InCharge™

ASL Reference Guide

Version 6.2

smarts

Cisco Part Number: OL-6120-01

Contents

Preface

The purpose of this guide is to provide information about the development and construction of adapters using the Adapter Scripting Language (ASL). This guide includes several chapters on the syntax of ASL. It also includes information about how to build adapters.

Intended Audience

This guide is intended to be read by any user who needs to create or modify custom adapters for InCharge Domain Managers.

Prerequisites

It is assumed that an InCharge Domain Manager has been properly installed and configured.

Document Organization

The chapters are organized as shown in Table 1.

1.	ADAPTER OVERVIEW	Describes basic adapter concepts.
2.	ASL RULE SETS	Describes how ASL rules are structured.
3.	DATA, VARIABLES, OPERATORS, AND EXPRESSIONS	Describes data and basic operations.
4.	ASL PATTERN MATCHING	Describes the purpose and syntax of ASL pattern matching.
5.	ASL FILTERS	Describes the purpose and syntax of ASL filters.
6.	ASL ACTIONS	Describes the purpose and syntax of ASL actions.
7.	INTERFACING WITH AN INCHARGE DOMAIN MANAGER	Describes how to use ASL to interact with InCharge Domain Manager objects.
8.	RUNNING ADAPTERS	Describes how to run adapters using the **sm_adapter** command.
A.	ASL REFERENCE	Quick reference to ASL functions and operators.
B.	DMCTL REFERENCE	Quick reference to dmctl.
C.	CARD-PORT MODEL CODE	Code listing of a correlation model developed to demonstrate how ASL interacts with InCharge Domain Manager objects.

Table 1: Document Organization

Documentation Conventions

Several conventions may be used in this document as shown in Table 2.

CONVENTION	EXPLANATION
`sample code`	Indicates code fragments and examples in Courier font
keyword	Indicates commands, keywords, literals, and operators in bold
`%`	Indicates C shell prompt
#	Indicates C shell superuser prompt
<parameter>	Indicates a user-supplied value or a list of non-terminal items in angle brackets
[option]	Indicates optional terms in brackets

CONVENTION	EXPLANATION
/InCharge	Indicates directory path names in italics
yourDomain	Indicates a user-specific or user-supplied value in bold, italics
File > Open	Indicates a menu path in italics
▼▲	Indicates a command that is formatted so that it wraps over one or more lines. The command must be typed as one line.

Table 2: Documentation Conventions

Directory path names are shown with forward slashes (/). Users of the Windows operating systems should substitute back slashes (\) for forward slashes.

Also, if there are figures illustrating consoles in this document, they represent the consoles as they appear in Windows. Under UNIX, the consoles appear with slight differences. For example, in views that display items in a tree hierarchy such as the Topology Browser, a plus sign displays for Windows and an open circle displays for UNIX.

Finally, unless otherwise specified, the term InCharge Manager is used to refer to InCharge programs such as Domain Managers, Global Managers, and adapters.

InCharge Installation Directory

In this document, the term ***BASEDIR*** represents the location where InCharge software is installed.

- For UNIX, this location is: */opt/InCharge<n>/<productsuite>.*
- For Windows, this location is: *C:\InCharge<n>\<productsuite>*.

The *<n>* represents the InCharge software platform version number. The *<productsuite>* represents the InCharge product suite that the product is part of.

Table 3 defines the *<productsuite>* directory for each InCharge product.

PRODUCT SUITE	INCLUDES THESE PRODUCTS	DIRECTORY
InCharge IP Management Suite	• IP Availability Manager • IP Performance Manager • IP Discovery Manager • InCharge Adapter for HP OpenView NNM • InCharge Adapter for IBM/Tivoli NetView	/IP
InCharge Service Assurance Management Suite	• Service Assurance Manager • Global Console • Business Dashboard • Business Impact Manager • Report Manager • SAM Failover System • Notification Adapters • Adapter Platform • SQL Data Interface Adapter • SNMP Trap Adapter • Syslog Adapter • XML Adapter • InCharge Adapter for Remedy • InCharge Adapter for TIBCO Rendezvous • InCharge Adapter for Concord eHealth • InCharge Adapter for InfoVista • InCharge Adapter for NetIQ AppManager	/SAM
InCharge Application Management Suite	• Application Services Manager • Beacon for WebSphere • Application Connectivity Monitor	/APP
InCharge Security Infrastructure Management Suite	• Security Infrastructure Manager • Firewall Performance Manager • InCharge Adapter for Check Point/Nokia • InCharge Adapter for Cisco Security	/SIM
InCharge Software Development Kit	• Software Development Kit	/SDK

Table 3: Product Suite Directory for InCharge Products

For example, on UNIX operating systems, InCharge IP Availability Manager is, by default, installed to */opt/InCharge6/IP/smarts*. This location is referred to as ***BASEDIR****/smarts*.

Optionally, you can specify the root of ***BASEDIR*** to be something other than */opt/InCharge6* (on UNIX) or *C:\InCharge6* (on Windows), but you cannot change the *<productsuite>* location under the root directory.

For more information about the directory structure of InCharge software, refer to the *InCharge System Administration Guide*.

Additional Resources

In addition to this manual, SMARTS provides the following resources.

InCharge Commands

Descriptions of InCharge commands are available as HTML pages. The *index.html* file, which provides an index to the various commands, is located in the ***BASEDIR****/smarts/doc/html/usage* directory.

Documentation

Readers of this manual may find other SMARTS documentation (also available in the ***BASEDIR****/smarts/doc/pdf* directory) helpful.

InCharge Documentation

The following SMARTS documents are product independent and thus relevant to users of all InCharge products:

- *InCharge Release Notes*
- *InCharge Documentation Roadmap*
- *InCharge System Administration Guide*
- *InCharge ICIM Reference*
- *InCharge ASL Reference Guide*
- *InCharge Perl Reference Guide*

Software Development Kit Documentation

The following SMARTS documents are relevant to users of the Software Development Kit.

- *InCharge Software Development Kit Remote API for Java* (in HTML format)
- *InCharge Software Development Kit Remote API Programmer's Guide*
- *InCharge ICIM Reference* (in HTML format)
- *InCharge Software Development Kit MODEL Reference Guide*

Common Abbreviations and Acronyms

The following lists common abbreviations and acronyms that are used in the InCharge guides.

ASL	Adapter Scripting Language
CDP	Cisco Discovery Protocol
ICIM	InCharge Common Information Model
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MSFC	Multilayer Switch Feature Card
MIB	Management Information Base
MODEL	Managed Object Definition Language
RSFC	Router Switch Feature Card
RSM	Router Switch Module
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
VLAN	Virtual Local Area Network

Technical Support

SMARTS provides technical support by e-mail or phone during normal business hours (8:00 A.M.—6:00 P.M. U.S. Eastern and Greenwich Mean Time). In addition, SMARTS offers the InCharge Express self-service web tool. The web tool allows customers to access a personalized web page and view, modify, or create help/trouble/support tickets. To access the self-service web tool, point your browser to:

https://websupport.smarts.com/SelfService/smarts/en-us

U.S.A Technical Support

E-Mail: *support@smarts.com*
Phone: +1.914.798.8600

EMEA Technical Support

E-Mail: *support-emea@smarts.com*
Phone: +44 (0) 1753.878140

Asia-Pac Technical Support

E-Mail: *support-asiapac@smarts.com*

You may also contact SMARTS at:

	U.S.A World Headquarters	**United Kingdom**
Address	SMARTS 44 South Broadway White Plains, New York 10601 U.S.A	SMARTS Gainsborough House 17-23 High Street Slough Berkshire SL1 1DY United Kingdom
Phone	+1.914.948.6200	+44 (0)1753.878110
Fax	+1.914.948.6270	+44 (0)1753.878111

For sales inquiries, contact SMARTS Sales at:
sales@smarts.com

SMARTS is on the World Wide Web at:
http://www.smarts.com

1

Adapter Overview

Purpose of Adapters

Adapters facilitate the communication of information between devices or applications and an InCharge Domain Manager. Adapters exist as either inflow adapters or outflow adapters. Inflow adapters deliver information to an InCharge Domain Manager. Outflow adapters pass information from an InCharge Domain Manager to other applications.

- Inflow adapters collect information and send it to an InCharge 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 an InCharge Domain Manager for a set of notifications. These adapters then pass the information to devices or other applications.

Figure 1 shows the flow of information through an inflow adapter, an InCharge Domain Manager, and an outflow adapter. The inflow adapter collects information and passes it to the InCharge Domain Manager. The domain manager generates notifications. The outflow adapter subscribes to a set of notifications and the domain manager passes those notifications to the adapter. Then, the outflow adapter passes the information to another entity.

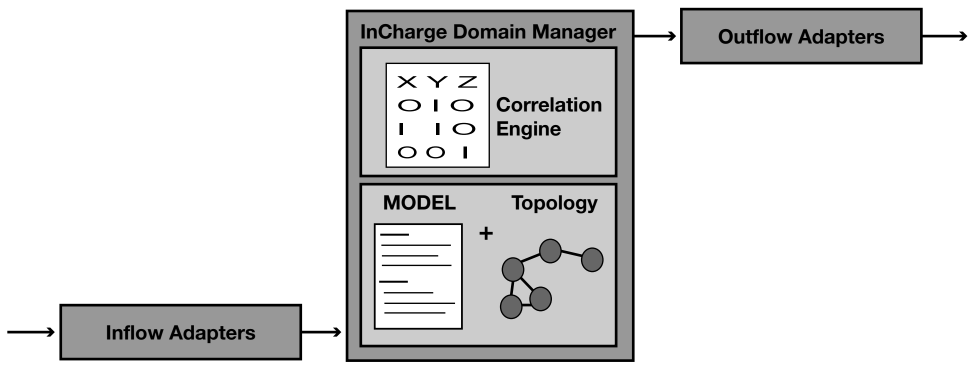


Figure 1: Adapter Overview

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

For example, in Figure 2, if an adapter is designed to move SNMP data to an InCharge Domain Manager, its Front-End communicates with a SNMP Device and its Back-End passes data to an InCharge Domain Manager.

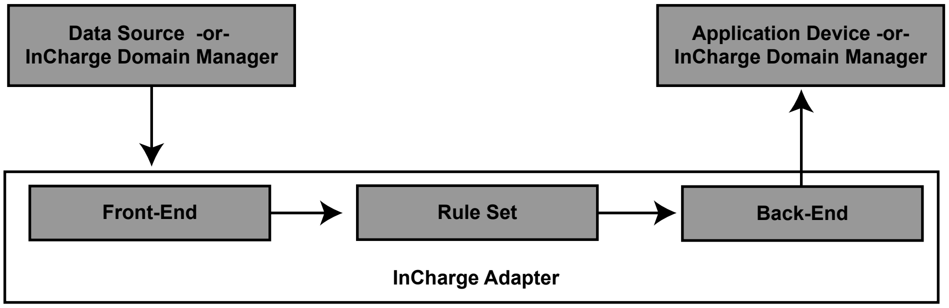

Data Source -or- InCharge Domain Manager
Application Device -or- InCharge Domain Manager
Front-End
Rule Set
Back-End
InCharge Adapter

Figure 2: Adapter Components

Adapter Scripting Language

The Adapter Scripting Language (ASL) is used to construct the rule sets of adapters. These rulesets 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 an InCharge Domain Manager.

Quick Introduction to Running Adapters

The **sm_adapter** command starts ASL scripts. The file *sm_adapter* (*sm_adapter.exe* for Windows) 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.

See "Running Adapters" on page 115 for more information about the **sm_adapter** command.

2

ASL Rule Sets

This chapter describes the construction of ASL rule sets. The chapter covers the basic format of rule sets and special rules.

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 pattern matching to the incoming data stream. 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. (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. See "Reserved Words" on page 136.

Curly braces surround the pattern list and demarcate the pattern block. A pattern block can contain one or more do 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 curly brace ({) marks the beginning of a filter block. Another curly 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 curly brace ({) marks the beginning of an action block. Another curly 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.

Using the guidelines above, 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 above encompasses all of the patterns and all of the action blocks. In order to add a filter block, the curly brace marking the end of the pattern block has to follow the last pattern list. Filter information can be added before the last do 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 can not be called from a filter.

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

Patterns and called rules can be interspersed with action (do) blocks.

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

Rules referenced by other rules are referred to as *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 rules called by the START rule, a rule named DEFAULT runs automatically. The purpose of the DEFAULT rule is to re-synchronize 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. For information about patterns, see "ASL Pattern Matching" on page 27.

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>
}
```

Note: The EOF rule shown above does not include a pattern matching section. The braces associated with the pattern block are not needed if there is no pattern.

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.

3

Data, Variables, Operators, and Expressions

Data

ASL's input data stream consists of characters and markers. Characters include any character in the extended ASCII character set. Markers demarcate the boundaries of fields and records.

Variables and Their Values

Variables are assigned values in an ASL script or at adapter startup. (To specify variable values at startup, see "Variable Assignment at Startup" on page 21.) ASL variables do not have a declared type, the values assigned to them are typed. Table 4 lists the types of values.

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. See "Reserved Words" on page 136.

VALUE TYPE	NOTE
numeric	ASL stores all numbers as double floating point.
string	
Boolean	
list	
table	
object handle	See "Interfacing With an InCharge Domain Manager" on page 89 for more information.
datetime	See "Time Function" on page 80 for more information.

Table 4: Types of Values for Variables

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 uses of values, ASL converts one type of value to the appropriate data type for use in a function or expression. For example, the variable below 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 semi-colon 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. The following table shows the 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.)

Table 5: Type Conversion Functions

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 `list` function initializes a list.

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 last 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.

Multi-column tables can be implemented in two ways. The first method is 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 uses 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"]);
}
```

Note: The use of the vertical line character ("|") to join the keys has no special meaning. Any character can be used.

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 only accessible 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 running on the same process where the variable is assigned a value.

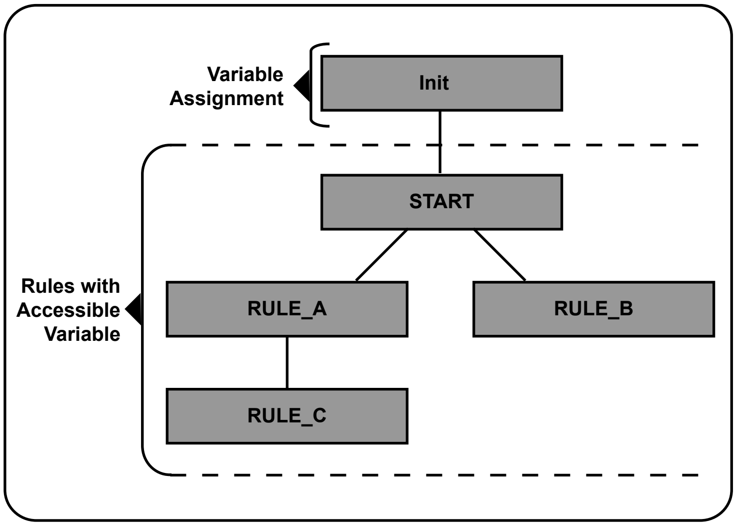
Driver Scope

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 Figure 4, 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);}
```

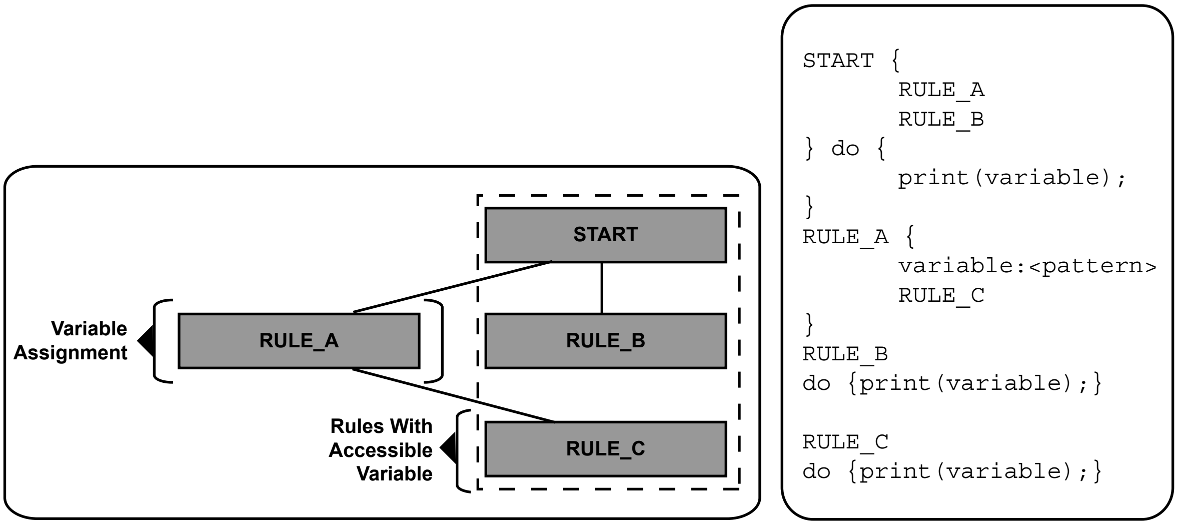
Figure 3: Example of a Variable With a Static Lifetime

In the next 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 Figure 4, each time the START rule is invoked, the variable is undefined.



```
START {
        RULE_A
        RULE_B
} do {
        print(variable);
}
RULE_A {
        variable:<pattern>
        RULE_C
}
RULE_B
do {print(variable);}

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

Figure 4: 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 rule HI runs (`x=TRUE`). This variable holds its value when the action block of the program is invoked after the rule HI. 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
$
```

Note: A driver scoped variable with a static lifetime remains static, even when a new value is assigned in a rule.

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.

If a local variable is used in a recursive rule, each recursion contains an exclusive value for the variable. See Figure 5.

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>;
```

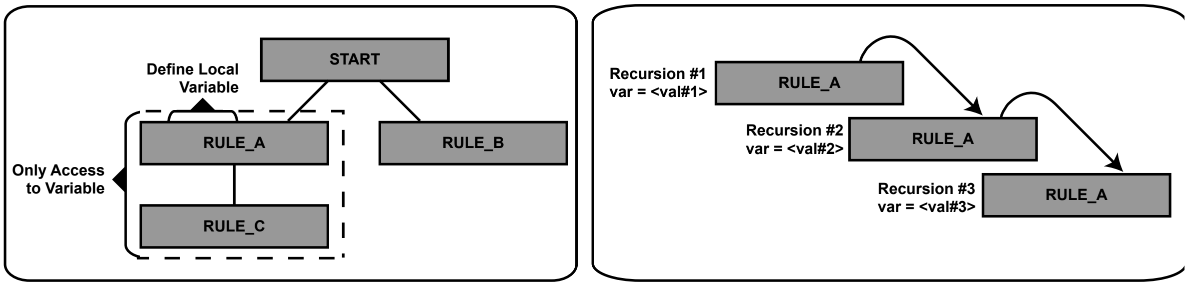


Figure 5: Local Variables Used in a Recursive Rule

The following example demonstrates how using 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 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. (For more information about eol markers, see "End of Line Matches" on page 44.)

```
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

Several adapters using the same process can share global variables. Global variables can contain numeric, string, and Boolean values only. They cannot be used as lists or tables. Use this syntax to declare a global variable:

```
global <variable_name>;
```

The global variable is always declared before the execution of the START rule.

Default Variable Values

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

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

Default variables can have any scope. 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 variable values 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 assigned using the `-d` option.

The following code fragment prints the number 20 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 using the `-D` option. Variables assigned using this method have a static lifetime. Default variables that are locally scoped cannot be assigned a value using this method.

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

This value assigned by the switch can be overridden by a standard variable assignment through either pattern matching or 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 and Expressions

Arithmetic Operators

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

OPERATOR	DESCRIPTION
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Table 6: Arithmetic Operators

Note: You can isolate the decimal portion of a number with the modulus operator 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. For more information see "ASL Pattern Matching" on page 27.

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:

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

Table 7: Relational and Logical Operators

Precedence and Order of Evaluation

ASL evaluates operators in the order listed below. 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.

OPERATOR
`( )`
`* /`
`%`
`+ -`
`== != < > <= >=`
`&& || .`

Comments

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

```
// ASL Script to read animal information
```

A comment can also be marked as a block 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 animal information
   created 6/30/2000 by FTW */
```

4

ASL Pattern Matching

In this chapter, the pattern matching syntax of the Adapter Scripting Language is described.

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 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 matches. If any component of a pattern fails to match its corresponding data component, the pattern does not 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.

In Figure 6, for the first match, the pattern is compared, starting with the leftmost 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 the beginning of the data.

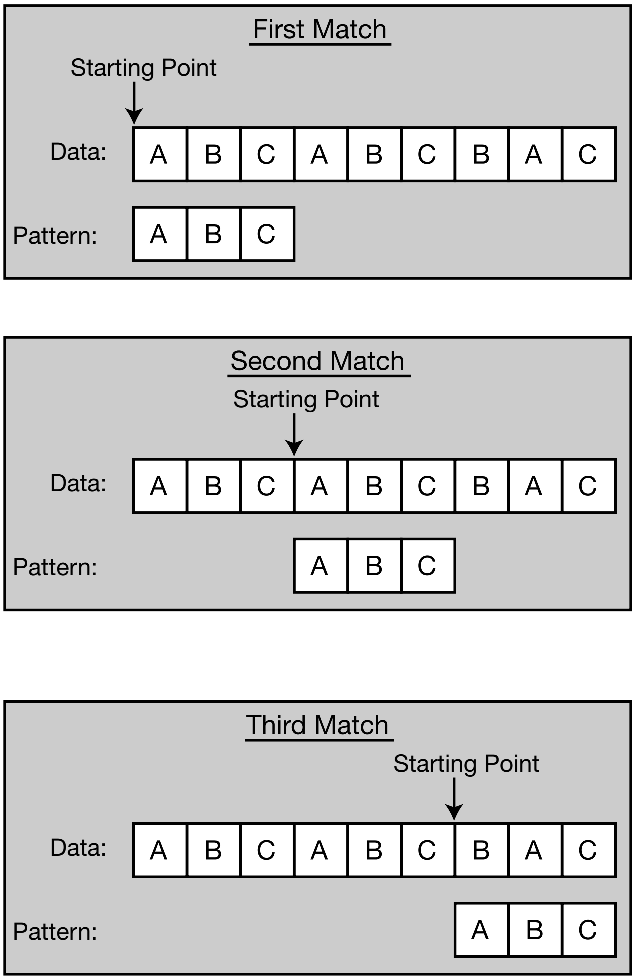


Figure 6: Progression of a Pattern Being Compared to Data

Pattern Operators

White Space Operator

Whenever two elements of a pattern are separated by a white space, ASL matches any delimiters that may occur between the data that match the two elements of the pattern. Two elements of a pattern separated by a white space matches data that may or may not include delimiters. In an ASL script, the white space operator includes spaces, tabs, or an end of line (eol) between two elements of a pattern.

Note: Do not confuse the end of line in an ASL script with an eol in input data. In an ASL script, an end of line is another white space operator. As input data, an eol is not skipped as a delimiter.

The following script matches a word followed by an integer and another word. White space separates the pattern elements 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 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. Everything inside of the braces must successful 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 data file. The last variable is assigned an integer followed by two words.

This last variable demonstrates how patterns can be grouped by using curly 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 whitespace. However, in this case, the whitespace 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 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 following pattern.

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 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 attempt 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 whitespace 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 above, 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"`. For more information on controlling pattern evaluation, see "Grouping Patterns" on page 34.

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 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, a question mark (?). The question mark operator indicates that a pattern matches regardless of whether the matching data exists or not. 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 using curly 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 only bound 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:

OPERATOR
`{}`
`:`
`whitespace . ..`
`?`
`|`

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 above matches an input of A followed by B or it matches an input of C followed by D.

Grouping patterns using the curly 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”
}
```

Note: In the example, the string is surrounded by double quotation marks. For every instance where quotation marks are used, they can be single or double quotation marks.

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:

```
a:{
}
```

This assigns the matched data to the variable a. See "Assignment Operator" on page 30 for more 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. Table 8 contains the special characters:

CHARACTER	SYNTAX	NOTES
tab	`\t`	
single quotation mark (')	`\'`	The quotation marks surrounding the string containing this code must be double quotes.
double quotation mark (")	`\"`	The quotation marks surrounding the string containing this code must be single quotes.
backward slash (\)	`\\`	
carriage return	`\r`	In most cases, use the eol pattern match instead of carriage return. See "End of Line Matches" on page 44 for more information.
line feed	`\n`	In most cases, use the eol pattern match instead of line feed. See "End of Line Matches" on page 44 for more information.

Table 8: Match Syntax for Special Characters

Note: Pattern matches are by default case-sensitive. To override the default, see "Making Patterns Case Sensitive/Insensitive" on page 56.

Any Character Matches

The `any` function returns a match if the character 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 being evaluated does not match any character in 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 matches 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.

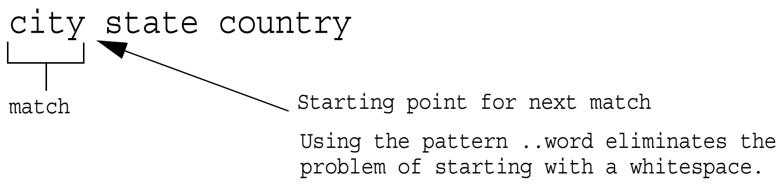
```
ASL Script (char_match.asl):
START {
    a:char
}
do {
    print("Matched with ".a);
}
DEFAULT {
    ..eol
}
do {
    print("Failed match");
}
Input (char_match.txt):
a <tab> b c

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. See "Customizing the Delimiter" on page 55 for more information.

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
$
```


city state country
match
Starting point for next match
Using the pattern ..word eliminates the problem of starting with a whitespace.

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 and 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 eol. To pattern match a field separator, use `fs`. (See "Field-Separator Translation" on page 117 for more information.)

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. The name and address 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 `--field-separator` option. 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 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.

Table 9 shows some sample uses of the `rep` function. Each sample uses the letter P to denote a pattern.

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..P ...	The double dot operator ignores all patterns except for P.

Table 9: Samples of the `rep` function

The following 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 a 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. They are:

- The `tab` function
- The `len` function

The tab Function

Using the `tab` function, it is possible to skip characters of input data. The syntax is:

```
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 data stream. 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.

Using the assignment operator (:) assigns all of the characters skipped to the variable or the current position if no characters are specified. The syntax is:

```
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 right to the last name of each person in the list. In the input data file, 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
$
```

The 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:

```
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, using the assignment operator assigns all characters skipped to the variable. The `len` function does not skip over markers.

In Figure 7, because of the whitespace 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. See "Customizing the Delimiter" on page 55 for more information.

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

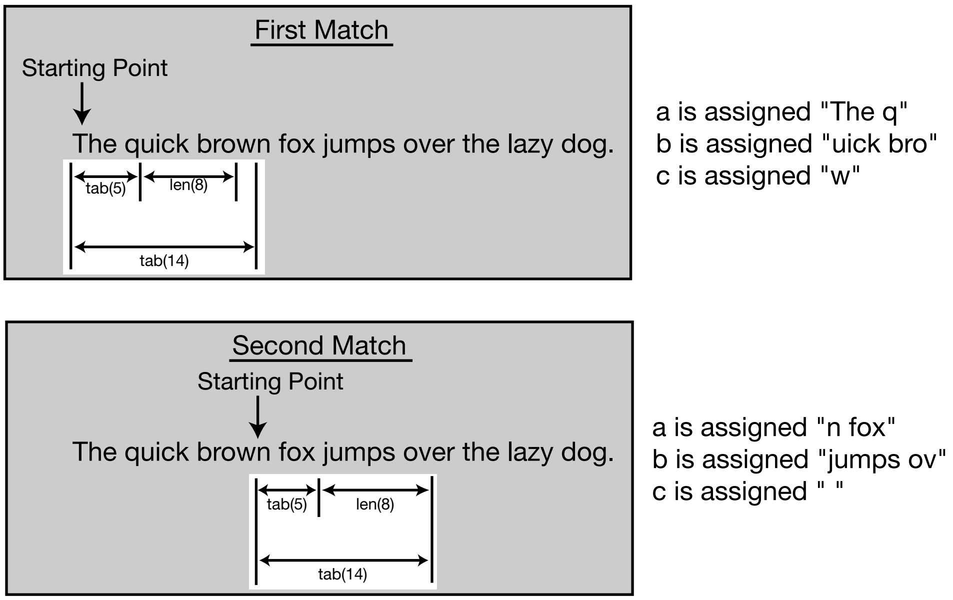


Figure 7: Example of Positional Matching

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. Using 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. The DEFAULT rule executes.

```
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 lines are 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. See "Special Rules" on page 7 for more information about this rule. For an example of the EOF rule, see the example script in "Switching the Input of a Pattern" on page 56.

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`. 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 is specified in a rule, ASL overrides any previous delimiter definitions.

Use the following syntax to define new delimiters:

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

The string defines the set of characters to use as delimiters. 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 carriage returns as delimiters, use the special syntax within the quotation marks as described in Table 8, “Match Syntax for Special Characters,” on page 37.

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 alternate to using the delim variable is to use a combination of the `rep` function and the `notany` function.

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

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

Making Patterns Case Sensitive/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 using the input variable. The input variable is assigned to a value which ASL uses for the pattern matching following 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:

```
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
$
```

Other ASL Functions That Work 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 semi-colon with functions.

5

ASL Filters

This chapter describes the uses of filter blocks in an ASL script and the basic syntax.

Filters

A filter determines whether the action block of a rule executes. When a filter fails, the following 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 Table 10. 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.

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

Table 10: Filter Operators

In the following script, 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
}
```

6

ASL Actions

This chapter describes the use of action blocks in an ASL script. This chapter contains information about the basic syntax used in action blocks. See "Interfacing With an InCharge Domain Manager" on page 89 to learn about using ASL to view and manipulate objects stored in an InCharge Domain Manager.

Actions

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 has the form:

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

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

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

Operators

In the action block, statements are terminated with a semicolon (;). The semicolon does not have to immediately follow a statement, but it must occur before the next statement.

All other operators used in the action block are described in "Operators and Expressions" on page 22.

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 (listortablename) {statements}
```

When the foreach statement is used with a loop, 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 Statements

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

```
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 Statements

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

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

If the conditional test is true, the statements following it are executed. If the conditional test is false, the statements following the else are executed. 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 to see 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 to see 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

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

This example is similar to the if statement example (see "If Else Statements" on page 67) except that a break statement has been added. 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

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 non-positive number, 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

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:

```
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 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:

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, if the first character of [set] is (^), any single character that is *not* in the set. 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.
|	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.

Table 11: Symbols for a Glob Pattern

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:

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

is equivalent to

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

Both of these expressions match any string not containing any digits.

A trailing instance of an ampersand character (&) can only match 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 adapter. No arguments are passed to the function. When the `stop` function is encountered, the ASL script immediately ceases execution. A call to the `stop` function does not return. The syntax is:

```
stop()
```

Quit Function

The `quit` function stops the backend, or the InCharge 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.

The syntax to stop a remote adapter is:

```
quit();
```

The syntax to stop the current adapter is:

```
self->quit();
```

Defined Function

The `defined` function returns TRUE if a variable has a value.

This example is a refinement of the if example (see "If Else Statements" on page 67). By using the `defined` function, several lines of the script are removed, including a foreach statement and an if statement.

The single if statement checks to see 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
$
```

Undefine Function

The `undef` function undefines 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:

```
sizeof(<value>)
```

or

```
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

This function returns a string. The returned string is a piece of the string passed to the function. The syntax is:

```
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
$
```

Lowercase Function

This function returns a string, converting any uppercase letters in the original string to lowercase. The syntax is:

```
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
$
```

Uppercase Function

This function returns a string, converting any lowercase letters in the original string to uppercase. The syntax is: 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 have truly found paradise
New York City
Output:
$ sm_adapter --file=toUpper_do.txt toUpper_do.asl
IHTFP
NYC
$
```

Print Function

The `print` function sends strings passed to it to an output location. The print destination is standard out (`stdout`). The syntax is:

```
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 Table 12.

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`	

Table 12: `Print` **Function Special Characters**

Sleep Function

This 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 Jan 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 which represents the number of seconds since Jan 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 numeric function

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

returns a number like,

```
958159632
```

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
$
```

Converting Hexadecimal to a String

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:

```
hexToString(value)
```

getRuleFileName Function

The `getRuleFileName` function returns the file name of the currently-executing rule file. The syntax is:

```
getRuleFileName([fullname])
```

The fullname argument is optional. If it is omitted or passed as FALSE, only the base name of the rule file name 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 using the return statement. (See"Returning Values" on page 82 for information on the return statement.) The syntax is:

```
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 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 datatype supported by ASL.

Functions optionally return a value, which are also of any ASL datatype. The return value may be a constant, or a variable declared within the scope of the function do-block. Refer to the section 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 that 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 ...
}
```

Note: A function may only be defined 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:

```
<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:

LOG	The exception is reported to the standard error file and system error log.
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 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" on page 34).

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
$
```

7

Interfacing With an InCharge Domain Manager

ASL and the MODEL Language

In order to create adapters that interact with an InCharge Domain Manager, it is necessary to understand how the domain manager is configured. The adapter creates, deletes, and interacts with instances of objects defined 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 these failures cause, and the effect of failures. Symptoms can be local, observed in the instance of the class, or propagated, 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. See Figure 8.

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.

For more information about MODEL, see the *InCharge Managed Object Definition Language Reference Guide*.

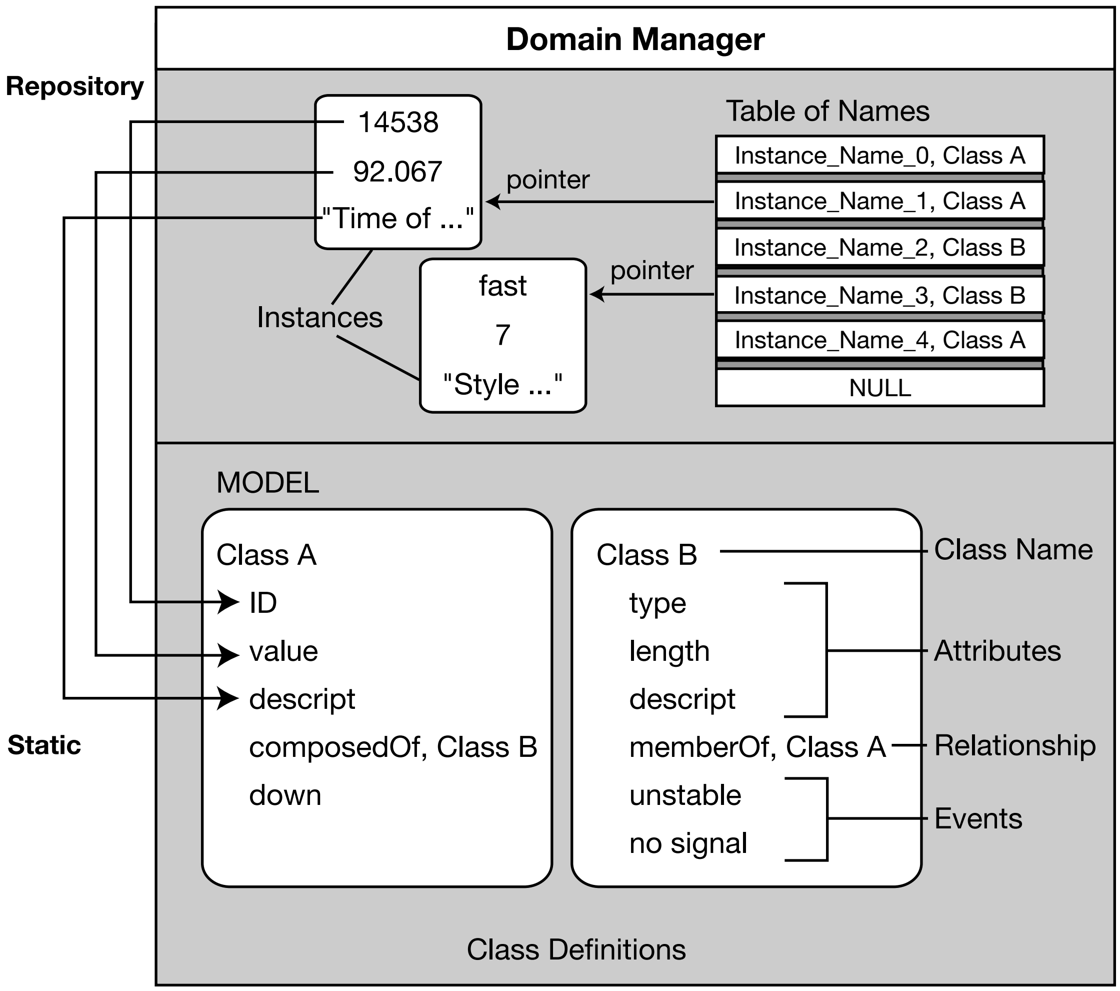


Figure 8: Domain Manager With a Model and Repository

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.

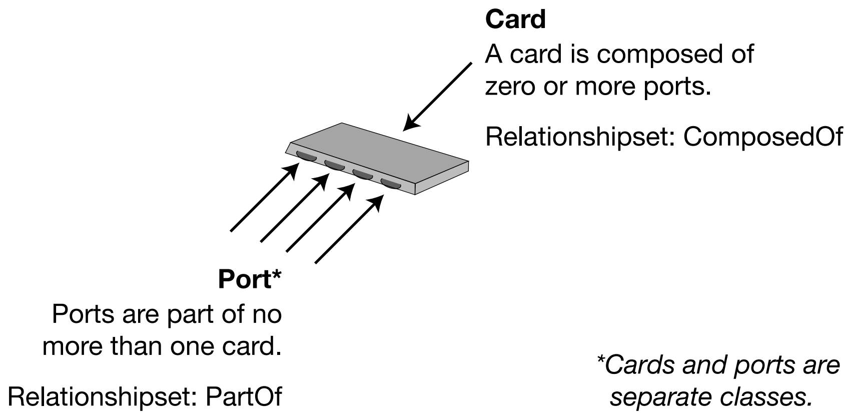

Card
A card is composed of zero or more ports.
Relationshipset: ComposedOf
Port*
Ports are part of no more than one card.
Relationshipset: PartOf
*Cards and ports are separate classes.

Figure 9: 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 the other class.

Cards have:

- 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 of event 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 also have

- 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 Figure 10, 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.

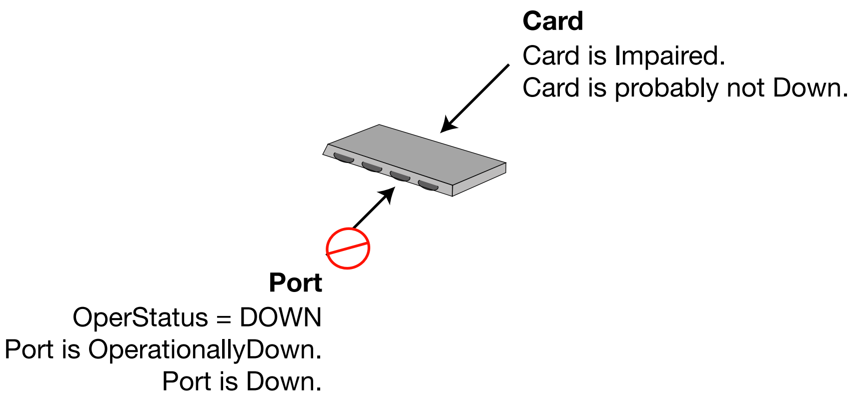


Figure 10: Events That Affect Ports and Their Related Card

Refer to *Card-Port MODEL Code* on page 145 to see the MODEL language code for this example. Figure 11 is a diagram of this card and port model.

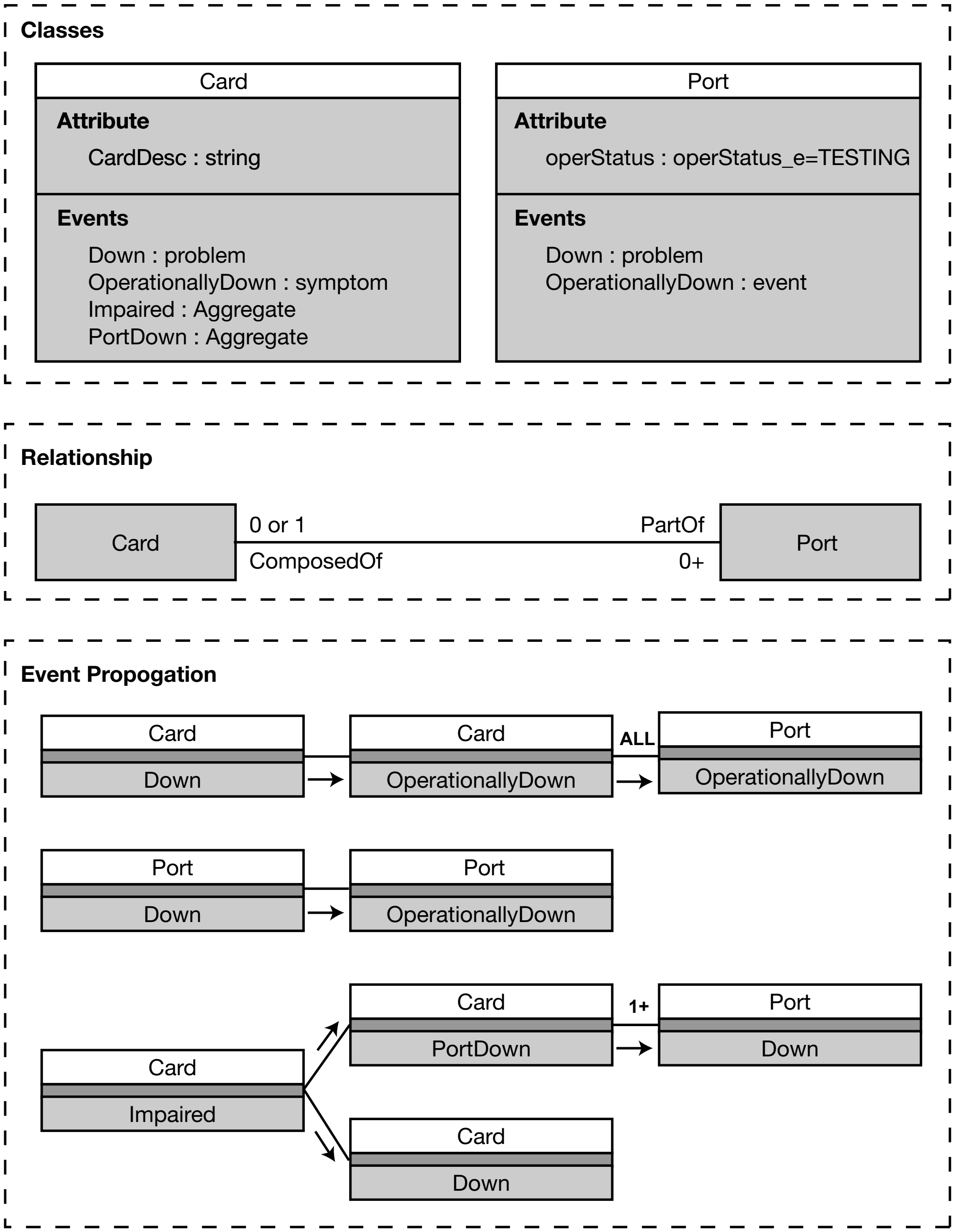


Figure 11: Diagram of the Card and Port Model

Objects and Instances

Creating Objects

The `create` function creates an instance of a class or an instance with an object handle. The syntax is:

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

or

```
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 loads instances of ports and cards. The name of InCharge Domain Manager containing 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

This `getInstances` function returns a list of object names for the given class, but does not return object handles. The syntax is:

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

The following script 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:

```
objRef = object([classname,] <objectname>);
```

or

```
objRef = object(<objhandle>);
```

For the first syntax, since the name of an instance must be unique, the classname is optional.

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.

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`. (See "Listing Instances" on page 96 for 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 using the following syntax:

```
objRef->property = value;
```

The objRef is the object handle for the object. You assign the object handle using the `create` or `object` functions or by through a relationship.

The following script loads the names of all instances of the classes “Card” and “Port” into an InCharge Domain Manager. The data file 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 and 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 `objRef->property` can also be assigned to an ASL variable.

```
variable = objRef->property;
```

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

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

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:

```
objRef->delete();
```

Note: The `undefine` function does not affect objects. Using it, removes an assigned value from a variable.

Note: 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. See "Modifying Relationshipsets" on page 103.

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 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:

```
objRef->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:

```
is(objRef->Relationship,objRef2)
```

The relationship used in the `is` function must be a valid relationship for the object or an error occurs. If objRef2 is related to objRef 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 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
$
```

Note: In MODEL, all classes have a built-in property named CreationClassName. This property contains the class name.

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 using 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.

Table 13: List Operators

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]);
```

Tables

AccessingTables 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:

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

Structures

Updating and Accessing Structure Attributes in MODEL from ASL

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 above example is,

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

Enumerated Data Types

Accessing Enumerated Data Types in the MODEL

Enumerated data types are represented as strings in ASL.

To update an enumerated attribute with an enum value of TESTING, use the following syntax.

```
x->enum_variable = "TESTING";
```

Type Conversions Between ASL and MODEL

MODEL cannot convert data from one type to another. The data types available in MODEL are not the same as the data types available in ASL. Table 14 shows how data types in MODEL and ASL correspond.

MODEL VALUE	ASL VALUE
Numeric (Any type)	Numeric (double)
Table	List
Relationship	Object handle
Relationshipset	List of object handles
Boolean	Boolean
String	String
External (User-defined) Type	String
Structure	List
Enum	String

Table 14: Data Types for MODEL and ASL

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

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 InCharge Domain Manager. The syntax is:

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

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining 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 InCharge Domain Manager. The syntax is:

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

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining 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:

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

getEventDescription Function

The `getEventDescription` function returns a string, defined in MODEL, that describes an event. The syntax is:

```
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:

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

getExplainedBy Function

The `getExplainedBy` function is the inverse of the `getExplains` function: It returns those problems which the MODEL developer has listed as explaining this problem. The syntax is:

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

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining 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> }
```

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:

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

The oneHop parameter is optional. If it is omitted or passed as FALSE, the full list of problems explaining 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.

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 an InCharge 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 either commit or abort a transaction:

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

or

```
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 the following:

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.

Table 15: Transaction Keyswords

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

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

getRuleFileName Function->

Returns the file name of the currently executing rule file as an argument.

Usage:

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

where <fullname> is a string containing the complete path-qualified file name.

getServerName Function->

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

thread Function->

Returns the operating system thread ID of the thread associated with adapter invoking the ASL script.

8

Running Adapters

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 the adapter follows when it receives information.

The options control which domain manager and broker the adapter connects to as well as configuring how the adapter runs. An adapter can run using a variety of front-end and back-end components. The options to use with **sm_adapter** are described throughout the rest of this chapter.

The file, *sm_adapter* (*sm_adapter.exe* for Windows), is found in ***BASEDIR***/*smarts/bin.*

Basic Options

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

OPTION	DESCRIPTION
`--broker=<location>`	Alternate Broker location as `host:port`. Also `-b <location>`.
`--model=<model>`	Name of MODEL library to load. Also `-M <model>`.
`--name=<name>`	Start a server registered under <name>. Also -n <name>.

OPTION	DESCRIPTION
`--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.

Table 16: sm_adapter Basic Options

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 an InCharge 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. For a list of options, see "sm_adapter Front-End Options" on page 124. Examples of the **sm_adapter** command with front-ends are provided in "Sample Front-End Invocations" on page 122.

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 Windows systems, `\r\n` represents an end of line. 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." In a Windows system, this file contains the characters:

```
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 (for example, 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, 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-end is 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 re-opened. 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 an SQL query issued using a command-line SQL program.

The `--program` option of the **sm_adapter** command invokes the program front-end. This command string is executed using the native command interpreter for the system. On UNIX, this is normally */bin/sh.* On Windows, this is *CMD.EXE*. 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 an InCharge 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 InCharge 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 identifying 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 `fs` markers. The first field of all notifications is an integer time stamp 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 InCharge 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 InCharge Domain Manager sends a single notification message when an event becomes active and a 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 InCharge Domain Manager clears an event, the ASL format of the 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 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 and another 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.

File Front-End

```
$ sm_adapter --file=/etc/passwd --field-separator=:
passwd.asl
```

File Tail Front-End

```
$ sm_adapter --tail=/var/adm/messages syslog.asl
```

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

Option	Description
`--file=<path>`	U 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' a subscription for topology changes is requested. If <sub> is of the form '<name>/n' then a subscription to NL <name> is requested. Note that 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; that profile specifies what subscriptions are to be requested. A profile name may optionally be followed by the /v qualifier. Multiple –subscribe options can be specified.
`--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. <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 at the before this adapter connected are not fed to the input stream.

Table 17: sm_adapter Front-End Options

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 containing the ASL parse rules is added at the end of the **sm_adapter** command:

```
sm_adapter [options] myRules.asl
```

sm_adapter Rule Set Options

OPTION	DESCRIPTION
`--D<var>=<value>`	Override the default value for a rule set variable.
`--verify`	Validate rules only.

Table 18: sm_adapter Rule Set Options

Back-End

The back-end of an adapter represents an InCharge Domain Manager. Object manipulations (for example, setting an attribute) within an ASL rule set are translated into InCharge 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 InCharge Domain Manager.
- Restartable domain manager — Also specifies a remote InCharge 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 restartable 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, if the domain manager is unavailable when the adapter starts or if the domain manager disconnects while the adapter is running, the adapter terminates. The restartable domain manager should be used in cases where you want the adapter to remain active when the domain manager is unavailable. The restartable domain manager back-end periodically attempts to reconnect to the domain manager. Once it succeeds, the connection is restored and the adapter continues to function.

The restartable back-end is most useful in conjunction with the subscriber front-end. See "Subscriber Front-End With a Restartable Domain Manager" on page 122 for more information about using the restartable domain manager with the subscriber front-end.

Behavior of the Restartable Domain Manager

The restartable 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 delivered to the rule set. This exception normally terminates the adapter. With a restartable 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 Restartable Domain Manager From the Command Line

You can use the `--rserver` option of the **sm_adapter** command to create a restartable domain manager back-end.

```
$ sm_adapter --rserver=myServer rules.asl
```

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.

Table 19: sm_adapter Back-End Options

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 a timestamp, the name of the domain manager, and a 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;
```

executed in a rule set connected to the domain manager, ServerName results in this trace output:

```
22-Apr-1998 14:26:11 ServerName:
put(MyClass,MyObject,attr,TRUE)
```

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`.

Table 20: Trace Options

Other Options

OPTION	DESCRIPTION
`--help`	Print help and exit.
`--version`	Print program version and exit.
`--daemon`	Run process as a daemon.
`--logname=<name>`	Use `name` to identify sender in the system log. Default: The program's name.
`--loglevel=<level>`	Minimum system logging level. Default: `Error`.
`--errlevel=<level>`	Minimum error printing level. Default: `Warning`.
`--tracelevel=<level>`	Minimum stack trace level. Default: `Fatal`. `level` can be one of `None`, `Emergency`, `Alert`, `Critical`, `Error`, `Warning`, `Notice`, `Informational`, or `Debug`. `Fatal` is a synonym for `Critical`.
`--output[=<file>]`	Redirect output (stdout and stderr) to ***BASEDIR****/logs/<****file****>.log*. If `file` is omitted, the `--logname` value is used.
`--accept=<host-list>`	Accept connections only from hosts on `host-list`, a comma-separated list of host names and IP addresses. You can also specify `any` instead of `host-list` to allow any host to connect. Default: `--accept=any`.

Table 21: Other sm_adapter Options

Stopping Adapters

Adapters that do not stop on their own can be stopped using SIGTERM for UNIX systems and the Task Manager for Windows.

You can create ASL scripts that include a `stop` or `quit` function. For information, see *Stop Function* on page 73 or *Quit Function* on page 74.

A

ASL Reference

This appendix provides a summary of ASL syntax and a list of reserved words.

ASL Syntax

The following table lists available ASL syntax.

SYNTAX	DESCRIPTION
!=	Not equal to.
+	Addition operator.
-	Subtraction operator.
*	Multiplication operator.
/	Division operator.
%	Modulus operator. Calculates using integer or floating point numbers.
&&	Logical AND.
.	(Pattern matching) Indicates that the next pattern must be matched immediately.
.	(Action block) Concatenates two strings.
..	Used to indicate an undefined string of characters up to the next pattern match.
||	Logical OR.

SYNTAX	DESCRIPTION
`+=`	Adds an object to a relationship.
`?`	(Pattern matching) Match one or zero times.
`?`	(Action block) Exception handling operator.
`<`	Less than.
`<=`	Less than or equal to.
`-=`	Removes an object from relationship.
`==`	Equal to.
`>`	Greater than.
`->`	Used to reference properties of an object.
`->*`	Used to reference properties of an object using a variable for the property name.
`>=`	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`	Use to break out of a loop.
`case = [exact]|[ignore]`	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>)`	Create an object.
`create(<objhandle>)`	Create 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.

SYNTAX	DESCRIPTION
`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 rule to fail when an exception occurs.
`FALSE`	Boolean false.
`feError()`	Returns a Boolean value. TRUE if the front-end has reported a failure to read data.
`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,eventname` of the root cause that causes that event.
`getChildren(<classname> [,recursive])`	Retrieve 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), return 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 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.

SYNTAX	DESCRIPTION
`getExplains(<classname>, <objectname>,<eventname> [,<oneHop>])`	Given a root cause, return the alternate closure as defined in MODEL.
`getInstances(<classname>)`	Returns a list of strings (not object handles) which are the names of the instances of that class.
`getRuleFileName([<fullname>])`	Returns the file name of the currently-executing rule file.
`getServerName()`	Returns the name of the domain manager.
`glob(<pattern>,<string>)`	Enables glob style pattern matching. Standard glob syntax. Returns a Boolean.
`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}`	Conditional statement.
`ignore`	Used in conjunction with `case` to make all string matches NOT case sensitive.
`IGNORE`	Exception handling, ignore 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.

SYNTAX	DESCRIPTION
`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 using peek.
`print(<string>)`	Prints the argument string to the screen.
`quit()`	Shuts down the InCharge process the adapter is talking to. This can be the adapter or a domain manager.
`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`	see `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.
`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 ASL script.
`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 running the adapter.

SYNTAX	DESCRIPTION
`time()`	Returns the system time.
`toLower(<string>)`	Converts string to lowercase letters
`toUpper(<string>)`	Converts string to uppercase letters
`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.

Table 22: ASL Syntax

Reserved Words

Table 23 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.

KEYWORD	KEYWORD	KEYWORD	KEYWORD
`any()`	`filter`	`input`	`rep()`
`boolean()`	`float`	`integer`	`repository`
`break`	`foreach`	`is()`	`sleep()`
`case`	`fs`	`isNull()`	`STOP`
`char`	`getCauses()`	`len()`	`stop()`
`clear()`	`getClosure()`	`list()`	`string()`
`consistencyUpdate()`	`getEventClassName()`	`local`	`substring()`
`continue`	`getEventDescription()`	`LOG`	`tab()`

KEYWORD	**KEYWORD**	**KEYWORD**	**KEYWORD**
`correlate()`	`getEventType()`	`NEXT`	`table()`
`create()`	`getExplainedBy()`	`NO_LOCK`	`thread()`
`default`	`getExplains()`	`not()`	`time()`
`defined()`	`getChildren()`	`notany()`	`toLower()`
`delete()`	`getInstances()`	`notify()`	`toUpper()`
`delim`	`getRuleFileName()`	`numeric()`	`transaction()`
`do`	`getServerName()`	`object()`	`TRUE`
`eol`	`glob()`	`peek()`	`undef()`
`exact`	`global`	`print()`	`while`
`FAIL`	`hex`	`quit()`	`word`
`FALSE`	`hexToString()`	`READ_LOCK`	`WRITE_LOCK`
`feError()`	`if`	`self`	
`feErrorMsg()`	`ignore`	`sizeOf()`	
`feErrorReset()`	`IGNORE`	`sizeof()`	

Table 23: ASL Reserved Words

B

dmctl Reference

Description

The Domain Manager Control Program or dmctl is a command-line tool for interacting with an InCharge Domain Manager (see "dmctl Syntax" on page 140). It can be used to query, modify, or receive notifications from a domain manager.

It can either execute commands typed at the command line, execute commands read from a batch file, or 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 (for example, **getE** is **getEvents**; use at least **getEventD** to **getEventDescription**); and, conversely, to prevent the accidental typing of a dangerous command (**quit** and **exit** cannot be abbreviated, and **shutdown** must appear as at least **shut**). Uppercase letters in command names are shown for clarity; 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 an InCharge Domain Manager. When you debug ASL scripts, dmctl is an effective tool to monitor the state of a domain manager.

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:

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, 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 InCharge 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>`.

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.
`--commands`	List **dmctl** commands and exit. Also `-c`.
`--help`	Print help and exit.
`--version`	Print program version and exit.

Table 24: dmctl Options

At any given time, dmctl can be attached to (at most) one domain manager. 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:

COMMAND	DESCRIPTION
`attach domain`	Attach to the specified domain manager. Once a domain manager is attached, other commands can be invoked.
`clear <class::instance::event>`	Force clear of the specified event.
`create <class>::<instance>`	Create a new instance in the repository.
`consistencyUpdate`	Re-compute the codebook.
`correlate`	Correlate now.
`delete <class>::<instance>`	Delete an instance from the repository.
`detach`	Detach from the domain manager. Another domain manager can now be attached with the **attach** command.
`execute <program> [<arg1> ...]`	Execute 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`	Exit dmctl.
`findInstances <class-pattern>::<instance-pattern>`	List instances that match given class and instance patterns.
`get <class>::<instance> [::<property>]`	List all instance properties values or a given property value.
`getClasses`	List all classes in the repository.

COMMAND	DESCRIPTION
`getEvents <class>`	List all exported events defined in given class.
`getEventDescription <class>::<event>`	Print description of given event.
`getInstances [<class>]`	List all instances in the repository, or all instances of the given class.
`getModels`	List all models loaded to the domain manager.
`getOperations <class>`	List all operations defined in given class.
`getPrograms`	List all programs loaded to the domain manager.
`getProperties <class>`	List all properties defined in given class.
`getThreads`	List all threads running in the domain manager.
`insert <class>::<instance>::<property> <value>`	Insert a value into a table or relationship.
`invoke <class>::<instance> <op> [<arg1> ...]`	Invoke an operation of given instance.
`loadModel <model>`	Load a new MODEL library. `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* (Windows)). After a MODEL library is loaded, the repository can be populated with instances of classes defined in that library.
`loadProgram <program>`	Load a new program. `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 `execute`.
`notify <class::instance::event>`	Force notification of a given event.
`ping`	Verify that the domain manager is still alive.
`put <class>::<instance>::<property> <value1> [<value2> ...]`	Set value of given property.
`quit`	Quit dmctl.
`remove <class>::<instance>::<property> <value>`	Remove a value from a table or relationship.
`restore <file>`	Restore the repository from a file. `file` should not contain a directory portion; it is read from ***BASEDIR****/repos/*.
`shutdown`	Shut down the domain manager.

COMMAND	DESCRIPTION
`save <file> [<class>]`	Save the repository to a file. `file` should not contain a directory portion; it is saved to ***BASEDIR****/repos/*. If `class` is specified, save only the sub-tree rooted at class.
`status`	Display the connection status.
`subscribe <class-regexp>::<instance-regexp>::<event-regexp> ...`	Subscribe to problems and events that match the given pattern(s). 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.
`subscribeEvents <class-regexp>::<instance-regexp>::<event-regexp> ...`	Subscribe to events that match the given pattern(s). 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> ...`	Subscribe to problems that match the given pattern(s). 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 25: dmctl Commands

C

Card-Port MODEL Code

This appendix provides the Card-Port MODEL as described in "Correlation Model Used for Example Scripts" on page 91.

```
/* card.mdl -
 *
 * Copyright (c) 2000, System Management ARTS (SMARTS)
 * 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;

   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;

}
```

Index