **void subroutine** followed by a user defined identifier and contains zero or more statements within a pair of open and close braces.

Parser errors will be generated for improper or incomplete declarations.

Example 40.2-1. **Subroutine** declaration example

```
void subroutine SubroutineName1
{

}
```

40.3 Usage

Example File: **UsingSubroutines.slogic**

The example script showcases how subroutines can be invoked and used in different blocks when the same code needs to be re-used in different contexts. This performs common actions (for example, collects some state transition logic in, loading a single subroutine GetNextStateNumber().

Example 40.3-1. **Subroutine** usage example

```
registers holding
{
    01: StateNumber { ... }
    02: NextStateNumber { ... }
}

task Task1
{
    initial state State1
    {
        onEnter { StateNumber = 1; }

        onLoop
        {
            GetNextStateNumber();

            if (NextStateNumber == 2)
               changestate StateName2;
        }

        onExit { }
    }

    state State2
    {
        onEnter { StateNumber = 2; }

        onLoop
        {
            GetNextStateNumber();

            if (NextStateNumber == 1)
               changestate StateName1;
        }

        onExit { }
    }
}

void subroutine GetNextStateNumber()
{
   if (StateNumber == 1)
   {
      NextStateNumber = 2;
      return;
   }

   NextStateNumber = 1;
}
```

41 System Objects

41.1 General Description

System objects are built-in instances of global system object types. Their object names are pre-defined in a Scanner Logic Script program, and you cannot re-use their names as an identifier for resource or register objects, tasks, states, or subroutines.

System objects provide access to obtain information from the Scanner 3100 device, to affect some Scanner 3100 options, and to initiate certain actions in the Scanner 3100.

41.2 System Object Types

The table below summarizes the types of system objects available from user code in the program. The object identifier names, descriptions of object properties and methods, and usage examples are contained in the subsequent chapters.

Table 41.2-1. Types of system objects

Object Type	Qty	Description	Ref.
`System_RealTime`	1	Accesses the real-time clock of the Scanner host environment	*Ch. 42*
`System_FlowRun`	2	Allows control of flow run accumulation	*Ch. 43*
`System_FlowArchive`	1	Allows creation of partial records	*Ch. 44*
`System_TriggeredArchive`	1	Allows publishing of triggered archive records	*Ch. 45*
`System_UserEventRecord`	8	Allows publishing of user event records	*Ch. 46*
`System_PrintedTicket`	16	Allows printing pre-defined tickets or reports	*Ch. 47*
`System_Display`	1	Allows control of current display group on LCD display	*Ch. 48*
`System_Math`	1	Provides mathematical constants and functions	*Ch. 49*

42 Real Time System Object

42.1 General Description

The global **RealTime** object is an instance of the **System_RealTime** object type. It provides access to the current calendar and time values in the real-time clock of the Scanner host environment.

42.2 Properties

Table 42.2-1. **System_RealTime** object properties

Property	Data Type	Access	Description
Year	uint	RO	Current year
Month	uint	RO	1 = January, ..., 12 = December
Day	uint	RO	Current Day of the month
Hour	uint	RO	0 = 12:00AM, ..., 23 = 23:00PM
Minute	uint	RO	Current minute (0-59)
Second	uint	RO	Current second (0-59)
DayOfWeek	uint	RO	1 = Sunday, ..., 7 = Saturday

42.3 Usage

Example File: **UsingRealTime.slogic**

In this sample script, the **RealTime** object is used to check which day of the month it is. In this case the program is set to generate an archive report on the 15th of every month

Example 42.3-1. **System_RealTime** object usage example

```
task Task1
{
   initial state CheckDayOfMonth
   {
      onEnter { }

      onLoop
      {
         if (RealTime.Day == 15) { changestate CreateArchiveReport; }
      }

      onExit { }
   }
```

```
   state CreateArchiveReport
   {
      ...
   }
}
```

43 Flow Run System Object

43.1 General Description

The global `FlowRun1` and `FlowRun2` objects are instances of the `System_FlowRun` object type. They provide access to the accumulation control of the Flow Runs in the Scanner host environment.

43.2 Required S3100 Device Configuration

For these objects to be used, the respective Flow Runs must be enabled and configured to be under the control of Scanner Logic Script in the Web Interface.

The Scanner 3100 has 2 Flow Runs. Flow Run 1 must be configured to use the `FlowRun1` object. Flow Run 2 must be configured to use the `FlowRun2` object. For each Flow Run System object defined, the Enable Flow Run option and the Accumulation Control Mode will be verified at the beginning of the program and at the start of each script execution cycle. If the Enable Flow Run option specified is not "Yes", or the Accumulation Control Mode option specified is not "Controlled by Scanner Logic Script program", the Scanner Logic Script will encounter a run time error and the `failState` State will be invoked. The default configuration for Enable Flow Run is "Yes" for Flow Run 1 and "No" for Flow Run 2. The default configuration for Accumulation Control Mode is "Always Accumulate" for both Flow Runs.

Table 43.2-1. **`System_FlowRun`** object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Flow Run	Accumulation Control Mode	Controlled by Scanner Logic Script program
Flow Run	Enable Flow Run	Yes

43.3 Properties

Table 43.3-1. **`System_FlowRun`** object properties

Property	Data Type	Access	Description
`IsAccumulating`	`bool`	`RO`	Indicates whether the `System_FlowRun` object is currently accumulating. This is a read-only property, and the accumulating state is changed with the `EnableAccumulation()` and `DisableAccumulation()` methods. The default value is false.

Property	Data Type	Access	Description
AccumulatingTime	uint	RO	Number of consecutive seconds IsAccumulating has been true. <does this go to zero once not accumulting, or does it maintain the last count and start back at zero once accumulating again?>
NotAccumulatingTime	uint	RO	Number of consecutive seconds IsAccumulating has been false

43.4 Methods

Table 43.4-1. **System_FlowRun** object methods

Method	Return Type	Description
EnableAccumulation()	void	Start flow run accumulating flow
DisableAccumulation()	void	Stop flow run accumulating flow

43.5 Usage

Example File: **UsingFlowRun.slogic**

This program demonstrates how to use the flow run system object. When the program starts it ensures only one flow run is actively accumulating. If flow run 1 is active it takes priority and flow run 2 is disabled.

Example 43.5-1. **System_FlowRun** object usage example

```
registers holding
{
   01: InactiveTime { ... }
}

task Task1
{
   initial state FlowRun_Initialize
   {
      onEnter
      {
         if (FlowRun1.IsAccumulating) { FlowRun2.DisableAccumulation(); }
         if (!FlowRun1.IsAccumulating)
         {
            InactiveTime = FlowRun1.NotAccumulatingTime;
            FlowRun2.EnableAccumulation();
         }
      }

      onLoop
      {
```

```
         if (FlowRun1.IsAccumulating) { changestate FlowRun1_Accumulate; }
         if (FlowRun2.IsAccumulating) { changestate FlowRun2_Accumulate; }
      }

      onExit { }
   }

   state FlowRun1_Accumulate
   {
      ...
   }

   state FlowRun2_Accumulate
   {
      ...
   }
}
```

44 Flow Archive System Object

44.1 General Description

The global `FlowArchive` object is an instance of the `System_FlowArchive` object type. It provides access to the Flow Archive system in the Scanner host environment and allows a Scanner Logic Script program to signal the creation of partial archive records.

44.2 Required S3100 Device Configuration

For this object to be used, the creation of partial records must be enabled in the Web Interface.

The Scanner 3100 has 2 Flow Archives that must be configured to use the `FlowArchive` object. The Enable Partial Records option will be verified at the beginning of the program and at the start of each script execution cycle. If the Enable Partial Records option specified is not "Yes", the Scanner Logic Script will encounter a run time error and the `failState` State will be invoked. Enable Partial Records is configured by default as "Yes".

Table 44.2-1. **`System_FlowArchive`** object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Flow Archive 1 Flow Archive 2	Enable Partial Records	Yes

44.3 Methods

Table 44.3-1. **`System_FlowArchive`** object methods

Method	Return Type	Description
`CreatePartialRecords()`	`void`	Signals the Scanner archive system to create partial archive records

44.4 Usage

Example File: `UsingFlowArchives.slogic`

In this sample script, a single state waits for a digital input high to create a partial record. A cooldown timer starts to limit the number of records created.

Example 44.4-1. **`System_FlowArchive`** object usage example

```
resource digitalinputs
{
   01: ArchiveButton { ... }
}

resource timers
{
   01: CooldownTimer { ... }
}

registers configuration
{
   01: TotalTestTime { ... }
}

task Task1
{
   initial state WaitForTestStart
   {
      onEnter { CooldownTimer.Start(); }

      onLoop
      {
         if ((ArchiveButton.ActiveTime == 1) && (CooldownTimer.Time > 9))
         {
            FlowArchive.CreatePartialRecords();
            CooldownTimer.Reset();
         }
      }

      onExit { }
   }
}
```

45 Triggered Archive System Object

45.1 General Description

The global **`TriggeredArchive`** object is an instance of the **`System_TriggeredArchive`** object type. It provides access to the Triggered Archive system in the Scanner 3100 host environment and allows a Scanner Logic Script program to create triggered archive records.

45.2 Required S3100 Device Configuration

The Scanner Triggered Archive system must be configured for manual triggering for the **`TriggeredArchive`** object to function. The Archiving Mode of the Triggered Archive system will be verified at the beginning of the program and at the start of each script execution cycle. If the Archiving Mode specified is not "Log Manually (via serial protocol, web page, or Scanner Logic Script program)", the Scanner Logic Script will encounter a run time error and the **`failState`** State will be invoked. Archiving Mode is configured by default as "Disabled".

Table 45.2-1. **`System_Display`** object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Triggered Archive	Archiving Mode	Log Manually (via serial protocol, web page, or Scanner Logic Script program)

45.3 Properties

Table 45.3-1. **`System_TriggeredArchive`** object properties

Property	Data Type	Access	Description
`NumberOfRecords`	`uint`	`RO`	Number of Triggered Archive records stored by the Scanner

45.4 Methods

Table 45.4-1. **System_TriggeredArchive** object methods

Method	Return Type	Description
PublishRecord()	void	Signals the Scanner archive system to trigger the creation of a Triggered Archive record
ResetArchive()	void	Signals the Scanner archive system to reset the Triggered Archive, returning the NumberOfRecords to zero

45.5 Usage

Example File: **UsingTriggeredArchives.slogic**

This sample script waits for the ArchiveButton to be pressed; when pressed a **changestate** occurs. The program will then proceed to publish a **TriggeredArchive** record every second for a Test Time defined by the user (default set to 4 hours).

Example 45.5-1. **System_TriggeredArchive** object usage example

```
resource digitalinputs
{
   01: ArchiveButton { ... }
}

resource timers
{
   01: CurrentTestTime { ... }
}

registers configuration
{
   01: TotalTestTime
   {
      description: "User set test length (in minutes)";
      category: “No Units”;
      initial_Value: 240;
   }
}

task Task1
{
   initial state WaitForTestStart
   {
      onEnter {}

      onLoop
      {
```

```
        if (ArchiveButton.Activetime > 1) { changestate RunWellTest; }
    }

    onExit {}
}

state RunWellTest
{
    onEnter { CurrentTestTime.Start(); }

    onLoop
    {
        TriggeredArchive.PublishRecord();
        if (CurrentTestTime >= TotalTestTime * 60) { changestate WaitForTestStart; }
    }

    onExit { CurrentTestTime.StopReset(); }
}
```

46 User Event Record System Object

46.1 General Description

The global **UserEventRecord1...UserEventRecord8** objects are instances of the **System_UserEventRecord** object types. They provide access to the event logging system in the Scanner 3100 and allow creation of user-generated event records. These events are general purpose and are viewable in the Event Archive using the Scanner Data Manager software. When invoking the creation of the event archive record, the user can provide a floating-point value which will be available in the record.

46.2 Properties

Table 46.2-1. **System_UserEventRecord** object properties

Property	Data Type	Access	Description
Value	float	R/W	The event/alarm value added to the event archive record

46.3 Methods

Table 46.3-1. **System_UserEventRecord** object methods

Method	Return Type	Description
CreateEventRecord()	void	Creates a labeled event archive record containing with the current Value

46.4 Usage

Example File: **UsingUserEventRecords.slogic**

This sample program showcases user event records. Three user event records are declared: User event records 1 and 2 capture and record analog input 1 and 2 every minute; User event record 8 increments every second and then creates a record.

Example 46.4-1. **System_UserEventRecord** object usage example

```
resource registerinputs
{
   01: AnalogIn1 { ... }
   02: AnalogIn2 { ... }
}

resource timers
{
   01: EventTimer { ... }
```

```
}

task Task1
{
   initial state User_Event_Regulation
   {
      onEnter
      {
         EventTimer.Start();
         UserEventRecord8.Value = 0;
      }

      onLoop
      {
         if (EventTimer >= 60)
         {
            UserEventRecord1.Value = AnalogIn1.Value;
            UserEventRecord2.Value = AnalogIn2.Value;

            UserEventRecord1.CreateEventRecord();
            UserEventRecord2.CreateEventRecord();

            EventTimer.Reset();
         }

         UserEventRecord8.Value++;
         UserEventRecord8.CreateEventRecord();
      }

      onExit { }
   }
}
```

47 Printed Ticket System Object[1]

47.1 General Description

The global **PrintedTicket1...PrintedTicket16** objects are instances of the **System_PrintedTicket** object type. They provide access to stored ticket definitions in the Scanner 3100.

47.2 Methods

Table 47.2-1. **System_PrintedTicket** object methods

Method	Return Type	Description
Print()	void	Signals the print of the ticket.

[1] Implemented in Logic Script but not yet supported by Scanner 3100. Planned for a future release.

48 Display System Object

48.1 General Description

The global **Display** object is an instance of the **System_Display** object type. It provides access to the Scanner 3100 LCD Display system and allows a Scanner Logic Script program to access and control the current Display Group being presented on the Local LCD Display.

A Display Group is a user configurable collection of register values for presentation on the Local LCD Display. A user can design multiple Display Groups to be displayed in a sequence using the Web Interface. Each Display Group is numbered, and allows for the definition of Display Selections and their Display Position within the group. There are several Display Modes which determine the behavior and timing of display updates and Display Group changes.

48.2 Required S3100 Device Configuration

The Display system in the Scanner host environment must be configured for control by Scanner Logic Script for this object to function. The Message Display Mode of the Display system will be verified at the beginning of the program and at the start of each script execution cycle. If the Message Display Mode specified is not "Controlled by Scanner Logic Script program", the Scanner Logic Script will encounter a run time error and the **failState** State will be invoked. Message Display Mode is configured by default as "Grouped Display Selections".

Table 48.2-1. **System_Display** object required S3100 device configuration

Scanner Resource	Configuration Parameter	Requirement
Display	Message Display Mode	Controlled by Scanner Logic Script program

48.3 Properties

Table 48.3-1. **System_Display** object properties

Property	Data Type	Access	Description
CurrentDisplayGroup	uint	RO	The current Display Group being displayed
SetValue	uint	R/W	The Display Group to be set with the **SetDisplayGroup()** method (1 - 32)

48.4 Methods

Table 48.4-1. **System_Display** object methods

Method	Return Type	Description
`AdvanceDisplayGroup()`	`void`	Advances the current Display Group to the next available group
`SetDisplayGroup()`	`void`	Sets the current Display Group to the `SetValue`

48.5 Usage

Example File: **UsingDisplaySystemObjects.slogic**

In this sample script, the first 11 display groups have been defined on the web interface. The program will cycle through them all at 10 seconds a display group until it reaches display group 11. It then resets the display group to 1, waits 10 seconds, and reruns the cycle.

Example 48.5-1. **System_Display** object usage example

```
resource timers
{
   01: DisplayTimer
   {
      description: "A timer to count the display time.";
      ...
   }
}

task Task1
{
   initial state Initialize_Display_Groups
   {
      onEnter
      {
         Display.SetValue = 1;
         Display.SetDisplayGroup();
         DisplayTimer.Start();
      }

      onLoop { changestate Cycle_First_10_Display_Groups; }

      onExit { }
   }


   state Cycle_First_10_Display_Groups
   {
      onEnter { DisplayTimer.Reset(); }
```

```
      onLoop
      {
         if (DisplayTimer.Time > 10)
         {
            Display.AdvanceDisplayGroup();
            DisplayTimer.Reset();
         }
         if (Display.CurrentDisplayGroup > 10) { changestate Reset_Display_Cycle; }
      }

      onExit { }
   }

   state Reset_Display_Cycle
   {
      ...
   }
}
```

49 Math Object

49.1 General Description

The global `Math` object is an instance of the `System_Math` object type. It contains constants and methods for use in calculations and expressions.

49.2 Math Constants

This chapter describes Scanner Logic Script math constants. The `Math` object defines a number of constant members that you can use in expressions. The constants are all `float` type.

Table 49.2-1. `System_Math` Object Constants

Constant	Value	Definition
`Math.E`	2.71828182845904523536	Euler's Number (e): $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$
`Math.LOG2E`	1.44269504088896340736	$\log_2(e)$
`Math.LOG10E`	0.434294481903251827651	$\log_{10}(e)$
`Math.LN2`	0.693147180559945309417	$\ln 2$
`Math.LN10`	2.3025850929940456840 2	$\ln 10$
`Math.PI`	3.14159265358979323846	π: The ratio of a circle's circumference to its diameter

49.3 Math Library Functions

Scanner Logic Script implements a library of common math functions. In the following tables, x and y may be literal values or properties of resource or registers objects. The type of the x and y arguments is `float`, and the methods all return `float` values.

Table 49.3-1. **`System_Math`** Object Library Functions

Method	Return Type	Description
`Math.Abs(x)`	`float`	Absolute Value Returns x if x >= 0, -x if x<0
`Math.Ceil(x)`	`float`	Ceiling Returns the smallest integer value greater than or equal to x
`Math.Floor(x)`	`float`	Floor Returns the largest integer value less than or equal to x
`Math.Sqrt(x)`	`float`	Square Root Error if x < 0
`Math.Log(x)`	`float`	Natural Logarithm Error if x <= 0
`Math.Log10(x)`	`float`	Base 10 Logarithm Error if x <= 0
`Math.Exp(x)`	`float`	Exponential e to the power of x
`Math.Pow(x,y)`	`float`	Power x to the power of y
`Math.Rand()`	`float`	Random Create random value x, where (0 < x < 1)

49.3.1 Math Trigonometric Functions

Trigonometric function arguments (x) are expressed in radians.

Table 49.3-2. **System_Math** Object Trigonometric Functions

Method	Return Type	Description
`Math.Sin(x)`	`float`	Sine
`Math.Cos(x)`	`float`	Cosine
`Math.Tan(x)`	`float`	Tangent
`Math.Asin(x)`	`float`	Arcsine Range [-π/2, π/2], x ∈ [-1, 1]
`Math.Acos(x)`	`float`	Arccosine Range [0, π], x ∈ [-1, 1]
`Math.Atan(x)`	`float`	Arctangent Range [-π/2, π/2]
`Math.Hsin(x)`	`float`	Hyperbolic Sine
`Math.Hcos(x)`	`float`	Hyperbolic Cosine
`Math.Htan(x)`	`float`	Hyperbolic Tangent

50 User HMI Field Object

50.1 General Description

`HmiField` objects are used to allow a Scanner Logic program to create a user-designed human-to-machine interface (HMI). Based on the information provided by the Scanner Logic program, the web interface creates the User HMI page. Using the `HmiFields` objects, groups of fields with custom headers can be created. This allows for a designing of specific presentations to operators and a means to provided them with all of the operational data on a single page.

The declaration for this object type allows the specification of a property belonging to an object which was declared by the user or is a member of a system object. A user-declared description can be added to provide extra information for the operator. If the selected property has the read/write attribute, the `HmiFields` can be declared to permit modification from the web interface. Declaring the object with the optional heading parameter will cause the insertion of the header text and start a new field group with the object as the first field.

`HmiFields` objects act as a direct bridge between the object properties declared within a Scanner Logic Script program and the Scanner host environment. At the beginning of each script execution cycle, any changes to the HMI fields tags in the host environment are reviewed. If the `HmiFields` was declared to permit external modification, the new value is adopted into the Logic Script environment. At the end of each script execution cycle, any changes to the watched properties are copied out to the `HmiFields` value.

`HmiFields` objects publish their values to the Scanner host environment for use in archiving, displaying on the LCD, acting as inputs for flow runs and calculators, and driving Analog and Digital outputs.

All declared headers and fields groups will appear in a User HMI webpage in the Logic Controller section of the web interface. The User HMI webpage is accessible by all users with valid access. The access level required to write to a field and modify it from the User HMI webpage is defined with the `ProgInfo`.

Up to 64 `HmiFields` objects may be declared for use in a Scanner Logic Script program.

Each field must be assigned to a specific field number (01 – 64). When accessing these fields through Scanner systems that are unaware of the user defined name (e.g.

serial protocol, User Event Archive), these field are identified using their defined number with the format **m32_LM_HF_1_Holding_RN** (where N is the field number).

50.2 Declaring HmiFields Objects

The **HmiFields** declaration group begins with the keywords **hmifields user**, and contains a hmifields item declaration for each **user** field to be used in the program within a pair of open and close braces. Each hmifields item declaration consists of a unique field index number between 01 and 64 and a system defined identifier separated by a colon, and contains a list of parameter assignment statements within a pair of open and close braces.

Parameter assignments that are omitted will have the parameters set to default values according to the parameter's data type (i.e. string : "", uint : 0, float : 0.0, bool : false). Unused hmifields items may be omitted from the hmifields declaration group. Omitting the header parameter will place the field into the existing field group. Including the header parameter will cause the insertion of the new header text and start a new field group with the declared field as the first field. If no **HmiFields** items are required, the entire hmifields declaration group may be omitted.

Parser errors will be generated for improper or incomplete declarations.

Example 50.2-1. **HmiFields** object declaration example

```
hmifields user
{
   01: UserHMI_01
   {
      propertyname: "MyTask.MyState.TotalActiveTime";
      header: "Program Statistics";
      description: "Total time spent in MyState state (seconds)";
      webmodify: false;
   }
   02: UserHMI_02
   {
      propertyname: "MinFlowingDP.Value";
      description: "User defined configuration register value property.";
      webmodify: true;
   }

   ...

   64: UserHMI_64
   {
      propertyname: "FlowingAlarm.HoldOffTime";
      description: "Property of user defined alarm object.";
      webmodify: false;
   }
}
```

50.3 Declaration Parameters

Table 50.3-1. **HmiFields** object declaration parameters

Parameter	Type Code	Description
propertyname	<propname>	A string containing the complete specification of a property including the user declared object name or system object name, and the property name.
header	<str64>	A user-provided string describing the optional field group header text.
description	<str256>	A user-provided string describing the user hmifield object
webmodify	bool	The permission parameter for allowing the field to be modified from the web interface if the property described in propertyname has the R/W access attribute. If the described property does not allow write access, this parameter is ignored.

50.4 Properties

Table 50.4-1. **HmiFields** object properties

Property	Data Type	Access	Description
Value * [default]	float	RO	Holds the value of the hmifields object; since this is the default property of the object, this property name can be omitted when referencing the object, and the compiler will automatically use the Value property

* These properties can be initialized with **initial_*** parameters.

50.5 Usage

HmiFields are not used within a Logic Script program but provide the Scanner host environment the information required to create the User HMI webpage in the Logic Controller section of the web interface. It is not recommended to use the value property of a HmiFields object within statements and expressions. Unexpected results may occur because the hmi field Value property is not updated until after the execution cycle has completed. In other words, the Value property will always contain the final contents of the assigned propertyname from the previous execution cycle.

This page is left blank intentionally.

Part IV—Appendix A

51 Scanner 3100 Unit Categories

51.1 General Description

Measurement units are broken down by Measurement Category and Unit Type.

51.2 Measurement Categories

All values in the Scanner 3100 have a measurement category. A category comprises numerator and denominator unit types, and defines the base metric units and default imperial units.

Table 51.2-1. Measurement categories

Name	Numerator Type Index	Denominator Type Index	Scanner Base Units	Metric Default	U.S. Customary Default	Rate Scalable?
No Units	0	0	—	—	—	Yes
Uncorrected Gas Volume	1	0	m^3	m^3	ft^3	Yes
Uncorrected Liquid Volume	1	0	m^3	m^3	bbl	Yes
Gas Volume	1	0	m^3	m^3	MCF	Yes
Liquid Volume	1	0	m^3	m^3	bbl	Yes
Static Pressure (absolute)	2	0	Pa	kPa	psia	No
Static Pressure (gauge)	3	0	Pa(g)	kPa(g)	psig	No
Differential Pressure	4	0	Pa	kPa	"H2O@68F	No
Temperature	5	0	K	°C	°F	No
Mass	6	0	kg	kg	lbm	Yes
Energy	7	0	J	MJ	Btu	Yes
Length	10	0	m	mm	inch	No
Frequency	11	0	Hz	Hz	Hz	No

Name	Numerator Type Index	Denominator Type Index	Scanner Base Units	Metric Default	U.S. Customary Default	Rate Scalable?
Resistance	12	0	Ohm	Ohm	Ohm	No
Current	13	0	A	mA	mA	No
Voltage	8	0	V	V	V	No
Fraction	20	0	—	—	—	No
Viscosity	17	0	kg/m•s	cP	lbm/ft•s	No
Density (Absolute)	6	1	kg/m^3	kg/m^3	lbm/ft^3	No
Molar Density	18	1	$kg•mol/m^3$	$kg•mol/m^3$	$lb•mol/ft^3$	No
Molar Mass	9	0	kg/kg•mol	kg/kg•mol	lb/lb•mol	No
Mass heating value	7	6	J/kg	MJ/kg	MMBtu/lbm	No
Volume heating value	7	1	J/m^3	MJ/m^3	$MMBtu/ft^3$	No
Energy per Mol (Molar heat values)	7	18	J/kg•mol	J/kg•mol	MMBtu/lb•mol	No
Thermal Expansion Factor	24	5	1/K	1/K	1/°F	No
Pulses per Volume (Vol. K-factor)	16	1	$pulses/m^3$	$pulses/m^3$	pulses/gal	No
Pulses per Mass (Mass K-factor)	16	6	pulses/kg	pulses/kg	pulses/lbm	No
Percent	15	0	%	%	%	No
Factor	0	0	—	—	—	No
Mole Fraction	20	0	—	—	—	No
Base Sediment and Water	15	0	%	%	%	No
Power	25	0	W	W	W	Yes

Name	Numerator Type Index	Denominator Type Index	Scanner Base Units	Metric Default	U.S. Customary Default	Rate Scalable?
Charge	26	0	Ah	Ah	Ah	Yes

51.3 Unit Types

Each unit type has a set of allowable units. The Scale and Offset are the conversion factor and offset to apply to a value in base units to calculate the value in each of the other units.

51.3.1 None

Table 51.3-1. Units for Unit Type None (Index 0)

Slogic	Name	Scale	Offset
	None	1	0

51.3.2 Volume

Table 51.3-2. Units for Unit Type Volume (Index 1)

Slogic	Name	Scale	Offset
m3	meter cubed	1	0
E3m3	thousand meters cubed	0.001	0
E6m3	million meters cubed	0.000001	0
MCF	thousand feet cubed	0.035314667	0
MMCF	million feet cubed	0.0000353146667214887	0
ft3	feet cubed	35.31466672	0
l	litre	1000	0
igal	imperial gallon	219.9692483	0
gal	US gallon	264.1720524	0
bbl	barrel	6.28981077	0
SCF	standard cubic foot	35.31466672	0
cm3	centimeter cubed	1000000	0
10m3	ten meters cubed	0.1	0
100m3	hundred meters cubed	0.01	0

51.3.3 Static Pressure (Absolute)

Table 51.3-3. Units for Unit Type Static Pressure (Index 2)

SLogic	Name	Scale	Offset
Pa	Pascal	1	0
kPa	kilopascal	0.001	0
MPa	megapascal	0.000001	0
psia	pounds per square inch (absolute)	0.000145038	0
inHg	inch of mercury	0.000296134	0
inH2O@68F	inch of water @ 68F	0.004021863	0
ftH2O@68F	foot of water @ 68F	0.000334883	0
atm	atmosphere	0.00000986923266716023	0
bar	bar	0.00001	0
mbar	millibar	0.01	0
kg/cm2	kilogram per centimeter squared	0.0000101971621297793	0
inH2O@60F	inch of water @ 60F	0.0040186	0
inH2O@39.167F	inch of water @ 39.167F	0.004014737	0

51.3.4 Static Pressure (Gauge)

Table 51.3-4. Units for Unit Type Static Pressure (Gauge) (Index 3)

SLogic	Name	Scale	Offset
Pa(g)	Pascal (gauge)	1	0
kPa(g)	kilopascal (gauge)	0.001	0
MPa(g)	megapascal (gauge)	0.000001	0
psig	pounds per square inch (gauge)	0.000145038	0
inHg(g)	inch of mercury (gauge)	0.000296134	0
inH2O(g)@68F	inch of water @ 68F (gauge)	0.004021863	0
ftH2O(g)@68F	foot of water @ 68F (gauge)	0.000334883	0
atm(g)	atmosphere (gauge)	0.00000986923266716023	0
bar(g)	bar (gauge)	0.00001	0

SLogic	Name	Scale	Offset
mbar(g)	millibar (gauge)	0.01	0
kg/cm2(g)	kilogram per centimeter squared (gauge)	0.0000101971621297793	0
inH2O(g)@60F	inch of water @ 60F (gauge)	0.0040186	0
inH2O(g)@39.167F	inch of water @ 39.167F (gauge)	0.004014737	0

51.3.5 fferential Pressure

Table 51.3-5. Units for Unit Type Differential Pressure (Index 4)

SLogic	Name	Scale	Offset
Pa	Pascal	1	0
kPa	kilopascal	0.001	0
MPa	megapascal	0.000001	0
psi	pounds per square inch	0.000145038	0
inHg	inch of mercury	0.000296134	0
inH2O@68F	inch of water @ 68F	0.004021863	0
ftH2O@68F	foot of water @ 68F	0.000334883	0
mmH2O@68F	millimeter of water @ 68 F	0.102155312	0
atm	atmosphere	0.00000986923266716023	0
bar	bar	0.00001	0
mbar	millibar	0.01	0
inH2O@60F	inch of water @ 60F	0.0040186	0
inH2O@39.167F	foot of water @ 39.167F	0.004014737	0
mmH2O@60F	millimeter of water @ 60 F	0.102072439	0
mmHg	inch of mercury	0.007500617	0
kg/cm2	kilogram per centimeter squared	0.000010197162129779	0

51.3.6 Temperature

Table 51.3-6. Units for Unit Type Temperature (Index 5)

SLogic	Name	Scale	Offset
K	Kelvin	1	0

SLogic	Name	Scale	Offset
degC	Celsius	1	-273.15
degF	Fahrenheit	1.8	-255.372
degR	Rankine	1.8	0

51.3.7 Mass

Table 51.3-7. Units for Unit Type Mass (Index 6)

SLogic	Name	Scale	Offset
kg	kilogram	1	0
lbm	pound	2.204622622	0
g	gram	1000	0

51.3.8 Energy

Table 51.3-8. Units for Unit Type Energy (Index 7)

SLogic	Name	Scale	Offset
J	Joule	1	0
kJ	kilojoule	0.001	0
MJ	megajoule	0.000001	0
GJ	gigajoule	0.000000001	0
Btu	British thermal unit	0.000947817	0
MBtu	thousand British thermal unit	0.00000094781712031 3317	0
MMBtu	million British thermal unit	0.00000000094781712031 3317	0
kWh	kilowatt-hour	0.00000027777777777778	0
kcal	kilocalorie	0.000238846	0
10MJ	ten megajoule	0.0000001	0
100MJ	hundred megajoule	0.00000001	0
BtuC	British thermal unit (thermochemical)	0.000948213	0

51.3.9 Voltage

Table 51.3-9. Units for Unit Type Voltage (Index 8)

SLogic	Name	Scale	Offset
V	Volt	1	0
mV	millivolt	1000	0

51.3.10 Molar Mass

Table 51.3-10. Units for Unit Type Molar Mass (Index 9)

SLogic	Name	Scale	Offset
kg/kg*mol	kg/kg·mol	1	0
lb/lb*mol	lb/lb·mol	1	0
g/g*mol	g/g·mol	1	0

51.3.11 Length

Table 51.3-11. Units for Unit Type Length (Index 10)

SLogic	Name	Scale	Offset
m	meter	1	0
cm	centimeter	100	0
mm	millimeter	1000	0
km	kilometer	0.001	0
inch	inch	39.37007874	0
ft	foot	3.280839895	0
yard	yard	1.093613298	0
mile	mile	0.000621371	0

51.3.12 Frequency

Table 51.3-12. Units for Unit Type Frequency (Index 11)

SLogic	Name	Scale	Offset
Hz	Hertz	1	0
kHz	kilohertz	0.001	0
MHz	megahertz	0.000001	0

51.3.13 Resistance

Table 51.3-13. Units for Unit Type Resistance (Index 12)

SLogic	Name	Scale	Offset
Ohm	Ohm	1	0
kOhm	kiloohm	0.001	0
MOhm	megaohm	0.000001	0

51.3.14 Current

Table 51.3-14. Units for Unit Type Current (Index 13)

SLogic	Name	Scale	Offset
A	Ampere	1	0
mA	milliampere	1000	0

51.3.15 Time

Table 51.3-15. Units for Unit Type Time (Index 14)

SLogic	Name	Scale	Offset
s	second	1	0
ms	millisecond	1000	0
mins	minute	0.016666667	0
hours	hour	0.000277778	0
days	day	0.0000115740740740741	0
weeks	week	0.00000165343915343915	0
months	month	0.0000003802651757	0
years	year	0.0000000316887646	0

51.3.16 Percent

Table 51.3-16. Units for Unit Type Percent (Index 15)

SLogic	Name	Scale	Offset
%	percent	1	0
fraction	fraction	0.01	0

51.3.17 Pulse

Table 51.3-17. Units for Unit Type Pulse (Index 16)

SLogic	Name	Scale	Offset
pulses	pulses	1	0

51.3.18 Viscosity

Table 51.3-18. Units for Unit Type Viscosity (Index 17)

SLogic	Name	Scale	Offset
kg/m*sec	kilogram per meter-second	1	0
P	Poise	10	0
cP	centipoise	1000	0
lbm/ft*s	pound per foot-second	0.671968975	0

51.3.19 Mole

Table 51.3-19. Units for Unit Type Mole (Index 18)

SLogic	Name	Scale	Offset
kg*mol	kilogram-mole	1	0
lb*mol	pound-mole	2.204622622	0
g*mol	gram-mole	1000	0

51.3.20 Relative Density

Table 51.3-20. Units for Unit Type Relative Density (Index 19)

SLogic	Name	Scale	Offset
ADen	absolute density	1	0
RDL	relative density liquid	0.001000985	0
RDG	relative density gas	0.816051772	0

51.3.21 Fraction

Table 51.3-21. Units for Unit Type Fraction (Index 20)

SLogic	Name	Scale	Offset
fraction	fraction	1	0
%	percent	100	0

51.3.22 System Ticks

Table 51.3-22. Units for Unit Type System Ticks (Index 21)

SLogic	Name	Scale	Offset
ticks	system tick	1	0
µs	microsecond	10000	0
ms	millisecond	10	0
s	second	0.01	0

51.3.23 Real Date

Table 51.3-23. Units for Unit Type Real Date (Index 22)

SLogic	Name	Scale	Offset
YYYYMMDD	Date	1	0

51.3.24 Real Time

Table 51.3-24. Units for Unit Type Real Time (Index 23)

SLogic	Name	Scale	Offset
HHMMSSCC	Time	1	0

51.3.25 Unity

Table 51.3-25. Units for Unit Type Unity (Index 24)

SLogic	Name	Scale	Offset
1	Unity	1	0

51.3.26 Power

Table 51.3-26. Units for Unit Type Power (Index 25)

SLogic	Name	Scale	Offset
W	Watt	1	0
mW	milliwatt	1000	0
kW	kilowatt	0.001	0

51.3.27 Charge

Table 51.3-27. Units for Unit Type Charge (Index 26)

SLogic	Name	Scale	Offset
Ah	amp-hour	1	0
mAh	milliamp-hour	1000	0
C	Coulomb	3600	0

51.4 Rate Scalar

Table 51.4-1. Units for Rate Scalar

SLogic	Name	Scale	Offset
/sec	per second	1	0
/min	per minute	60	0
/hr	per hour	3600	0
/day	per day	86400	0

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.