VMware Smart Assurance ASL Reference Guide

VMware Smart Assurance 10.0.0

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2019 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

1 Overview 5

Adapters and ASL 5

Setting the locale 8

Setting the locale 9

String support 10

Printing and logging 11

2 Rule Sets 14

Rule set construction 14

Rule execution 17

3 Operators, and Expressions 18

Data and comments 18

Variables and their values 18

Operators 28

4 Pattern Matching and Filters 31

Patterns 31

Pattern operators 32

Pattern elements 39

Special variables 56

Filters 58

5 Actions 61

Action block 61

Iteration and control statements 62

Function reference 68

Returning values 79

Passing arguments to functions 81

Calling rules as functions from do blocks 81

Exception handling 83

6 Interfacing with a Domain Manager 85

ASL and the MODEL language 85

Objects and instances 89

Tables, structures and enumerations 98

Type conversions between ASL and MODEL 100

Domain Manager control 100

Events 101
Transactions, commit, and abort 103
Error handling 104
Repositories 105
Naming and identity 105
Tracing ASL scripts 106

7 Running Adapters 108
The sm_adapter command 108
Front end 110
Rule set 117
Back end 117
Tracing operations 118
Stopping adapters 120

8 ASL Reference 121
Syntax 121
Reserved words 126

9 dmctl Reference 128
Description 128
dmctl syntax 129

10 Card-Port MODEL Code 134
Card-Port MODEL example 134

Overview

1

This chapter includes the following topics:

- Adapters and ASL
- Setting the locale
- Setting the locale
- String support
- Printing and logging

Adapters and ASL

Adapters facilitate the communication of information between devices or applications and a Domain Manager. Adapters exist as either inflow adapters or outflow adapters.

- Inflow adapters collect information and send it to a Domain Manager. These adapters can be used to initialize a Domain Manager with data such as topology information. They also can send, as they occur, event information and topology changes to a Domain Manager.
- Outflow adapters subscribe to a Domain Manager for a set of notifications. These adapters then pass the information to devices or other applications.

  Adapter overview shows the flow of information through an inflow adapter, a Domain Manager, and an outflow adapter.

  a The inflow adapter collects information and passes it to the Domain Manager.

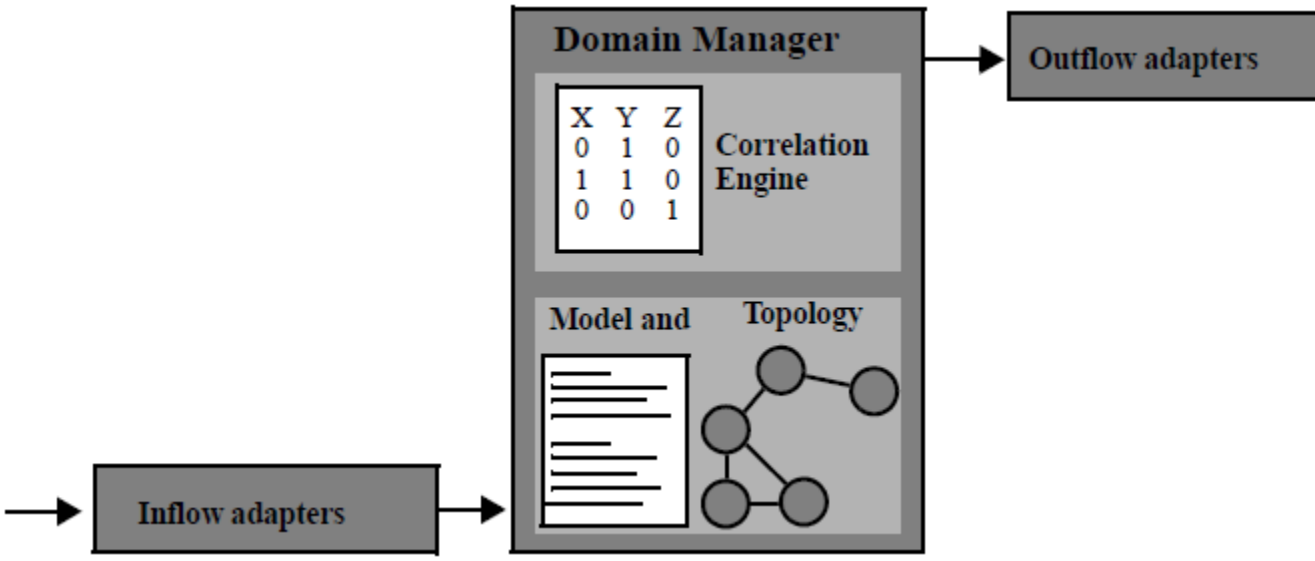
  b The Domain Manager generates notifications.

  c The outflow adapter subscribes to a set of notifications.

  d The Domain Manager passes those notifications to the adapter.

e The outflow adapter passes the information to another entity.

Figure 1-1. Adapter overview



Smart Assurance installation directory

In this document, the term BASEDIR represents the location where Smart Assurance software is installed.

- For UNIX, this location is /opt/InCharge/<*product*>.

Optionally, you can specify the root of BASEDIR to be something other than:

- UNIX: /opt/InCharge

However, you cannot change the <*product*> location under the root directory.

The *VMware Smart Assurance System Administration Guide* provides more information about the directory structure.

Adapter components

All adapters, regardless of type, consist of three basic components: front end, rule set and back end. These three components must be included as part of any adapter.

Front end

The front end collects information for processing in the rule set of the adapter.

- The front end of an inflow adapter collects information an external entity.
- The front end of an outflow adapter collects information from Domain Manager notifications.

Rule set

Rule sets match, filter, and process information received from the front end. The processed information is handled by the back end.

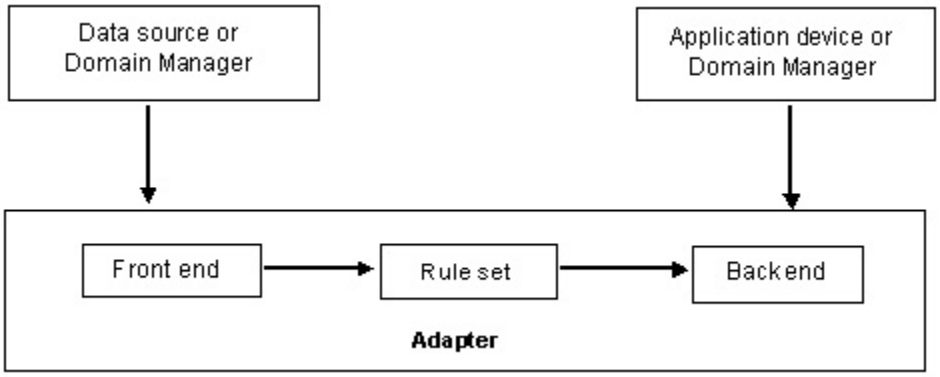
Back end

The back end passes information processed by the rule set to other programs or devices.

- The back end of an inflow adapter passes information to a Domain Manager.
- The back end of an outflow adapter passes information from a Domain Manager to another device or location.

  For example, in Adapter components, if an adapter is designed to move SNMP data to a Domain Manager, its front end communicates with an SNMP device and its back end passes data to a Domain Manager.

  Figure 1-2. Adapter components



Adapter Scripting Language

The VMware Smart Assurance Adapter Scripting Language (ASL) is used to construct the rule sets of adapters. These rule sets are responsible for matching incoming data with patterns and processing the matched data. ASL provides an easy-to-use method of moving data into or out of a Domain Manager.

Introduction to running adapters

The sm_adapter command starts ASL scripts. The file sm_adapter is found in BASEDIR/smarts/bin. Adapters can receive information from files and send output to a file or to the screen. To run most of the sample scripts found in this guide, use this syntax:

```
sm_adapter --file=<input_file>
         <ASL_script_file>
```

The command starts an adapter that runs a user-supplied script *<ASL_script_file>*, reading data from the file, *<input_file>*.

Chapter 7 Running Adapters provides additional information about the sm_adapter command.

Setting the locale

A *locale* is the RFC-1766/3066 compliant specification for language and country.

The scope of execution of an ASL script is within the context of a Domain Manager. Once connected, the locale can be changed by using the setLocale()function. The syntax is:

```
<var> = setLocale(<locale>)
```

The return value is a string representation of the previous setting of the locale, or a default value if setLocale() has not yet been called.

The following example sets the locale to French as spoken in France.

```
old_locale = setLocale(”fr_FR”);
print(“The old locale was “.old_locale);
```

When setLocale() is called in ASL, the value of the locale is saved in thread-local storage. When an ASL script connects to a Domain Manager, the same thread is used for the lifetime of the connection session. Therefore, if you call setLocale(“fr_FR”) from an ASL script, then the locale is fr_FR for the duration of the ASL script. When the script ends, the Domain Manager thread goes away. Other ASL scripts, and other clients each have their own connection/session in their own thread, so each thread is independent of the others.

Locale effects on message rendering

Currently, the scope of the setLocale() operation influences text sent from the Domain Manager to the ASL script. Specifically, setLocale() affects textual data of these types:

- MODEL descriptions obtained from the following operations on SM_System:
  - getOpDescription
  - getEventDescription
  - getPropDescription
  - getClassDescription
- Error text obtained from the following operations on SM_API_Support:
  - getErrorText

    The scope of the setLocale() operation has no influence on any other textual data communicated between the ASL script and the Domain Manager. Textual elements defined as part of modeled applications (aside from entity description data) are not affected by this operation.

Default locale when setLocale() is not called

The default locale to set for an ASL script is determined as follows:

1 The value of the SM_LOCALE environment variable.

2 If SM_LOCALE is not set equal to a value, then the default locale is set equal to en_US.

Setting the locale

A *locale* is the RFC-1766/3066 compliant specification for language and country.

The scope of execution of an ASL script is within the context of a Domain Manager. Once connected, the locale can be changed by using the setLocale()function. The syntax is:

```
<var> = setLocale(<locale>)
```

The return value is a string representation of the previous setting of the locale, or a default value if setLocale() has not yet been called.

The following example sets the locale to French as spoken in France.

```
old_locale = setLocale(”fr_FR”);
print(“The old locale was “.old_locale);
```

When setLocale() is called in ASL, the value of the locale is saved in thread-local storage. When an ASL script connects to a Domain Manager, the same thread is used for the lifetime of the connection session. Therefore, if you call setLocale(“fr_FR”) from an ASL script, then the locale is fr_FR for the duration of the ASL script. When the script ends, the Domain Manager thread goes away. Other ASL scripts, and other clients each have their own connection/session in their own thread, so each thread is independent of the others.

Locale effects on message rendering

Currently, the scope of the setLocale() operation influences text sent from the Domain Manager to the ASL script. Specifically, setLocale() affects textual data of these types:

- MODEL descriptions obtained from the following operations on SM_System:
  - getOpDescription
  - getEventDescription
  - getPropDescription
  - getClassDescription
- Error text obtained from the following operations on SM_API_Support:
  - getErrorText

    The scope of the setLocale() operation has no influence on any other textual data communicated between the ASL script and the Domain Manager. Textual elements defined as part of modeled applications (aside from entity description data) are not affected by this operation.

Default locale when setLocale() is not called

The default locale to set for an ASL script is determined as follows:

1 The value of the SM_LOCALE environment variable.

2 If SM_LOCALE is not set equal to a value, then the default locale is set equal to en_US.

String support

The ASL interpreter assumes that ASL Script files are UTF-8 encoded. The interpreter will recognize a valid UTF-8 BOM if it is present. The file may contain a valid UTF-8 BOM, but it is not required.

ASL is assumed to be working with UTF-8 strings all the time. The string type in ASL is assumed to be a UTF-8 string.

You can print a string by performing the following:

- Extract the localized string.
- Apply substitutions to the string.
- Print the final rendered string.

Obtaining externalized strings

A built-in function for use by internal applications only (SDK and other VMware adopting applications) may be used to retrieve an externalized string. The getExternalString() function extracts a string, based on the table name, key, and locale that is passed to the function. The getExternalString() function then renders the extracted string given a list of substitutions (up to six).

```
<var> = getExternalString(<table>, <key>, <locale>)
```

The *<table>* and *<key>* arguments are required. The *<locale>* argument is optional. The default value is the value of the SM_LOCALE environment variable. The return value is the UTF-8 encoded string that contains the text of the message defined in the table.

The ASL script may then use the returned string as an argument in the print() built-in function, as shown in the following example:

```
msg = getExternalString("SMS_MSG","SUCCESS","fr_FR");
print(msg);
```

Applying substitutions

Applying substitutions may be accomplished by using the formatString() built-in function. The formatString() function takes up to seven arguments:

```
<var> = formatString(<msg>[,<sub_1>][,<sub_2>][,<sub_3>][,<sub_4>][,<sub_5>][,<sub_6>])
```

The *<msg>* argument is the string variable which was extracted from a message table. Up to six substitution arguments may be specified.

The example below shows the use of the getExternalString() and formatString() built-in functions.

```
# message table entry:
#
message_table MY_TABLE
message_code  OPER_FAIL_MSG,
              "Operation &1 failed.  Reason: &2"
---------------------------------------
// ASL fragment.   An attempt to perform some operation failed.
// the variable 'operation' holds the name of the operation that
// failed, the variable 'reason' indicates the reason for the failure.
//
// extract the string using the default locale (SM_LOCALE)
//
 format = getExternalString("MY_TABLE","OPER_FAIL_MSG");
//
// format the substitutions
//
 output = formatString(format,operation,reason)
//
// print the resulting string
//
 print(output);
```

Printing and logging

Messages that are printed by using the ASL print() built-in function are printed to each of the active logs in the Domain Manager process. There is no automatic translation of the message text. For example, if the message was obtained by calling getExternalString() by using a French locale, the message will appear both in the English log and the French log in French. The section print() function provides additional information.

Retrieving and setting log, error and trace levels at runtime

There are three computed attributes available to get and set the log, error, and trace levels of a Domain Manager at runtime. These computed attributes, described in Computed attributes to retrieve and set log, error, and trace levels at runtime, are available on the SM_JIIM_Support object. The SM_System object is a subclass of SM_JIIM_Support and also inherits these computed attributes.

Table 1-1. Computed attributes to retrieve and set log, error, and trace levels at runtime

Computed Attributes	Description
logLevel	The minimum exception level for sending messages to the system error logger.The logLevel attributeis a string, and can be any one of the values set for the **--**loglevel command line option:
errLevel	The minimum exception level for writing messages to the log files. The errLevel attributeis a string, and can be any one of the values set for the **--**errlevel command line option.
traceLevel	Used to print a stack trace to the SDK log file when an exception at this level or above occurs. Exceptions below this level do not write a stack trace. The traceLevel attributeis a string, and can be any one of the values set for the **--**tracelevel command line option.

The values of these computed attributes can be retrieved and set, and valid values are:

- None
- Emergency
- Alert
- Critical
- Error
- Warning
- Notice
- Informational,
- Debug

Note Fatal is a synonym for Critical.

Retrieving the current level

You can retrieve the current levels of SM_System::SM-System::logLevel, SM_System::SM-System::errLevel, or SM_System::SM-System::traceLevel. A string is returned which represents the current level, such as "Warning", "Error", or "Fatal". For example:

```
sm_system = object("SM_System", "SM-System");
curr_error_level = sm_system->errLevel;
```

Setting the level

To change the current levels, obtain a pointer to the object, and then set the value of SM_System::SM-System::logLevel, SM_System::SM-System::errLevel, or SM_System::SM-System::traceLevel to the appropriate level.

In this example, the trace level setting is changed to None.

```
sm_system = object("SM_System", "SM-System");
sm_system->traceLevel = "None";
```

When you change the log, error or trace levels a message is printed in the log file. The log message will appear similar to the following:

```
[April 8, 2009 5:03:41 PM EDT +122ms] t@1149000000 SM_ProtocolEngine-6
JM_MSG-*-JM_TRACE_LEVEL_CHANGED-User 'user1', using remote dmctl client (id 6), on host host1 with
credentials tpadmin1 has changed the Trace level to None; in file
"/mypath/repos/jiim/SM_JIIM_Support_Impl.c" at line 458
```

Rule Sets

2

This chapter includes the following topics:

- Rule set construction
- Rule execution

Rule set construction

A rule set is a group of rules that *may* match, filter, and execute actions to process data received from an adapter's front end.

An ASL rule consists of blocks of patterns, filters, and actions. Patterns select the data to process by applying a pattern matching to the incoming datastream. Filters control whether actions are performed on matched data. Actions process the data.

An ASL rule has the basic form:

```
          <RULE-NAME> {
<pattern-list>
}
filter {
<filter-list>
}
do {
<action-list>
}
```

Rules begin with a rule name. A rule name must consist of alphanumeric characters and/or an underscore (_). The first character of a rule cannot be a number.

Note In this guide, by convention, no rule names contain lowercase letters.

Pattern, filter, and action blocks comprise the contents of a rule. None of the blocks are required.

Note Certain ASL words are reserved and should not be used, as described in *"Reserved words" on page 149*.

Braces surround the pattern list and demarcate the pattern block. A pattern block can contain one or more action blocks, but cannot contain any filter blocks. For actions performed before any pattern matching, an action block can be included inside of the pattern block before any pattern matching statements.

The word “filter” followed by a brace ({) marks the beginning of a filter block. Another brace (}) marks the end. A rule can contain no more than one filter block, and that block must appear after the pattern block.

The word “do” followed by a brace ({) marks the beginning of an action block. Another brace (}) marks the end. A rule can contain multiple action blocks placed inside a pattern block or outside of a pattern block. However, only one action block can appear outside of a pattern block.

In this example, the following rule associates a specific pattern with a specific action:

```
          <RULE-NAME> {
<pattern-list> do {<action-list>}
<pattern-list> do {<action-list>}
<pattern-list> do {<action-list>}
}
```

The pattern block shown in the example encompasses all of the patterns and all of the action blocks. In order to add a filter block, the brace marking the end of the pattern block has to follow the last pattern list. Filter information can be added before the last action block.

```
          <RULE-NAME> {
<pattern-list> do {<action-list>}
<pattern-list> do {<action-list>}
<pattern-list>
}
filter {<filter-list>}
do {<action-list>}
```

Rules can be called from other rules or in an action block. Rules cannot be called from a filter.

```
          <RULE-NAME> {
<OTHER-RULE>
}
filter {
<filter-list>
}
do {
<action-list>
}
```

Patterns and called rules can be interspersed with action blocks.

```
          <RULE-NAME> {
<pattern-list>
do {<action-list>}
<OTHER-RULE>
}
```

Rules referenced by other rules are considered *subordinate* to the rule that references them. The rule that references a subordinate rule is referred to as *superior*. The flow of control of ASL processing passes from superior rules to subordinate rules and back.

Special rules

Three special rules exist: START, DEFAULT and EOF. Each of these rules must be in uppercase letters.

START rule

All rule sets must include a rule named START. The START rule marks the starting point of the ASL script when the adapter receives input. However, the START rule does not have to be the first rule listed in the script.

```
START {
<pattern-list>
}
filter {
<filter-list>
}
do {
<action-list>
}
```

DEFAULT rule

If the input fails to match any patterns specified by the START rule and the rules called by the START rule, a rule named DEFAULT runs automatically. The purpose of the DEFAULT rule is to resynchronize the input stream. The DEFAULT rule has the same structure as any other rule. However, if the pattern matching of the DEFAULT rule fails, the ASL script aborts.

If the DEFAULT rule is not explicitly defined and the START rule fails, there is an implicit DEFAULT rule that is executed. This implicit DEFAULT rule matches the current line of input.

If the implicit DEFAULT rule was added to an ASL script, it would look like this:

```
DEFAULT {
..eol
}
```

The string "..eol" is an ASL pattern that, from a starting point, matches all characters up to and including an end-of-line. Chapter 4 Pattern Matching and Filters provides additional information about patterns.

EOF rule

A rule named EOF runs at the end of an input file. The EOF rule can but should not include a pattern matching block. At the end of a file, there is no data for a pattern to match. The EOF rule is not required in an ASL script.

```
EOF
do {
<action-list>
}
```

Rule execution

The first rule executed in an ASL script is the START rule. The START rule runs repeatedly until all of the data input is processed. Data input is processed as it is matched with all of the patterns of a rule.

Patterns are components either of the START rule or of other rules called from the START rule. As data matches patterns, the starting position for the next pattern match moves to the end of the matched data. The next match is tested from this new position and so forth.

The new starting position for pattern matches is permanent when all of the patterns associated with the START rule match. Patterns associated with the START rule include patterns from all of the rules subordinate to START. If the START rule and all of its subordinate rules have executed, the START rule executes again at the new starting position.

If patterns in subordinate rules fail to match, control passes to the immediately superior rule where an alternate, if it exists, is tried. If an alternate does not exist, the starting point for the match is reset to the point when the superior rule executed. Control is passed to the next superior rule and the next alternate is tested. This continues until a match is made or until the START rule does not match.

The DEFAULT rule runs when the START rule fails. The DEFAULT rule contains its own pattern matching. When the DEFAULT rule matches, the starting point is permanently advanced and the START rule is executed. If the DEFAULT rule pattern match fails, the ASL script aborts.

The EOF rule runs when the data input is exhausted. It is not necessarily the last rule to execute because other rules might be subordinate.

Operators, and Expressions

3

This chapter includes the following topics:

- Data and comments
- Variables and their values
- Operators

Data and comments

This section describes the input datastream and the format of comments.

Data

The ASL input datastream consists of characters and markers. Characters include any character in the extended ASCII character set. Markers demarcate the boundaries of fields and records.

Comments

There are two different formats for comments. A comment is text that begins with two slashes (//) and all information to the right of the slashes is ignored.

```
// ASL Script to read card information
```

A comment can also be marked as a block by using a slash followed by an asterisk at the beginning (/*) and an asterisk followed by a slash at the end (*/) of the comment block.

```
/* ASL Script to read card information
   created 1/30/2009 by FTW */
```

Variables and their values

Variables are assigned values in an ASL script or at adapter startup. Variable assignment at startup describes how to specify variable values at startup.

ASL variables do not have a declared type, the values assigned to them are typed. Variable values can have the following types:

- numeric

  ASL stores all numbers as double-floating point.
- string
- binary
- Boolean
- list
- table
- object handle

  The section Chapter 6 Interfacing with a Domain Manager provides additional information.
- datetime

  The section time() function provides additional information.

  A variable name can consist of a letter or an underscore, followed by any number of letters, underscores, and digits. A variable name is case-sensitive. Uppercase and lowercase letters are distinct.

  Note Certain ASL words are reserved and should not be used as identifiers or variables, as described in *"Reserved words" on page 149*.

  Any variable can be assigned different types of values. For example, starting with a basic ASL statement:

```
x = "string";
```

  The variable x can then be used to store a value:

```
x = 5.62;
```

  It can also store a Boolean value:

```
x = TRUE;
```

  For most values, ASL converts one type of value to the appropriate data type for use in a function or expression. In this example, the var_w variable is assigned a string of numeric characters.

```
var_w = "3498";
```

  The variable, var_w, can be added to a number.

```
var_y = var_w+100;
```

This statement is valid. The number 100 is added to the numeric equivalent of the value stored in var_w. There is no intermediate step.

Note The semicolon terminates the end of these assignment actions in ASL.

Type conversions

In some instances, automatic type conversion does not occur. Some functions return values of different types depending on the type of the value passed. For these cases, there are variable type conversion functions. Type conversion functions shows the type of conversion functions:

Table 3-1. Type conversion functions

Syntax	Description
string(value)	Converts the values as a string type.
boolean(value)	Converts the values as a Boolean type. Any nonzero number is converted to TRUE. The only strings that convert are “true” and “false” including all capitalization variations.
numeric(value)	Converts the values as a numeric type. All numeric values are stored as double-floating point.

Lists and tables

Lists and tables are special types of values that can store sets of values. Any variable can store lists and tables. A value of any type can be assigned to a list or table.

List values

A list is an ordered set of values that are indexed numerically, starting with zero (0). The value assigned to a member of a list can be any value regardless of whether it is variable, constant value, a table, a list, or other value type. The list() function initializes the variable type to be of type list. An example is:

```
x = list();
```

Using this syntax, the list is empty.

There are three methods to add members to a list.

- The first method is:

  ```
  x = list(<value_1>,<value_2>,<value_3>,...,<value_n>);
  ```

- The second method specifies an index and assigns a value. This method can also be used to change the existing value of a member in a list.

  ```
  x[100] = value_1;
  x[57] = value_2;
  ```

- The third method appends the value to the list. Using this method, ASL keeps track of the index value.

  x += value_1;

Table values

Tables are associative lists. They exist as hashed arrays with a key and value pair (*<key>*,*<value>*) for each member. The keys for a table must be unique.

A variable may be initialized to be of type table. An example is:

```
y = table();
```

To add a value to a table, use the following syntax:

```
        <table_name>[<key>] = <value>;
```

The value specified for the *<key>* or for the *<value>* can be of any value type.

Multicolumn tables can be implemented in two ways.

The first method implements the multicolumn table as a table of tables:

```
START {
        .. eol
} do {
        x = table();
        x["a"] = table();
        x["a"]["b"] = 1;
        x["a"]["c"] = 2;
        print(x["a"]["b"]);
        print(x["a"]["c"]);
}
```

The second method implements the multicolumn table as a single table with joined key:

```
START {
        .. eol
} do {
        x = table();
        x["a|b"] = 1;
        x["a|c"] = 2;
        print(x["a|b"]);
        print(x["a|c"]);
}
```

Scope of variables

The scope of variables determines where the value associated with the variable can be accessed. In ASL, there are three scope levels:

- Local-scoped variables are accessible only from the rule where the variable is assigned a value and can be referenced in subordinate rules.
- Driver-scoped variables are accessible from any rule in an ASL script.
- Global-scoped variables are accessible from any adapter that runs on the same process where the variable is assigned a value.

Note A local-scope variable cannot have the same name as a global-scoped variable.

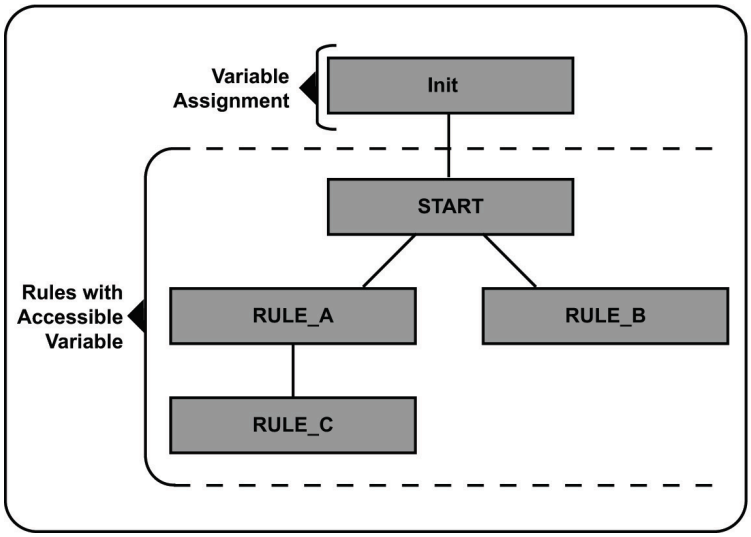
Driver-scope

Driver-scoped variables are accessible from any rule in an ASL script. There are two lifetimes for driver-scoped variables:

- Static
- Record

The difference between the two lifetimes depends on where the variable is assigned a value.

The longer lifetime for a driver-scoped variable is static. When a variable is assigned a value at the beginning of an ASL script, before the START rule is executed, it is static. Static variables maintain their value across uses of the START rule or until a new value is assigned. In Example of a variable with a static lifetime, each time the START rule is invoked, the variable retains its value.




```
variable=<value>;
START {
        RULE_A
        RULE_B
} do {
        print(variable);
}
RULE_A {
        RULE_C
}
RULE_B
do {print(variable);}

RULE_C
do {print(variable);}
```

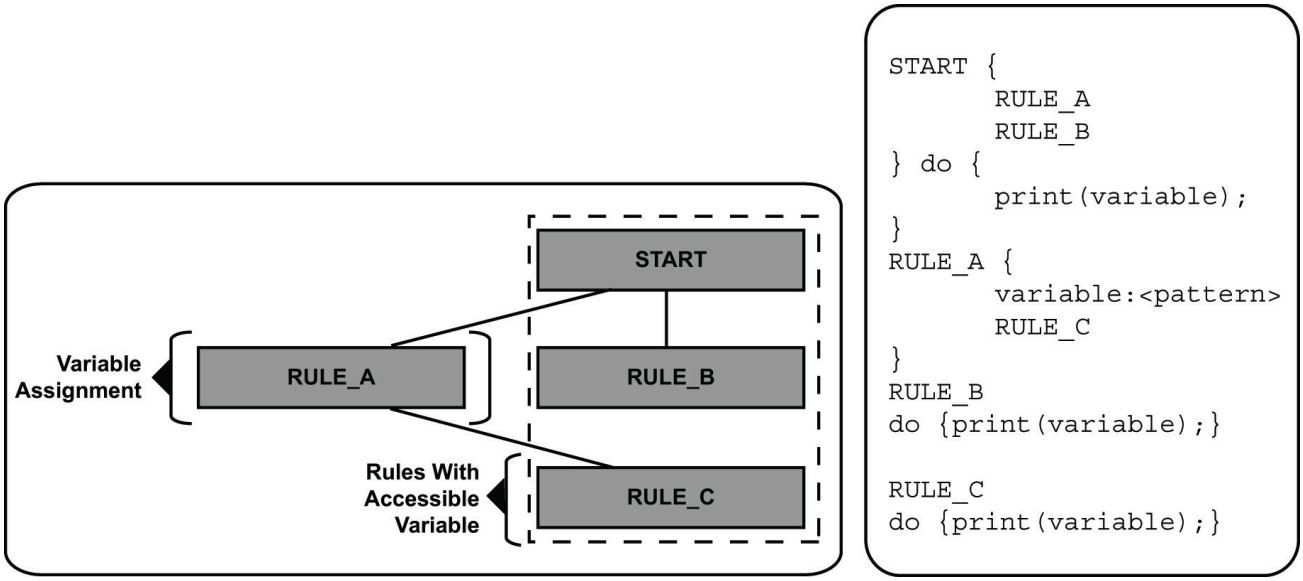
Figure 3-1. Example of a variable with a static lifetime

In the following example, the script counts the number of lines of data in a file. A variable is assigned to zero at the beginning of the script. The variable's value is incremented for each line of input. At the end of the script, the value of the variable is printed. In this script, the START rule is invoked five times and the variable is not reset.

```
ASL Script (static_var.asl):
lines=0;
START {
 .. eol
}
do {
 lines = lines+1;
}
EOF do{
 print("Number of lines ".lines);
}
Input: (static_var.txt):
line 1
line 2
line 3
line 4
line 5
Output:
$ sm_adapter --file=static_var.txt static_var.asl
Number of lines 5
$
```

The shorter lifetime for a driver-scoped variable is record. A driver-scoped variable has a record lifetime when it is assigned a value in a rule. The value of this variable does not change until it is reassigned a value or when the START rule exits. A driver-scoped variable with a record lifetime is undefined each time the START rule exits. In Example of a variable with a record lifetime , each time the START rule is invoked, the variable is undefined.

Figure 3-2. Example of a variable with a record lifetime



In the next example, the script demonstrates that driver-scoped variables keep their value until the START rule is invoked. In the script, a variable is assigned when the HI rule runs (x=TRUE). This variable holds its value when the action block of the program is invoked after the HI rule. When the START rule is invoked again, the input fails to match the pattern specified by the HI rule so the alternate rule to HI, the THERE rule, is invoked and the variable is not assigned a value in this rule. When the execution of the script reaches the action block, no variable exists.

```
ASL Script (record_var.asl):
START {
 print("Starting again")
 /* print returns TRUE so can be
  * used outside of a do*/
 HI|THERE
}
do {
 if (defined(x))
  //Tests whether variable exists
  {print("x is defined");}
 else {print("x is not defined");}
}
HI {
 "hi" eol
}
do { x=TRUE;}
THERE {
 "there" eol
}
Input (record_var.txt):
hi
there
Output:
$ sm_adapter --file=record_var.txt record_var.asl
Starting again
```

```
x is defined
Starting again
x is not defined
$
```

Local-scope

Driver-scoped variables have some limitations. For instance, they are not suitable for recursion, but local-scoped variables are suitable.

The following example shows a script that uses recursion and a driver-scoped variable. At first glance, the script looks like it reverses the order of words that are passed to it. A driver-scoped variable x with a record lifetime is used to store a value each time a rule is executed. Unfortunately, each time the recursion executes, the word assigned to the variable overwrites the existing value. The last value assigned to x is the value that is used for all of the recursions. Only the last word appears as output.

```
ASL Script (2ndrule_var.asl):
START {
 READANDPRINT
}
READANDPRINT {
 x:word READANDPRINT|eol
}
do {
 print(x);
}
Input (2ndrule_var.txt):
The dog ran.
Output:
$ sm_adapter --file=2ndrule_var.txt 2ndrule_var.asl
ran.
ran.
ran.
ran.
$
```

Local variables are defined in a rule and have a separate value each time the rule runs. Local variables can be referenced in any subordinate rules. They cannot be accessed in any other rule. A local variable cannot have the same name as a global variable.

If a local variable is used in a recursive rule, each recursion contains an exclusive value for the variable, as shown in *Figure 5 on page 37*.

Local variables are declared in the pattern block of a rule. These declarations must occur before any pattern matching statements. Local variables override variables with the same name. Use this syntax to declare a local variable:

```
local <variable_name>;
```

The following example demonstrates how use of a local variable instead of a driver-scoped variable changes the output of the script.

The script reverses the order of words. The rule READANDPRINT iterates until it does not match. For each iteration, the value of the word scanned is stored in a local variable. When the end-of-line (eol) marker is reached, the READANDPRINT rule no longer matches and the recursive rules are completed with the action block. Each value of x is printed, which has the effect of reversing the order of the input. The section End-of-line matches provides additional more information about eol markers.

```
ASL Script (local_var.asl):
START {
 READANDPRINT
}
READANDPRINT {
 local x = “end”;
 x:word
 READANDPRINT|eol
}
do {
 print(x);
}
Input: (local_var.txt):
The dog ran.
Output:
$ sm_adapter --file=local_var.txt local_var.asl
end
ran.
dog
The
$
```

Global-scope

Adapters in the same server or threads in a single process can share global variables. Global variables can be of type numeric, string or boolean.

Lists and tables can also be used as global variables but it is not recommended. The ASL programmer must exercise caution when using global lists or tables to ensure that a single thread or adapter is the "writer" of the list or table. If more than one thread or adapter attempts to assign values to a list or table, the likely result will be data loss due to rewriting the variables’ values. There is no inherent mutex or lock arrangement in ASL to prevent this and it is up to the programmer to ensure against this.

Global variable declarations must be made in the static section of the script file, before any rules are declared.

Note A local variable cannot have the same name as a global variable.

Adapters in the same server or threads in a single process can share global variables. Global variables can be of type numeric, string or boolean.

Use this syntax to declare a global variable:

```
global <variable_name>;
```

Default variable values

A default assignment assigns a value only if the variable is undefined.

```
default <variable_name>=<value_or_expression>;
```

The scope of a default variable is restricted to driver-scoped with a static lifetime. A value must be assigned to a variable declared as default. The value cannot be blank. An example of declaring a default variable is:

```
default x = 5;
```

Default variables assigned in an ASL script can be overridden by the value of the variables specified during the startup of an adapter with the -D option.

Any variable assignment *not* defined as default can override the default value and, if one exists, the value that was assigned by using the -d option.

The following code fragment prints the number 1 if there is no integer value to assign to y.

```
x = 1;
default y = 1;
START
do
{       print("x=".x);
        print("y=".y);
        stop();
}
Output with the –D option:
$ sm_adapter –Dx=2 –Dy=2 default.asl
x=1
y=2
Output without the –D option:
$ sm_adapter default.asl
x=1
y=1
```

Variable assignment at startup

Variable values can be assigned when the adapter is started by using the -D option. Variables assigned by using this method have a static lifetime. Default variables that are locally scoped cannot be assigned a value by using this method.

```
sm_adapter –D<variable_name>=<value>
```

The value that is assigned by the switch can be overridden by a standard variable assignment through either pattern matching or by a variable assignment statement. To define a variable so that the -D option overrides the value set in the ASL script, define the variable as a default variable in the ASL script.

Operators

This section describes operators and their order of evaluation.

Arithmetic operators

The addition, subtraction, multiplication, division, and modulus operators can only be used with numeric values. The arithmetic operators are described in Arithmetic operators.

Table 3-2. Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Note You can isolate the decimal portion of a number with the modulus operator by using the syntax: number%1.

For example, this script assigns numeric values to two variables. These numbers are used in addition, subtraction, multiplication, division, and modulus operations.

```
ASL Script (mathematic_do.asl):
START
do {
 a = 36;
 b = 4;
 print("Addition ".a+b);
 c = a-b;
 print("Subtraction ".c);
 d = a*b;
 print("Multiplication ".d);
 print("Division ".b/a);
 e = 10%b;
 print("Modulus ".e);
 stop();
}
Output:
$ sm_adapter mathematic_do.asl
Addition 40
Subtraction 32
Multiplication 144
Division 0.111111111111111
Modulus 2
$
```

String operators

The concatenation operator is the period (.). The concatenation operator forms a new string composed of two values. Variable conversion is handled automatically.

Note In the pattern block of a rule, a period is not an operator that concatenates two strings. It forces two patterns to match together, as described in Chapter 4 Pattern Matching and Filters.

For example, in this script, four variables are assigned values. The first concatenation occurs within the first print statement. The second concatenation is a combination of two numbers and is assigned to the variable c. The new string represents a number and can be used in calculations.

```
ASL Script (concat_do.asl):
START
do {
 a = 657;
 b = 9283;
 x = “cat”;
 y = ”dog”;
 print(x.y);
 c = a.b;
 print(c);
 stop();
}
Output:
$ sm_adapter concat_do.asl
catdog
6579283
$
```

Relational and logical operators

The relational and logical operators are described in Relational and logical operators.

Table 3-3. Relational and logical operators

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&&	Logical AND
||	Logical OR

Precedence and order of evaluation

ASL evaluates operators in the following order:

1 ( )

2 * /

3 %

4 + -

5 == != < > <= >=

6 && || .

Parentheses are evaluated first. The logical AND, logical OR, and the concatenation operator are evaluated last. ASL evaluates operators with the same level of precedence from left to right.

Pattern Matching and Filters

4

This chapter includes the following topics:

- Patterns
- Pattern operators
- Pattern elements
- Special variables
- Filters

Patterns

A pattern is a mechanism for matching input. A combination of markers, operators, and characters describes a pattern. This combination includes the data to match and the operators that control how and when the data matches. More complicated patterns are made from simpler patterns.

ASL compares patterns to an input of markers and characters. As each component of a pattern matches with data, the next pattern component is compared with the next segment of input. If all of the components of a pattern match, the pattern is considered a match. If any component of a pattern fails to match its corresponding data component, the pattern is not considered a match.

A pattern that does not match fails. In most cases, when a pattern fails, the rest of the rule following that pattern does not execute.

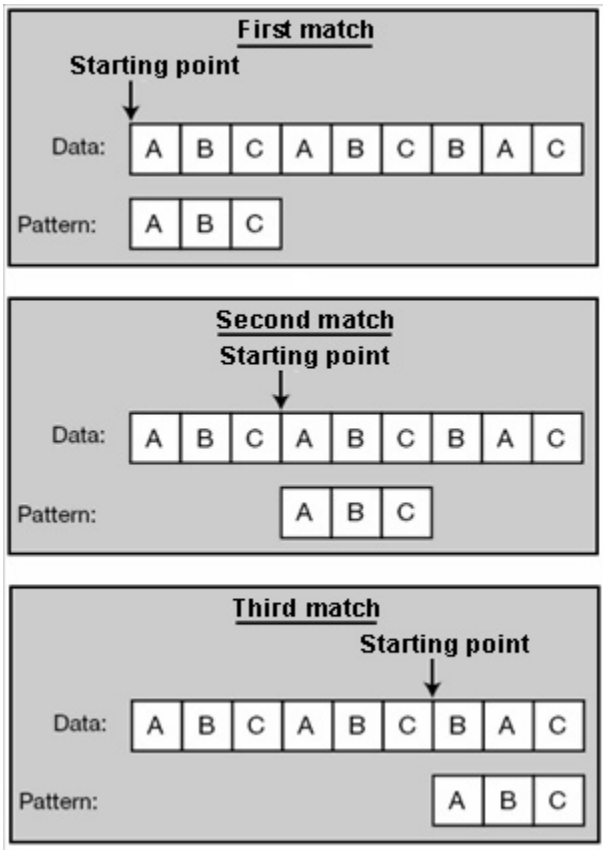
As ASL matches patterns with data, the starting point for the next match moves to the position after the data that was matched. When a pattern fails, the starting position does not advance and may or may not go back to its original position, depending on how the rule containing the pattern is defined.

Progression of a pattern being compared to data shows that for the first match, the pattern is compared, starting with the left-most position in the data. The pattern matches.

As the result of the successful match, the starting point for the next comparison, the second match, is immediately after the last successful data match. The pattern matches.

For the third match, the starting position is in a different location again because of the previous successful match. This comparison fails and the start position resets. Depending on the pattern, the start position resets to either the starting point for this comparison or to the beginning of the data.

Figure 4-1. Progression of a pattern being compared to data



Using a function in a pattern

Most ASL functions used in the action block of a script can be included in an ASL pattern match. The syntax of these functions, when used in a pattern block, is slightly different than when used in an action block. To use a function in a pattern block, use the following syntax:

```
function([<value>])
```

In a pattern block, do not use a semicolon with functions.

Pattern operators

This section describes pattern operators and their order of their order of precedence.

White space handling

White space is defined as one or more of any of the characters assigned to the delimiter variable, as described in Customizing the delimiter. By default, the space and tab characters are assigned to the delimiter variable. That means any number of contiguous space or tab characters constitute white space in the data.

A pattern is made up of one or more pattern elements. There are many types of pattern elements. For example, keywords such as “word”, and “integer” are elements. An element can be a literal string (text surrounded by quotes). By default, ASL will ignore any white space in the data as it tries to match pattern elements. You can use the dot operator to keep ASL from ignoring white space in a pattern match, as described in Dot operator.

Note When a literal pattern element contains whitespace, ASL does not ignore it.

For example, if the pattern to match is:

```
"John" "Doe" integer “st” “Street”
```

Then ASL will match the following data equally.

```
John Doe 121st street
JohnDoe121stStreet
John    Doe  121  st   Street
```

The following script matches a word followed by an integer and another word. The dot operator is not used, so the delimiters between the words in the data are automatically skipped (if they exist). The end-of-line (eol) for each line of data is not matched, which causes the START rule to fail. Notice that the second line of input matches even though there is no delimiter between the number, 42343, and the title, Manager.

```
ASL Script (wspace_match.asl):
START {
 a:{word integer word}
}
do {
 print("Matched with ".a");
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (wspace_match.txt):
Tony 25234 Employee
John 42343Manager
Output:
$ sm_adapter --file=wspace_match.txt wspace_match.asl
Matched with Tony 25234 Employee
Failed match
Matched with John 42343Manager
Failed match
```

Assignment operator

The characters matched to a pattern can be placed into a variable by using a colon (:) as an operator. The syntax of a variable assignment is:

```
<variablename>:<pattern>
```

ASL assigns all of the characters that match the pattern to the variable. If the pattern does not match, the variable is not assigned a value.

To assign patterns that have more than one element, use braces to group the elements. All the components within the braces must successfully match before the variable is assigned a value. The syntax is:

```
<variablename>:{<pattern_1>
<pattern_2>
<pattern_3>
<pattern_4>}
```

The following script contains three variable assignments. The first and second variables are assigned the first and second word, respectively, in the datafile. The last variable is assigned an integer followed by two words.

This last variable demonstrates how patterns can be grouped by using braces. Without the braces, the variable is assigned only the integer.

The last name in the input file causes the pattern to fail because there is no address information.

```
ASL Script (assign_match.asl):
START {
 f_name:word l_name:word
 address:{integer word word}
 eol
}
do {
 print(f_name);
 print(l_name);
 print(address);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (assign_match.txt):
John Doe 11 Main St.
Jane Doe 35 Oak Dr.
Bill Jones
Output:
```

```
$ sm_adapter --file=assign_match.txt assign_match.asl
John
Doe
11 Main St.
Jane
Doe
35 Oak Dr.
Failed Match
$
```

Dot operator

A single dot (.) between two elements of a pattern indicates that the second element must be matched immediately after the first. The dot can be separated from the two elements with white space. However, in this case, the white space does not indicate an optional delimiter. If a delimiter exists in the input data, this delimiter must be included as part of either the first or second element when using the dot operator.

The following script matches an integer followed by a word. Only the integer is assigned to a variable and printed. In this example, the first two lines of data match this pattern. When a space is added between the integer and the word, as in line 3 of the data, the pattern fails.

```
ASL Script (sngldot_match.asl):
START {
 a:integer.word
 eol
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (sngldot_match.txt):
95627XFR
34036TFR
11536 GBH
Output:
$ sm_adapter --file=sngldot_match.txt sngldot_match.asl
Matched with 95627
Matched with 34036
Failed match
$
```

Double-dot operator

The double-dot operator (..) matches all characters except for an end-of-line (eol). White space can surround the double-dot, but the dots must be together. This operator cannot stand alone. Another pattern must follow. The parser matches as few characters as possible before matching the pattern that follows.

The following script matches any string of characters up to and including the word, Smith. The pattern fails on the second line of data because there is no Smith before the end-of-line (eol) is reached. In the DEFAULT rule, the double-dot operator matches everything until the end of the line.

```
ASL Script (dbldot_match.asl):
START {
 a:{
 .."Smith"} eol
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (dbldot_match.txt):
3400 John V. Smith
3502 Kathy McCue
1401 Bruce Smith
Output:
$ sm_adapter --file=dbldot_match.txt dbldot_match.asl
Matched with 3400 John V. Smith
Failed match
Matched with 1401 Bruce Smith
$
```

Alternative operator

The alternative operator (|) is used to specify an alternate pattern to match if the first pattern fails. This operator can be used in a series to create a list of alternatives. When a pattern in a list of alternatives matches, the remaining alternatives are not tested.

The alternative operator has the lowest level of precedence of all the pattern matching operators. Even the white space operator has a higher level of precedence.

Once an alternate matches, ASL does not back up to test other combinations. For example, if the input to an ASL script is:

```
abc
```

and the pattern matching is:

```
{"a"|"ab"} "c"
```

No match occurs because once “a” matches, “ab” is not tested. ASL compares the pattern “c” with the input character “b” and the pattern fails. In general, when constructing a series of alternates, it is better to place the more complex (or longer) patterns ahead of other patterns.

Note In the example, braces are used around the alternate expression to control how the pattern is evaluated because the alternate operator has the lowest level of precedence. Without the braces, the alternate to the pattern "a" is the pattern "ab" "c". Grouping patterns provides additional information on controlling pattern evaluation.

The following script matches one of three numbers followed by an end-of-line. For the first number of the input file, the data matches the first alternative so nothing else is checked. The second number of the input file does not match any of the alternatives so the pattern fails. The third number does not match either the first or second alternative, but it does match the third.

```
ASL Script (alt_match.asl):
START {
 a:{
 "3400"|"4500"|"4127"} eol
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (alt_match.txt):
3400
3908
4127
Output:
$ sm_adapter --file=alt_match.txt alt_match.asl
Matched with 3400
Failed match
Matched with 4127
$
```

Maybe operator

In cases where data to be matched might or might not exist in the input, use the maybe operator, which is a question mark (?). The maybe operator indicates that a pattern matches regardless of whether the matching data exists. ASL assigns a NULL string to a variable if a pattern with a maybe operator has no match.

The following script matches an employee number (an integer) and a name (of multiple words). The first and third lines of input match. An employee ID and a name exists. Even though there is no employee ID number, the second input line matches because ASL assigns the variable a NULL string.

```
ASL Script (0to1_match.asl):
START {
/* (The rep keyword indicates Repeated Pattern Matches) */
 a:integer? b:rep(word) eol
}
do {
 print("Employee ".b);
 print("Employee ID ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (0to1_match.txt):
4120 Kathy Jones
John L. Doe
3901 David Smith
Output:
$ sm_adapter --file=0to1_match.txt 0to1_match.asl
Employee Kathy Jones
Employee ID 4120
Employee John L. Doe
Employee ID
Employee David Smith
Employee ID 3901
$
```

Grouping patterns

Patterns can be grouped by using braces {}.

Grouping patterns together can extend the information assigned to a variable. For example,

```
x:{word word word}
```

The variable is assigned if the input consists of three strings. Without the braces, x is bound only to the value of the first word match.

Braces also can control how patterns are evaluated. The use of braces can be combined with operators:

```
{word integer}
```

The braces used in this example force the pattern to match a word followed by an integer or nothing at all. A single word or a single integer will not match this pattern.

Precedence of pattern operators

The order of precedence for pattern matching is as follows:

1 {}

2 :

3 whitespace . ..

4 ?

5 |

The alternative operator has the lowest precedence. For example, suppose there are four patterns: A, B, C, and D.

```
A B|C D
```

is equivalent to

```
{A B}|{C D}
```

The example matches an input of A followed by B, or it matches an input of C followed by D.

Grouping patterns by using the braces has the highest level of precedence. For the patterns A, B, C, and D:

```
{A|B} C D
```

is *not* equivalent to

```
A|B C D
```

Pattern elements

Pattern elements are the building blocks for more complex patterns.

String matches

A string match is a match to one or more characters. An example of a string match is:

```
RULE {
 “def”
}
```

If the “def” pattern is compared with an input string of “defdefeedef,” the rule matches twice and fails when it reaches the “eedef.”

In the following script, the pattern used for matching is def. The pattern matches def twice and fails with the input of eed. When the pattern fails in this script, the DEFAULT rule runs, which matches the entire line.

In most of the examples in this chapter, the pattern to match is surrounded by the following, which assigns the matched data to the variable a:

```
a:{
}
```

The section Assignment operator provides additional information.

It is not necessary to include a variable assignment as part of pattern matching. The matched data is printed by printing the contents of the variable a.

```
ASL Script (str_match.asl):
START {
 a:{
 "def"}
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (str_match.txt):
defdefead
Output:
$ sm_adapter --file=str_match.txt str_match.asl
Matched with def
Matched with def
Failed match
$
```

The match for some characters requires a special syntax. Match syntax for special characters contains the special characters:

Table 4-1. Match syntax for special characters

Character	Syntax	Notes
tab	\t	
single quotation mark (')	\'	The quotation marks surrounding the string that contains this code must be double quotes.
double quotation mark (")	\"	The quotation marks surrounding the string that contains this code must be single quotes.
backward slash (\)	\\	

Table 4-1. Match syntax for special characters (Continued)

Character	Syntax	Notes
carriage return	\r	In most cases, use the eol pattern match instead of carriage return. End-of-line matches provides additional information.
line feed	\n	In most cases, use the eol pattern match instead of line feed. End-of-line matches provides additional information.

Note Pattern matches are by default case-sensitive. Making patterns case-sensitive or case-insensitive discusses how to override the default.

Any character matches

The any() function returns a match if the character that is being evaluated matches any character in *<character_string>*.

```
any(<character_string>)
```

In the following script, the any() function matches with the first four characters of the input file. The next character causes the pattern to fail and the DEFAULT rule runs.

```
ASL Script (any_match.asl):
START {
 a:any("abc")
}
do {  print("Matched with ".a);
}
DEFAULT {  ..eol
}
do {
 print("Failed match");
}
Input (any_match.txt):
bbacxy
Output:
$ sm_adapter --file=any_match.txt any_match.asl
Matched with b
Matched with b
Matched with a
Matched with c
Failed match
$
```

Notany character matches

The notany() function returns a match if the character that is being evaluated does not match any character in the *<character_string>*.

```
notany(<character_string>)
```

In the following script, the notany() function matches the first three characters of the input file. The next character causes the pattern to fail and the DEFAULT rule runs.

```
ASL Script (notany_match.asl):
START {
 a:notany("cx")
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (notany_match.txt):
bbacbxy
Ouput:
$ sm_adapter --file=notany_match.txt notany_match.asl
Matched with b
Matched with b
Matched with a
Failed match
$
```

Char matches

A char match is a match to any character except for a field-separator or an end-of-line. To replace the char() function, the any() function has to have a character string that is 256 characters long. In the following script, the character pattern matches with the letter a, the tab, the letter b, the space and finally the letter c before failing when it tries to match the eol.

```
ASL Script (char_match.asl):
START {
 a:char
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (char_match.txt):
a <tab> b c
Output:
$ sm_adapter --file=char_match.txt char_match.asl
Matched with a
Matched with
```

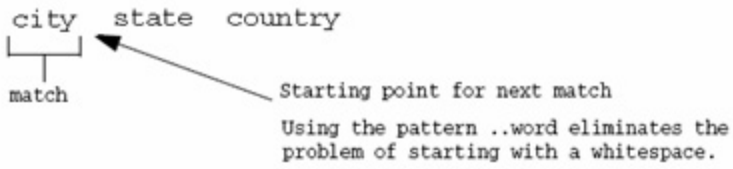
```
Matched with b
Matched with
Matched with c
Failed match
```

Word matches

A word match is a match to a sequence of characters that ends with, but does not include, a delimiting character. Spaces, tabs, field-separators, and ends of line are the default delimiting characters. The delimiting characters can be redefined, as described in Customizing the delimiter.

The following script is an example of a word pattern that is a match of all characters up to, but not including the delimiter. The second time the rule is evaluated, the pattern matching starts with the delimiter. The second word match fails because there are no characters found before the delimiter is found.

```
ASL Script (word_match.asl):
START {
 a:word
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (word_match.txt):
city state country
Output:
$ sm_adapter --file=word_match.txt word_match.asl
Matched with city
Failed match
$
```



Integer matches

The integer pattern matches a string of numeric characters that may or may not be preceded by a minus sign. Any non-numeric character except for a dash (minus sign) is not valid for integer matches. The integer pattern matches the first part of the string:

```
83294IVBXR
```

For example, the following script matches each integer and the end-of-line. The match fails on the third line of data because there is a decimal point. At that point, the integer is matched with 214 and the pattern fails because eol is not a match for .56.

```
ASL Script (int_match.asl):
START {
 a:integer eol
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (int_match.txt):
12300
-375
214.56
Output:
$ sm_adapter --file=int_match.txt int_match.asl
Matched with 12300
Matched with -375
Failed match
$
```

Floating-point number matches

The floating-point number pattern matches a string of numeric characters that may or may not be preceded by a minus sign, and that may or may not include a decimal point followed by other numbers. Any non-numeric character except for a period or a dash (minus sign) is not valid for floating-point matches.

For example, this script matches each number and the corresponding end-of-line. Only the value of float is assigned to the variable so there are no extra lines between the lines of output.

```
ASL Script (float_matches.asl):
START {
 a:float eol
}
```

```
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (float_match.txt):
173
-3.95
3453.45
Output:
$ sm_adapter --file=float_match.txt float_match.asl
Matched with 173
Matched with -3.95
Matched with 3453.45
$
```

Hexadecimal matches

The hexadecimal pattern matches a string of hexadecimal characters. These characters cannot be preceded by a minus sign. The hexadecimal pattern matches any numeric character and the letters a through f. The pattern does not match anything else. When ASL assigns a hexadecimal pattern to a variable, the numeric value is kept but not the hexadecimal value.

The following script matches each hexadecimal number and the corresponding end-of-line.

```
ASL Script (hex_match.asl):
START {
 a:hex eol
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (hex_match.txt):
ff2234
FF
23
Output:
$ sm_adapter --file=hex_match.txt hex_match.asl
Matched with 16720436
Matched with 255
Matched with 35
$
```

Field-separator matches

A field-separator is inserted into input data by an adapter's front end. When using input files, a field-separator is defined to replace a character of input such as a comma, colon, or a vertical bar (|). A field-separator represents a division of data that can include multiple words, integers, strings, and even end-of-line (eol). You can use fs to pattern match a field-separator, as described in Field-separator translation.

Note Do not confuse a field-separator with a delimiter. Even though a delimiter can be used to separate multiple words, it is best used as a word separator. A field-separator separates fields that can contain one or more words. Regardless of the value of the delimiting characters, a field-separator is implicitly defined as a delimiter.

For example, the following script matches a name and an address, which are separated by a field-separator. In the input, the field-separator is a colon (:). The name is assigned from one or more repeated words that come before the field-separator. The address is assigned from the repeated words that follow the field-separator.

The field-separator is defined when the adapter is run with the following option:

```
--field-separator
```

The input is parsed by the front end and the field-separator is placed into the data.

```
ASL Script (fs_match.asl):
START {
 a:rep(word) fs
 b:rep(word) eol
}
do {
 print("Name ".a);
 print("Address ".b);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (fs_match.txt):
Jay Ray:11 Main St
Jane Doe:34 Oak Dr
Output:
$ sm_adapter --field-separator=: --file=fs_match.txt fs_match.asl
Name Jay Ray
Address 11 Main St
Name Jane Doe
Address 34 Oak Dr
$
```

End-of-line matches

One of the markers included in the input is the end-of-line (eol). The eol is added to the input data by the front end of the adapter. To match an end-of-line, use eol.

The following script matches a word and an end-of-line. Notice that the output is different from many of the previous examples.

- The first difference is that there were no messages for Failed match. All of the data matched.
- The second difference is that there is a space between successive lines of output. This is because the eol match is included as part of the output.

```
ASL Script (eol_match.asl):
START {
 a:{word eol}
}
do {
 print("Matched with ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (eol_match.txt):
switch
router
card
Output:
$ sm_adapter --file=eol_match.txt eol_match.asl
Matched with switch
Matched with router
Matched with card
$
```

Repeated pattern matches

The rep() function repeats pattern matches or a rule for a specific number or for one or more matches. The syntax is:

```
rep(<ruleorpattern>[,<number>])
```

The rule or pattern to repeat is specified by using *<ruleorpattern>*. The *<number>* is optional and indicates the number of times to match the pattern. If the *<number>* is not included, the pattern is matched one or more times until it fails or until the pattern following it is matched. The pattern must match at least once.

Samples of the rep() function shows some sample uses of the rep() function. Each sample uses the letter P to denote a pattern.

Table 4-2. Samples of the rep() function

Example	Behavior	Note
rep(P)	P P P P P P ...	Behaves as if a white space operator appears between each occurrence of P.
rep(.P)	P.P.P.P.P.P.P. ...	Behaves as if a dot operator appears between each occurrence of P.
rep(..P)	..P..P..P..P..P ...	The double-dot operator ignores all patterns except for P.

The following ASL script matches two numbers, one or more words, and then an end-of-line.

- The first line of input is matched. There are two numbers followed by two words.
- The second line of input is matched. There are two numbers and three words. The second rep() function is repeated until the end-of-line is reached.
- The third line of input fails. The line does not contain two numbers at the front of the line.

```
ASL Script (repeat_match.asl):
START {
 a:rep(integer,2)
 b:rep(word) eol
}
do {
 print("Matched numbers ".a);
 print("Matched with ".b);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (repeat_match.txt):
3400 4127 cat dog
4 5 goat lamb cow
1 chicken horse
Output:
$ sm_adapter --file=repeat_match.txt repeat_match.asl
Matched numbers 3400 4127
Matched with cat dog
Matched numbers 4 5
Matched with goat lamb cow
Failed match
$
```

Boolean expressions

Boolean expressions can be added to a pattern list. If a Boolean expression fails, the pattern match also fails.

Note Boolean expressions are more commonly used in the filter section of a rule than in the pattern matching block. Unlike Boolean expressions in a pattern, a failing Boolean expression in a filter does not cause the rule to fail.

The following script matches a name and an integer. The Boolean expression is:

```
number>2
```

If the integer is greater than two, the match succeeds.

For every line in the input, the word, the integer and the eol are successfully matched. When the Boolean expression causes the pattern to fail, the DEFAULT rule is executed and Failed Match is printed.

```
ASL Script (bool_match.asl):
START {
 animal:word
 number:integer
 eol
 number>2
}
do {
 print("Animal ".animal);
 print("Number ".number);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (bool_match.txt):
elephants 4
moose 1
tigers 2
giraffes 3
Output:
$ sm_adapter --file=bool_match.txt bool_match.asl
Animal elephants
Number 4
Failed match
Failed match
Animal giraffes
Number 3
$
```

Positional matching

As ASL matches input data, each character is evaluated until a pattern matches or fails. You can use two positional matching functions to specify the position where the parser starts to test a pattern.

- The tab() function
- The len() function

tab() function

With the tab() function, you can skip characters of input data by using the following syntax:

```
tab(<char_num>)
```

The value of *<char_num>* is the number of characters to skip from the starting position when the START rule is invoked. If *<char_num>* is not specified, the function returns the value of the position where the parser starts to test a pattern.

It is not possible to use the tab() function to move backward (right to left) in the datastream. For example, from the first position of a string, you use the tab() function to go to position 20, but, you then cannot go back to position 15. You cannot use the tab() function to return to data already parsed. Also, the tab() function does not skip over markers.

With the assignment operator (:), you can assign all of the characters skipped to the variable or the current position if no characters are specified by using the following syntax:

```
        <variable>:tab(<char_num>)
```

In the following script, the tab() function in this pattern is used to skip the first and middle names. The parser goes directly to the last name of each person in the list. In the input datafile, each field has a fixed length. The last name field starts at position 16.

At position 16, a single word is matched and assigned to a variable. Then, the current starting position of the next pattern is returned. This position varies because the word lengths are not equal.

```
ASL Script (tab_match.asl):
START {
 tab(16) lname:word
 locate:tab() eol
}
do {
 print("Last Name ".lname);
 print("Tab ".locate);}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (tab_match.txt):
```

```
John            Doe
Jane    Deborah Smith
Output:
$ sm_adapter --file=tab_match.txt tab_match.asl
Last Name Doe
Tab 19
Name Smith
Tab 21
$
```

len() function

Another positional matching function is len(). The len() function advances the starting position from its current position, unlike the tab() function that works from the original starting position when the START rule is invoked. The syntax is as follows:

```
len(<char_num>)
```

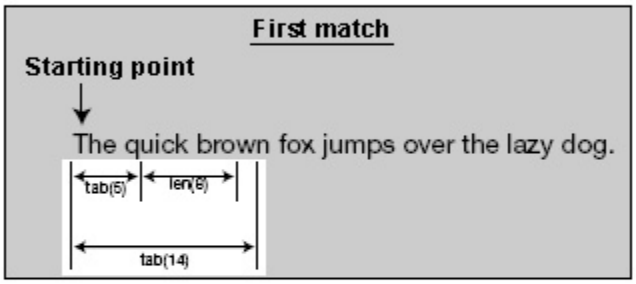
The value of <*char_num*> is the number of characters to advance from its current position. If <*char_num*> is not specified, the function returns the value of the position where the parser would start to test a pattern.

Like the tab() function, use of the assignment operator assigns all characters skipped to the variable. The len() function does not skip over markers.

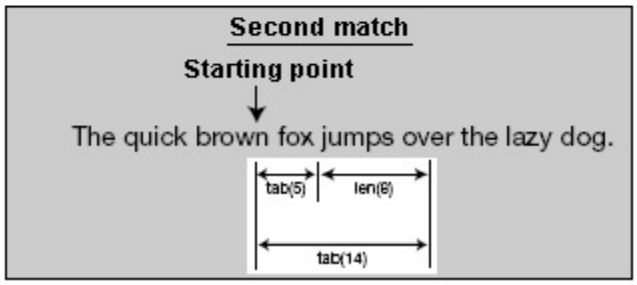
In Example of positional matching, because of the white space operator between each variable assignment, the space between fox and jumps is skipped. The len() function starts with the letter j. It ends at position 14, forcing tab (14) to return an empty string. This problem could be avoided by eliminating the delimiters, as described in Customizing the delimiter.

Figure 4-2. Example of positional matching

```
START {
   a:tab(5)
   b:len(8)
   c:tab(14)
}
```



a is assigned "The q"
b is assigned "uick bro"
c is assigned "w"



a is assigned "n fox"
b is assigned "jumps ov"
c is assigned " "

Peek() function

The peek() function returns a value of TRUE or FALSE depending on whether the pattern passed to it matches the current input string. Use of the peek() function does not advance the starting position, so input can be scanned before other pattern matching. The peek() function stops scanning for a pattern when it reaches a marker.

```
peek(<pattern>)
```

The following script looks for the word horse in each input string. The double-dot operator is necessary in the pattern so that the peek() function will match any occurrence of the word horse in a pattern and not just when it appears first.

If the word horse is found in the input string, the peek() function is TRUE and the rule GETLINE executes. GETLINE assigns every word in the string to the variable a. The value of a is printed and the peek() function starts with the next input string. If the peek() function is FALSE, the rule GETLINE does not execute, but the DEFAULT rule does execute.

```
ASL Script (peek_match.asl):
START {
 peek(..'horse') GETLINE
}
GETLINE{
 a:rep(word) eol
```

```
}
do {
 print("Horse found: ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (peek_match.txt):
moose horse camel
elephant mule camel
goat llama horse
horse goat llama
Output:
$ sm_adapter --file=peek_match.txt peek_match.asl
Horse found: moose horse camel
Failed match
Horse found: goat llama horse
Horse found: horse goat llama
$
```

Not() function

The not() function acts as a logical NOT. If the pattern argument of the function matches, the not() function returns failure to the match. When the argument does not match, the not() function returns a successful match. The starting point does not advance when the not() function is used.

For example, the following script looks for input strings that do not contain the word horse.

- If the word horse is found in the input string, the peek() function matches but the not() function returns a failed match. In this case, the rule GETLINE does not execute.
- If the peek() function does not match, the not() function returns a successful match so the rule GETLINE executes.

```
ASL Script (not_match.asl):
START {
 not(peek(..'horse'))
 GETLINE
}
GETLINE{
 a:rep(word) eol
}
do {
 print("No horse: ".a);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
```

```
Input (not_match.txt):
moose horse camel
elephant mule camel
goat llama horse
horse goat llama
Output:
$ sm_adapter --file=not_match.txt not_match.asl
Failed match
No horse: elephant mule camel
Failed match
Failed match
$
```

Matching through other rules

Pattern matching can be divided among various rules. From the START rule, other rules can be called. Rules that are called can have any or all of these sections: patterns, filters, or actions.

Operators or functions that work with patterns also work with rules. For example, the rep function can repeat a rule:

```
START {
rep(<RULE1>) <RULE2>
}
```

In the following script, two rules are called from the START rule, PERSON and LOCATION. Each rule uses the same pattern to find words until an end-of-line, but assigns values to different variables. The values of the variables are printed from an action block under START.

ASL Script (other_rules.asl):

```
START {
 PERSON LOCATION
}
do {
 print("Name: ".name);
 print("Address: ".address);
}
PERSON {
 name:rep(word) eol
}
LOCATION {
 address:rep(word) eol
}
Input (other_rules.txt):
Jay Ray
11 Main St
Jane Doe
34 Oak Dr
Output:
$ sm_adapter --file=other_rules.txt other_rules.asl
Name: Jay Ray
```

```
Address: 11 Main St
Name: Jane Doe
Address: 34 Oak Dr
$
```

In the next example, the DATE and PLACE rules are called from START. The DATE and PLACE rules are separated by an alternative operator. For each line of text where DATE matches, PLACE is not tested. If DATE does not match, PLACE is tested.

Notice how the variable assignment works in this example. When either DATE or PLACE matches, the entire matched string is returned and assigned to x. The end-of-line is added to the string, which affects the output.

```
ASL Script (operators_rules.asl):
START {
 x:{DATE|PLACE}
}
do {
 print("Day or place ".x);
}
DATE {
 integer.rep("/".integer,2)
 eol
}
PLACE {
 integer word word eol
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (operators_rules.txt):
01/05/99
11 Main St
camel horse mule
10/31/01
34 Oak Dr
Output:
$ sm_adapter --file=operators_rules.txt operators_rules.asl
Day or place 01/05/99
Day or place 11 Main St
Failed match
Day or place 10/31/01
Day or place 34 Oak Dr
$
```

End-of-file matches

The end of a file cannot be matched through pattern matching. However, there is a special rule, EOF, that runs when the end of a file is reached. Special rules provides additional information about the EOF rule.

Special variables

Special variables control how ASL evaluates patterns. These variables define the delimiters between words, case-sensitivity, and even the data input.

Customizing the delimiter

You can redefine the delimiters by assigning a string to the built-in variable delim. Use the following syntax to define new delimiters:

```
delim=<string>;
```

The *<string>* defines the set of characters to use as delimiters.

The delimiter is specified at the beginning of a script or at the beginning of a rule, before any pattern matching.

- If a delimiter statement is placed before all of the rules, the delimiters apply to all rules.
- If a delimiter statement is placed in a rule, ASL overrides any previous delimiter definitions.

  In this example, the delimiter is ":?" for RULE2, but "|" for RULE1 and RULE3.

  ```
  delim=":?";
  START {
      RULE1 RULE2
  }
  RULE1 {
     delim="|";
   RULE3
  }
  ```

  The new delimiters override all of the default delimiters except for eol and fs. Each character included in the *<string>* is a separate delimiter.

  ```
  delim = ":?";
  ```

  When the string :? is assigned to delim, it defines two delimiters: a colon and a question mark. If input data contained the combination (“:?”), ASL interprets the data as two delimiters in series.

  The match for some characters requires a special syntax. To use tabs and returns as delimiters, use the special syntax within the quotation marks, as described in Customizing the delimiter.

  Note For most cases, use the eol pattern match instead of line feed or carriage return. The eol character is a delimiter by default and cannot be overridden.

An alternative to using the delim variable is to use a combination of the rep() function and the notany() function. For example:

```
x:rep(notany(":?"))
```

In the example, ASL assigns to the variable x everything that appears up to a question mark or a colon.

Making patterns case-sensitive or case-insensitive

By default, pattern matching is case-sensitive. To override the default case-sensitivity, use the case variable. This variable can be used at the beginning of a script or at the beginning of a rule, before any pattern matching. The syntax of the case variable is:

```
case=[exact]|[ignore];
```

To match case-sensitive patterns, set case equal to exact. For patterns where case does not matter, set case equal to ignore.

If a case variable is placed before all of the rules, the case setting applies to all rules. This can be overridden in a rule by using another case variable.

Switching the input of a pattern

For complicated inputs, the pattern matching can be divided among different rules by using the input variable. The input variable is assigned to a value which ASL uses for the pattern matching that follows the input. The input variable must come at the beginning of a rule, before any pattern matching but not at the beginning of a script. If the value assigned to the input variable does not end with an eol, an eol is appended to the value. The syntax is as follows:

```
input=<value>;
```

For example, the following script looks for lines of input that start with the string “Error:”. When it finds those strings, the rest of the words are assigned to the variable desc and the rule PROCESSDESC is called.

PROCESSDESC takes the variable desc as its input and performs pattern matching based on the input. The error level and error message are printed.

The START rule processes the next line data and, if it is an error, PROCESSDESC runs again. At the end of the input file, the rule EOF runs. This rule prints a statement that the errors are processed.

```
ASL Script (input_ex.asl):
START {
 "Error:" desc:rep(word) eol
 PROCESSDESC
}
PROCESSDESC {
 input=desc;
```

```
 errornumber:integer
 errorlevel:word
 errormsg:rep(word)
 eol
}
do {
 print(errorlevel." ".errormsg);
}
DEFAULT {
 ..eol
}
do {
 print("No Error");
}
EOF
do {
 print();
 print("Errors Processed");
}
Input (input_ex.txt):
Error: 2568 Severe Can't process
Status: 2358 Starting backup
Error: 1202 Warning Bad data
Error: 923 Critical Wrong Number
Output:
$ sm_adapter --file=input_ex.txt input_ex.asl
Severe Can't process
No Error
Warning Bad data
Critical Wrong Number
Errors Processed
$
```

Filters

A filter determines whether the action block of a rule executes. When a filter fails, the subsequent action block does not execute. Unlike a pattern, if a filter fails, the rule it is called from does not fail.

A filter has the form:

```
filter {
{<filter-list>}
}
```

The *<filter-list>* must be a single Boolean expression. ASL evaluates this expression from left to right. If it is TRUE, ASL executes the actions in the do block. If it is FALSE, ASL does not execute the actions in the do block.

Filter operators are listed in Filter operators.

- Parentheses have the highest level of precedence.

- The logical operators && and || have the lowest level of precedence.

  Although the filter-list must be a single Boolean expression, you can use the logical AND and logical OR to test for more than one condition.

Table 4-3. Filter operators

Operator	Definition
()	Grouping operators
+ - * / %	Arithmetic operators
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&&	Logical AND
||	Logical OR

In the following script example, the filter controls whether an action is performed. The script matches a word and an eol. The value of the word is assigned to x.

The filter is true when x has the value “switch”. For the input file, this occurs in the first line and the fourth line. The action block is executed twice. The DEFAULT rule is never executed because the pattern always matches.

```
ASL Script (simple_filter.asl):
START {
 x:word eol
}
filter {
 x=="switch"
}
do {
 print("Filter match: ".x);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (simple_filter.txt):
switch
router
card
switch
Output:
```

```
$ sm_adapter --file=simple_filter.txt simple_filter.asl
Filter match: switch
Filter match: switch
$
```

In the next example, the script matches a word and an eol. The value of the word is assigned to x. The filter is true when x has the value “switch,” or when it has the value “router.”

```
ASL Script (other_filter.asl):
START {
 x:word eol
}
filter {
 x=="switch" || x=="router"
}
do {
 print("Filter match: ".x);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (other_filter.txt):
switch
router
card
switch
Output:
$ sm_adapter --file=simple_filter.txt simple_filter.asl
Filter match: switch
Filter match: router
Filter match: switch
$
```

Filters can be useful for debugging. Setting a single variable at the beginning of a script can affect the actions that a script performs.

```
debug = TRUE;
START {
x:word eol
}
filter {
debug==TRUE
}
```

Actions

5

This chapter includes the following topics:

- Action block
- Iteration and control statements
- Function reference
- Returning values
- Passing arguments to functions
- Calling rules as functions from do blocks
- Exception handling

Action block

The action block is executed for a rule when pattern matching succeeds (or does not exist) and the filter is TRUE (or does not exist). An action block uses the following form:

```
do {
<action-list>
}
```

Action lists can occur only within a do block. An action list is composed of statements. Statements include variable assignments, conditional actions, and rules. Each statement must end with a semicolon. An example of an action block is as follows:

```
do {
statement_1;
statement_2;
statement_3;
}
```

Operators

In the action block, statements are terminated with a semicolon (;). Although a semicolon is not required to immediately follow the statement, it must occur before the next statement.

All other operators used in the action block are described in *"Operators" on page 39*.

Iteration and control statements

Iteration and control statements are not followed by a semicolon. They may control other statements that require the use of semicolons.

foreach statement

The foreach statement iterates through all of the members of a list or a table. The syntax is:

```
foreach <variable> (<list_or_table_name>) {statements}
```

When the foreach statement is used with a list, the variable stores the value of each member of the list every time the foreach statement loops. The members returned are in numerical order by their list indices.

The following script matches numbers from an input file and stores the values in a list. The foreach statement cycles through every value stored in the list. Each value is printed and added to a subtotal, which is also printed.

```
ASL Script (foreachlist_do.asl):
data = list();
total = 0;
START {
 rep(NUMBERS)|eol
}
NUMBERS {
 x:integer
}
do {
 data += x;
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
EOF
do {
 foreach i (data) {
  total = total + i;
  print("Number: ".i);
  print("Subtotal ".total);
 }
}
Input (foreachlist_do.txt):
2 4 8
Output:
$ sm_adapter --file=foreachlist_do.txt foreachlist_do.asl
Number: 2
Subtotal: 2
Number: 4
```

```
Subtotal: 6
Number: 8
Subtotal: 14
$
```

When the foreach statement is used with a table, the variable receives the keys to the members of the table. Tables have no inherent order.

The next script matches numbers and words from an input file and stores the values in a table. The foreach statement cycles through every key stored in the table. Each key is printed as well as the associated value. The order in which the members of the table are printed is not necessarily the order in which the members were added to the table.

```
ASL Script (foreachtable_do.asl):
pop = table();
total = 0;
START {
 rep(NUMBERS)
}
NUMBERS {
 count:integer animal:word
 eol
}
do {
 pop[animal] = count;
}
DEFAULT {..eol}
do {print("Failed match");}
EOF
do {
 foreach w (pop)
 {
  print("Animal ".w);
  print("Number ".pop[w]);
 }
}
Input (foreachtable_do.txt):
2 cows
4 cats
6 dogs
8 chickens
Output:
$ sm_adapter --file=foreachtable_do.txt foreachtable_do.asl
Animal chickens
Number 8
Animal cows
Number 2
Animal cats
Number 4
Animal dogs
Number 6
$
```

while statement

The while statement repeats a block of statements while a condition is true. The syntax is as follows:

```
while <condition> {<statements>}
```

The following script has no pattern matching. The variable x is printed and incremented by one until it is no longer less than five.

```
ASL Script (while_do.asl):
x=0;
START
do {
 while x < 5
 {
  print(x);
  x=x+1;
 }
 stop();
}
Input:
none
Output:
$ sm_adapter while_do.asl
0
1
2
3
4
$
```

if else statement

The if else statements control what actions are performed based on the results of a conditional test. The syntax is as follows:

```
if (<conditional_test>) {<statements>} [else {<statements>}]
```

The else portion of the statement is optional.

Note The braces around the statements are required, even if there is only one statement.

In the following example, the if statements check whether a value to use in a table exists as a key or not. Each key in a table must be unique. In the script, if a key exists, the value associated with the key is added to the new value.

This script uses two if statements.

- The first if statement tests whether any of the existing keys match the name of the animal ready to be loaded into the table. If the animal name matches any of the existing keys, a flag is set to TRUE.

- The second if statement tests whether the flag is FALSE. If the flag is FALSE, the animal name and number are added to the table. If the flag is TRUE, the number of animals are added to the animal count already stored in the table.

```
ASL Script (if_do.asl):
z = table();
START {
 rep(IMPORT)
}
IMPORT {
 animal:word
 count:integer
 eol
}
do {
 test="FALSE";
 foreach i (z)
 {
  if (i == animal)
  {
   test="TRUE";
  }
 }
 if (test=="FALSE")
 {
  z[animal]=count;
 }
 else
 {
  z[animal]=z[animal]+count;
 }
}
EOF
do {
 foreach i (z)
 {
  print(i." count ".z[i]);
 }
}
Input (if_do.txt):
dog 3
cat 4
canary 3
cat 2
dog 5
dog 1
cat 1
Output:
$ sm_adapter --file=if_do.txt if_do.asl
dog count 9
cat count 7
canary count 3
$
```

break statement

The break statement stops processing statements inside of a loop and exits the loop. The break statement is valid only inside of foreach or while loops.

This example is similar to the if statement example except that a break statement has been added. if else statement provides additional information about the if statement.

The break statement causes the test in the foreach statement to stop once the condition is TRUE rather than continuing to test.

```
ASL Script (break_do.asl):
z = table();
START {
 rep(IMPORT)
}
IMPORT {
 animal:word
 count:integer
 eol
}
do {
 test="FALSE";
 foreach i (z)
 {
  if (i == animal)
  {
   test="TRUE";
   break;
  }
 }
 if (test=="FALSE")
 {
  z[animal]=count;
 }
 else
 {
  z[animal]=z[animal]+count;
 }
}
EOF
do {
 foreach i (z)
 {
  print(i." count ".z[i]);
 }
}
break_do.txt (Input):
dog 3
cat 4
canary 3
cat 2
dog 5
```

```
dog 1
cat 1
Output:
$ sm_adapter --file=break_do.txt break_do.asl
dog count 9
cat count 7
canary count 3
$
```

continue statement

The continue statement starts the next iteration of a foreach statement or while statement. When ASL invokes the continue statement, the remaining statements in the foreach or while loop are skipped. The continue statement is not valid outside of a loop.

In the following example, an if statement checks whether values in a table are positive. For any number less than or equal to 0, the continue statement advances the foreach loop.

```
ASL Script (continue_do.asl):
x = table();
START {
 LOAD
}
LOAD {
 y:word z:integer eol
}
do {
 x[y] = z;
}
EOF
do {
 foreach w (x)
  //skip if not positive
  {if (x[w]<=0)
   {continue;
   }
  print(w." ".x[w]);
  }
}
Input (continue_do.txt):
elephant 0
goat -1
worm 100
chicken 7
dog 2
cat 0
moose 9
Output:
$ sm_adapter --file=continue_do.txt continue_do.asl
dog 2
```

```
moose 9
chicken 7
worm 100
$
```

Function reference

This section discusses the different types of functions that are available.

glob() function

The glob() function returns a TRUE or FALSE based on whether a pattern matches a string. The glob() function can be used as a conditional test for an if or while statement. The syntax is as follows:

```
glob(<pattern>, <string>)
```

The expression for a glob() function is a series of characters (or patterns) that are matched against incoming character strings. You use these expressions when you define matching criteria.

Matching is done strictly from left to right, one character or basic glob expression at a time. Characters that are not part of match constructs will match themselves. The pattern and the incoming string must match completely. For example, the pattern abcd does not match the input “abcde” or” abc”.

A compound glob pattern consists of one or more basic wildcard patterns separated by ampersand (&) or tilde (~) characters. A compound wildcard pattern is matched by attempting to match each of its component basic wildcard patterns against the entire input string.

The pattern used in the glob() function is not the same pattern used in the pattern block of an ASL script. The symbols that describe the pattern are described in Symbols for a glob pattern .

Table 5-1. Symbols for a glob pattern

Symbol	Description
*	Matches an arbitrary string of characters. The string can be empty.
?	Matches any single character.
^	Acts as a NOT. Use this in conjunction with other symbols or characters.
[set]	Matches: ■ Any single character that appears within [set], or ■ Any single character that is not in the set if the first character of [set] is (^). A hyphen (-) within [set] indicates a range, so that [a-d] is equivalent to [abcd]. The character before the hyphen (-) must precede the character after it or the range will be empty. The character (^) in any position except the first, or a hyphen (-) at the first or last position, has no special meaning.
<n1-n2>	Matches numbers in a given range. Both n1 and n2 must be strings of digits, which represent non-negative integer values. The matching characters are a non-empty string of digits whose value, as a non-negative integer, is greater than or equal to n1 and less than or equal to n2. If either end of the range is omitted, no limitation is placed on the accepted number.

Table 5-1. Symbols for a glob pattern (Continued)

Symbol	Description
|	Matches alternatives. For example, ”ab|bc|cd” without spaces matches exactly the three following strings: “ab”, “bc”, and “cd”. A vertical bar (|) as the first or last character of a pattern accepts an empty string as a match.
\	Removes the special status, if any, of the following character. Backslash (\) has no special meaning within a set ([set]) or range (<n1-n2>) construct.
&	“And Also” for a compound wildcard pattern. If a component basic wildcard pattern is preceded by & (or is the first basic wildcard pattern in the compound wildcard pattern), it *must* successfully match.
~	“Except” for a compound wildcard pattern (opposite function of &). If a component basic wildcard pattern is preceded by ~, it *must not* match.

Note Spaces are interpreted as characters and are subject to matching even if they are adjacent to operators like “&.”

If the first character of a compound wildcard expression is an ampersand (&) or tilde (~) character, the compound is interpreted as if an asterisk (*) appeared at the beginning of the pattern. For example, the following two expressions are equivalent to each other:

```
~*[0-9]*
*~*[0-9]*
```

Both of these expressions match any string that does not contain any digits.

A trailing instance of an ampersand character (&) can match only the empty string. A trailing instance of a tilde character (~) can be read as “except for the empty string.”

The following script prints sentences that contain the characters “Ship” or “ship.”

```
ASL Script (glob_do.asl):
START {
 sentence:rep(word) eol
}
do {
 if (glob("*[Ss]hip*",sentence))
  {print(sentence);}
}

Input (glob_do.txt):
I have a ship that floats.
Shipping is a big industry in Hong Kong.
The dog ate my homework.
I fell down and hurt my hip.
A boat is much smaller than a ship.
Output:
$ sm_adapter --file=glob_do.txt glob_do.asl
```

```
I have a ship that floats.
Shipping is a big industry in Hong Kong.
A boat is much smaller than a ship.
$
```

stop() function

The stop() function stops the current thread. No arguments are passed to the function. A call to the stop() function does not return.

If you call stop() in an ASL thread, and that thread is in a process that has other threads, the effect is to simply stop that thread. However, when the thread that is being stopped is the only thread that is running, then stopping that thread effectively shuts down the adapter (because the adapter shuts down when there are no more active threads).

The syntax is as follows:

```
stop()
```

quit() function

The quit() function stops the back end, or the Domain Manager in the case of inflow adapters. No arguments are passed to the function. When the quit() function is encountered, the ASL script immediately ceases execution.

A call to the quit() function does not return, except when you run sm_adapter with the **-**s option to connect the adapter to a remote Domain Manager, and then you call quit(). In this case, the remote Domain Manager initiates its shutdown procedure and the function does return. The calling script continues running while the remote Domain Manager shuts down (which may take more than a few seconds). The script may even interact with the back-end Domain Manager during the interval between calling quit() and the Domain Manager actually shutting down. And so long as the script never tries to interact with the back-end Domain Manager once it has shut down, the script can continue running indefinitely.

The syntax to stop a remote adapter is as follows:

```
quit();
```

The syntax to stop the current adapter is as follows:

```
self->quit();
If you launch sm_adapter and do not specify a back end, the adapter becomes its own back end.  In such
cases, quit() does the same thing as self->quit().
```

defined() function

The defined() function returns TRUE if a variable has a value.

This example is a refinement of the if example. The section if else statement provides additional information.

By using the defined() function, several lines of the script are removed, including a foreach statement and an if statement.

The single if statement verifies whether a table key entry exists in the table. If the name exists, the number of animals is added to the animal count already stored in the table. If the name does not exist, the animal name and number are added to the table.

```
ASL Script (defined_do.asl):
z = table();
START {
 rep(IMPORT)
}
IMPORT {
 animal:word
 count:integer
 eol
}
do {
 if (defined(z[animal]))
 {
  z[animal]=z[animal]+count;
 }
 else
 {
  z[animal]=count;
 }
}
EOF
do {
 foreach i (z)
 {
  print(i." count ".z[i]);
 }
}
Input (defined_do.txt):
dog 3
cat 4
canary 3
cat 2
dog 5
dog 1
cat 1
Output:
$ sm_adapter --file=defined_do.txt defined_do.asl
dog count 9
cat count 7
canary count 3
$
```

undef() function

The undef() function clears a variable, including lists and tables, or a table member. A list entry cannot be undefined. An undefined variable has no value. Other statements and functions cannot use it until it is reassigned a value.

Note The undef() function cannot be used to undefine a global or static variable.

sizeof() function

The sizeof() function returns a number depending on the value passed to the function. The sizeof() function converts any values (except for a list or a table) to a string, and returns the number of characters in the string. For lists and tables, the sizeof() function returns the number of defined members.

The syntax is one of the following:

- sizeof(<*value*>)
- sizeOf(<*value*>)

  The following script matches a line of text in a file. The length of each line is measured and printed.

```
ASL Script (sizeOf_do.asl):
START {
 x:rep(word) eol
}
do {
 y = sizeof(x);
 print("Length ".y);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (sizeOf_do.txt):
This has a size of 21
size of 9
Output:
$ sm_adapter --file=sizeof_do.txt sizeof_do.asl
Length 21
Length 9
$
```

substring() function

The substring() function returns a string. The returned string is a piece of the string passed to the function. The syntax is as follows:

```
substring(<string>, <starting_position>, <characters_to_return>)
```

The *<starting_position>* value indicates the beginning of the new substring taken from the *<string>*. The first position of a string is position 0. The *<characters_to_return>* value indicates the number of characters to return starting with starting_position.

For the following script, the input is a 12-line text file. Each line has 10 characters.

For the first line of input, the substring() function returns the entire string. For subsequent lines of input, a character is removed from the beginning of the string.

When the starting position is greater than the number of characters in the original string, no characters are returned.

```
ASL Script (substring_do.asl):
y=0;
START {
 x:word eol
}
do {
 newstring=substring(x,y,10);
 print("New string ".newstring);
 y = y+1;
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (substring_do.txt):
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
Output:
$ sm_adapter --file=substring_do.txt substring_do.asl
New string 0123456789
New string 123456789
```

```
New string 23456789
New string 3456789
New string 456789
New string 56789
New string 6789
New string 789
New string 89
New string 9
New string
New string
$
```

toLower() function

The toLower() function returns a string, converting any uppercase letters in the original string to lowercase. The syntax is as follows:

```
toLower(<string>);
```

For example, this script converts strings so that all of the letters following the first character in each word are lowercase.

For each word in a person's name, the initial character is assigned to the initial variable and the remaining characters (if there are any) are assigned to the rest variable. The toLower() function converts the string stored in the rest variable to lowercase letters. Then, the characters stored in the initial and the rest variables are assigned to the fullname variable.

Once each name has been read and converted, ASL prints it before the next name is read and converted.

```
ASL Script (toLower_do.asl):
START {
do {fullname="";}
 rep(READNAME) eol
}
do {
 print(fullname);
}
READNAME {
 initial:char.rest:word?
}
do {
 rest = toLower(rest);
 name=initial.rest;
 fullname = fullname.name." ";
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (toLower_do.txt):
```

```
JOHN DOE
JANE Q PUBLIC
Output:
$ sm_adapter --file=toLower_do.txt toLower_do.asl
John Doe
Jane Q Public
$
```

toUpper() function

The toUpper() function returns a string, converting any lowercase letters in the original string to uppercase. The syntax is as follows:

```
toUpper(<string>);
```

The following script assigns the first letter of each word to the variable a. The a variable is concatenated with another variable, name, to form an acronym. The toUpper() function capitalizes the entire acronym before it is printed.

```
ASL Script (toUpper_do.asl):
name="";
START {
 rep(FIRSTLETTER) eol
}
do {
 print(toUpper(name));
}
FIRSTLETTER {
 a:char.word?
}
do {
 name=name.a;
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (to Upper_do.txt):
I see several tall buildings
New York City
Output:
$ sm_adapter --file=toUpper_do.txt toUpper_do.asl
ISSTB
NYC
$
```

print() function

The print() function sends strings it receives to each of the active logs in the Domain Manager process. There is no automatic translation of the message text. The syntax is as follows:

```
print(<argument>);
```

The value of the argument passed to the print() function can be any data type. The argument passed to a print() function is converted to a string before it is printed.

To print special characters, use the syntax listed in Print function special characters.

Table 5-2. Print function special characters

Character	Syntax	Notes
tab	\t	
single quote (')	\'	The quotation marks surrounding the string containing this code must be double quotes.
double quote (")	\"	The quotation marks surrounding the string containing this code must be single quotes.
backward slash (\)	\\	
carriage return	\r	
line feed	\n	

The output of the print() function is dependent on the value of the computed attribute, enableLegacyPrint, in GA_Driver. When ASL scripts are run by using sm_adapter, the enableLegacyPrint attribute is set to TRUE. In all other cases it is set to FALSE. The default value of the enableLegacyPrint attribute is FALSE.

When the enableLegacyPrint is set to FALSE, the print output includes the time, date, thread id, and ASL script. When enableLegacyPrint is set to TRUE, only the string that is sent to the print() function is printed.

You need an object handle to a GA_Driver object to set the enableLegacyPrint attribute. You can use "Null GA_Driver" which always exists.

In this example, the value of the enableLegacyPrint attribute is changed. The output of the print() function is also included.

```
START { } do {
 print("default");
 drv = object("GA_Driver", "Null GA_Driver");
 drv->enableLegacyPrint = FALSE ? LOG,STOP;
 print("new style");
 drv->enableLegacyPrint = TRUE ? LOG,STOP;
 print("legacy");
 stop();
}
```

Output for "sm_adapter --output /tmp/foo.asl" :

```
default
[February 6, 2009 12:19:03 AM EST +527ms] t@1088432480 main
ASL_MSG-*-ASLP-/tmp/foo.asl: new style
legacy
```

sleep() function

The sleep() function temporarily stops the adapter. The syntax is:

```
sleep(<time_in_seconds>);
```

The sleep() function is passed a number, which represents the amount of seconds to sleep. The sleep() function returns TRUE.

time() function

The time() function works with and without an argument. Without an argument, this function returns the current system time. With a numeric argument, the time() function adds the argument (as seconds) to the date and time January 1, 1970 00:00:00 GMT and returns the new time and date based on the current time zone information.

The output type of the time() function can be controlled through type casting. By default, the function returns a string. If the function is converted to a numeric, the output is converted into an integer that represents the number of seconds since January 1, 1970 00:00:00 GMT.

For example:

```
x = string(time());
print(x);
```

The string() function returns the time in the following format:

```
DD-MONTH-YYYY HH:MM:SS
```

However, the following numeric function returns a number similar to 958159632.

```
x = numeric(time());
print(x);
```

Type conversions also work when a value is passed as an argument to the function.

The following script prints the date and time, waits five seconds, and prints the date and time again. It also converts a number read from the input file into a date and time.

```
ASL Script (time_do.asl):
START {
 y:integer eol
}
```

```
do {
 print(time());
 sleep(5);
 print(time());
 x = time(y);
 print(x);
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
Input (time_do.txt):
987517893
Output:
$ sm_adapter --file=time_do.txt time_do.asl
12-May-2000 03:28:47
12-May-2000 03:28:52
17-Apr-2001 10:31:33
$
```

trace() function

The trace() function turns on or off an internal trace() function that traces how the code is parsing the passed parameters. You can turn on the function at a specific point in the code to validate that line. This function is useful when creating a rule and validating what is being matched.

The trace() function gives you access, from within ASL, to command line options available with the sm_adapter command with the following exceptions:

- There is no analog of sm_adapter - -traceRules because all rules have already been compiled by the time you could turn on tracing of rule compilation.
- Calling the trace() function as trace("input",TRUE), turns on tracing at the front end, and has no command line equivalent under sm_adapter.

  The syntax is as follows:

  ```
  trace(<string>, <BOOLEAN>)
  ```

  The *<BOOLEAN>* argument determines whether tracing is turned on or off.

  Valid values for string lists valid values for *<string>*.

Table 5-3. Valid values for string

Value	Description
parser	Matches the input stream against the rule.
input	Traces input from the front end.
server	Traces the interaction of the remote wire protocol with the server.

For example:

```
START {
 x:rep(word) eol
 }
 do {
 trace ("parser", TRUE);
 trace ("input", TRUE);
 trace ("server", TRUE);
 }
```

hexToString() function

The hexToString() function converts values with a hexadecimal format into their equivalent ASCII characters. The function pairs the hexadecimal values and converts each pair. The syntax is as follows:

```
hexToString(<value>)
```

getRuleFileName() function

The getRuleFileName() function returns the filename of the currently executing rule file. The syntax is as follows:

```
getRuleFileName([<fullname>])
```

The *<fullname>* argument is optional. If it is omitted or passed as FALSE, only the base name of the rule filename is returned. Otherwise, the full name, including the path, is returned.

Returning values

By default, a rule returns the characters it matches. A patternless rule returns an empty string. Like other patterns, the returned value from a rule can be assigned to a variable.

```
a:RULE_A
```

The return statement overrides values returned by a rule. The syntax is:

```
return <value>;
```

ASL exits a rule immediately after a return statement. It does not execute any statements that follow a return statement.

The next script reads input that consists of a person's name followed by a field-separator and an address. Both the name and address are corrected for capitalization: an uppercase letter at the beginning of a word, followed with lowercase letters for the rest of the word.

In the START rule, two variables are assigned values returned from the READWORDS rule. Every time the pattern in the START rule matches, the two variables are printed.

The READWORDS rule is a repetition of pattern matching followed by an action block. The action block converts the data so that the first letter of a word is capitalized and the rest of the letters are lowercase. The word is added to a string called tempstr.

At the end of the READWORDS rule, the return statement returns the correctly capitalized string. If there was no return statement, the READWORDS rule would return the string as it was read during the pattern matching.

```
ASL Script (return_do.asl):
START {
 fullname:READWORDS fs
 address:READWORDS eol
}
do {
 print(fullname.address);
}
READWORDS {
 do {tempstr = "";}
 rep(READ_FIX)
do {return tempstr;
 }
}
READ_FIX {
 initial:char.rest:word?
}
do {
 initial = toUpper(initial);
 rest = toLower(rest);
 fixed=initial.rest;
 tempstr = tempstr.fixed." ";
}
DEFAULT {
 ..eol
}
do {
 print("Failed match");
}
Input (return_do.txt):
John Q Doe:11 MAin St.
JANE PUblic:387 OAK DR.
HENry HUDSON:9 ELM rd.
Output:
$ sm_adapter --field-separator=: --file=return_do.txt return_do.asl
John Q Doe 11 Main St.
Jane Public 387 Oak Dr.
Henry Hudson 92 Elm Rd.
$
```

Passing arguments to functions

Rules can act as functions, which can be called from both a pattern block and an action block. You can pass values to rules in ASL and receive values back by using the return statement, as described in *"Returning values" on page 92*. The syntax is as follows:

```
RULE(<variablename>)
```

The argument of a rule is a local variable.

The following script reads and counts the words in a file. The READWORD rule is passed an argument that is used as a counter. The count variable behaves as a local variable.

```
ASL Script (rule_arg.asl):
START {
 READWORD(1)
}
READWORD(count) {
 local y = "end";
 y:word
 READWORD(count+1)|eol
}
do {
 print(y." ".count);
}
Input (rule_arg.txt)
dog cat goat
Output:
$ sm_adapter --file=rule_arg.txt rule_arg.asl
end 4
goat 3
cat 2
dog 1
$
```

Calling rules as functions from do blocks

Another rule may be called within any do block. Functions have the following generalized syntax.

```
FunctionName (arguments)
do
{
 ... ASL statement(s) ...
}
```

Any valid ASL expression may be used in the body of a function. This includes calls to other functions and recursive function calls.

Function may have any number of arguments, including no arguments, which are any data type that is supported by ASL.

Functions optionally return a value, which are also of any ASL data type. The return value may be a constant, or a variable declared within the scope of the function do block, as described in Returning values.

A function must be defined in the same ASL script where it is called, and may be defined before or after the location where it is referenced. The general syntax used to invoke a function references the function by name, passes the number and type of arguments expected by the function, and handles the return value, as shown.

Note Not all functions are required to return a value as shown in the example.

```
do {
  ... ASL statement(s) ...
  arg1 = x;
  arg2 = y;
  ReturnValue = FunctionName (arg1, arg2);
  ... handle ReturnValue ...
}
```

A function may be defined only outside of a do block. That is, a function may not be defined within another function with the intent of referencing the inner function from another do block.

The following is an example of a recursive function definition and call.

```
START
{  .. eol  }
do {
  foreach i (list(0,1,2,3,4,5)) {
   print (i.”! = “.factorial(i));
  }
}
factorial(x)
do {
  if (x > 1)
   return (x * factorial(x-1));
  else
   return (x);
```

}

This example produces the following results.

```
0! = 0
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Exception handling

Exception handling can be added to any action components. Exception handling determines how errors from statements or functions are treated. The syntax is as follows:

```
<action> ? [LOG][,][FAIL|STOP|NEXT|IGNORE]
```

The question mark handles errors generated by the function. The keywords, that control how an exception is handled are described in Exception handling keywords.

Table 5-4. Exception handling keywords

Keyword	Description
LOG	The exception is reported to the English log file and the non-English log file, if specified. When a remote exception chain is logged by ASL, the locale defined in the SM_LOCALE environment variable will be used. The VMware Smart Assurance System Administration Guide provides further information concerning the log files.
FAIL	The current rule exits with a failure status. If the failed expression is associated with the START rule, the DEFAULT rule is executed.
STOP	This is equivalent to calling the stop() function.
NEXT	The processing of actions in the current expression block stops. The rule, however, is not treated as failed rule.
IGNORE	The current action is ignored and processing continues as if nothing happened.

You can specify the LOG keyword with any of the other keywords.

To define the severity of the exception, arguments can be passed to LOG. Logging levels are set by using the syntax:

```
LOG("<logging_level>")
```

Valid logging keywords are (in increasing level of severity):

- Debug
- Notice or Informational
- Warning (Default log value)
- Error
- Fatal or Critical

When the exception is not specified, the default behavior for exceptions is LOG, NEXT. In other words, when an exception occurs, the error is logged and no other actions are performed in the current action block.

Note Do not confuse the question mark (?) used for exception handling with the question mark (?) used with patterns. Maybe operator provides additional information.

The following script demonstrates two different keywords to use for exception handling. The first exception occurs for:

```
print(x) ? IGNORE;
```

There is no value assigned to the variable x, so the function causes an exception. Since the keyword is IGNORE, the current line is skipped and the next line is executed.

The second exception occurs for:

```
print(y) ? FAIL;
```

There is no value assigned to the variable y, so the function causes an exception. Since the keyword is FAIL, the current action block is not completed (and the following print statement is not executed) and the START rule fails. Whenever the START rule fails, the DEFAULT rule is executed.

```
ASL Script (exception_do.asl)
START
do {
 print("Hello");
 print(x) ? IGNORE;
 print("OK");
 print(y) ? FAIL;
 print(“Here I am”);
}
DEFAULT
do {
 print("Default rule");
 stop();
}
Input:
none
Output:
$ sm_adapter exception_do.asl
Hello
OK
Default rule
$
```

Interfacing with a Domain Manager

6

This chapter includes the following topics:

- ASL and the MODEL language
- Objects and instances
- Tables, structures and enumerations
- Type conversions between ASL and MODEL
- Domain Manager control
- Events
- Transactions, commit, and abort
- Error handling
- Repositories
- Naming and identity
- Tracing ASL scripts

ASL and the MODEL language

In order to create adapters that interact with a Domain Manager, it is necessary to understand how the Domain Manager is configured. The adapter creates, deletes, and interacts with instances of objects that are defined by using the MODEL language.

The MODEL language is an object-oriented language used to construct a correlation model to describe a managed domain. The language is used to define a set of classes and the attributes, relationships, and events that are associated with the classes.

Classes describe the objects that are modeled for use in a Domain Manager. Instances are specific occurrences of a class. For example, a class might describe a human and an instance of the class is someone named Bill.

Attributes describe a class and, for an instance of the class, include information about its present state. Examples of attributes include an element's name and a counter that counts the number of packets traversing an interface.

Relationships define how instances are related to other instances. Relationships can be one-to-one, one-to-many, many-to-one, or many-to-many. When only a single instance can be related to another instance (or instances), it is a relationship. When multiple instances can be related to another instance (or instances), it is a relationshipset.

Events describe the failures that can occur for a class, the symptoms that these failures can cause, and the effect of failures.

Symptoms can be described a follows:

- Local, meaning they are observed in the instance of the class.
- Propagated, meaning they are observed in instances related to the failing instances.

Once classes are specified, the model is loaded and run on a Domain Manager. Instances are created for each entity that is modeled. Each instance is associated with a class and has values for its attributes, relationships, and events.

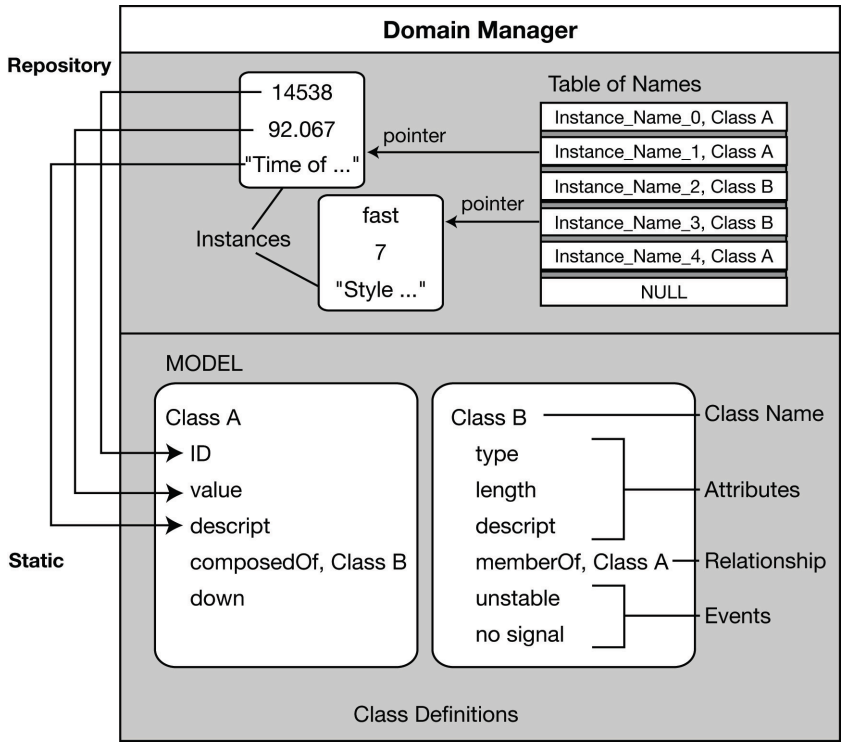
The models stored in a Domain Manager are static. Instances are dynamic and are stored in the repository, as shown in Domain Manager with a model and repository.

Instances in a Domain Manager consist of a table entry and data. The table entry includes the name of each instance and its class. Each instance name must be unique. The table of names always contains an entry for the NULL object.

The data associated with each instance includes properties such as attributes, operations, and the relationships between instances. The data also includes event information such as problems, symptoms, and events.

VMware Smart Assurance MODEL Reference Guide provides additional information about MODEL.

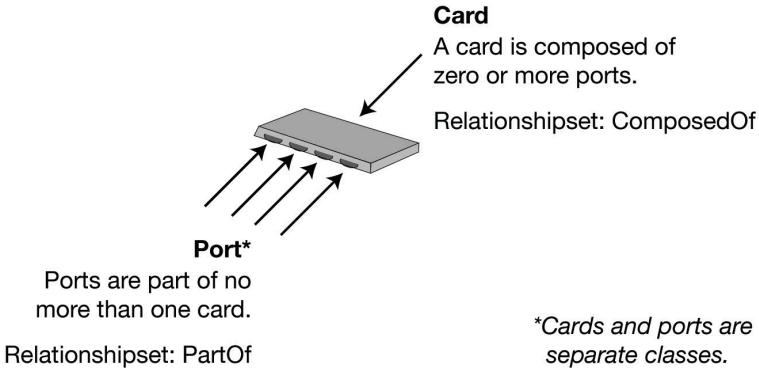
Figure 6-1. Domain Manager with a model and repository



Correlation model used for example scripts

All of the ASL script examples used in this chapter interact with a small correlation model. The model defines two types of objects: cards and ports, as shown in Relationship between cards and ports.

Figure 6-2. Relationship between cards and ports



Cards and ports are separate classes. Each class has its own set of attributes and events. An instance of one class can be related to an instance of another class.

Cards have the following:

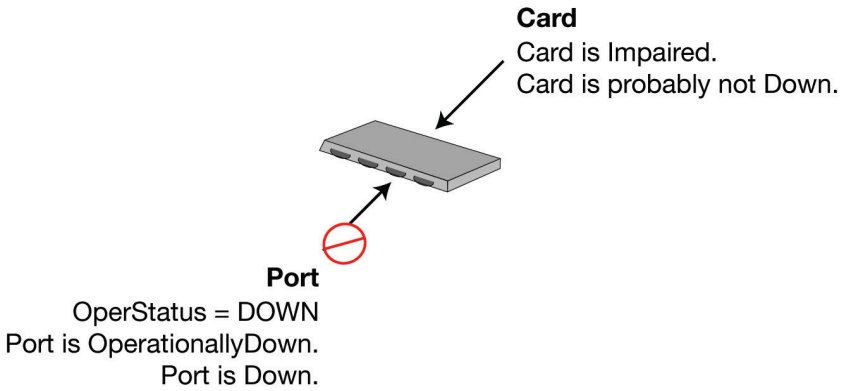
- A single attribute, CardDesc. This attribute is a string and has no default value.

- Two types of events.
  - The first type is a problem called Down. When all of a card's ports are OperationallyDown, the card is Down.
  - The second type is an aggregate called Impaired. If any of the ports associated with a card is Down or if the card is Down, the card is Impaired.

    Ports have the following:
- A single attribute, OperStatus. This attribute has a special data type that limits the values of the attribute to TESTING, UP, and DOWN. DOWN is the default value.
- Two types of events.
  - The first type is a problem called Down. A port is Down when it is OperationallyDown.
  - A second type is a symptomatic event called OperationallyDown. When the port's OperStatus attribute is set to DOWN, it causes the OperationallyDown event.

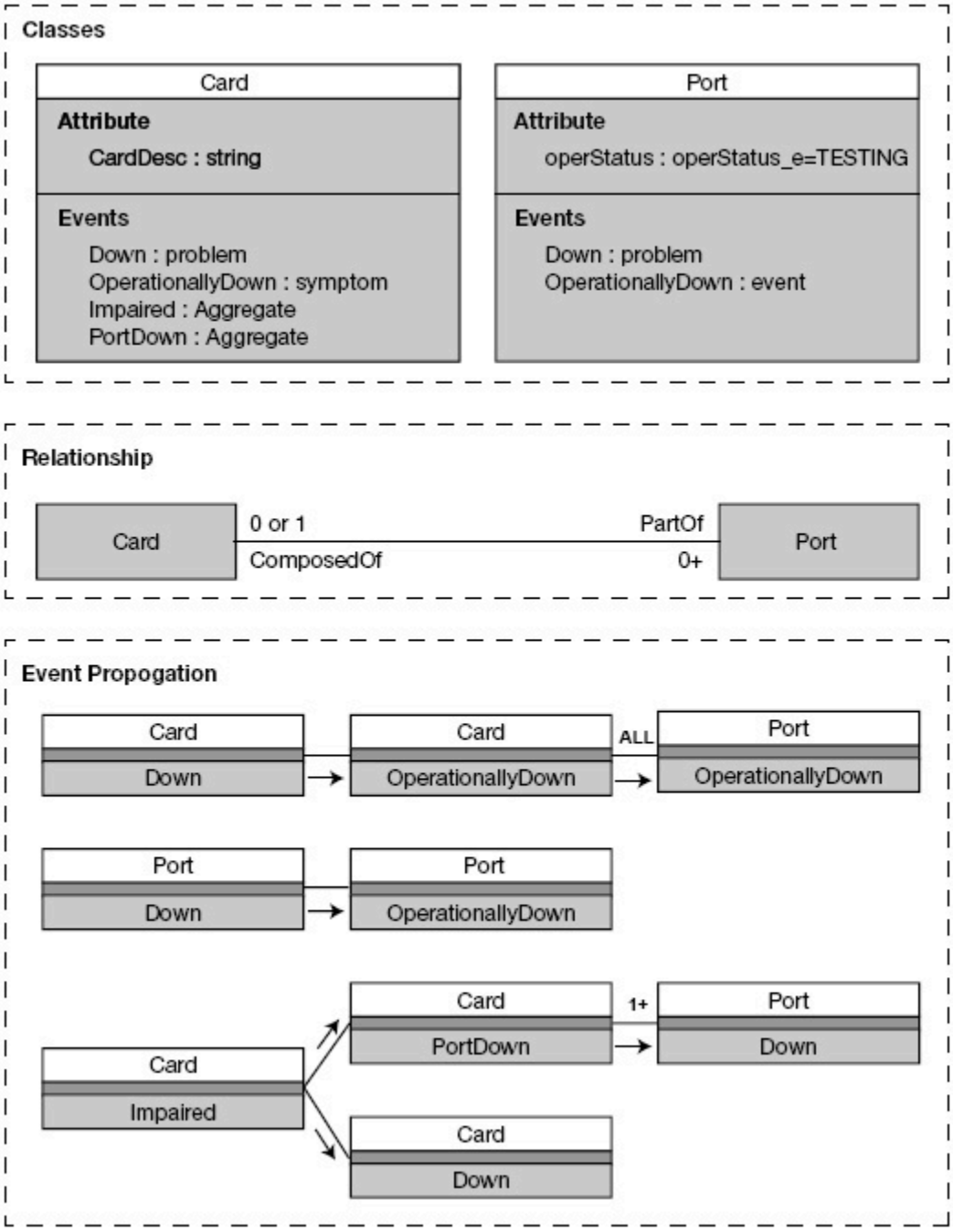
    In Events that affect ports and their related card, the port's attribute OperStatus is equal to DOWN. The attribute's value causes the event OperationallyDown, which, in turn, causes the problem Down. Since the port participates in a PartOf relationship with the card, the card is also affected. The port has the problem Down and, as a result, the card has the compound event (aggregate), Impaired. The card is probably not down, because only one port (not all ports) is experiencing a problem.

    Figure 6-3. Events that affect ports and their related card



The section *Appendix C, "Card-Port MODEL Code,"* describes the MODEL language code for this example. Diagram of the card and port model is a diagram of this card and port model.

Figure 6-4. Diagram of the card and port model



Objects and instances

This section discusses how to create, delete and manipulate objects and instances.

Creating objects

The create() function creates an instance of a class or an instance with an object handle. The syntax is one of the following:

- create(<*classname*>, <*objectname*>);
- create(<*objhandle*>);

  If the object being created already exists, the create() function returns a reference to that object or an error if the object exists and belongs to different class.

  If you specify an object handle (objhandle) and the object does not exist, the create() function creates the object and the object handle.

  You can assign the result of the function to a variable (for example, objRef). This defines an object handle for the object.

  ```
  objRef = create(<classname>, <objectname>);
  ```

  The following script example loads instances of ports and cards. The name of the Domain Manager that contains the model is JS1.

  ```
  ASL Script (create_obj.asl):
  START {
   CARD rep(PORT)
  }
  CARD {
   "CARD:" cardname:word eol
  }
  do {
   create("Card",cardname);
  }
  PORT {
   portname:word eol
  }
  do {
   create("Port",portname);
  }
  DEFAULT {
   err:{..eol}
  }
  do {
   print(err." Failed");
  }
  EOF
  do {print("Complete");
  }
  Input (create_obj.txt):
  CARD: CARD0
   PORT00
   PORT01
   PORT02
  ```

```
 PORT03
CARD: CARD1
 PORT10
 PORT11
 PORT12
CARD: CARD2
 PORT20
 PORT21
 PORT22
 PORT23
 PORT24
 PORT25
Output:
$ sm_adapter --server=JS1 --file=create_obj.txt create_obj.asl
Complete
$
```

Listing instances

The getInstances() function returns a list of object names for the given class, but does not return object handles. The syntax is as follows:

```
        <listname> = getInstances(<classname>);
```

The following script example retrieves the names of all instances of the class “Card” and stores them in a list. The names stored in the list cannot be used by ASL to reference objects.

```
ASL Script (get_obj.asl):
START {
 ..eol
}
do {
 x = getInstances("Card");
 print(x);
}
Input:
none
Output:
$ sm_adapter --server=JS1 get_obj.asl
{ CARD0, CARD1, CARD2 }
$
```

Creating handles for existing objects

The object() function returns an object handle for existing objects. An object handle is a distinguished data type in ASL that represents a model class name and an object name. The syntax is one of the following:

- <*objectRef*> = object([<*classname*>,] <*objectname*>);

  The name of the instance must be unique, therefore, the class name is optional.

- *<objectRef>* = object(*<objhandle>*);

  There are three ways to obtain a handle for an existing object:

- If you specify a class name and an object name, then the object() function returns the object handle for the specified object. ASL stores that information as *<class>*::*<name>*.
- If you specify an object handle (objhandle), then the object() function returns the object handle unchanged.
- If you specify an object name only, then the object() function returns the handle object and assumes the object name and the class name, MR_Object.

  You can also create an object handle for future use for an object that does not exist.

  For example:

```
objref = object();
```

  The following script retrieves the names of all instances of the class “Card” and stores them in a list. A foreach loop cycles through the list and adds each object, including its class, to a different list. The script prints both lists.

  The lists demonstrate the difference between the getInstances() function and the object() function. The getInstances() function returns only the name of the object. The object() function returns the object handle for the object and ASL stores that information as *<class>*::*<name>*. The section Listing instances describes the getInstances() function.

```
ASL Script (handle_obj.asl):
y = list();
START {
 ..eol
}
do {
 x = getInstances("Card");
 print(x);

 foreach mem (x)
  {y += object("Card",mem);
  }

 print(y);
}
Input:
none
Output:
$ sm_adapter --server=JS1 handle_obj.asl
{ CARD0, CARD1, CARD2 }
{ Card::CARD0, Card::CARD1, Card::CARD2 }
```

Attributes, relationships, and operations of objects

The attributes, relationships, and operations of an object are its properties. ASL assigns values to properties by using the following syntax:

```
<objectRef>-><property> = <value>;
```

The *<objectRef>* is the object handle for the object. You assign the object handle by using the create() or object() functions or through a relationship.

The following script loads the names of all instances of the classes “Card” and “Port” into a Domain Manager. The datafile contains a card name followed by the ports associated with the card. As this script loads cards and ports, it creates a relationship between a card and its ports.

Relationships between objects in MODEL are paired. It is not necessary to define both relationships. Define one relationship. Its converse is defined by default.

```
ASL Script (properties_obj.asl):
START {
 CARD rep(PORT)
}
CARD {
 "CARD:" cardname:word eol
}
do {
 cardObj = create("Card",cardname);
}
PORT {
 portname:word eol
 }
do {
 portObj = create("Port",portname);
 cardObj->ComposedOf += portObj;
 print(portObj." ".cardObj);
}
Input (create_obj.asl):
CARD: CARD0
 PORT00
 PORT01
 PORT02
 PORT03
CARD: CARD1
 PORT10
 PORT11
 PORT12
CARD: CARD2
 PORT20
 PORT21
 PORT22
 PORT23
 PORT24
 PORT25
```

```
Output:
$ sm_adapter --server=JS1 --file=create_obj.txt properties_obj.asl
Port::PORT00 Card::CARD0
Port::PORT01 Card::CARD0
Port::PORT02 Card::CARD0
Port::PORT03 Card::CARD0
Port::PORT10 Card::CARD1
Port::PORT11 Card::CARD1
Port::PORT12 Card::CARD1
Port::PORT20 Card::CARD2
Port::PORT21 Card::CARD2
Port::PORT22 Card::CARD2
Port::PORT23 Card::CARD2
Port::PORT24 Card::CARD2
Port::PORT25 Card::CARD2
$
```

The value stored in *<objectRef>-><property* >can also be assigned to an ASL variable.

```
        <variable> = <objectRef>-><property>;
```

You can refer to properties indirectly. This is a two-step process. First, assign the property to a variable. Second, the variable can be used in place of the property in conjunction with an indirection operator (->*).

```
propertyname = “property”;
objRef->*propertyname = "UP";
```

The syntax objRef->*variable allows you to access an attribute by variable. The following code sample assigns the attribute name to a variable, then makes a reference to that attribute through the indirection operator. The output is the creation class name of the given object.

```
variable = “CreationClassName”;
c = objRef->*variable;
print(c);
```

When you run this ASL script on a router object, the output is “Router”.

Deleting objects

The delete() method deletes an instance of a class. This function does not delete objects that are related to the deleted object. ASL creates an error if the object handle used with the delete() method points to an object that does not exist. The syntax of the delete() method is:

```
        <objectRef>->delete();
```

You can use += or -= operators to add or delete an object from a list of object handles in ASL or from a relationshipset in MODEL. Modifying relationshipsets provides additional information.

The following script deletes a card and its related ports. The script contains a default variable that specifies the card to delete. When the ComposedOf relationship is used, the ASL script creates a list of port objects to delete. The card is deleted first, followed by its ports. Exception handling causes the script to stop if the card object does not exist.

```
ASL Script (delete_obj.asl):
default delthis = "CARDX";
START {
 ..eol
}
do {
 delthisObj = object(delthis);
 relObj = delthisObj->ComposedOf?LOG,STOP;
 x = delthisObj->delete();
 foreach mem (relObj)
  {
  mem->delete();
  }
 print("Deleted ".delthis." and related ports");
}
Input:
none
Output:
$ sm_adapter --server=JS1 -Ddelthis="CARD2" delete_obj.asl
Deleted CARD2 and related ports
$
```

Testing for null objects

The isNull() method tests whether an object handle points to a valid object. If this function returns TRUE, the object does not exist. The syntax is as follows:

```
          <objectRef>->isNull();
```

For example, this script deletes the object PORT25 from the Domain Manager if it exists. The if statement uses the isNull() function to test if the object exists. An exclamation point is a logical NOT, so that if the object does exist, the condition is true for the if statement. The output demonstrates the case where the object exists and is deleted.

```
ASL Script (isnull_obj.asl):
START {
 ..eol
}
do {
 delthisObj = object("Port","PORT25");
 if (!delthisObj->isNull())
  {
   delthisObj->delete();
     print("Deleted   ".delthisObj);
  }
 else { print(delthisObj." does not exist");}
```

```
}
Input:
none
Output:
$ sm_adapter --server=JS1 isnull_obj.asl
Deleted   Port::PORT25
$
```

Testing relationships

The is() function tests whether an object is a member of a relationship. The syntax is as follows:

```
is(<objectRef>->Relationship>,<objectRef2>)
```

The *<relationship>* used in the is() function must be a valid relationship for the object or an error occurs. If *<objectRef2>* is related to *<objectRef>* by *<Relationship>*, the function returns TRUE. Otherwise, the function returns FALSE.

This script tests objects of the class Port to see if they are related to CARD2. The script gets the list of ports by using the class Port. When the ports are printed, the class that is printed is MR_Object, which is the parent class of the Port and Card classes. The MR_Object class appears because a class is not specified for the object() function.

```
ASL Script (relation_obj.asl):
START {
 ..eol
}
do {
 cardObj = object("CARD2");
 x = getInstances("Port");
 foreach mem (x)
  {
  portObj = object(mem);
  if (is(cardObj->ComposedOf,portObj))
   {print(portObj." is related to ".cardObj);
   }
  }
}
Input:
none
Output:
$ sm_adapter --server=JS1 relation_obj.asl
MR_Object::PORT20 is related to MR_Object::CARD2
MR_Object::PORT21 is related to MR_Object::CARD2
MR_Object::PORT22 is related to MR_Object::CARD2
MR_Object::PORT23 is related to MR_Object::CARD2
MR_Object::PORT24 is related to MR_Object::CARD2
MR_Object::PORT25 is related to MR_Object::CARD2
$
```

Modifying relationshipsets

When a model relates many objects to one or more objects, it is a relationshipset. A relationshipset in MODEL is a list of object handles in ASL. You can manipulate this list by using the list operators described in List operators

Table 6-1. List operators

Syntax	Description
objRef->rela_prop += value;	Adds a value to a relationshipset.
objRef->rela_prop -= value;	Removes a value from a relationshipset.
x->rela_prop = object("");	Clears a single relationship.
x->rela_prop = list(y,z);	Adds a list of object handles to a relationshipset.
x->rela_prop=list();	Clears a relationshipset.

Accessing a particular relationship in a relationshipset is a two-step process. First, load relationshipset into a list in ASL. Then, access a specific element in the list. For example:

```
x = objRef->rela_prop;
print(x[0]);
```

Retrieving description texts

There are severaloperations of the class SM_System that can be used to retrieve description texts or objects. The texts that are retrieved will be in the locale that was set either by the initial connection, or by the locale set by the most recent call to the setLocale().

If these operations are run outside of the context of a remote session, then the effective locale is CI_Locale::DEFAULT.

The getClassDescription() operation retrieves the descriptive text associated with the MODEL class name.

```
/// get model class description
```

string getClassDescription(in string className);

The getOpDescription() operation retrieves the descriptive text associated with a MODEL operation.

```
/// get model operation description
```

string getOpDescription(in string className, in string opName);

The getPropDescription() operation retrieves the descriptive text associated with a class property, such as an attribute.

```
/// get model property description
string getPropDescription(in string className, in string propName);
```

The getEventDescription operation retrieves the descriptive text associated with a MODEL event.

```
/// get model event description
string getEventDescription(in string className, in string eventName);
```

In this example, the description of the Attr1 attribute in the MyClass class is printed, in Japanese.

```
sysObj = object("SM_System","SM-System");
setLocale("ja_JP");
foo = sysObj->getPropDescription("MyClass","Attr1");
print(foo);
```

Tables, structures and enumerations

This section describes how to access, and how to clear the members of a MODEL table. It also describes how to access and update structures and enumerations.

Accessing tables in MODEL from ASL

Tables in MODEL are represented as lists in ASL. In MODEL, a table of structures is a list of lists in ASL.

To return the structure in the MODEL table given the key (“F”), use the following syntax:

```
print(x->f[“F”]);
```

Clearing the members of a table

To clear the members of a MODEL table, use the following syntax:

```
x->rela_prop=list();
```

Updating and accessing structure attributes in MODEL

Structures in MODEL are represented as lists in ASL. The number of items in the list in the ASL is equal to the number of fields in the corresponding structure of the MODEL. The first item in the list will correspond to the first field defined within the structure, and so on for each subsequent item.

For example, if the following structure is defined in Example.mdl, the following ASL code, Example.asl, shows how to reference the structure.

```
Example.mdl
interface Example_Struct:MR_ManagedObject
{
 struct model_struct
 {
  int  valueType;
  string  Value;
 };
 attribute model_struct asl_list;
};
Example.asl
START
{
 do
 {
  // create an instance, ‘example’, of Example_struct
  example = create (“Example_Struct”, “Example-Struct”);
  // put values into a ‘model_struct’ structure:
  // structure element ‘valueType’ is list element [0]
  // structure element ‘value’ is list element [1]
  struct_value = list();
  struct_value[0] = 1;
  struct_value[1] = “Structure Example”;
  // set the value in the Example_struct instance
  example->asl_list = struct_value;
  // get values from a ‘model_struct’ structure
  accessed_struct = list();
  accessed_struct = example->asl_list;
  // display the retrieved structure elements
  print (accessed_struct);
  stop();
 }
}
```

The output of the example is the following:

```
$sm_adapter -M Example Example.asl
{1, Structure Example }
```

Accessing enumerated data types in MODEL

Enumerated data types are represented as strings in ASL.

The following example shows the syntax to update an enumerated attribute with an enum value of TESTING:

```
x->enum_variable = “TESTING”;
```

Type conversions between ASL and MODEL

MODEL cannot convert data from one type to another. The types of data available in MODEL are not the same as the types of data available in ASL. Correspondence between types of data in MODEL and ASL shows how types of data in MODEL and ASL correspond.

Table 6-2. Correspondence between types of data in MODEL and ASL

MODEL	ASL
Numeric (any type)	Numeric (double)
binary	binary
Table	List
Relationship	Object handle
Relationshipset	List of object handles
boolean	boolean
string	string
External (User-defined) Type	string
struct (a structure)	List
enum	string

Note There is no MODEL type that corresponds to an ASL table.

Domain Manager control

The following functions allow you to trigger Domain Manager actions from ASL. These actions either rebuild the correlation model or cause the Domain Manager to correlate events.

consistencyUpdate() function

The consistencyUpdate() function causes the Domain Manager to recompute the correlation rules. This function always returns, as soon as the request has been registered, with a TRUE value. The actual recomputation can take some time, and continues on the Domain Manager independently of the adapter. The syntax is:

```
consistencyUpdate()
```

correlate() function

The correlate() function causes the Domain Manager to correlate events. This function always returns, as soon as the request has been registered, with a TRUE value. The actual correlation computation continues on the Domain Manager independently of the adapter. The syntax is:

```
correlate()
```

Events

This section describes how to retrieve event information.

getCauses() function

The getCauses() function returns a list of problems that cause an event. The function receives three arguments: class, instance, and event. The function returns the problems that cause the event based on the relationships among instances defined in the Domain Manager. The syntax is as follows:

```
getCauses(<classname>,<instancename>,<eventname>,[<oneHop>])
```

The *<oneHop>* parameter is optional. If it is omitted or passed as FALSE, the full list of problems that explain *<eventname>*, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list *<eventname>* among the events they explain are returned.

The function returns a list of lists with the format:

```
{ <classname>,<instancename>,<problemname> },
{ <classname>,<instancename>,<problemname> },
...
```

getClosure() function

The getClosure() function returns a list of symptoms associated with a problem or aggregation. The function receives three arguments: class, instance, and event. The function returns the symptoms associated with the problem or aggregate based on the relationships among instances defined in the Domain Manager. The syntax is as follows:

```
getClosure(<classname>,<instancename>,<eventname>,[<oneHop>])
```

The *<oneHop>* parameter is optional. If it is omitted or passed as FALSE, the full list of problems that explain *<eventname>*, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list *<eventname>* among the events they explain are returned.

The function returns a list of lists with the format:

```
{ <classname>,<instancename>,<symptomname> },
{ <classname>,<instancename>,<symptomname> }
```

getEventClassName() function

The getEventClassName() function returns a string with the name of the ancestor class associated with a class and an event. The ancestor class is where the event was originally defined. That is, the class in which the event definition statement, not any refinement, appeared. The syntax is as follows:

```
getEventClassName(<classname>,<eventname>)
```

getEventDescription() function

The getEventDescription() function returns a string, defined in MODEL, that describes an event. The syntax is as follows:

```
getEventDescription(<classname>,<eventname>);
```

getEventType() function

The getEventType() function returns a string that classifies an event as a PROBLEM, EVENT, AGGREGATION, or SYMPTOM. The syntax is as follows:

```
getEventType(<classname>,<eventname>);
```

getExplainedBy() function

The getExplainedBy() function is the inverse of the getExplains() function. It returns those problems that the MODEL developer has listed as explaining this problem. The syntax is as follows:

```
getExplainedBy(<classname>,<instancename>,<eventname>[,<oneHop>])
```

The *<oneHop>* parameter is optional. If it is omitted or passed as FALSE, the full list of problems that explains *<eventname>*, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list *<eventname>* among the events that they explain are returned.

The function returns a list of lists with the format:

```
{ <classname>,<instancename>,<symptomname> },
{ <classname>,<instancename>,<symptomname> }
```

getExplains() function

MODEL developers can add information to a problem in order to emphasize events that occur because of a problem. The getExplains() function returns a list of these events. The syntax is as follows:

```
getExplains(<classname>,<instancename>,<eventname>,[<oneHop>])
```

The *<oneHop>* parameter is optional. If it is omitted or passed as FALSE, the full list of problems that explains *<eventname>*, whether directly or indirectly, is returned. If it is passed as TRUE, only those problems that directly list *<eventname>* among the events that they explain are returned.

getChildren() function

The getChildren() function provides the list of classes derived from a particular class.

```
getChildren(<classname>[,recursive])
```

The recursive parameter is optional. If omitted or false, only the immediate child classes are retrieved. If true, all children, including those of derived classes are retrieved.

Transactions, commit, and abort

When you modify objects in ASL scripts, the objects change as each modification occurs. Using transactions, you can commit many changes to the objects in a Domain Manager as a single change or choose to abort all of them. Use the following syntax to create a transaction:

```
        <variable> = transaction();
```

You cannot assign a transaction to a global variable or to a statically scoped variable. After you assign the transaction to a variable, every change made to an object does not affect the object until you commit the transaction. If the you abort the transaction, any changes made will not affect the object. Use the following syntax to commit a transaction:

```
        <variable>->commit()
```

Use the following syntax to abort a transaction:

```
        <variable>->abort()
```

The changes made with a transaction are not visible outside of the ASL script until you commit the changes. Within a transaction, the same ASL script can see the proposed changes. Transactions also can control how other applications see objects before changes are committed or aborted by adding a single keyword. The syntax of a transaction with a keyword is:

```
        <variable> = transaction([WRITE_LOCK|READ_LOCK|NO_LOCK]);
```

A keyword can be any one of those listed in Transaction keywords.

Table 6-3. Transaction keywords

Keyword	Description
WRITE_LOCK	While the transaction is open, no other process can modify or access information in the repository.
READ_LOCK	Currently behaves as WRITE_LOCK.
NO_LOCK	This is the default behavior. No locks exist until the ASL commits the transaction.

You can nest transactions. When you nest a transaction, you must commit or abort the nested transaction before you commit or abort the previous transaction.

ASL aborts any open transactions when the START rule completes.

Note A maximum of 16 transactions may be open concurrently.

The following script deletes a card and its related ports. The script contains a default variable that specifies the card to delete. Using the ComposedOf relationship, the ASL script creates a list of port objects to delete. The script deletes the card and its related ports at the same time through a transaction.

```
ASL Script (deltrans_obj.asl):
default delthis = "CARDX";
START
do {
 delthisObj = object(delthis);
 relObj = delthisObj->ComposedOf?LOG,STOP;
 deltrans=transaction();
  x = delthisObj->delete();
  foreach mem (relObj)
   {
   mem->delete();
   }
 deltrans->commit();
 print("Deleted ".delthis." and related ports");
 stop();
}
Input:
none
Output:
$ sm_adapter --server=JS1 -Ddelthis="CARD2" deltrans_obj.asl
Deleted CARD2 and related ports
$
```

Error handling

This section describes functions that allow you to determine whether an error occurred, to get a description of the error and to reset the error state.

feError() function

Invoke the feError() function to determine if the front end has reported a failure to read data.

The function returns a Boolean value, where TRUE indicates an error occurred and FALSE indicates no error occurred. If an error occurred, invoke feErrorMsg() for a description of the error.

feErrorMsg() function

Invoke the feErrorMsg() function to get a description of the error indicated by the feError() function. feError() returns TRUE.

This function returns a string that describes the error. If no error condition exists, an empty string is returned.

feErrorReset() function

The feErrorReset() function is used with the feError() and feErrorMsg() functions to reset the error state. There is no return value for this function.

Repositories

self->

Directs functions that interact with the repository to use the adapter's repository and not the Domain Manager's. Used in conjunction with adding objects to the adapter's repository.

Naming and identity

This section describes functions used to retrieve names and identity of the rule which is running, what Domain Manager the script is running o,n and the thread ID of the adapter.

getRuleFileName() function

The getRuleFileName() function returns the filename of the currently executing rule file as an argument.

Usage:

```
getRuleFileName ([<fullname>]);
```

where *<fullname>* is a string containing the complete path-qualified filename.

getServerName() function

The getServerName() function returns a string, which is the name of the Domain Manager associated with adapter that is invoking the ASL script.

thread() function

The thread() function returns the operating system thread ID that is associated with the adapter that is invoking the ASL script.

Tracing ASL scripts

The ASL_Trace Model object is used to start and stop a trace from dmctl, the Java API, bootstrap.conf, and ASL. The ASL_Trace model object contains three methods.

- start() method starts the tracing
- stop() method stops the tracing
- isActive() method checks to see if a trace is in progress

```
interface ASL_Trace
start()
stop()
isActive()
```

The trace contains timestamps for:

- Entry into ASL scripts
- Exit to ASL scripts
- Exit from ASL scripts

For example, to use the tracing capability from dmctl:

a Create an instance of the Model object. An instance named ASL-Trace is customary.

```
dmctl -s serverName create ASL_Trace::ASL-Trace
```

b Invoke start/stop/isActive as needed.

```
dmctl -s serverName invoke ASL_Trace::ASL-Trace start
dmctl -s serverName invoke ASL_Trace::ASL-Trace stop
```

The output will appear similar to the following:

```
1 2009/02/26 13:26:15.175714600 1 000:00:00.000000000 NoteASLThread ThreadName=adapter
1 2009/02/26 13:26:15.175714600 2 000:00:00.000000000 NoteLayout Record prefix fields:
1 2009/02/26 13:26:15.175714600 3 000:00:00.000000000 NoteLayout Thread id, Date, Time, Seq,
1 2009/02/26 13:26:15.175714600 4 000:00:00.000000000 NoteLayout Delta since prior record in
thread,
1 2009/02/26 13:26:15.175714600 5 000:00:00.000000000 NoteLayout RecordType.
1 2009/02/26 13:26:15.175714600 6 000:00:00.000000000 NoteLayout Trace records continue with:
1 2009/02/26 13:26:15.175714600 7 000:00:00.000000000 NoteLayout Entry|Exit|StopExit|ThrowExit|
ThrowExit1|ThrowExit2,
1 2009/02/26 13:26:15.175714600 8 000:00:00.000000000 NoteLayout Constructor|Destructor|
```

```
Initialize|Multiplex,
1 2009/02/26 13:26:15.175714600 9 000:00:00.000000000 NoteLayout this for ASL_Parser, Nest
depth, Rule file name.
1 2009/02/26 13:26:15.175714600 10 000:00:00.000000000 StartTrace
1 2009/02/26 13:26:15.186840900 11 000:00:00.011126300 NoteDriver CreationClassName=GA_Driver
Name=ASL-Trace1-Driver
1 2009/02/26 13:26:15.187012400 12 000:00:00.000171500 Trace Entry Constructor
00000001002CA120 0 tasl_trace1.asl
1 2009/02/26 13:26:15.187184600 13 000:00:00.000172200 Trace Exit Constructor 00000001002CA120
0 tasl_trace1.asl
1 2009/02/26 13:26:15.187535500 14 000:00:00.000350900 Trace Entry Initialize 00000001002CA120
0 tasl_trace1.asl
1 2009/02/26 13:26:15.187703000 15 000:00:00.000167500 Trace Exit Initialize 00000001002CA120
0 tasl_trace1.asl
1 2009/02/26 13:26:15.188291200 16 000:00:00.000588200 Trace Entry Destructor 00000001002CA120
0 tasl_trace1.asl
1 2009/02/26 13:26:15.188457500 17 000:00:00.000166300 Trace Exit Destructor 00000001002CA120
0 tasl_trace1.asl
1 2009/02/26 13:26:45.193438300 18 000:00:30.004980800 StopTrace
```

The log file is written to BASEDIR/smarts/local/logs as <*serverName*>.ASL_Trace.log.

- If the BASEDIR/smarts/local/logs directory cannot be found, the log is written to the current working directory.
- If the <*serverName*> is not available, the file name becomes ASL_Trace.log_<*pid*>.

  The trace is written chronologically. Thus, traffic for several threads may be intermingled. A sort of the first 4 fields will produce traces for individual threads.

  The record layout is noted when the trace is started. The prolog of each record is fixed. After that, each record contains a record type to provide summary program processing.

  The ‘this’ pointer is logged, and can be used when tying the ASL trace to pstack samples.

  Startup is automatic if the SM_ASL_TRACE environment variable is set to anything that does not lower case to 0, false, no, or off.

Running Adapters

7

This chapter includes the following topics:

- The sm_adapter command
- Front end
- Rule set
- Back end
- Tracing operations
- Stopping adapters

The sm_adapter command

The sm_adapter command starts ASL scripts. To run the command, you must specify the ASL rules file to run as well as choosing how the front end and back end of the adapter operate. The command syntax is:

```
sm_adapter [options...] [<rule set>]
```

The *<rule set>* is the ASL script that the adapter follows when it receives information.

The options control which Domain Manager and Broker the adapter connects to, and it also configures how the adapter runs. An adapter can run by using a variety of front end and back end components. The options are described in The sm_adapter command.

The file, sm_adapter is found in BASEDIR/smarts/bin.

Basic options

sm_adapter basic options shows the options which enable you to name the adapter and control which Broker, MODEL library, and port the adapter uses.

Table 7-1. sm_adapter basic options

Option	Description
--broker=*<location>*	Alternate Broker location as host:port. Also -b *<location>*.
--model=*<model>*	Name of MODEL library to load. Also -M *<model>*.

Table 7-1. sm_adapter basic options (Continued)

Option	Description
--name=<*name*>	Start a server registered under <name>. Also -n <*name*>.
--port=<*xxxx*>	Alternate registration port. Use with --name.
--timeout=<*secs*>	Set the timeout for server interaction. The timeout applies to the back end connection except when using the subscriber front end, in which case it applies to the front end. The argument is in seconds, and can be a decimal value. If the --timeout option appears with no value, 600 seconds is used. By default, there is no timeout.
--wait	Wait for initial driver to complete.

Other sm_adapter options describes additional sm_adapter options.

Table 7-2. Other sm_adapter options

Option	Description
--accept=<*host-list*>	Accept connections only from hosts on host-list, a comma-separated list of hostnames and IP addresses. You can also specify any instead of host-list to allow any host to connect. Default: --accept=any.
--daemon	Run process as a daemon.
--errlevel=<*level*>	Specify the minimum level at which error events are written to the standard error output. Possible values include: none, emergency, alert, fatal, critical, error, warning, notice, informational, debug. The default is warning. (Note that fatal is equivalent to emergency, alert, or critical.)
--help	Print help and exit.
--loglevel=<*level*>	Specify the minimum level at which event messages are written to the system logging facility. Possible values include: none, emergency, alert, fatal, critical, error, warning, notice, informational, debug. The default is error. (Note that fatal is equivalent to emergency, alert, or critical.)
--logname=<*name*>	Specify the name used to identify the SDK program in the system log. On UNIX systems, the default is the program name.
--output[=<*file*>]	Messages generated from internal logging calls are directed to a log file. Log files are written to the BASEDIR/smarts/logs directory. If <*file*> is not specified, the value of --logname is used. If a log file with specified name already exists, it is moved to a backup file, BASEDIR/smarts/local/logs/<*file*>.bak. If the backup file already exists, it is deleted.
--tracelevel=<*level*>	Specify the minimum level at which a process stack trace is written to the standard error output. Possible values include: none, emergency, alert, fatal, critical, error, warning, notice, informational, debug. The default is fatal. (Note that fatal is equivalent to emergency, alert, or critical.)
--version	Print program version and exit.

Front end

The front end of the adapter is responsible for reading data from an input source and formatting it for processing by an ASL rule set. The adapter can run with one of the following front ends:

- File—Reads data from an ASCII file.
- File tail—Is designed to read the log files of long-running Domain Managers.Like the file front end, the file tail front end reads data from an ASCII file. But instead of opening the file, reading the contents and closing the file, this front end opens the file, skips to its end and waits for additional lines to be written to the file.
- Program—Reads data from the output of a command pipeline.
- Subscribe—Is used in outflow adapters. This front end connects to a Domain Manager and issues a set of subscription requests. Subsequent notifications received from the Domain Manager are formatted and delivered to the ASL rule set. There are many configuration options available with this front end. The options include selecting the subscription set to performing sophisticated smoothing and filtering of events.

  Each front end is described in a subsequent subsection. A list of options is provided in sm_adapter front end options. Examples of the sm_adapter command with front ends are provided in Sample front end invocations.

File front end

The file front end reads data from an ASCII file. The front end opens and reads the file, feeds its contents to the rule set, then closes the file and terminates. This front end performs two special formatting translations:

- End-of-line translation
- Field-separator translation

  The **--**file option of the sm_adapter command invokes this front end.

End-of-line translation

One of the more troublesome aspects of using a cross-platform file adapter is dealing with the end-of-line problem. On UNIX, a single \n represents an end-of-line. The file front end translates either a single \n or the \r\n sequence to an ASL end-of-line marker (eol), which eliminates this problem.

Consider a file with one line, “hello world":

```
h  e  l  l  o   w  o  r  l  d  \r  \n
```

Whereas, on a UNIX system, this file contains the characters:

```
h  e  l  l  o   w  o  r  l  d  \n
```

Using the file front end on either system feeds the following input stream into the rule set:

```
h  e  l  l  o   w  o  r  l  d  eol
```

Field-separator translation

Many input files are structured as a series of records, where each record is represented as one line in the file. Each record (line) may be further separated into fields. The fields are normally delimited by a special character, such as the TAB character or perhaps the colon (:) character.

To simplify the parsing of such files, you can specify a field-separator character with the **--**field-separator option of the sm_adapter command. Each occurrence of this character in the input file is replaced with an ASL field-separator marker (fs). By default, ASL does not translate characters to field-separator markers.

Consider the following line taken from the UNIX /etc/passwd file:

```
root:*:0:0:Admin:/:/bin/sh
```

Reading this line and using the file front end with the field-separator character set to the colon (:), feeds the following input stream into the rule set:

```
r o o t fs * fs 0 fs 0 fs A d m i n fs/fs / b i n / s h eol
```

This line can then be parsed with an ASL pattern:

```
word fs word fs integer fs integer fs word fs word fs word eol
```

The character replaced with an fs can never be seen by the rule set of the ASL script. The following pattern does not match the input stream:

```
word “:” word “:” integer fs integer fs word fs word fs word eol
```

File tail front end

The file tail front endis designed to read the log files of long-running Domain Managers. All of the characteristics of the file front end are present in the file tail front end. Unlike the file front end, however, the file tail front end does not simply open the file, read its contents, and then terminate. Rather it opens the file, seeks to the end of the file, and waits for additional lines to be written to the end of the file. As new lines appear, the front end feeds them to the rule set. The **--**tail option of the sm_adapter command invokes this front end.

Rotating Domain Manager log files

One of the difficulties in parsing Domain Manager log files is dealing with log file rotation. By their nature, Domain Manager log files grow over time. Most system administrators set up periodic jobs to remove or rename the log files to avoid consuming too much disk space. The file tail front end periodically checks if the file has been rotated. If so, the file is closed and reopened. The file is then read from the beginning. All entries in the newly opened file are fed into the rule set.

Program front end

The program front end reads data from the output of a command pipeline. Like the file front end, the program front end performs end-of-line and field-separator translations. This front end, for example, can parse the results of a SQL query issued by using a command-line SQL program.

The **--**program option of the sm_adapter command invokes the program front end. This command string is executed by using the native command interpreter for the system. On UNIX, this is normally /bin/sh. When the front end object is created, the command is executed and a pipeline is created between the command and the front end. This pipeline combines both the stdout and stderr streams from the command into a single stream.

Once the command exits (or once it closes stderr and stdout) the front end terminates.

Subscriber front end

The subscriber front end subscribes to a Domain Manager and feeds the subsequent notifications from the Domain Manager to the rule set. Subscriptions can be used for events, topology changes, and attribute changes.

Notification formatting

The Domain Manager generates a number of different kinds of notifications. Each notification received from the Domain Manager is streamed as a single record with an eol marker that identifies the end of the notification. Individual fields within each notification, for example the class name or instance name in an event notification, are separated by fsmarkers. The first field of all notifications is an integer timestamp in the standard UNIX format which indicates when the Domain Manager originally created the notification. The remaining fields of the notification record differ depending on the type of record.

Event notification records

Event notifications are received from the Domain Manager when the status of an event changes. In order to receive these notifications, you must subscribe to events in the Domain Manager.

The ASL format of the notification record is:

```
timestamp:    integer      fs
              “NOTIFY”     fs
class:        word         fs
instance:     word         fs
event:        word         fs
certainty:    float        fs
                           eol
```

Normally, the Domain Manager sends the following:

- Single notification message when an event becomes active

- Single clear message when the event is no longer active.

  If an event corresponds to a root-cause problem, it is possible that the certainty of the diagnosis will change over time. If the diagnosis certainty changes, the Domain Manager generates another notification. Notifications of this type are streamed in a slightly different manner:

```
timestamp:    integer            fs
              “CERTAINTY_CHANGE” fs
class:        word               fs
instance:     word               fs
event:        word               fs
certainty:    float              fs
                                 eol
```

  The subscriber front end normally discards certainty changes that are less than 1% (0.01). This behavior can be disabled.

  When the Domain Manager clears an event, the ASL format of the notification record is:

```
timestamp:    integer      fs
              “CLEAR”      fs
class:        word         fs
instance:     word         fs
event:        word         fs
                           eol
```

Object create record

When a new object is created in the Domain Manager’s repository, the Domain Manager sends an object create message. The ASL format of an object create record is:

```
timestamp:    integer       fs
              “CREATE”      fs
class:        word          fs
instance:     word          fs
                            eol
```

Object delete record

When an object is deleted from the Domain Manager’s repository, the Domain Manager sends an object delete message. The ASL format of an object delete record is:

```
timestamp:    integer       fs
              “DELETE”      fs
class:        word          fs
instance:     word          fs
                            eol
```

Class load record

When a new class is created in the Domain Manager's repository, the Domain Manager sends a class load message. Classes are created when new MODEL-generated libraries are loaded. The ASL format of a class load record is:

```
timestamp:    integer          fs
              "CLASS_LOAD"     fs
class:        word             fs
                               eol
```

Class unload record

When a class is deleted from the Domain Manager's repository, the Domain Manager sends a class unload message. Deleting of classes is not currently supported in the Domain Manager so this message is not used.

```
timestamp:    integer          fs
              "CLASS_UNLOAD"   fs
class:        word             fs
                               eol
```

Relation change record

When a relationship changes between objects, the Domain Manager sends a relation change message. The ASL format of a relation change record is:

```
timestamp:    integer          fs
              "RELATION_CHANGE" fs
class:        word             fs
instance:     word             fs
relation:     word             fs
                               eol
```

Property change record

When an object's property changes, the Domain Manager sends a property change message. The ASL format of a property change record is:

```
              "ATTR_CHANGE"    fs
class:        word             fs
instance:     word             fs
attribute:    word             fs
value:        word             fs
                               eol
```

Subscriber front end with a restartable Domain Manager

If the Domain Manager that is being sent subscriptions by the subscriber front end is a restartable Domain Manager, two additional messages are sent to the rule set:

- One when a connection is made to the Domain Manager.
- The other when the connection to the Domain Manager is lost.

Domain Manager connect record

When the connection to the Domain Manager is established, the Domain Manager sends a Domain Manager connect record. These records are sent to the rule set even if the front end issues no subscriptions. In fact, it is sometimes useful to use a subscriber front end with no subscriptions just for the purpose of being notified when the Domain Manager terminates and restarts. The ASL format of the Domain Manager connect record is:

```
timestamp:    integer          fs
              “CONNECT”        fs
server:       word             fs
                               eol
```

Domain Manager disconnect record

When the connection to the Domain Manager is lost, the Domain Manager sends a Domain Manager disconnect record. Like the Domain Manager connect record, these records are generated even if no subscriptions to the Domain Manager are issued. The ASL format of the Domain Manager disconnect message is:

```
timestamp:    integer          fs
              “DISCONNECT”     fs
server:       word             fs
                               eol
```

Sample front end invocations

This section provides sample front end invocations, as described in Sample front end invocations.

Table 7-3. Sample front end invocations

Type of front end	Invocation
File front end	$ sm_adapter --file=/etc/passwd --field-separator=: passwd.asl
File tail front end	$ sm_adapter --tail=/var/adm/messages syslog.asl

Table 7-3. Sample front end invocations (Continued)

Type of front end	Invocation
Program front end	$ sm_adapter --program="/bin/ps -ef" ps.asl
Subscriber front end	$ sm_adapter --subscribe=".*::.*::.*" --server=myDomainManager notify.asl The value of the subscribe parameter is parsed into class, instance, and event expressions. The "::" string separates the individual patterns. Subscriptions are sent to the Domain Manager specified with the --server option. In addition to the event patterns, the subscribe option also allows you to qualify the types of events included in the subscription set. A qualification has the form: --subscribe=".*::.*::.*/paev" The trailing /paev qualifies the subscription to include problems (p), aggregates (a), symptoms (e) in the subscription set, and verbose mode (v) which turns on subscription control messages. You can specify any combination of the letters p, a, e, and v. If no qualifier is specified, the default is problems only (/p). To use the subscription front end in conjunction with a restartable Domain Manager, use the --rserver option in place of the --server option.

sm_adapter front end options

sm_adapter front end options describes the available sm_adapter front end options.

Table 7-4. sm_adapter front end options

Option	Description
--file=<*path*>	Read input from a file. Also -f <*path*>.
--tail=<*path*>	Read input by tailing a file. Also -t <*path*>.
--program=<*cmd*>	Read input from a command pipeline. Also -p <*cmd*>.
--field-separator=C	Translate 'C' to the field-separator (FS) marker. Valid only in conjunction with --file, --tail or --program. Also -F <C>.
--subscribe=<*sub*>	Use the subscriber front end. Subscriptions are sent to the server specified with the --server option. The <*sub*> parameter is the subscription request. ■ If <*sub*> is 'topology', then a subscription for topology changes is requested. ■ If <*sub*> is of the form '<*name*>/n', then a subscription to NL <*name*> is requested. **Note** Only one NL subscription may be specified. ■ If <*sub*> is of the form C::I::E[/paev], 'C', 'I', 'E' are regexp patterns representing the classes, instances, and events to which to subscribe. The letters following a slash (/) are subscription qualifiers: 'p' means subscribe to problems; 'a' means subscribe to aggregates (impacts); and 'e' means subscribe to events. If none of these are present, 'p' is assumed. 'v' means run in verbose mode, which turns on subscription control messages. Otherwise, <*sub*> is a profile name in which the profile specifies what subscriptions are to be requested. A profile name may be optionally followed by the /v qualifier. Multiple --subscribe options can be specified.

Table 7-4. sm_adapter front end options (Continued)

Option	Description
--smoothing=<*num*>	Event smoothing interval. This parameter is used by the subscriber front end to smooth event notifications (and clears) received from the server. Only events (or clears) that stay active (or cleared) for <*num*> seconds are fed into the input stream. The <*num*> must be a non-negative integer. The default value is 0 which disables smoothing.
--ignoreOld	Ignore old notifications. This parameter is used by the subscriber front end. Notifications for events that were active before this adapter connected are not fed to the input stream.

Rule set

The rule set parameter is used to specify the ASL parse rules that are applied to the input data. The name of the file that contains the ASL parse rules is added at the end of the sm_adapter command. For example:

```
sm_adapter [<options>] myRules.asl
```

sm_adapter rule set options

The sm_adapter rule set options are listed in sm_adapter rule set options.

Table 7-5. sm_adapter rule set options

Option	Description
--D<*var*>=<*value*>	Override the default value for a rule set variable.
--verify	Validate rules only.

Back end

The back end of an adapter represents a Domain Manager. Object manipulations (for example, setting an attribute) within an ASL rule set are translated into VMware Smart Assurance Remote API requests. The Domain Manager to which the API transmits the request is determined by the back end. Two back ends are available:

- Remote Domain Manager —Specifies a remote Domain Manager.
- Restart-enabled Domain Manager — Specifies a remote Domain Manager, but the adapter handles automatic reconnects if the connection to the Domain Manager is broken.

  The --server option to the sm_adapter command creates a remote Domain Manager back end.

  ```
  % sm_adapter --server=myServer rules.asl
  ```

  The restart-enabled Domain Manager back end is a specialization of the remote Domain Manager that adds automatic reconnect capabilities. When using a normal remote Domain Manager back end, the adapter terminates if one of the following occurs:

- Domain Manager is unavailable when the adapter starts.

- Domain Manager disconnects while the adapater is running.

  The restart-enabled Domain Manager should be used in cases where you want the adapter to remain active when the Domain Manager is unavailable. The back end of a restart-enabled Domain Manager periodically attempts to reconnect to the Domain Manager. Once it succeeds, the connection is restored and the adapter continues to function.

  The restart-enabled back end is most useful in conjunction with the subscriber front end. The section Subscriber front end with a restartable Domain Manager provides additional information about using the restart-enabled Domain Manager with the subscriber front end.

Behavior of the restart-enabled Domain Manager

The restart-enabled Domain Manager provides a way to create an adapter that is robust in the face of communication problems with the Domain Manager.

Normally, errors encountered during the transmission of an operation to the Domain Manager result in an exception to the rule set. This exception normally terminates the adapter. With a restart-enabled Domain Manager, however, the exception is treated as a non-fatal error and it does not automatically terminate the parser. Once the Domain Manager connection is re-established, operations no longer result in exceptions but begin working properly again.

Using the restart-enabled Domain Manager from the command line

You can use the **--**rserver option of the sm_adapter command to create a restart-enabled Domain Manager back end.

```
$ sm_adapter --rserver=<Domain Manager_name> rules.asl
```

Back end options

sm_adapter back end options describes the available back end options.

Table 7-6. sm_adapter back end options

Option	Description
--server=self	Connect driver to local repository, the default.
--server=null	Do not connect to any server. Useful for debugging offline in combination with --traceServer.
--server=<*name*>	Connect driver to remote server. Also -s <*name*>.
--rserver=<*name*>	Auto-reconnect driver to remote server. Also -S <*name*>.
--description=<*desc*>	Description of this adapter, sent to remote server.
--mcast=<*name*>	Connect driver to a local subscription server.

Tracing operations

Rule set

When the adapter starts, rules contained in the rule set file are read in and compiled into an internal form. The compilation enables the adapter to parse the input data efficiently. If the trace attribute of the rule set object is set to TRUE, the adapter dumps a trace of the compiled rules after it has converted them to the internal form. This option is used for debugging purposes only.

Specify the **--**traceRules option with the sm_adapter command to set the trace attribute of the rule set object to TRUE.

Back end

All back ends of an adapter can trace all API operations transmitted to the Domain Manager. This is a very useful option for debugging a rule set. If you are using the sm_adapter command, specify the **--**traceServer option to enable back end tracing. The trace output includes the following:

- Timestamp
- Name of the Domain Manager
- Description of the operation sent to the Domain Manager

  If the operation returns a value (for instance, if you are retrieving an object property), the retrieved value is also printed.

  For example, when the following ASL commands,

  ```
  obj = object("MyClass","MyObject");
  obj->attr = TRUE;
  ```

  are executed in a rule set that is connected to a Domain Manager with the name, ServerName, the commands result in the following trace output:

  ```
  2-Feb-2009 14:26:11 ServerName: put(MyClass,MyObject,attr,TRUE)
  ```

Trace options

Trace options describes the available trace options.

Table 7-7. Trace options

Option	Description
--traceRules	Trace rule compilation.
--traceServer	Trace interactions with the back end server.
--traceParse	Trace rule matching.
--trace	Enable all tracing. Also -d.

Stopping adapters

Adapters that do not stop on their own can be stopped by using SIGTERM for UNIX systems.

You can create ASL scripts that include a stop() or quit() function. stop() function and quit() function provide additional information for the stop() and quit() functions, respectively.

ASL Reference

8

This chapter includes the following topics:

- Syntax
- Reserved words

Syntax

ASL syntax lists available ASL syntax.

Table 8-1. ASL syntax

Syntax	Description
!=	Indicates not equal to.
+	Indicates an addition operator.
-	Indicates a subtraction operator.
*	Indicates a multiplication operator.
/	Indicates a division operator.
%	Indicates a modulus operator. Calculates by using integer or floating-point numbers.
&&	Indicates a logical AND.
.	(Period) pattern matching. Indicates that the next pattern must be matched immediately.
.	(Period)action block. Concatenates two strings.
..	(Double period) Used to indicate an undefined string of characters up to the next pattern match.
||	Indicates a logical OR.
+=	Adds an object to a relationship.
?	Pattern matching. Matches one or zero times.
?	Action block. Exception handling operator.
<	Indicates less than.
<=	Indicates less than or equal to.
-=	Removes an object from relationship.

Table 8-1. **ASL syntax** (Continued)

Syntax	Description
==	Indicates equal to.
>	Indicates greater than.
->	Used to reference properties of an object.
->*	Used to reference properties of an object by using a variable for the property name.
>=	Indicates greater than or equal to.
any(*<string>)*	Represents any character in its argument string.
boolean(*<value>*)	Converts the argument to a TRUE or FALSE. All nonzero numbers are TRUE. Any other type is converted to an uppercase string and compared to TRUE or FALSE. If it does not match either, it returns an error.
break	Used to break out of a loop.
case = [exact]|[ignore]	A variable that determines whether string matches are case-sensitive (default is exact) or not.
char	Represents a character, not an eol or fs.
consistencyUpdate()	Causes the Domain Manager to recompute the correlation rules.
continue	Used to move to start of loop and start with the next element.
correlate()	Causes the Domain Manager to correlate events.
create(*<classname>*,*<objectname>*)	Creates an object.
create(*<objhandle>*)	Creates an object handle which represents an instance.
default	Defines the value to use for a variable if the variable is not assigned a value.
defined(*<variable>*)	Determines whether a variable is defined.
delete()	Deletes an object on the Domain Manager.
delim	Defines delimiters.
do	Marks the beginning of an action block.
else {*<statements>*}	Alternative actions when an if statement fails.
eol	Represents the end of a line of data.
exact	Used in conjunction with “case” to make all string matches case-sensitive.
FAIL	Keyword for exception handling. Causes a rule to fail when an exception occurs.
FALSE	A Boolean false.
feError()	Returns a Boolean value. TRUE if the front end has reported a failure to read data.

Table 8-1. ASL syntax (Continued)

Syntax	Description
feErrorMsg()	If the feError() function is true, the feErrorMsg() function returns a string that describes the error.
feErrorReset()	Resets the error state so that there is no error.
filter	Marks the beginning of a filter block.
float	Represents a floating number, including an optional minus sign.
foreach <*variable*> (<*list_or_table*>) {<*statements*>}	Iterates over the values of a list or the index of a table. Variable is assigned successive values of the list of table.
fs	Represents a field-separator.
getCauses(<*classname*>,<*objectname*>, <*eventname*>[,<oneHop>])	Returns a list of problems that can cause that event. Each element of the list is a list that contains classname, objectname, and eventname of the root cause that causes that event.
getChildren(<*classname*>[,recursive])	Retrieves the list of classes derived from the specified class. The recursive parameter is optional. ■ If omitted or false, only the immediate child classes are retrieved. ■ If true, all children, including those of derived classes are retrieved.
getClosure(<*classname*>,<*objectname*>,<*eventname*>[,<*oneHop*>])	Given a root cause or aggregation (compound), returns a list of symptoms for that root cause. Returned list is similar to the getCauses() function.
getEventClassName(<*classname*>, <*eventname*>)	Returns a string with the name of the ancestor class associated with a class and an event.
getEventDescription(<*classname*>, <*eventname*>)	Returns a description for an event. The description string is defined in MODEL.
getEventType(<*classname*>, <*eventname*>)	Returns a string that indicates the type of the event (PROBLEM, EVENT, AGGREGATE).
getExplainedBy(<*classname*>, <*instancename*>,<*eventname*>[,<*oneHop*>])	Returns those problems which the MODEL developer has listed as explaining this problem.
getExplains(<*classname*>, <*objectname*>,<*eventname*>[,<*oneHop*>])	Given a root cause, returns the alternate closure as defined in MODEL.
getInstances(<*classname*>)	Returns a list of strings (not object handles) that are the names of the instances of that class.
getRuleFileName([<*fullname*>])	Returns the filename of the currently executing rule file.
getServerName()	Returns the name of the Domain Manager.
getExternalString(<*table*>, <*key*>, <*locale*>)	Extracts a string given a table name, key, and locale, and renders the string given a list of substitutions (up to six).
glob(<*pattern*>,<*string*>)	Enables glob style pattern matching. Standard glob syntax. Returns a Boolean.

Table 8-1. ASL syntax (Continued)

Syntax	Description
global	Defines the scope of a variable as global. If more than one adapter for a repository, global values can be shared.
hex	Represents a hexidecimal number. There is no minus sign.
hexToString(<*hexadecimal*>)	Converts a hexidecimal number (the argument) to a string.
if (<*conditional expression*>) {<*statements*>}	Represents a conditional statement.
ignore	Used in conjunction with “case” to make all string matches NOT case-sensitive.
IGNORE	Exception handling, ignores exception and continue.
input=string	Defines the input for parsing.
integer	Represents an integer, including an optional minus sign.
is(<*objecthandle*>-><*Relate*>,<*object2handle*>)	Tests whether an object is a member of a relationship.
isNull()	Tests whether an object handle points a valid object. If TRUE, object does not exist.
len(<*number*>)	Moves the current starting position of an input string a number of characters.
list(<*listitem1*>,<*listitem2*>,<*listitem3*>, *etc.*)	Creates a list variable. Can either be used with arguments or without.
local	Variable scoping keyword.
LOG	Keyword for exception handling. Writes to the system log when an exception occurs.
LOG(<*loglevel*>)	Keyword for exception handling. Writes to the system log when an exception occurs and allows the classification of the exception’s severity.
NEXT	Keyword for exception handling. Skips remaining actions in do block and goes to next rule.
NO_LOCK	Argument passed to the transaction() function.
not(<*pattern*>)	Does not match if pattern matches. Matches if the pattern does not.
notany(<*string*>)	Matches any character not included its argument <*string*>.
numeric()	Attempts to convert the argument to a number. If it is a Boolean, it returns 1 if TRUE and 0 is FALSE. If it is a string, it tries to interpret it as a number. If it cannot, an error occurs.
object([<*classname*>,]<*objectname*>)	Converts a name to an object handle.
object(<*objhandle*>)	Returns an object handle.
peek(<*pattern*>)	Prescan input for a pattern and match or fail it. The search position does not change by using peek.
print(<*string*>)	Prints the argument string to the screen.

Table 8-1. **ASL syntax** (Continued)

Syntax	Description
quit()	Shuts down the VMware Smart Assurance process the adapter is talking to. This can be the adapter or a Domain Manager. Tells the back end to shut down.If you launch sm_adapter and do not specify a back end, the adapter becomes its own back end. In such cases, quit() does the same thing as self->quit().
READ_LOCK	Argument passed to the transaction() function.
rep(*<pattern>[,<Number>]*)	Repeat pattern or rule a defined number of times or until it fails.
return *<string>*	Returns a value from a do block.
self	Same as self->
self->	Directs functions that interact with the repository to use the adapters repository and not the Domain Manager's. Used in conjunction with adding objects to the adapter's repository.
setLocale(*<locale>*)	The *<locale>* is the RFC-1766/3066 compliant specification for language and country. The return value is a string representation of the previous setting of the locale, or a default value if setLocale() has not yet been called.
sizeOf(*<string>*)	Counts the number of characters in a string.
sizeof(*<string>*)	Counts the number of characters in a string.
sleep(*<number>*)	Causes the adapter to sleep for a certain number of seconds.
STOP	Exception handling. Stops the ASL script.
stop()	Stops the current thread. If you call stop() in an ASL thread, and that thread is in a process that has other threads, the effect is to simply kill that thread. But if that thread is the only thread, which is the typical case when using sm_adapter, stopping that thread effectively kills the adapter (because the adapter shuts down when there are no more active threads).
string(*<value>*)	Converts the argument to a string.
substring(*<string>*,*<start_pos>*,*<num_chars_to_remove>*)	Returns a new string that is a piece of the string passed to it.
tab(*<integer>*)	Moves the starting position in an input string to the position passed to the function. This cannot be used to go backwards. If no argument is specified, this function returns the starting position for pattern matching in an input string.
table()	Creates a table variable.
thread()	Returns the thread ID of the thread that is running the adapter.
time()	Returns the system time.

Table 8-1. ASL syntax (Continued)

Syntax	Description
toLower(*<string>*)	Converts string to lowercase letters.
toUpper(*<string>*)	Converts string to uppercase letters.
trace*(<string>*,*BOOLEAN*)	Makes tracing available from with ASL
transaction([<WRITE_LOCK|READ_LOCK|NO_LOCK>])	Starts a repository transaction. Allows updates to a Domain Manager to be entered and then committed all at once. Needs to be committed before changes in the Domain Manager are accepted. Use the abort() function instead of the commit() function to remove changes. If a START rule begins before things are committed, they are automatically aborted. NO_LOCK is the default.
TRUE	Boolean true.
undef()	Undefines a variable. Appears as if the variable was never assigned.
while	Conditional statement causes loop while condition is true.
word	Represents a series of characters ending with, but not including, a delimiter.
WRITE_LOCK	Argument passed to the transaction() function.

Reserved words

ASL reserved words lists the ASL words that are reserved and should not be used as identifiers or variables. The reserved words are case-sensitive. Parentheses () indicate functions. Not all of the reserved words are currently used.

any()	float	ignore	sizeOf()
boolean()	foreach	IGNORE	sizeof()
break	formatString()	input	rep()
case	fs	integer	repository
char	getCauses()	is()	sleep()
clear()	getChildren()	isNull()	STOP
consistencyUpdate()	getClassDescription()	len()	stop()
continue	getClosure()	list()	string()
correlate()	getEventClassName()	local	substring()
create()	getEventDescription()	LOG	tab()
default	getEventType()	NEXT	table()
defined()	getExplainedBy()	NO_LOCK	thread()
delete()	getExplains()	not()	time()
delim	getExternalString()	notany()	toLower()

do	getInstances()	notify()	toUpper()
else	getOpDescription()	numeric()	trace()
eol	getPropDescription()	object()	transaction()
exact	getRuleFileName()	peek()	
FAIL	getServerName()	print()	TRUE
FALSE	glob()	quit()	undef()
feError()	global	READ_LOCK	while
feErrorMsg()	hex	return()	word
feErrorReset()	hexToString()	self	WRITE_LOCK
filter	if	setLocale()	

dmctl Reference

9

This chapter includes the following topics:

- Description
- dmctl syntax

Description

The Domain Manager Control Program or dmctl is a command line tool for interacting with a Domain Manager. *"dmctl syntax" on page 153* provides additional information. The dmctl can be used to query, modify, or receive notifications from a Domain Manager.

The tool can do any of the following:

- Execute commands typed at the command line
- Execute commands read from a batch file
- Interactively read commands typed in

  If no batch file or command is specified, dmctl enters an interactive mode in which it prints a prompt and accepts typed user commands. If a batch file is specified, dmctl executes the commands in the batch file. Single commands can also be specified.

  Command names can be abbreviated, usually to the shortest unique prefix. There are exceptions to allow a common command to be typed easily when an uncommon one conflicts with it, and conversely, to prevent the accidental typing of some commands. For example, getE is getEvents. The minimum abbreviation for getEventDescription is getEventD. The quit and exit commands cannot be abbreviated, and shutdown must appear as at least shut. Uppercase letters in command names are shown for clarity only. They can be typed in lowercase, with the same meaning.

  In non-interactive mode, dmctl also accepts commands that subscribe to notifications from the Domain Manager. In that case, dmctl does not return, but continuously waits for notifications and prints them to standard out (stdout).
- In a command-line mode, the Domain Manager to interact with must be specified with the **--**server=<name> or **-s** <name> option.
- In batch or interactive mode, the Domain Manager to interact with can be specified later with the attach command.

ASL and dmctl

Issuing dmctl instructions is a good method to use to find out about the classes, attributes, events, relationships, and methods available in a Domain Manager. When you debug ASL scripts, dmctl is an effective tool to monitor the state of a Domain Manager.

Setting the locale

Locale is the RFC-1766/3066 compliant specification for language and country. For example, “fr_FR” refers to French as spoken in France.

The setLocale command can be used in dmctl to set the locale of a session, as described in Setting the locale. The types of text that the locale affects is the same as those described for an ASL script. The section Locale effects on message rendering provides additional information.

dmctl syntax

The basic syntax is:

```
dmctl [options...] [<command>]
```

When given a command or batch file to run, dmctl executes the commands and exits.

The options include those listed in dmctl options:

Table 9-1. dmctl options

Option	Description
--server=<*name*>	Name of Domain Manager. This argument is used to identify the Domain Manager to connect to. If it is not specified, it can be set later, in interactive mode, by using the attach command. If it is in a host:port/name format, the specified host:port is used to locate Domain Manager name. Otherwise, if it is in a simple name format, name is located by the Broker. Also -s <*name*>.
--broker=<*location*>	Alternate Broker <*host*>:<*port*> location. If it is not specified, the Broker is located by the standard search order, as follows: If the SM_BROKER environment variable is defined, use its value. Otherwise, use the default smarts-broker:426 location. Either <*host*> or :<*port*> portions may be omitted, in which case the defaults smarts-broker and :426 are used, respectively. Also -b <*location*>.
--file=<*file*>	Input batch file. Given this option, dmctl executes the commands in the file and exits. Also -f <*file*>.

Table 9-1. dmctl options (Continued)

Option	Description
--timeout[=<*seconds*>]	Set a timeout on the remote execution of each command. A value of 0 specifies no limit. The default is 0 (no limit) in interactive mode, 60 (1 minute) in non-interactive mode. If **--**timeout is specified without an argument, a value of 60 seconds is used. If a remote command takes too long, an error message is printed and dmctl immediately exits with the status ETIME.
--traceServer	Trace actions performed in the Domain Manager.
--commands	List dmctl commands and exit. Also **-**c.
--help	Print help and exit.
--version	Print program version and exit.

At any given time, dmctl can be attached to (at most) one Domain Manager. The dmctl forwards all accepted commands to the attached Domain Manager, receives a response, and prints it to stdout. It is important to remember that the commands are invoked in the server process. The commands include those in dmctl commands :

Table 9-2. dmctl commands

Command	Description
attach domain	Attaches to the specified Domain Manager. Once a Domain Manager is attached, other commands can be invoked.
clear <*class*> :: <*instance*> :: <*event*>	Forces a clear of the specified event.
create <*class*>::<*instance*>	Creates a new instance in the repository.
consistencyUpdate	Recomputes the codebook.
correlate	Correlates now.
delete <*class*>::<*instance*>	Deletes an instance from the repository.
detach	Detaches from the Domain Manager. Another Domain Manager can now be attached with the attach command.
execute <*program*> [<*arg1*> ...]	Executes a program. Program should be the base name of the program file, without the suffix or directory. For example, use “name” to load the program name.po.
exit	Exits dmctl.
findInstances <*class-pattern*>::<*instance-pattern*>	Lists instances that match given class and instance patterns.

Table 9-2. dmctl commands (Continued)

Command	Description
get <*class*>::<*instance*>[::<*property*>]	Lists all properties values or a given property value for a particular class instance.
getClasses	Lists all classes in the repository.
getEvents <*class*>	Lists all exported events defined in given class.
getEventDescription <*class*>::<*event*>	Prints description of given event.
getInstances [<*class*>]	Lists all instances in the repository, or all instances of the given class.
getModels	Lists all models loaded to the Domain Manager.
getOperations <*class*>	Lists all operations defined in given class.
getPrograms	Lists all programs loaded to the Domain Manager.
getProperties <*class*>	Lists all properties defined in given class.
getThreads	Lists all threads that are running in the Domain Manager.
insert <*class*>::<*instance*>::<*property*> <*value*>	Inserts a value into a table or relationship.
invoke <*class*>::<*instance*> <*op*> [<*arg1*> ...]	Invokes an operation of given instance.
loadModel <*model*>	Loads a new MODEL library. The <*model*> should be the base name of the library. Do not specify a prefix or suffix with the name. Once a MODEL library is loaded, a prefix or suffix is added to the name. For example, libname.so (Solaris), libname.sl (HP-UX), or name.dll. After a MODEL library is loaded, the repository can be populated with instances of classes defined in that library.
loadProgram <*program*>	Loads a new program.The <*program*> should be the base name of the program file. Do not specify a prefix or suffix with the name. Once a program is loaded, a prefix or suffix is added to the name. For example, *name.po*. After a program is loaded, it can be executed with the execute command.
notify <*class*> :: <*instance*> :: <*event*>	Forces notification of a given event.

Table 9-2. dmctl commands (Continued)

Command	Description
ping	Verifies that the Domain Manager is still alive.
put <*class*>::<*instance*>::<*property*> <*value1*> [<*value2*> ...]	Sets value of given property.
quit	Quits dmctl.
remove <*class*>::<*instance*>::<*property*> <*value*>	Removes a value from a table or relationship.
restore <*file*>	Restores the repository from a file. The <*file*> should not contain a directory portion; it is read from BASEDIR/smarts/repos/.
shutdown	Shuts down the Domain Manager.
save <*file*> [<*class*>]	Saves the repository to a file. The <*file*> should not contain a directory portion. It is saved to BASEDIR/smarts/repos/. If <*class*> is specified, then save only the sub-tree rooted at class.
setLocale <*locale*>	Allows you to set the locale before a Domain Manager attach operation, and to change the locale while the session is open. The setLocale command will print the value of the previous setting on completion of the API call. The default locale is set for a dmctl session as follows: 1 The value of the SM_LOCALE environment variable. 2 If SM_LOCALE is not set equal to a value, then the default locale is set equal to en_US.
status	Displays the connection status.
subscribe <*class-regexp*>::<*instance-regexp*>::<*event-regexp*> ...	Subscribes to problems and events that match the given patterns. The dmctl sends the subscription requests, and then loops indefinitely, printing the received notifications. The program exits only when the Domain Manager is shut down, or when interrupted.

Table 9-2. dmctl commands (Continued)

Command	Description
subscribeEvents <*class-regexp*>::<*instance-regexp*>::<*event-regexp*> ...	Subscribes to events that match the given patterns. The dmctl sends the subscription requests, and then loops indefinitely, printing the received notifications. The program exits only when the Domain Manager is shut down, or when interrupted.
subscribeProblems < *class-regexp*>::<*instance-regexp*>::<*event-regexp*> ...	Subscribes to problems that match the given patterns. The dmctl sends the subscription requests, and then loops indefinitely, printing the received notifications. The program exits only when the Domain Manager is shut down, or when interrupted.

Card-Port MODEL Code

10

Card-Port MODEL example

The following example provides the Card-Port MODEL as described in *Correlation model used for example scripts*.

```
/* card.mdl -
*
* Copyright (c) 2019, VMware Inc.
* All Rights Reserved
*
* A simple model file for use as an example of writing and
* building a small model.
*
*/
// Include the "resource" class from the netmate heirarchy.
#include "nm/nm.mdl"
// Since we include nm.mdl for purposes of derivation, we must
// have the generated .h file include nm.h
#pragma include_h "nm/nm.h"
//////////////////////////////////////////////////////////////////
//
// This is a very simple card.
//
//////////////////////////////////////////////////////////////////
// The class Card
```

```
interface Card : MR_ManagedObject {
// Attributes maintained for the class Card
attribute string CardDesc
"A brief description of the card";
// Relationship between the class Card and
// the class Port
relationshipset ComposedOf, Port, PartOf
"The ports in this card";
// The notifications for the class Card
export
Down, // Problems
Impaired; // Compound Notification
problem Down
"The card is down, causing all its ports to be "
"operationally down"
= OperationallyDown;
propagate symptom OperationallyDown
"Symptom observed on the ports in this card"
= Port, ComposedOf, OperationallyDown;
// Compound notification
aggregate Impaired
"The card or a port on this card is Down"
= Down,
PortDown;
Card-Port MODEL example 149
Card-Port MODEL Code
propagate aggregate PortDown
"The Down problem on ports in this card"
= Port, ComposedOf, Down;
}
// The class Port
interface Port : MR_ManagedObject {
```

```
// Attributes maintained for the class Port
enum operStatus_e {
TESTING = 0,
UP = 1,
DOWN = 2
};
attribute operStatus_e operStatus
"The operational status of the port"
= TESTING;
// relationship between the class Port and
// the class Card
relationship PartOf, Card, ComposedOf
"The card this port is part of";
// The notifications for the class Port
export
OperationallyDown, // Symptom
Down; // Problem
event OperationallyDown
"This port is not operational"
= operStatus == DOWN;
// Problem
problem Down
"The port is down"
= OperationallyDown;
}
```