



APPENDIX **B**

Writing Macros for Use in Templates

Macros are sets of commands or instructions that perform specific operations on target devices. They contain proprietary VFrame subroutines, Cisco IOS software commands, and Perl subroutines and syntax.

This appendix provides detailed information about writing macros for use in templates, and includes the following sections:

- [Procedures for Writing Macros, page B-1](#)
- [Expressing Variables in Macros, page B-19](#)
- [Subroutine Reference, page B-21](#)

Procedures for Writing Macros

You can write macros that configure CSMs and FWSMs, verify CSMs and FWSMs, and perform LOM operations on servers and blade servers. This section provides step-by-step procedures for writing specific types of macros, provides information about the structure of a macro, and includes the following topics:

- [Writing an FWSM Configuration Macro, page B-2](#)
- [Writing a CSM Configuration Macro, page B-5](#)
- [Writing an FWSM Verification Macro, page B-6](#)
- [Writing a CSM Verification Macro, page B-9](#)
- [Writing an IBM Blade Server Power On Macro, page B-10](#)
- [Writing an IBM Blade Server Power Off Macro, page B-12](#)
- [Writing an FWSM Configuration Removal Macro, page B-14](#)
- [Writing a CSM Configuration Removal Macro, page B-17](#)
- [Putting It All Together, page B-18](#)

Writing an FWSM Configuration Macro

An FWSM configuration macro typically includes the following operations:

- Configuring the firewall interface
- Enabling firewall interface security permissions
- Configuring the BVI interface
- Configuring the default gateway
- Creating the outside ACL for the VIP
- Creating the outside ACL for DHCP traffic
- Creating the outside ACL for the NFS filers
- Creating the inside ACL for the NFS filers
- Creating the inside ACL for VFrame traffic
- Configuring the inside ACL to allow pings from the CSM
- Applying the ACLs to the interface
- Configuring ports for communication

The following procedure provides the steps for writing an FWSM configuration macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Initialize any arrays and declare any variables you use.

Use the following code as a reference:

```
my @verification_config_exceptions = (".*");
my $config_from_verify=0;
my @exceptions = ("INFO: Security level for .*", "INFO: converting.*", "WARNING:.*");
```

Step 2 Write a subroutine that configures the firewall interfaces.

Use the following code as a reference:

```
config("
    interface vlan $FWOutsideInterfaceGroup->{Vlan} bridge-group 1 nameif
    $FWOutsideInterfaceGroup->{Name} security-level
    $FWOutsideInterfaceGroup->{SecurityLevel} no shutdown
    interface vlan $FWOutsideInterfaceGroup->{Vlan} bridge-group 1 nameif
    $FWInsideInterfaceGroup->{Name} security-level
    $FWOutsideInterfaceGroup->{SecurityLevel} no shutdown
);
```



Note For more information, see the [config\(\)](#) subroutine.

Step 3 Write a subroutine that enables firewall interface security permissions.

Use the following code as a reference:

```
config("
  if ($FWOutsideInterfaceGroup->{SecurityLevel} eq
    $FWInsideInterfaceGroup->{SecurityLevel}) {
    config("
      same-security-traffic permit inter-interface
    ");
  }
");
```

Step 4 Write a subroutine that configures the BVI interface.

Use the following code as a reference:

```
config("
  interface BVI 1 ip address ${\&get_ip($BVIInterfacePrimIpAddrGroup->{IPAddress},
    INFO)}
  ${\&get_mask($BVIInterfacePrimIpAddrGroup->{IPAddress})} standby
  ${\&get_ip($BVIInterfaceSecondaryIpAddrGroup->{IPAddress}, INFO)}
");
```



Note For more information, see the [get_ip\(\)](#) and [get_mask\(\)](#) subroutines.

Step 5 Write a subroutine that configures the default gateway.

Use the following code as a reference:

```
config("
  route $FWOutsideInterfaceGroup->{Name} 0.0.0.0 0.0.0.0
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)}
");
```

Step 6 Write a subroutine that creates the outside ACL for the VIP.

Use the following code as a reference:

```
config("
  access-list OUTSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)}
  access-list OUTSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)} eq www
");
```

Step 7 Write a subroutine that creates the outside ACL for DHCP traffic.

Use the following code as a reference:

```
config("
  access-list OUTSIDE_VCC_ACL extended permit udp any eq bootps any eq bootpc
");
```

Step 8 Write a subroutine that creates the outside ACL for the NFS filers.

Use the following code as a reference:

```
config("
  access-list OUTSIDE_VCC_ACL extended permit icmp host
  ${\&get_ip($NFSSfiler->{IPAddress}, INFO)} any echo-reply
");
```

Step 9 Write a subroutine that creates the inside ACL for the NFS filers.

Use the following code as a reference:

```
config("
  access-list INSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
  access-list INSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
  access-list INSIDE_VCC_ACL extended permit udp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
");
```

Step 10 Write a subroutine that creates the inside ACL for VFrame traffic.

Use the following code as a reference:

```
config("
  access-list INSIDE_VCC_ACL extended permit udp any eq bootpc any eq bootps access-list
  INSIDE_VCC_ACL extended permit tcp any host ${\&get_ip($VCC->{IPAddress}, INFO)}
  eq 3000
  access-list INSIDE_VCC_ACL extended permit tcp any host ${\&get_ip($VCC->{IPAddress},
  INFO)} eq 3010
  access-list INSIDE_VCC_ACL extended permit udp any host ${\&get_ip($VCC->{IPAddress},
  INFO)} eq tftp
");
```

Step 11 Write a subroutine that configures the inside ACL to allow pings from the CSM.

Use the following code as a reference:

```
config("
  access-list INSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)}
  access-list OUTSIDE_VCC_ACL extended permit icmp host
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)} any echo-reply
");
```

Step 12 Write a subroutine that applies the ACLs to the interfaces.

Use the following code as a reference:

```
config("
  access-group INSIDE_VCC_ACL in interface $FWInsideInterfaceGroup->{Name}
  access-group OUTSIDE_VCC_ACL in interface $FWOutsideInterfaceGroup->{Name}
");
```

Step 13 Write a subroutine that configures various ports for communication.

Use the following code as a reference:

```
config("
  fixup protocol dns maximum-length 512
  fixup protocol ftp 21
  fixup protocol h323 H225 1720
  fixup protocol h323 ras 1718-1719
  fixup protocol http 80
  fixup protocol rsh 514
  fixup protocol sip 5060
  fixup protocol sip udp 5060
  fixup protocol skinny 2000
  fixup protocol smtp 25
  fixup protocol sqlnet 1521
");
```

Writing a CSM Configuration Macro

A CSM configuration macro typically includes the following operations:

- Configuring load balancing on interfaces
- Creating server farms
- Creating the probe
- Creating the virtual server

The following procedure provides the steps for writing an CSM configuration macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Write a subroutine that configures load balancing on interfaces.

Use the following code as a reference:

```
config("
    vlan $ClientInterfaceVariableType->{Vlan}
    $ClientInterfaceVariableType->{InterfaceType}
    ip address ${\&get_ip($LBPrimaryIpAddrGroup->{IPAddress}, INFO)}
    ${\&get_mask($LBPrimaryIpAddrGroup->{IPAddress})}
    alt ${\&get_ip($LBSecondaryIpAddrGroup->{IPAddress}, INFO)}
    ${\&get_mask($LBSecondaryIpAddrGroup->{IPAddress})}
    alias ${\&get_ip($ClientVlanAliasIpAddress->{IPAddress}, INFO)}
    ${\&get_mask($ClientVlanAliasIpAddress->{IPAddress})}
    gateway ${\&get_ip($Gateway->{IPAddress}, INFO)}
    vlan $ServerInterfaceVariableType->{Vlan}
    $ServerInterfaceVariableType->{InterfaceType}
    ip address ${\&get_ip($LBPrimaryIpAddrGroup->{IPAddress},
    INFO)}${\&get_mask($LBPrimaryIpAddrGroup->{IPAddress})}
    alt ${\&get_ip($LBSecondaryIpAddrGroup->{IPAddress}, INFO)}
    ${\&get_mask($LBSecondaryIpAddrGroup->{IPAddress})}
");
```



Note For more information, see the [config\(\)](#), [get_ip\(\)](#), and [get_mask\(\)](#) subroutines.

Step 2 Write a subroutine that creates server farms.

Use the following code as a reference:

```
config("
    serverfarm $ServerFarm->{Name}
    predictor $ServerFarm->{Predictor}
    probe PING
");
```

Step 3 Write a subroutine that creates the probe.

Use the following code as a reference:

```
config("
  probe PING icmp
  interval 5
  retries 2
  failed 5
  receive 2
");
```

Step 4 Write a subroutine that creates the virtual server.

Use the following code as a reference:

```
config("
  vserver $Vserver->{Name}
  virtual ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)} $Vserver->{Protocol}
  vlan $ClientInterfaceVariableType->{Vlan}
  serverfarm $ServerFarm->{Name}
  inservice
");
```

Writing an FWSM Verification Macro

An FWSM verification macro typically includes the following operations:

- Retrieving the current configuration
- Verifying the existence of the firewall interfaces
- Verifying the existence of the BVI interface
- Verifying the existence of the outside ACL for DHCP traffic
- Verifying the existence of the outside ACL for the load balancing VIP
- Verifying the existence of the outside ACL for the NFS filers
- Verifying the existence of outside ACL ping permissions
- Verifying the existence of the inside ACL for VFrame traffic
- Verifying the existence of the inside ACL for the NFS filers
- Verifying the existence of inside ACL ping permissions
- Verifying that the ACLs are applied to the interfaces
- Verifying the existence of a route to the gateway

The following procedure provides the steps for writing an FWSM verification macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Declare any variables you use.

Use the following code as a reference:

```
use vars qw'$autofix';
use vars qw'$config_from_verify';
```

Step 2 Write a subroutine that retrieves the current configuration.

Use the following code as a reference:

```
reset_parser({error_def=>WARNING});
cmd_config ("show running-config", ': end'.set_prompt_to_cisco_ios(),
{time_out=>60, parse_flag=>1});
```



Note For more information, see the [reset_parser\(\)](#), [cmd_config\(\)](#), and [set_prompt_to_cisco_ios\(\)](#) subroutines.

Step 3 Write a subroutine that verifies the existence of the firewall interfaces.

Use the following code as a reference:

```
find_and_parse("interface Vlan$FWInsideInterfaceGroup->{Vlan}", {begin=>0});
find_and_parse("nameif $FWInsideInterfaceGroup->{Name}", {current_only=>1});
find_and_parse("bridge-group 1", {current_only=>1});
find_and_parse("security-level $FWInsideInterfaceGroup->{SecurityLevel}",
{current_only=>1});
find_and_parse("interface Vlan$FWOutsideInterfaceGroup->{Vlan}", {begin=>0});
find_and_parse("nameif $FWOutsideInterfaceGroup->{Name}", {current_only=>1});
find_and_parse("bridge-group 1", {current_only=>1});
find_and_parse("security-level $FWOutsideInterfaceGroup->{SecurityLevel}",
{current_only=>1,end_of_block=>'!'});
```



Note For more information, see the [find_and_parse\(\)](#) subroutine.

Step 4 Write a subroutine that verifies the existence of the BVI interface.

Use the following code as a reference:

```
find_and_parse("interface BVI1", {begin=>0});
find_and_parse(" ip address
${\&get_ip($BVIInterfacePrimIpAddrGroup->{IPAddress}, INFO)}
${\&get_mask($BVIInterfacePrimIpAddrGroup->{IPAddress})} standby
${\&get_ip($BVIInterfaceSecondaryIpAddrGroup->{IPAddress}, INFO)}", {current_only=>1});
```



Note For more information, see the [find_and_parse\(\)](#), [get_ip\(\)](#), and [get_mask\(\)](#) subroutines.

Step 5 Write a subroutine that verifies the existence of the ACL for DHCP traffic.

Use the following code as a reference:

```
find_and_parse("access-list OUTSIDE_VCC_ACL extended permit udp any eq bootps
any eq bootpc", {begin=>0});
```

Step 6 Write a subroutine that verifies the existence of the outside ACL for the load balancing VIP.

Use the following code as a reference:

```
find_and_parse("access-list OUTSIDE_VCC_ACL extended permit icmp any host
${\&get_ip($VipAddrGroup->{IPAddress}, INFO)}", {begin=>0});
find_and_parse("access-list OUTSIDE_VCC_ACL extended permit tcp any host
${\&get_ip($VipAddrGroup->{IPAddress}, INFO)} eq www", {begin=>0});
```

Step 7 Write a subroutine that verifies the existence of the outside ACL for the NFS filers.

Use the following code as a reference:

```
find_and_parse("access-list INSIDE_VCC_ACL extended permit icmp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit tcp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit udp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
```

Step 8 Write a subroutine that verifies the existence of ping permissions for the outside ACL.

Use the following code as a reference:

```
find_and_parse("access-list OUTSIDE_VCC_ACL extended permit icmp host
${\&get_ip($DefaultGateway->{IPAddress}, INFO)} any echo-reply", {begin=>0});
```

Step 9 Write a subroutine that verifies the existence of the inside ACL for VFrame traffic.

Use the following code as a reference:

```
find_and_parse("access-list INSIDE_VCC_ACL extended permit udp any eq bootpc
any eq bootps", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit tcp any host
${\&get_ip($VCC->{IPAddress}, INFO)} eq 3000", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit tcp any host
${\&get_ip($VCC->{IPAddress}, INFO)} eq 3010", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit udp any host
${\&get_ip($VCC->{IPAddress}, INFO)} eq tftp", {begin=>0});
```

Step 10 Write a subroutine that verifies the existence of the inside ACL for the NFS filers.

Use the following code as a reference:

```
find_and_parse("access-list INSIDE_VCC_ACL extended permit icmp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit tcp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
find_and_parse("access-list INSIDE_VCC_ACL extended permit udp any host
${\&get_ip($NFSFiler->{IPAddress}, INFO)}", {begin=>0});
```

Step 11 Write a subroutine that verifies the existence of ping permissions for the inside ACL.

Use the following code as a reference:

```
find_and_parse("access-list INSIDE_VCC_ACL extended permit icmp any host
${\&get_ip($DefaultGateway->{IPAddress}, INFO)}", {begin=>0});
```

Step 12 Write a subroutine that verifies that the ACLs are applied to the interfaces.

Use the following code as a reference:

```
find_and_parse("access-group OUTSIDE_VCC_ACL in interface
$FWOutsideInterfaceGroup->{Name}", {begin=>0});
find_and_parse("access-group INSIDE_VCC_ACL in interface $FWInsideInterfaceGroup->{Name}",
{begin=>0});
```

Step 13 Write a subroutine that verifies the existence of a route to the gateway.

Use the following code as a reference:

```
find_and_parse("route $FWOutsideInterfaceGroup->{Name} 0.0.0.0 0.0.0.0
${\&get_ip($DefaultGateway->{IPAddress}, INFO)}");
```

Writing a CSM Verification Macro

A CSM verification macro typically includes the following operations:

- Retrieving the current configuration
- Verifying the existence of any load balancing interfaces
- Verifying the existence of the probe
- Verifying the existence of any server farms
- Verifying the existence of any virtual servers

The following procedure provides the steps for writing a CSM verification macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Declare any variables you use.

Use the following code as a reference:

```
use vars qw'$CSM_Module';
use vars qw'$autofix';
```

Step 2 Write a subroutine that retrieves the current configuration.

Use the following code as a reference:

```
reset_parser({error_def=>WARNING});
cmd_config ("show running-config module $CSM_Module", {parse_flag=>1});
find_and_parse("module ContentSwitchingModule $CSM_Module");
```



Note For more information, see the [reset_parser\(\)](#), [cmd_config\(\)](#), and [find_and_parse\(\)](#) subroutines.

Step 3 Write a subroutine that verifies the existence of any load balancing interfaces.

Use the following code as a reference:

```
find_and_parse("vlan $ClientInterfaceVariableType->{Vlan}
$ClientInterfaceVariableType->{InterfaceType}");
${\&get_mask($LBPrimaryIpAddrGroup->{IPAddress})} alt
${\&get_ip($LBSecondaryIpAddrGroup->{IPAddress}, INFO)}
${\&get_mask($LBSecondaryIpAddrGroup->{IPAddress})}", {current_only=>1});
my $primary = "ip address ${\&get_ip($LBPrimaryIpAddrGroup->{IPAddress}, INFO)}
${\&get_mask($LBPrimaryIpAddrGroup->{IPAddress})}"
." alt ${\&get_ip($LBSecondaryIpAddrGroup->{IPAddress}, INFO)}
${\&get_mask($LBSecondaryIpAddrGroup->{IPAddress})}";
my $secondary = "ip address ${\&get_ip($LBSecondaryIpAddrGroup->{IPAddress}, INFO)}
```

```

${\&get_mask($LBSecondaryIpAddrGroup->{IPAddress})}"
." alt ${\&get_ip($LBPrimaryIpAddrGroup->{IPAddress}, INFO)}
${\&get_mask($LBPrimaryIpAddrGroup->{IPAddress})}";
my $primary_or_secondary = '('.$primary.'|'.$secondary.')';
find_and_parse($primary_or_secondary, {current_only=>1});
find_and_parse("gateway ${\&get_ip($Gateway->{IPAddress}, INFO)}", {current_only=>1});
find_and_parse("alias ${\&get_ip($ClientVlanAliasIpAddress->{IPAddress}, INFO)}
${\&get_mask($ClientVlanAliasIpAddress->{IPAddress})}", {current_only=>1});
find_and_parse("vlan $ServerInterfaceVariableType->{Vlan}
$ServerInterfaceVariableType->{InterfaceType}");
find_and_parse($primary_or_secondary, {current_only=>1});

```



Note For more information, see the [find_and_parse\(\)](#), [get_ip\(\)](#), and [get_mask\(\)](#) subroutines.

Step 4 Write a subroutine that verifies the existence of the probe.

Use the following code as a reference:

```

find_and_parse("probe PING icmp");
find_and_parse("interval 5", {current_only=>1});
find_and_parse("retries 2", {current_only=>1});
find_and_parse("failed 5", {current_only=>1});
find_and_parse("receive 2", {current_only=>1});
find_and_parse("!", {current_only=>1});

```

Step 5 Write a subroutine that verifies the existence of any server farms.

Use the following code as a reference:

```

find_and_parse("serverfarm $ServerFarm->{Name}");
find_and_parse("nat server", {current_only=>1});
find_and_parse("no nat client", {current_only=>1});
find_and_parse("probe PING", {end_of_block=>"!"});

```

Step 6 Write a subroutine that verifies the existence of any virtual servers.

Use the following code as a reference:

```

find_and_parse("vserver $Vserver->{Name}");
find_and_parse("virtual ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)}
$Vserver->{Protocol}", {current_only=>1});
find_and_parse("vlan $ClientInterfaceVariableType->{Vlan}", {current_only=>1});
find_and_parse("serverfarm $ServerFarm->{Name}", {current_only=>1});
find_and_parse("inservice");
find_and_parse("!", {current_only=>1});

```

Writing an IBM Blade Server Power On Macro

An IBM Blade Server power on macro typically includes the following operations:

- Creating a connection to the blade server
- Creating a password
- Retrieving the current power state
- Verifying the existence of the blade server
- Powering on the blade server
- Creating a waiting period for boot to complete

- Retrieving the current power state
- Verifying that the blade server is powered on

The following procedure provides the steps for writing an IBM blade server power on macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Declare any variables you use.

Use the following code as a reference:

```
use vars qw'$LOM_User';
use vars qw'$LOM_Password';
my $src;
```

Step 2 Write a subroutine that creates a connection to the blade server.

Use the following code as a reference:

```
cmd_config ("ssh $lom_connection_parameters->{IPAddress} -x -o
'StrictHostKeyChecking no' -l $LOM_User", "[Pp]assword:", {timeout=>50, parse_flag=>1});
```



Note For more information, see the [cmd_config\(\)](#) subroutine.

Step 3 Write a subroutine that creates a password.

Use the following code as a reference:

```
cmd_config (" $LOM_Password", {parse_flag=>1});
```

Step 4 Write a subroutine that retrieves the current power state.

Use the following code as a reference:

```
cmd_config ("power -state -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 5 Write a subroutine that verifies the existence of the blade server.

Use the following code as a reference:

```
$src = find_and_parse("empty", {error=>NONE});
if ($src ne -1) {
    cmd_config("exit\r\n", {parse_flag=>1});
    error_report(ERROR, "Blade is not present in chassis");
    return 0;
}
```



Note For more information, see the [find_and_parse\(\)](#), [cmd_config\(\)](#), and [error_report\(\)](#) subroutines.

Step 6 Write a subroutine that powers on the blade server.

Use the following code as a reference:

```
cmd_config("power -on -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 7 Write a subroutine that creates a waiting period for boot process to complete.

Use the following code as a reference:

```
for (my $i = 0; $i < 5; $i++) {
    wait_sec(2);
}
```

Step 8 Write a subroutine that retrieves the current power state.

Use the following code as a reference:

```
cmd_config ("power -state -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 9 Write a subroutine to determine if the blade server is on.

Use the following code as a reference:

```
$rc = find_and_parse("On", {error=>NONE});
if ($rc ne -1) {
    cmd_config("exit\r\n", {parse_flag=>1});
    return 0;
}
```

Writing an IBM Blade Server Power Off Macro

An IBM Blade Server power off macro typically includes the following operations:

- Creating a connection to the blade server
- Creating a password
- Retrieving the current power state
- Verifying the existence of the blade server
- Powering off the blade server
- Creating a waiting period for power off to complete
- Retrieving the current power state
- Verifying that the blade server is powered off

The following procedure provides the steps for writing an IBM Blade Server power off macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Declare any variables you use.

Use the following code as a reference:

```
use vars qw'$LOM_User';
use vars qw'$LOM_Password';
my $rc;
```

Step 2 Write a subroutine that creates a connection to the blade server.

Use the following code as a reference:

```
cmd_config ("ssh $lom_connection_parameters->{IPAddress} -x -o
'StrictHostKeyChecking no' -l $LOM_User", "[Pp]assword:", {timeout=>50, parse_flag=>1});
```



Note For more information, see the [cmd_config\(\)](#) subroutine.

Step 3 Write a subroutine that creates a password.

Use the following code as a reference:

```
cmd_config (" $LOM_Password", {parse_flag=>1});
```

Step 4 Write a subroutine that retrieves the current power state.

Use the following code as a reference:

```
cmd_config ("power -state -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 5 Write a subroutine that verifies the existence of the blade server.

Use the following code as a reference:

```
$rc = find_and_parse("empty", {error=>NONE});
if ($rc ne -1) {
    cmd_config("exit\r\n", {parse_flag=>1});
    error_report(ERROR, "Blade is not present in chassis");
    return 0;
}
```



Note For more information, see the [find_and_parse\(\)](#), [cmd_config\(\)](#), and [error_report\(\)](#) subroutines.

Step 6 Write a subroutine that powers off the blade server.

Use the following code as a reference:

```
cmd_config("power -off -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 7 Write a subroutine that creates a waiting period for power off process to complete.

Use the following code as a reference:

```
for (my $i = 0; $i < 5; $i++) {
    wait_sec(2);
}
```

Step 8 Write a subroutine to retrieve the current power state.

Use the following code as a reference:

```
cmd_config ("power -state -T $lom_connection_parameters->{LomUserAttribute1}\r\n",
{parse_flag=>1});
```

Step 9 Write a subroutine to determine if the blade server is on.

Use the following code as a reference:

```
$rc = find_and_parse("On", {error=>NONE});
if ($rc ne -1) {
    cmd_config("exit\r\n", {parse_flag=>1});
    return 0;
}
```

Writing an FWSM Configuration Removal Macro

An FWSM configuration removal macro typically includes the following operations:

- Disabling the application of ACLs on interfaces
- Disabling the outside ACL for the VIP
- Disabling the outside ACL for DHCP traffic
- Disabling the outside ACL for the NFS filers
- Disabling the inside ACL for the NFS filers
- Disabling the inside ACL for VFrame traffic
- Disabling ping permissions for the inside and outside ACLs
- Removing port communication configurations
- Removing the gateway route configuration
- Removing the firewall interface configurations
- Remove the BVI interface configuration
- Disabling the ACL for ping and HTTP traffic to the VIP

The following procedure provides the steps for writing an FWSM configuration removal macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

Step 1 Initialize any arrays you use.

Use the following code as a reference:

```
my @exceptions= (".*");
```

Step 2 Write a subroutine that disables the application of ACLs on the interfaces.

Use the following code as a reference:

```
config ("
    no access-group INSIDE_VCC_ACL in interface $FWInsideInterfaceGroup->{Name}
    no access-group OUTSIDE_VCC_ACL in interface
    $FWOutsideInterfaceGroup->{Name}
");
```



Note For more information, see the [config\(\)](#) subroutine.

Step 3 Write a subroutine that disables the outside ACL for the VIP.

Use the following code as a reference:

```
config("
  no access-list OUTSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($VipAddrGroup->{IPAddress},
  INFO)}
  no access-list OUTSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($VipAddrGroup->{IPAddress},
  INFO)} eq www
");
```



Note For more information, see the [config\(\)](#) and [get_ip\(\)](#) subroutines.

Step 4 Write a subroutine that disables the outside ACL for DHCP traffic.

Use the following code as a reference:

```
config("
  no access-list OUTSIDE_VCC_ACL extended permit udp any eq bootps any eq bootpc
");
```

Step 5 Write a subroutine that disables the outside ACL for the NFS filers.

Use the following code as a reference:

```
config("
  no access-list OUTSIDE_VCC_ACL extended permit icmp host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)} any echo-reply
");
```

Step 6 Write a subroutine that disables the inside ACL for the NFS filers.

Use the following code as a reference:

```
config("
  no access-list INSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
  no access-list INSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
  no access-list INSIDE_VCC_ACL extended permit udp any host
  ${\&get_ip($NFSFiler->{IPAddress}, INFO)}
");
```

Step 7 Write a subroutine that disables the inside ACL for VFrame traffic.

Use the following code as a reference:

```
config("
  no access-list INSIDE_VCC_ACL extended permit udp any eq bootpc any eq bootps
  no access-list INSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($VCC->{IPAddress}, INFO)} eq 3000
  no access-list INSIDE_VCC_ACL extended permit tcp any host
  ${\&get_ip($VCC->{IPAddress}, INFO)} eq 3010
  no access-list INSIDE_VCC_ACL extended permit udp any host
  ${\&get_ip($VCC->{IPAddress}, INFO)} eq tftp
");
```

Step 8 Write a subroutine that disables ping permissions for the inside and outside ACLs.

Use the following code as a reference:

```
config("
  no access-list INSIDE_VCC_ACL extended permit icmp any host
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)}
  no access-list OUTSIDE_VCC_ACL extended permit icmp host
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)} any echo-reply
");
```

Step 9 Write a subroutine to remove any port communication configuration.

Use the following code as a reference:

```
config("
  no fixup protocol dns maximum-length 512
  no fixup protocol ftp 21
  no fixup protocol h323 H225 1720
  no fixup protocol h323 ras 1718-1719
  no fixup protocol http 80
  no fixup protocol rsh 514
  no fixup protocol sip 5060
  no fixup protocol sip udp 5060
  no fixup protocol skinny 2000
  no fixup protocol smtp 25
  no fixup protocol sqlnet 1521
");
```

Step 10 Write a subroutine to remove the gateway route configuration.

Use the following code as a reference:

```
config("
  no route $FWOutsideInterfaceGroup->{Name} 0.0.0.0 0.0.0.0
  ${\&get_ip($DefaultGateway->{IPAddress}, INFO)}
");
```

Step 11 Remove the firewall interfaces configuration.

Use the following code as a reference:

```
config("
  interface vlan$FWOutsideInterfaceGroup->{Vlan} no bridge-group 1
  no nameif $FWOutsideInterfaceGroup->{Name}
  interface vlan$FWInsideInterfaceGroup->{Vlan} no bridge-group 1
  no nameif $FWInsideInterfaceGroup->{Name}
");
```

Step 12 Write a subroutine to remove the BVI interface configuration.

Use the following code as a reference:

```
config("
  no interface BVI 1
");
```

- Step 13** Write a subroutine that disables ping permissions for the ACL and restricts HTTP traffic to the VIP. Use the following code as a reference:

```
config("
  no access-group VIP_ACL in interface outside
  no access-list VIP_ACL extended permit icmp any host
  ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)}
  no access-list VIP_ACL extended permit tcp any host
  ${\&get_ip($VipAddrGroup->{IPAddress}, INFO)} eq www
");
```

Writing a CSM Configuration Removal Macro

An CSM configuration removal macro typically includes the following operations:

- Disabling the virtual servers
- Disabling the server farms
- Disabling ping
- Removing the interface configurations

The following procedure provides the steps for writing a CSM configuration removal macro. Each step in the procedure provides the following instructions:

- The next operation to code
- An example subroutine that provides a reference to aid in the development of your macro

Procedure

- Step 1** Write a subroutine that disables any virtual server.

Use the following code as a reference:

```
config ("
  no vserver $Vserver->{Name}
");
```



Note For more information, see the [config\(\)](#) subroutine.

- Step 2** Write a subroutine that disables any server farm.

Use the following code as a reference:

```
config ("
  no serverfarm $ServerFarm->{Name}
");
```

- Step 3** Write a subroutine that disables ping.

Use the following code as a reference:

```
config ("
  no probe PING icmp
");
```

Step 4 Write a subroutine that removes any interface configurations.

Use the following code as a reference:

```
config ("
  no vlan $ClientInterfaceVariableType->{Vlan}
  $ServerInterfaceVariableType->{InterfaceType}
  no vlan $ServerInterfaceVariableType->{Vlan}
  $ServerInterfaceVariableType->{InterfaceType}
");
```

Putting It All Together

VFrame macros start operation at the line of code that calls the name of the macro, then executes line by consecutive line. Consequently, macros need to be structured in a certain way. We recommend that you use [Example B-1](#) to structure your macros.

Example B-1 Macro Structure

```
#####
declare all your variables
#####

my $varOne
...
my $varN

#####
list the subroutines used in the macro
#####

sub routine_One
...
sub routine_N

#####
call the macro by name
#####

sub macro_One

#####
call the subroutines
#####

routine_One
...
routine_N

#####
write out the subroutines
#####

sub routine_One

config ("
...
");
```

```

sub routine_N

if ($varOne->{name} eq $varN->{name}) {
...
}
#####

```

Expressing Variables in Macros

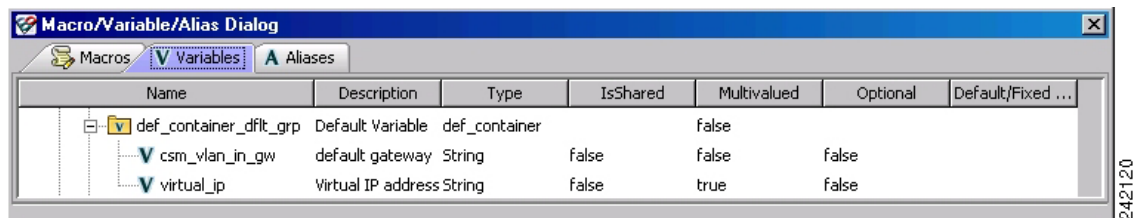
VFrame Data Center variables, as they appear in the GUI, are expressed differently in macros. When referring to a variable, macros use Perl notation to express the variables, but the GUI does not. How the VFrame Data Center variables are expressed in macros depends on the variable type. This section describes how each variable type appears in the GUI and is expressed in macros, and includes the following topics:

- [Single-Value Variables](#)
- [Single-Value Variable Groups](#)
- [Multivalue Variables](#)
- [Multivalue Variable Groups](#)

Single-Value Variables

When shown in the VFrame GUI, a single-value variable appears as part of a variable group, and its Multivalued field is set to false. For example, [Figure B-1](#) shows how the `csm_vlan_in_gw` single-value variable appears in the VFrame GUI.

Figure B-1 *csm_vlan_in_gw Single-Value Variable*



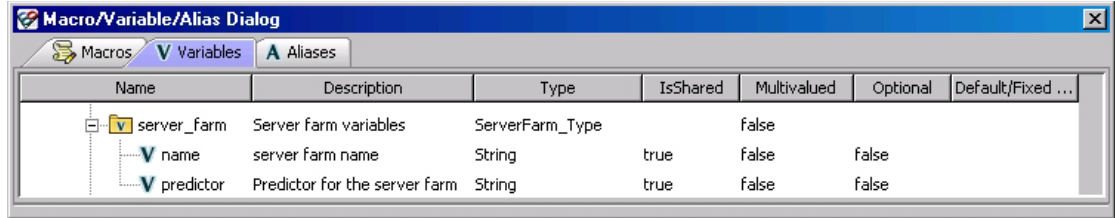
However, a single-value variable is expressed in a macro as a Perl variable. For example, the same `csm_vlan_in_gw` single-value variable is expressed as the following Perl variable in a macro:

```
our $csm_vlan_in_gw = "192.0.2.100";
```

Single-Value Variable Groups

When shown in the VFrame GUI, a single-value variable group appears as a folder containing one or more variables, and its Multivalued field is set to false. For example, [Figure B-2](#) shows how the `server_farm` single-value variable group appears in the VFrame GUI.

242120

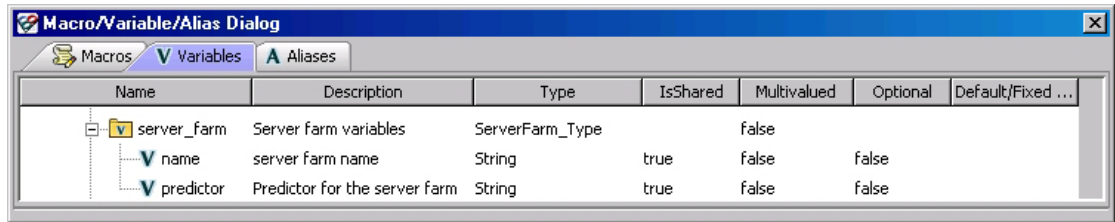
Figure B-2 *server_farm Single-Value Variable Group*

However, a single-value variable group is expressed in a macro as a Perl hash (associative array), where the hash keys are key/value pairs. For example, the same `server_farm` single-value variable group, which contains the `name` and `predictor` single-value variables, is expressed as the following Perl hash in a macro:

```
our $server_farm={
    name=>"serv_farm_west",
    predictor=>"weighted",
};
```

Multivalue Variables

When shown in the VFrame GUI, a multivalue variable appears as part of a variable group, and its Multivalued field is set to true. For example, Figure B-3 shows how the `virtual_ip` multivalue variable appears in the VFrame GUI.

Figure B-3 *virtual_ip Multivalue Variable*

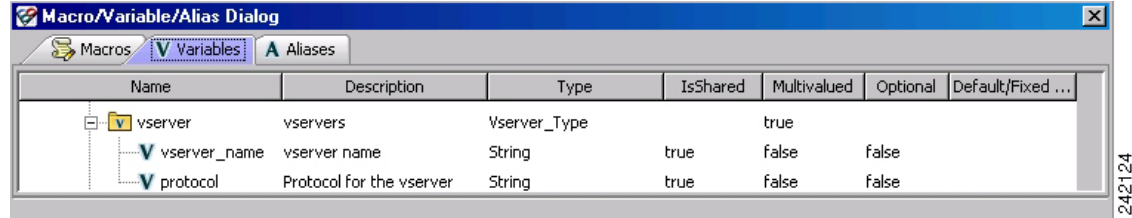
However, a multivalue variable is expressed in a macro as an array, where each element in the array is a value. For example, the same `virtual_ip` multivalue variable is expressed as the following array in a macro:

```
our @virtual_ip=(192.0.2.100, 192.0.2.101, 192.0.2.102)
```

Multivalue Variable Groups

When shown in the VFrame GUI, a multivalue variable group appears as a folder containing one or more multivalue variables, and its Multivalued field is set to true. For example, Figure B-4 shows how the `vserver` multivalue variable group appears in the VFrame GUI.

Figure B-4 vserver Multivalue Variable Group



However, a multivalue variable group is expressed in a macro as an array of Perl hashes, where each element in the array is a hash, and the hash keys for each hash are key/value pairs. For example, the same vserver multivalue variable group, which contains the vserver_name and protocol multivalue variables, is expressed as the following array of Perl hashes in a macro:

```
our @vserver=( {
    vserver_name=>"serv1",
    protocol=>"tcp www",
}, {
    vserver_name=>"serv2",
    protocol=>"tcp www",
});
```

Subroutine Reference

This section provides detailed reference information for the following Cisco VFrame Data Center macro subroutines:

- [cmd_config\(\)](#)
- [config\(\)](#)
- [error_report\(\)](#)
- [find_and_parse\(\)](#)
- [get_default_timeout\(\)](#)
- [get_ip\(\)](#)
- [get_line\(\)](#)
- [get_mask\(\)](#)
- [get_parser_status\(\)](#)
- [get_prompt\(\)](#)
- [increment_line\(\)](#)
- [increment_parser_status\(\)](#)
- [int2ip\(\)](#)
- [ip2int\(\)](#)
- [print_parser\(\)](#)
- [reset_parser\(\)](#)
- [set_config_check\(\)](#)
- [set_default_timeout\(\)](#)
- [set_line\(\)](#)
- [set_prompt\(\)](#)
- [set_prompt_to_cisco_ios\(\)](#)
- [set_prompt_to_linux\(\)](#)
- [wait_sec\(\)](#)

cmd_config()

```
cmd_config ($send_str, $match, {exception_re=>RegExp, parse_flag=>Value, timeout=>Interval})
```

Purpose

Sends a single CLI command to a device and checks the device response for unexpected text.

Parameters

\$send_str	Single CLI command to be sent to a device.
\$match	Optional. The match criteria (a string or regular expression) against which the device CLI output is matched. If the output fails to match the specified criteria, the macro fails. The default value is the CLI prompt. For more information about retrieving and configuring the CLI prompt, see the get_prompt() , set_prompt() , set_prompt_to_cisco_ios() , and set_prompt_to_linux() subroutines.
exception_re=>RegExp	Optional. Exception hash, where <i>RegExp</i> is a regular expression that defines text string which is allowed to appear in the device output without causing a match failure.
parse_flag=>Flag	Optional. Parse flag hash, where <i>Flag</i> specifies whether a parse array is generated. Valid Flag values include the following: <ul style="list-style-type: none"> • 0—No parse array is generated. • 1—A parse array is generated. All output resulting from the command being sent to the device is saved to the parse array. <p>Note You can access the text in the parse array using the parser-related subroutines described in this section.</p> <p>The default <i>Flag</i> value is 0.</p>
timeout=>Interval	Optional. Timeout hash, where <i>Interval</i> specifies the time, in seconds, to wait for a valid match. If a valid match is not made within the timeout interval, an error is generated. The default <i>Interval</i> value is 5 . <p>Note The default timeout interval can also be globally set using the <code>set_default_timeout()</code> subroutine, or the system preferences. If the timeout hash is not specified, then the <code>set_default_timeout()</code> value is used. If neither the timeout hash or the <code>set_default_timeout()</code> subroutine is specified, then the system preferences default timeout value is used.</p>

Return Values

This subroutine has no return values.

Usage Guidelines

Use the `cmd_config()` subroutine to send a single CLI command to a device and check the device response for unexpected text.



Note

The `cmd_config()` subroutine can send only one CLI command line. To send multiple command lines, use the `config()` subroutine,

Use the optional `$match` variable to change the match criteria used when checking the device response. That is, you can configure the `$match` variable to check the response for something other than the CLI prompt.

**Note**

When successfully run, the `cmd_config()` subroutine expects the device to respond only with the command sent to the device (specified by the `$send_str` variable), and the string that matches the `$match` variable, with nothing in between.

Use the optional `exception_re` hash to define a text string that is allowed to appear in the device response before the CLI prompt without generating an error. This is useful in situations where you expect the device to respond with informational text before the prompt.

**Note**

If you need to specify more than one exception regular expression, you can configure a global array of exception regular expressions that will be used by all `config()` and `cmd_config()` subroutines. For more information, see the [set_config_check\(\)](#) subroutine.

Set the optional `parse_flag` hash to 1 (`parse_flag=>1`) to send the device response to a parser array without generating an error when a match failure occurs. The parser array is an array of strings, with each string on a separate line. The parser array is useful for capturing information from the device that you want to see, such as **show** command output. The contents of parse array is overwritten each time the `cmd_config()` subroutine is run with the `parse_flag` hash set to 1. When the `cmd_config()` subroutine sends the device response to the parser array, the cursor is initially set to the beginning of the first line of the parser array.

When the `parse_flag` hash is set to 0 (or not specified) and a match failure occurs, nothing is sent to the parse array and the `NO_MATCH` error is generated. For more information about the `NO_MATCH` error, see the [error_report\(\)](#) subroutine.

**Note**

The `exception_re` and `parse_flag` hashes are mutually exclusive. Both hashes allow the macro to continue running without generating an error after a match failure, but the `exception_re` hash does not send any information to the parse array, whereas the `parse_flag` hash (when set to 1) does send information to the parse array for later viewing. We recommend that you do not specify both hashes in the same `cmd_config()` subroutine. If both are specified, the `parse_flag` hash overwrites the `exception_re` hash.

Use the optional `timeout` hash to specify the time interval, in seconds, to wait for a valid match. If the `timeout` hash value is not specified, a global `timeout` value set by either the [set_default_timeout\(\)](#) subroutine or the system preferences is used. If a match is not made within the `timeout` interval, the `TIMEOUT` error is generated. For more information about the `TIMEOUT` error, see the [error_report\(\)](#) subroutine.

Examples**Basic Subroutine Example**

The following example sends the **show running-config module \$CSM_Module** command to a device and prints the command output that was sent to the parse array:

```
sub cmd_config ("show running-config module $CSM_Module", {parse_flag=>1});
print_parser();
```

Successful Subroutine Example

By default, with no optional parameters specified, the `cmd_config()` subroutine generates an error when the device responds to a CLI command with anything other than the CLI prompt. For example, if the CLI prompt is “prompt#” and the following `cmd_config()` subroutine is run:

```
cmd_config("interface vlan230");
```

The **interface** command is entered and the device responds with the prompt, as follows:

```
interface vlan230
prompt#
```

The subroutine succeeds because the device responds with the CLI prompt, which matches the default match value.

Failed Subroutine Example

The following example shows a failed run of the `cmd_config()` subroutine, where the subroutine detects a mismatch and generates an error. If the following `cmd_config()` subroutine is run:

```
cmd_config("interface vlan230");
```

The **interface** command is entered and the device response contains an INFO message before prompt, as follows:

```
interface vlan230
INFO: Security level for \".*\\" set to .* by default.
prompt#
```

The subroutine fails because the INFO message did not match the default match value, which is the CLI prompt.

Exception Support Example

The following example runs the `cmd_config()` subroutine and allows any line with the “INFO: Security level” string to appear in the device output without causing a match failure:

```
cmd_config ("config", exception_re=>"INFO: Security level");
```

The **interface** command is entered and the device response contains an INFO message before prompt, as follows:

```
interface vlan230
INFO: Security level for \".*\\" set to .* by default.
prompt#
```

The subroutine succeeded because the `exception_re` hash allows the line with the INFO message to appear without causing a match failure. The subroutine skips that line and then matches the CLI prompt on the next line.

config()

```
config ($send_str, $match, {exception_re=>RegExp, timeout=>Interval})
```

Purpose

Sends one or more CLI commands to a device and checks the device response for unexpected information.

Parameters

\$send_str	One or more CLI commands to be sent to a device, with each command on a new line, as specified by a carriage return or the \n modifier.
\$match	Optional. The match criteria (a string or regular expression) against which the device CLI output is matched. If the output fails to match the specified criteria, the macro fails. The default value is the device CLI prompt. For more information about retrieving and configuring the CLI prompt, see the get_prompt() , set_prompt() , set_prompt_to_cisco_ios() , and set_prompt_to_linux() subroutines.
exception_re=>RegExp	Optional. Exception hash, where <i>RegExp</i> is a regular expression that defines text string which is allowed to appear in the device output without causing a match failure.
timeout=>Interval	Optional. Timeout hash, where <i>Interval</i> specifies the time, in seconds, to wait for a valid match. If a valid match is not made within the timeout interval, an error is generated. The default <i>Interval</i> value is 5 . Note The default timeout interval can also be globally set using the set_default_timeout() subroutine.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the `config()` subroutine to send one or more CLI commands to a device and check the device response for unexpected information. The subroutine sends the CLI commands to the device and checks the devices' response for matches to the `$match` value. The subroutine ends after a match is made.



Note

Unlike the `cmd_config()` subroutine, the device response cannot be sent to the parser array.

If all the CLI commands have been sent to the device and no match has been made, the subroutine waits a specified time interval for the device to respond and a valid match to be made. If a match is not made within the time interval, an error is generated and the subroutine ends. Use the optional timeout hash to specify the time interval, in seconds, to wait for a valid match.

Use the optional `$match` variable to change the match criteria used when checking the device response. That is, you can configure the `$match` variable to check the response for something other than the CLI prompt.



Note

When successfully run, the `config()` subroutine expects the device to respond only with the command(s) sent to the device (specified by the `$send_str` variable), and the string that matches the `$match` variable, with nothing in between.

Use the optional `exception_re` hash to define a text string that is allowed to appear in the device response before the CLI prompt without generating an error. This is useful in situations where you expect the device to respond with informational text before the prompt.

**Note**

If you need to specify more than one exception regular expression, you can configure a global array of exception regular expressions that will be used by all `config()` and `cmd_config()` subroutines. For more information, see the [set_config_check\(\)](#) subroutine.

Examples**Multiple CLI Command Example**

The following example sends two CLI commands to a device:

```
config("
  vlan $csm_ingress_vlan client
  ip address $ip_transp->{IPAddress} $ip_transp->{IPMask}
");
```

Successful Subroutine Example

The following example shows a successful run of the `config()` subroutine, where the device returns the command sent to it and a string that matches the `$match` variable, with nothing in between. If the following `config()` subroutine is run:

```
config("vlan 250 client");
```

The **vlan 250 client** command is entered and the device responds with a prompt, as follows:

```
vlan 250 client
6k-2-2(config-slb-vlan-client)#
```

The subroutine succeeds because the device responds with the CLI prompt, which matches the default match value.

Failed Subroutine Example

The following example shows a failed run of the `config()` subroutine, where the subroutine detects a mismatch and stops the macro. If the following `config()` subroutine is run:

```
config("iiii probe ping");
```

The **iiii probe ping** command is entered and the device responds with an invalid input message, as follows:

```
iiii probe ping
^
% Invalid input detected at '^' marker.
prompt#Couldn't compile code in box: NO_MATCH at macro_run.pl line 215.
prompt#
```

The subroutine fails because the invalid input message did not match the default match value, which is the CLI prompt.

Exception Support Example

The following example uses the `exception_re` hash to specify a string that excluded a line from the `config()` subroutine string-matching process. That is, the subroutine will not fail when it receives a line containing text that matches the `exception_re` hash. The `config()` subroutine is as follows:

```
config("config terminal", {exception_re=>"Enter configuration commands"});
```

The **config terminal** command is entered and the device response contains an informational message before prompt, as follows:

```
config terminal
Enter configuration commands, one per line. End with CNTL/Z.
prompt(config)#
```

The subroutine succeeded because the `exception_re` hash allows the line with the informational message to appear without causing a match failure. The subroutine skips that line and then matches the CLI prompt on the next line.

error_report()

```
error_report($Error, $user_str, {recv=>String, send=>String})
```

Purpose

Reports errors that occur when running a macro, and stops the macro when a fatal error type is specified.

Parameters

\$Error	Error type. The error types that can be reported to VFrame Data Center are grouped into the following three categories: <ul style="list-style-type: none"> • Informational error—Indicates that no failure has occurred and the macro is allowed to continue running. The error types included in this category are NONE, INFO, and WARNING. • Non-fatal error—Indicates that an action has failed, but the macro is allowed to continue running. The error type included in this category is ERROR. • Fatal error—Indicates that an action has failed and the macro is immediately stopped. The error types included in this category are CONNECTION_FAILURE, CREDENTIAL_FAILURE, FAILURE, NO_MATCH, SYSTEM_SCRIPT_FAILURE, TIMEOUT, and USER_SCRIPT_FAILURE.
\$user_str	Error message string that describes the specific error being reported.
send=>String	Optional. Send hash, where <i>String</i> is the command that caused the error.
recv=>String	Optional. Receive hash, where <i>String</i> is the text string sent from the device when the error occurred.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the error_report() subroutine to report errors that occur when running a macro, and to stop the macro when a fatal error type is specified.



Note

Running the error_report() subroutine when you have specified a fatal error type for that error report immediately stops the macro.

Use the \$Error variable to specify the error type to report and the \$user_str to specify a string that describes the error.

Use the optional send=>String or recv=>String parameters to include the command that caused the error or the text string sent from the device when the error occurred, respectively.

Examples

Informational Error Example

The following example reports an informational error stating that the configuration is valid. Because it is an informational error, the macro continues to run:

```
error_report (INFO,"Configuration is Valid - verify_csm");
```

Non-Fatal Error Example

The following example reports a non-fatal error stating that credential authentication has failed. Because it is a non-fatal error, the macro continues to run:

```
error_report (ERROR, "Credential failure - enable password");
```

Fatal Error Example

The following example reports a fatal error indicating that configuration verification has failed. Because it is a fatal error, the macro is immediately stopped:

```
error_report (FAILURE, "Configuration verification FAILURE (verify_csm)");
```

find_and_parse()

```
find_and_parse($expect_str, $var_regexp, {end_of_block=>RegExp, current_only=>Flag,
error=>ErrorType, match=>Type})
```

Purpose

Searches the contents of a parser array for a match to a specified string or regular expression, and optionally extracts a value after a match has been found.

Parameters

\$expect_str	The match criteria (a string or regular expression) against which the device CLI output is matched.
\$var_regexp	Optional. String or regular expression that specifies a value that is to be extracted from the parser array. A value can be extracted only if the subroutine finds a match to the \$expect_str variable, and the value must immediately follow the string match, on the same line in the parser array.
end_of_block=>RegExp	Optional. End of block hash, where <i>RegExp</i> is a regular expression that is used to search for a string that signifies the end of a block of strings in the parser array.
current_only=>Flag	Optional. Current line only hash, where <i>Flag</i> specifies whether the search is restricted to the current line: <ul style="list-style-type: none"> • 0—Does not restrict the search to the current line. • 1—Restricts the search to the current line.
error=>ErrorType	Optional. Error hash, where <i>ErrorType</i> specifies the error type reported to VFrame Data Center. Error types are grouped into the following three categories: <ul style="list-style-type: none"> • Informational error—Indicates that no failure has occurred and the macro is allowed to continue running. Valid <i>ErrorType</i> values included in this category are NONE, INFO, and WARNING. • Non-fatal error—Indicates that an action has failed, but the macro is allowed to continue running. The valid <i>ErrorType</i> value included in this category is ERROR. • Fatal error—Indicates that an action has failed and the macro is immediately stopped. The valid <i>ErrorType</i> values included in this category are CONNECTION_FAILURE, CREDENTIAL_FAILURE, FAILURE, NO_MATCH, SYSTEM_SCRIPT_FAILURE, TIMEOUT, and USER_SCRIPT_FAILURE.

The default *ErrorType* value is **FAILURE**.

<code>match=>Type</code>	<p>Optional. Match type hash, where <i>Type</i> specifies whether a match is performed against the device output using a regular expression, an exact match, or a substring. Valid <i>Type</i> values include the following:</p> <ul style="list-style-type: none"> • regexp—Regular expression • exact—Exact match • substr—Substring <p>The default <i>Type</i> value is regexp.</p>
-----------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Return Values

This subroutine optionally returns string that matches the `$var_regexp` variable.

Usage Guidelines

Use the `find_and_parse()` subroutine to search the contents of a parser array for a match to a specified string or regular expression, and optionally extracts a value after a match has been found.

When a match is found on a particular line in the parser array, the subroutine moves the cursor to the beginning of the next line. Running the `find_and_parse()` subroutine again starts the search from the new cursor position, not from the beginning of the parser array. However, if the `find_and_parse()` subroutine is run and no errors are found, the cursor remains at the beginning of the parser array, and running the `find_and_parse()` subroutine again starts the search from the beginning of the parser array.



Note

The `cmd_config()` subroutine saves output in a parser array, which is an array of strings, with each string on a separate line. The cursor is initially set to the beginning of the first line of the parser array. For more information, see the [cmd_config\(\)](#) subroutine.

Use the optional `$var_regexp` variable to specify a text string to extract from the line where the subroutine has found a match to the `$expect_str` variable. The text string to be extracted must follow the string that matches the `$expect_str` variable. After a text string is extracted from a line where a match has been found, the cursor is still moved to the beginning of the next line in the parser array. This option is useful in situations where you want to extract a value associated with a particular field. For example, if the parser array contains the “ip address 192.0.2.100” string, you can use the `find_and_parse()` subroutine to search for the IP address field, and then extract the value, which is the “192.0.2.100” string.

Use the optional `end_of_block` hash to stop searching through the parser array when a match is found to the search string or the `end_of_block` regular expression. This is useful in situations where the parser array may have multiple instances of the same string that you are searching for, but you only want to find a particular instance of that string in a specific block of lines in the parser array. You would first use the `find_and_parse()` subroutine to search for the string that signifies the beginning of the block that you want to search, then use another `find_and_parse()` subroutine to start searching for your search string until it either finds the string in that block or finds the string that signifies the end of that block.

Use the optional `current_only` hash to restrict search area to the line in the parser array where the cursor is currently positioned.

Use the optional `error` hash to specify the behavior that will occur when the `find_and_parse()` subroutine fails. By default, the `error` hash is set to `FAILURE`, and the `find_and_parse()` subroutine logs an error and terminates the macro when it finds the first error. To find multiple errors, you must set the `error` hash to a non-fatal or informational error type.

Examples

Some of the examples in this section use the following parser array contents:

```
#0#show running-config module 3
#1#Building configuration...
#2#
#3#Current configuration : 859 bytes
#4#module ContentSwitchingModule 3
#5# ft group 3 vlan 332
#6#!
#7# vlan 102 client
#8# ip address 192.0.2.100 255.255.255.0 alt 192.0.2.101 255.255.255.0
#9# gateway 192.0.2.1
.
.
.
#42# serverfarm SF_SG_0
#43# persistent rebalance
#44# inservice
#45#!
#46# vserver VS_SG_1
#47# virtual 192.168.28.221 tcp www
#48# vlan 102
#49# serverfarm SF_SG_1
#50# persistent rebalance
#51# inservice
#52#!
#53#end
```

Successful Subroutine Example

The following example shows a successful run of the find_and_parse() subroutine. The cursor is initially set to line 0 in the parser array and the following subroutine is run:

```
find_and_parse("vlan 102 client")
```

The subroutine searches for the “vlan 102 client” string and successfully finds it on line 7 in the parser array. The subroutine then moves the cursor to the beginning of the line 8 (the line following line 7).

Failed Subroutine Example

The following example shows a successful run of the find_and_parse() subroutine. The cursor is set to line 8 in the parser array and the following subroutine is run:

```
find_and_parse("vlan 102 client")
```

The subroutine fails because the search begins from line 8, and the parser array does not contain the “vlan 102 client” string from line 8 to the end of the parser array. To successfully run the same subroutine again, you must first use the set_line() subroutine to move the cursor to any line above line 7.

Current Line Only Example

The following find_and_parse() subroutine restricts its search to the current line, and succeeds only if the cursor is set to line 8, which is the line that contains the matching string:

```
find_and_parse("ip address 192.0.2.100 255.255.255.0 alt 192.0.2.101 255.255.255.0",
{current_only=>1});
```

If the cursor is set to any other line, and the search is restricted to the current line, then running the subroutine results in a failure.

End of Block Example

The following example shows how to find the “persistent rebalance” string that applies only to serverfarm SF_SG_0. The cursor is set to line 8, and the following subroutines are run:

```
find_and_parse("serverfarm SF_SG_0");
find_and_parse("persistent rebalance", {end_of_block=>"!"});
```

The first subroutine finds the “serverfarm SF_SG_0” string on line 42, and sets the cursor to the beginning of line 43. The next subroutine finds the “persistent rebalance” string on line 43. If the same subroutine is run again, it will find the “!” end of block string and stop before it finds the next instance of the “persistent rebalance” string on line 50, which applies to serverfarm SF_SG_1.

Value Extraction Example

The find_and_parse() subroutine also allows you to extract values from parser array that are associated with specific fields. For example, if you want to extract the values associated with the following fields:

```
Symmetrix ID: 000187400067
Director Ports Status : [ON,ON,N/A,N/A]
Device Configuration : RAID-5 (Meta Member,
```

The following example extracts the values located immediately after the text string matches (field names):

```
$array{UID} = find_and_parse("Symmetrix ID",NUMBER_RE);
$status = find_and_parse("Director Ports Status :",WORD_RE);
@port_status =split(/,/, $status);
my $line = find_and_parse("Device Configuration :", ".*"); #read whole line
if ($line =~ /(WORD_RE).*(Meta Member)/) else {
    $line =~ /(WORD_RE)/;
    $lun[$luns]->{RaidLevel} = $1;
}
```

get_default_timeout()

`get_default_timeout()`

Purpose Acquires the global default timeout value.

Parameters This subroutine has no parameters.

Return Values Default timeout, in seconds.

Usage Guidelines Use the `get_default_timeout()` subroutine to acquire the global default timeout value. This is useful in situations where you want to set the value of a variable equal to the default timeout value.

Initially, the global default timeout value is 5, but you can use the `set_default_timeout()` subroutine to change this value. For more information, see the [set_default_timeout\(\)](#) subroutine.

Examples The following example sets the `$seconds` variable to the global default timeout value:

```
$seconds=get_default_timeout()
```

get_ip()

```
get_ip($IPAddrResourceType)
```

Purpose

Extracts the IPv4 address from the \$IPAddrResourceType variable.

Parameters

\$IPAddrResourceType	IP address resource type variable. Contains the IP address and the subnet mask in the form <i>A.B.C.D/W.X.Y.Z</i> , where <i>A.B.C.D</i> is the IP address and <i>W.X.Y.Z</i> is the subnet mask.
----------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Return Values

IPv4 address string.

Usage Guidelines

Use the get_ip() subroutine to extract the IPv4 address from the \$IPAddrResourceType variable.

Examples

The following example extracts the IP address from the \$IPAddrResourceType variable, and sets the \$ip_addr variable to 192.0.2.100:

```
$IPAddrResourceType=192.0.2.100/255.255.255.0  
$ip_addr=get_ip($IPAddrResourceType)
```

get_line()

```
get_line({increment=>X, move=>X})
```

Purpose

Extracts the line number and the contents of a line in the parse array.

Parameters

increment=>X	Optional. Increment hash, where <i>X</i> is the number of lines to increment the cursor in the parser array after returning a value.
move=>X	Optional. Move hash, where <i>X</i> is the number of lines to increment the cursor in the parser array before returning a value.

Return Values

Line number and the contents of a line in the parse array.

Usage Guidelines

Use the `get_line()` subroutine to extract the line number and the contents of a line in the parse array.

Examples

The examples in this section use the following parser array contents:

```
#0#show running-config module 3
#1#Building configuration...
#2#
#3#Current configuration : 859 bytes
#4#module ContentSwitchingModule 3
#5# ft group 3 vlan 332
#6#!
#7# vlan 102 client
#8# ip address 192.0.2.101 255.255.255.0 alt 192.0.2.102 255.255.255.0
#9# gateway 192.0.2.1
```

Basic Get Line Example

The following example sets the cursor to line 9, sets `$line_number = 9`, and then sets `$line = "gateway 192.0.2.1"`

```
set_line(9);
($line_number, $line) = get_line()
```

Get Line with Increment Example

The following example sets the cursor to line 3, sets `$line_number = 3`, sets `$line = "Current configuration : 859 bytes"`, and then increments the cursor six lines to line 9:

```
set_line(3);
($line_number, $line) = get_line({increment=>6});
```

Get Line with Move Example

The following example sets the cursor to line 4, increments the cursor two lines to line 6, sets \$line_number = 6, and then sets \$line = “!”.

```
set_line(4)
($line_number, $line) = get_line({move=>2})
```

Get Line with Increment and Move Example

The following example sets the cursor to line 0, increments the cursor seven lines to line 7, sets \$line_number = 7, sets \$line = “vlan 102 client”, and then increments the cursor one line to line 8:

```
set_line(0);
($line_number, $line) = get_line({increment=>1, move=>7})
```

get_mask()

```
get_mask($IPAddrResourceType)
```

Purpose Extracts the subnet mask from the \$IPAddrResourceType variable.

Parameters `$IPAddrResourceType` IP address resource type variable. Contains the IP address and the subnet mask in the form *A.B.C.D/W.X.Y.Z*, where *A.B.C.D* is the IP address and *W.X.Y.Z* is the subnet mask.

Return Values Subnet mask string.

Usage Guidelines Use the get_mask subroutine to extract the subnet mask from the \$IPAddrResourceType variable.

Examples The following example extracts the subnet mask from the \$IPAddrResourceType variable, and sets the \$net_mask variable to 255.255.255.0:

```
$IPAddrResourceType=192.0.2.100/255.255.255.0  
$net_mask=get_mask($IPAddrResourceType)
```

get_parser_status()

get_parser_status()

Purpose

Acquires the total number of times the find_and_parse() subroutine has failed.

Parameters

This subroutine has no parameters.

Return Values

The find_and_parse() subroutine failure count.

Usage Guidelines

Use the get_parser_status() subroutine to acquire the total number of times the find_and_parse() subroutine has failed.

This subroutine is useful in situations where you want to stop a macro after allowing the find_and_parse() subroutine to find multiple errors in the macro, not just the first error. To do this, you must configure the find_and_parse() subroutine so that it does not immediately stop the macro after finding the first error. Instead, you let it find multiple errors in the macro. Then you use the get_parser_status() to see if the find_and_parse() subroutine failure count is greater than zero. If it is, you would then use the error_report() subroutine to stop the macro and report that multiple errors were found. For more information, see the [error_report\(\)](#) and [find_and_parse\(\)](#) subroutines.

Examples

Basic Get Parser Status Example

The following example sets the value of the \$parse_status variable to the total number of times the find_and_parse() subroutine has failed:

```
$parse_status=get_parser_status();
```

Finding Multiple Errors Example

The following parser array contents are used for the example:

```
#0#show running-config module 3
#1#Building configuration...
#2#
#3#Current configuration : 859 bytes
#4#module ContentSwitchingModule 3
#5# ft group 3 vlan 332
#6#!
#7# vlan 102 client
#8# ip address 192.0.2.100 255.255.255.0 alt 192.0.2.101 255.255.255.0
#9# gateway 192.0.2.1
```

The following example shows how to stop a macro after finding multiple errors. The find_and_parse() subroutine is run three times. The first find_and_parse() subroutine successfully finds a match on line 5. The second find_and_parse() subroutine begins searching from line 6 and fails to find a match. The third find_and_parse() subroutine also begins searching from line 6 and also fails to find a match. Because there were two failures, the \$parse_status value is set to 2 (using the get_parser_status() subroutine). The

if statement checks to see if the `$parse_status` value is greater than zero. Because the value is greater than zero, the `error_report()` subroutine stops the macro with a `FAILURE` error and sends the “Configuration verification FAILURE” string.

```
find_and_parse("ft group 3 vlan 332", {error=>ERROR});
find_and_parse("vlan 202 client", {error=>ERROR});
find_and_parse("gateway 192.0.2.101", {error=>ERROR});
$parse_status=get_parser_status();
if ($parse_status>0)
{
    error_report(FAILURE, "Configuration verification FAILURE");
}
```

get_prompt()

get_prompt()

Purpose Gets the currently set prompt.

Parameters This subroutine has no parameters.

Return Values Currently set prompt.

Usage Guidelines Use the get_prompt() subroutine to get the currently set prompt.

Examples The following example acquires the prompt from the current line in the parser array and sets the \$prompt_re variable equal to the prompt.

```
$prompt_re=get_prompt();
```

increment_line()

```
increment_line($inc)
```

Purpose Increments the cursor position in the parse array.

Parameters `$inc` Increment variable. Number of lines to increment in the parse array.

Return Values Line number after incrementing.

Usage Guidelines Use the `increment_line()` subroutine to increment the cursor position in the parse array.

Examples The following example sets the cursor to line 5 in the parser array, increments the cursor 15 lines to line 20, and sets the `$current_line` variable to 20.

```
set_line (5)  
$current_line = increment_line(15)
```

increment_parser_status()

```
increment_parser_status($inc)
```

Purpose Increments the parser status value.

Parameters `$inc` Parser status increment variable. Increases the parser status by the specified increment value.

Return Values Parser status value after incrementing.

Usage Guidelines Use the `increment_parser_status()` subroutine to increment the parser status value.

Examples Assuming that the `find_and_parse()` subroutine failed five times. failed, the following example sets the `$parse_status` variable equal to 5, and increments the value of the `$parse_status` variable by 3, to a total value of 8.

```
$parse_status = get_parser_status();  
$parse_status = increment_parser_status(3);
```

int2ip()

```
int2ip($ip_net_num)
```

Purpose

Converts a network number from an integer to an IPv4 address string.

Parameters

`$ip_net_num` Network number in integer format.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the `int2ip()` subroutine to convert a network number from an integer to an IPv4 address string.

This subroutine is useful in situations where you have converted IPv4 address strings to integer format so that you can easily perform logical operations with them, and want to convert the results of the logical operations back to IPv4 address strings.

Examples

The following example extracts the network number in integer format (3221225984) from the 192.0.2.100 IP address and 255.255.255.0 subnet mask, converts it to IPv4 address format (192.0.2.0), and then sets the `$ip_net_string` to equal the network number in IPv4 address:

```
$ip_net_num=ip2int("192.0.2.100") & ip2int("255.255.255.0");  
$ip_net_string=int2ip($ip_net_num);
```

ip2int()

```
ip2int($ip_net_string)
```

Purpose

Converts a network number from IPv4 address string to an integer.

Parameters

`$ip_net_string` Network number in IPv4 format.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the `ip2int()` subroutine to convert a network number from IPv4 address string to an integer.

This subroutine is useful in situations where you want to convert IPv4 address strings to integer format so that you can easily perform logical operations with them.

Examples

The following example sets the `$ip_net_string` variable equal to the network number in IPv4 address format (192.0.2.0), converts the network number to integer format (3221225984), and then sets the `$ip_net_num` variable equal to that network number in integer format:

```
$ip_net_string="192.0.2.0";  
$ip_net_num=ip2int($ip_net_string);
```

print_parser()

```
print_parser($lines, $begin)
```

Purpose Prints the contents of the parser array.

Parameters	<code>\$lines</code>	Optional. Specifies the number of lines to print. By default, this subroutine prints all lines in the parser array.
	<code>\$begin</code>	Optional. Specifies the line in the parser array from which to start printing. The default value is 1.

Return Values This subroutine has no return values.

Usage Guidelines Use the `print_parser()` subroutine to print the contents of the parser array. By default, this subroutine prints the entire contents of the parser array. Use the optional `$lines` and `$begin` variables when you want to print only a portion of the parser array contents.

Examples **Default Printing Example**
The following example prints the contents of the parser array:

```
print_parser();
```

Printing Specific Lines Example

The following example starts at line 6 in the parser array and prints ten lines (lines 6 through 15):

```
print_parser(10, 6);
```

reset_parser()

```
reset_parser({error=>X, line=>X, match=>X})
```

Purpose Sets default values used by find_and_parse() subroutine.

Parameters	error=>X	Optional. Error hash, where <i>X</i> is the default error.
	line=>X	Optional. Line hash, where <i>X</i> is the cursor position.
	match=>X	Optional. Match hash, where <i>X</i> is the default match type.

Return Values This subroutine has no return values.

Usage Guidelines Use the reset_parser() subroutine to set default values used by the find_an_parse() subroutine.

Examples The following example set the default error to WARNING, the cursor position to line 0, and the default match type to regular expression:

```
reset_parser({error=>WARNING, line=>0, match=>-re})
```

set_config_check()

```
set_config_check(@list)
```

Purpose

Configures a global array of exception strings that can be used by the `config()` and `cmd_config()` subroutines.

Parameters

@list	Array of exception strings.
-------	-----------------------------

Return Values

This subroutine has no return values.

Usage Guidelines

Use the `set_config_check()` subroutine to configure a global array of exception strings that can be used by the `cmd_config()` and `config()` subroutines. This is useful in situations where you know that sending a command to a device will result in the device returning an informational statement that would cause the `cmd_config()` and `config()` subroutine to fail. You can create an array of those informational statements and use the `set_config_check()` subroutine to globally make them exempt from the search process. For more information, see the [cmd_config\(\)](#) and [config\(\)](#) subroutines.

Examples

The following example creates a string array and then uses that array to configure a global array of exception strings that can be used by the `cmd_config()` and `config()` subroutines:

```
@list = (
    "Access Rules Download Complete: Memory Utilization:.*",
    "Warning: VLAN .* is not configured.",
    "Security level for .* changed to .*",
    "ERROR: Interface \'vlan.*\' already exists",
    "VLAN .* is not assigned a name yet.",
    "INFO: Security level for \".*\\" set to .* by default.",
    "INFO: converting \'fixup protocol .*\' to MPF commands",
    "WARNING: \<.*\> found duplicate element"
);
set_config_check(@list);
```

set_default_timeout()

```
set_default_timeout($seconds)
```

Purpose Sets the global default timeout interval.

Parameters `$seconds` Time interval, in seconds, before a timeout occurs.

Return Values This subroutine has no return values.

Usage Guidelines Use the `set_default_timeout()` subroutine to set the global default timeout interval.

Examples The following example sets the global default timeout interval to 50 seconds.

```
set_default_timeout(50);
```

set_line()

```
set_line ($line, $str)
```

Purpose

Moves the cursor to the specified line in the parser array, and optionally replaces the contents of that line with the specified string.

Parameters

\$line	Line number in the parser array.
\$str	Optional. String variable used to replace the contents of a line in the parser array.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the set_line() subroutine to move the cursor to the specified line in the parser array, and optionally replace the contents of that line with the specified string.

Examples

Basic Set Line Example

The following example moves the cursor to line 5 in the parser array, and sets the value of the \$current_line variable to 5.

```
$current_line=set_line(5);
```

Set Line and Change Line Contents Example

The following example moves the cursor to line 48 in the parser array, and changes the contents of line 48 to "vlan 102".

```
set_line(48,"vlan 102");
```

set_prompt()

```
set_prompt($prompt_re)
```

Purpose

Sets the CLI prompt to a format as specified by a string or regular expression.

Parameters

\$prompt_re	String or regular expression that defines the CLI prompt.
-------------	-----------------------------------------------------------

Return Values

This subroutine has no return values.

Usage Guidelines

Use the set_prompt() subroutine to set the CLI prompt to a format as specified by a string or regular expression.

Examples

The following example sets the CLI prompt to the format specified by the \$prompt_re variable:

```
$prompt_re='\r\n^.*[%>$#]\s{0,1}$';  
set_prompt($prompt_re);
```

set_prompt_to_cisco_ios()

set_prompt_to_cisco_ios()

Purpose

Sets the CLI prompt to work with the Cisco IOS format.

Parameters

This subroutine has no parameters.

Return Values

This subroutine has no return values.

Usage Guidelines

Use the set_prompt_to_cisco_ios() subroutine to set the CLI prompt to the Cisco IOS format.

Examples

The following example sets the CLI prompt to work with the Cisco IOS format.

```
set_prompt_to_cisco_ios()
```

set_prompt_to_linux()

```
set_prompt_to_linux()
```

Purpose Sets the CLI prompt to work with the Linux format.

Parameters This subroutine has no parameters.

Return Values This subroutine has no return values.

Usage Guidelines Use the set_prompt_to_linux() subroutine to set the CLI prompt to the Linux format.

Examples The following example sets the CLI prompt to work with the Linux format:

```
set_prompt_to_linux()
```

wait_sec()

```
wait_sec($seconds)
```

Purpose Pauses a macro for a specified time interval.

Parameters \$seconds Time interval, in seconds, to wait.

Return Values This subroutine has no return values.

Usage Guidelines Use the wait_sec() subroutine in instances where the macro must wait for a specified time interval before continuing to run.

Examples The following example pauses a macro for 10 seconds:

```
wait_sec(10)
```