# CA ACF2™ for z/ VM

## Command and Diagnose Limiting Guide

**r12 SP4**

# Contact CA Technologies

**Contact CA Support**

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

**Providing Feedback About Product Documentation**

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

# Documentation Changes

The following documentation updates have been made to indicate support for z/VM Version 6 Release 3.0:

- Command Models
- Control Statements

# Contents

## Chapter 3: Using the ACF Command (CMDLIM Setting) 49



## Chapter 4: Using the ACF Command (DIAGLIM Setting) 63

# Chapter 1: Introduction

This guide provides you with guidelines to follow when writing command limiting rules, including the basic structure of a command limiting rule set. We explain how to use the ACF command to create, display, change, and delete command limiting rule sets. We provide guidelines for protecting the VM spooling subsystem and the licensed VM/Directory Maintenance Product. We also explain what to do if a syntax error is detected and how to override the defaults. We also explain transposition routines and syntax model command language.

This guide also explains how to write rules to limit who can issue specific diagnose instructions. We show you how to use the ACF command to process these rules, including compiling, testing, storing, decompiling, and deleting these rules. We also provide special CA ACF2™ for VM (CA ACF2 for z/ VM) diagnose codes.

This section contains the following topics:



## Audience

This guide is targeted to users who are responsible for the following tasks:

- Limiting who can execute certain CP commands and diagnose instructions
- Controlling how syntax error processing is handled for command limiting
- Setting up transposition routines
- Command limiting and logging VM directory commands
- Modifying, creating, and deleting syntax models
- Converting command limiting and diagnose limiting rules to include the MDLTYPE.

# Required Reading

## IBM Publications

We recommend you have these IBM publications for reference:

| Guide | Number |
|---|---|
| *CP Command Reference for General Users* | SC19-6211 |
| *VM/Directory Maintenance Program Product for General Users* | SC20-1839 |
| *VM/Directory Maintenance Program Product: Installation and System Administrator's Guide* | SC20-1840 |

## CA Publications

We also recommend you be familiar with the following CA ACF2 for z/ VM documentation:

| Guide | Description |
|---|---|
| *Administrator Guide* | This guide describes all the various subcommands of the ACF command. You should be familiar with basic CA ACF2 for z/ VM concepts, such as the User Identification string (UID), production by default philosophy, and components of CA ACF2 for z/ VM. |

# Command Notation

This guide uses the following command notation. Enter the following exactly as they appear in command descriptions:

| Type of Characters | Description |
|---|---|
| UPPERCASE | Identifies commands, keywords, and keyword values that you must code exactly as shown. |
| MIXed Cases | Identify command abbreviations. The uppercase letters are the minimum abbreviation; lowercase letters are optional |

| Type of Characters | Description |
|---|---|
| Symbols | You must code all symbols, such as commas, equal signs, and slashes exactly as shown. |

The following clarify command syntax; do not type these as they appear:

| Type of Characters | Description |
|---|---|
| lowercase | Indicates a variable that you must supply. |
| [ ] | Identify optional keywords or parameters. |
| { } | Require that you choose one or more of the keywords or parameters listed. |
| underlining | Shows default values that you do not have to specify. |
| \| | Separates alternative keywords and parameters, choose one. |
| ... | Means you can repeat the preceding items or group of items more than once. |

| Sample Command | Explanation | |
|---|---|---|
| ACFNRULE{ruleid\|KEY(ruleid)} | ACFNRULE | Command abbreviation. |
| TYPE(rsrctype)]- | TYPE | Optional value you can specify. |
| {[ADD(ruleentry)…]- | ADD | Optional keyword. |
| [DELETE(ruleentry)…]}- | DELETE | Optional keyword. |
| [LIST\|NOLIST] - | LIST | Default, you do not have to specify. |
| [VERIFY\|NOVERIFY] | VERIFY | Default, you do not have to specify. |

# Chapter 2: Rule Writing Guidelines

This chapter explains what command limiting rules are, how to use them, how to write them, and how to use masking. Read this chapter thoroughly before writing command limiting rules.

This section contains the following topics:



## What Is Command Limiting?

In a VM environment, the Control Program (CP) component of the operating system controls the operation of the CPU. Privileged VM users can control, modify, and display sensitive portions of the CP through a set of powerful commands, called CP commands. CA ACF2 for z/ VM command limiting is a way to control who can execute specific operands of CP commands.

The CP command classification scheme is a standard security facility of VM. It lets you classify a user to CP. By default, CP classifies users through seven nonhierarchical privilege classes, ranging from the highest level of A (defining a system operator) to G (general user). CA ACF2 for z/ VM does not interfere with the normal CP privilege class security. Normal privilege class validation is done even with command limiting active. CA ACF2 for z/ VM command limiting provides a finer degree of control over the execution of CP commands. Some of the benefits you receive from command limiting include:

- Controlling the use of individual command operands. For example, the CHANGE and TRANSFER commands let users manipulate reader, punch, and printer files. You control user access to files on the VM spooling system.
- Preventing users from executing a particular CP command without having to modify CP. For example, all class G users can normally issue the TRACE command. Command limiting can prevent users from executing TRACE and other powerful CP commands.

- Command limiting controls are very flexible. For example, you can:
  - Log the execution of powerful commands and operands to maintain a complete audit trail of their use
  - Select the particular CP commands that you want to control.

# Components of Command and Diagnose Limiting

The major components of command limiting are listed below. Subsequent chapters in this guide explain each component in depth.

## Command Limiting Rules

Command limiting and diagnose limiting rules are stored on the CA ACF2 for z/ VM Infostorage data base. These rules, similar to access rules, describe the environment where a particular CP command is executed. The environment criteria include a combination of command operands present when a command is issued and the User Identification string (UID) of the user that issued the command.

## ACF Commands

To maintain command limiting rule sets, use the following ACF subcommands:

**COMPILE**

Converts a rule set into the CA ACF2 for z/ VM format

**DECOMP**

Lists a previously stored rule set

**DELETE**

Removes a rule set

**STORE**

Stores a set of compiled rules on the Infostorage database

**TEST**

Tests the correctness of a rule set.

## Command Models

Command models are an important component of command limiting. These models describe the valid syntax, format, and operands of a CP command to CA ACF2 for z/ VM. One command model exists for each CP command. However, there can be multiple command models for the same CP command with different operating system and release identifiers.

For example, the following command limiting rules can coexist:

```
$KEY(IPL) MDLTYPE(510)
$KEY(IPL) MDLTYPE(520)
$KEY(IPL) MDLTYPE(530)
$KEY(IPL) MDLTYPE(540)
$KEY(IPL) MDLTYPE(610)
$KEY(IPL) MDLTYPE(620)
$KEY(IPL) MDLTYPE(630)
```

While the $KEY identifying the IPL command is the same, the MDLTYPEs identifying the operating system and release are different.

We supply command models for each standard CP command. Typically, they are compiled and stored on the Infostorage database during the initial installation of CA ACF2 for z/ VM. You can modify these models. For information about using MDLTYPE, see the chapter "Syntax Model Command Language."

## Command Limiting Journal

The ACFRPTCL report formats the logging and violation records that are written when CA ACF2 for z/ VM validates the execution of a command. For more information about this report, see the R*eports and Utilities Guide*.

## Transposition Routines

Most CP commands accept many different types of operands that are translated (or transposed) in some way when CP interprets them. Transposition routines convert CP command operands and run time values into a common variable for rule writing. This lets you write general rules so you do not have to account for every command variation. You should be aware of the role transposition plays during command validation. Knowing the operands transposed and how they are transposed makes it easier to write rules that require minimal future maintenance while providing a high degree of CP command security.

As an example of how transposition routines work, assume you want to write a rule to allow the user ID MAINT to execute the ATTACH command. You create the following rule entry:

```
*- TO OWNER AS *- UID(MAINT) ALLOW
```

This rule entry is transposed to:

```
*- TO MAINT AS *- UID(MAINT) ALLOW
```

MAINT then issues the following command:

```
ATTACH 0381 TO * AS 381
```

CA ACF2 for z/ VM interprets this command as:

```
ATTACH 0381 TO MAINT AS 0381
```

Comparing the command to the rule, CA ACF2 for z/ VM makes the following determinations:

- 0381 matches any mask (*-)
- TO matches the keyword TO
- User ID MAINT matches MAINT
- AS matches the keyword AS
- 0381 matches any mask (*-)
- The command issuer (MAINT) matches UID(MAINT).

Since all operands and the user ID match the rule, CA ACF2 for z/ VM now checks the authorization that was specified in the rule entry (ALLOW) and the command is allowed.

# Who Can Write Command Limiting Rules?

Normally, an unscoped security officer (a user with the SECURITY privilege and no defined scopes) is responsible for writing command limiting rules.

# Command Limiting and CP Special Considerations

VM logon IDs can affect command limiting and CP command processing. As a general rule, do not assign user IDs that can be mistaken for a CP command or command operands. Do not assign a user ID of one to four numeric characters that could be interpreted by CP as a spool ID instead of a user ID. For example, a user ID of READER could be interpreted literally by CP as the reader queue or someone named Reader. So if a user wanted to see if READER was logged on, he would enter Q READER. However, instead of responding with information about the user ID, CP checks the reader queue of the user who issued the command.

Also be aware that you can command limit the ACFSERVE QUERY STATUS command. This command limiting prevents users from executing CA ACF2 for z/ VM reports, utilities, or the ACF command. If you command limit this ACFSERVE command, be sure to give responsible individuals the authority they need to perform their tasks.

# Basic Rule Set Structure

A key that is the full name of a CP command identifies a command limiting rule set. For example, the command limiting rule set shown below applies to the CP SPOOL command:

```
A $KEY(SPOOL) MDLTYPE(530)

B CON PURGE UID(****OPR) LOG

C CON START UID(****OPR) ALLOW

D PRT COPY - ALLOW

E PRT RSCS ALLOW
```

**A**

Specifies the full name of the CP command the rule set applies to.

**B**

Logs each time OPR issues a SPOOL CONSOLE PURGE command.

**C**

Lets OPR issue a SPOOL CONSOLE START command.

**D**

Lets all users issue a SPOOL PRT COPIES *nnn* command. The dash (-) after the COPY operand says that you can specify any number of copies. You could have included a value in the rule entry if you wanted to limit the number of copies a user could print (for example, COPY 5).

**E**

Lets all users issue a SPOOL PRT RSCS command to send printed output through the Remote Spooling Communications Subsystem (RSCS).

# How Commands Are Validated

To write effective command limiting rules, you must understand how rule entries are sorted and interpreted. You must also understand the matching environment concept. These concepts and the command validation process are explained in the following sections.

## Rule Entry Sorting

Rule entries are automatically sorted from most specific to most general. CA ACF2 for z/VM sorts command limiting rule sets as follows:

**Operands**

Alphabetically, masked then unmasked

**UID**

UID of the users the rule applies to

**SHIFT operands**

In alphabetical order, with ”none specified” last

**SOURCE operands**

In alphabetical order, with ”none specified” last

**UNTIL|FOR**

Last Gregorian date the rule is valid on.

Rule entries are sorted this way to make rule interpretation more efficient and rule writing simpler. To interpret a sorted rule, CA ACF2 for z/ VM simply compares the first rule entry to the actual user request, then the second, third, and so on.

The $NOSORT control statement stores and interprets rule entries in the exact order they are entered in the rule set. We do not recommend using $NOSORT. If you must use it, be sure to test the unsorted rule carefully to ensure it works as expected.

## Matching Environment Concept

A rule match occurs when the operand values defined in a rule entry correspond to the actual environment of the user's command request. The first rule entry that matches determines if the command is allowed to execute, allowed but logged, or prevented and logged.

To summarize, CA ACF2 for z/ VM validates access to a CP command and operand combination by:

- Searching for a command model that matches the syntax and format of the command the user issued. This command is then converted into the format CA ACF2 for z/ VM requires and the operands are sorted as defined in the command model. If no command model is found for a particular command, the MODE field of the CMDLIM VMO record defines how the command request is handled.
- Checking each entry in the rule set for a match of the UID, SHIFT, SOURCE, and UNTIL dates. If CA ACF2 for z/ VM finds no matching entry, the MODE field of the CMDLIM VMO record defines how the command request is handled.

  When CA ACF2 for z/ VM finds a matching rule entry, it checks the entry to determine if the command format in the rule entry matches the command entered by the user. If the formats match, the user's command is compared to the rule entry.
- Interpreting the first rule entry that matches the user's access environment (command format, UID, SHIFT, SOURCE, UNTIL, and operands). The command then executes, executes but is logged, or is denied and logged, based on the access permission values in the matching rule entry. If CA ACF2 for z/ VM finds no matching rule entry, the MODE field of the CMDLIM VMO record defines how the command request is handled.
- If the command affects a spool file, checking the system spool to ensure the user is authorized to manipulate all of the files in the scope of the command. For more information, see Command Limiting the CP Spooling System (see page 75).

# Syntax of a Rule Set

All command and diagnose limiting rule sets consist of three parts: Control statements, rule entries, and access permissions. The full syntax of a command limiting rule set is shown below.

# Control Statements

The control statements identify the command and diagnose limiting rule set and determine some rule set characteristics. The control statements and the syntax rules for coding them are:

```
  $KEY(command|diagnose) [MDLTYPE(mdltype)]
  $MDLTYPE(mdltype)
[ $MODEL(model)               ]
[ $MODE(QUIET|LOG|WARN|ABORT) ]
[ $NOSORT                     ]
[ $OWNER(ownerid)             ]
[ $USERDATA(localdata)        ]
[ %CHANGE uid|uidmask         ]
```

- Each control statement must begin in column one.
- The $KEY control statement is the only required control statement.
- You can use any number of $ or % control statements. If you use the same type of $ control statement more than once, CA ACF2 for z/ VM uses only the last control statement of that type. Enter all $ control statements before any rule entries in the rule set.
- Comment statements begin with an asterisk (*) in column one and can be anywhere in the input. They let you place text inside an uncompiled rule set. This text is lost when the rule set is compiled.
- You can continue all input to the compiler on multiple statements by using a dash (-) as the last nonblank character on the line. If you continue a %CHANGE statement, the next line is treated as a continuation of the %CHANGE control statement, even if that line has the format of another control statement.

You can specify the following control statements in a command and diagnose limiting rule set:

**$KEY(command|diagnose)**

Supplies the full name of the CP command or diagnose code. You cannot mask this name.

**$MDLTYPE(mdltype)**

Specifies a three-character name that identifies the appropriate command model. If you do not specify the $MDLTYPE as an operand of the $KEY control statement, you can specify it as a separate control statement.

Each command model defines a valid syntax of a CP command for the operating system and release specified in the $MDLTYPE. The $MDLTYPE lets you write rules for the same CP command for different multiple operating systems and releases. Doing so lets you create and test command limiting rules under operating systems other than the current release running at your site.

You can also use the $MDLTYPE to separate rule sets that apply to different CPUs in an CA ACF2 for z/ VM shared database environment. You can then create separate rules for the same diagnose for different CPUs.

We supply command model files for all supported releases of the VM operating system. Their file type is always MODEL. For more information about the command model files, see the *Installation Guide*.

For example:

- **ZVM510**—z/VM Version 5, Release 1.0 systems
- **ZVM520**—z/VM Version 5, Release 2.0 systems
- **ZVM530**—z/VM Version 5, Release 3.0 systems
- **ZVM540**—z/VM Version 5, Release 4.0 systems
- **ZVM610**—z/VM Version 6, Release 1.0 systems
- **ZVM620**—z/VM Version 6, Release 2.0 systems
- **ZVM630**—z/VM Version 6, Release 3.0 systems
- The models for DirMaint are located in a separate file, depending on the release of DirMaint you are running. For example, DIRMR410 contains the model for VM/Directory Maintenance Function Level 410.

If you do not define the MDLTYPE control statement, CA ACF2 for z/ VM command limiting uses the default MDLTYPE defined in the CMDLIM VMO record. Diagnose limiting uses the default MDLTYPE defined in the DIAGLIM VMO record.

**$MODEL(model)**

Specifies the name of a syntax model description record used when compiling this rule set. The $MODEL control statement is optional. You should not specify it when a syntax model exists that has the same name as this rule's $KEY. The $MODEL control statement is designed for use by NEXTKEY rule sets, where you define the $KEY of the NEXTKEY rule set and the $KEY control statement does not match a syntax model. For more information about NEXTKEY, see the Using NEXTKEY section.

**$MODE(QUIET|LOG|WARN|ABORT)**

Specifies the mode for this CP command validation. Whenever an access is denied through this rule set, the mode determines the CA ACF2 for z/ VM response. Valid modes are:

**QUIET**

Allow the access

**LOG**

Allow but log the access

**WARN**

Allow the access but issue a warning message

**ABORT**

Abort and log the access attempt.

If a rule entry does not permit the request, it is aborted. An exception is how the SYNERR field overrides the mode value. For more information about this field, see the chapter "Controlling Syntax Error Processing for Command Limiting." For more information about the $MODE control statement, see the *Administrator Guide*.

**$NOSORT**

Prevents standard CA ACF2 for z/ VM sorting of command limiting rules when you store a rule set. For more information about the $NOSORT control statement, see the *Administrator Guide*.

**$OWNER(ownerid)**

Provides an information-only field of up to 24 characters. For more information about the $OWNER control statement, see the *Administrator Guide*.

**$USERDATA(localdata)**

Specifies any text string of up to 64 characters that is stored with the rule set. For more information about the $USERDATA control statement, see the *Administrator Guide*.

**%CHANGE (uid|uidmask)**

Specifies a UID string or UID mask. This mask lets a SECURITY privileged user delegate the authority to change and recompile the rule set to other users through a UID string or UID string mask. For more information about the %CHANGE control statement, see the *Administrator Guide*.

## Access Environments

Individual command limiting rule entries follow the control statements in a rule set and specify the environment and access permissions when a CP command is executed. Each rule entry describes a unique access environment. When the actual user request matches the access environment defined in a rule entry, that rule determines if the command is executed, executed but logged, or not executed.

The syntax rules for individual rule entries and the rules for coding them are:

```
operandmask UID(uidmask) SHIFT(shift) SOURCE(source) -
UNTIL(date)|FOR(days) DATA(userdata) NEXTKEY(nextkey)
```

- A rule entry can span multiple lines. Each line is normally 72 characters, although the compiler honors the logical record length of the file.
- Start rule entries in column two. This avoids having a rule entry treated as a comment when that entry begins with an asterisk.
- A dash (-) at the end of a line unconditionally continues a rule entry from one line to the next. For example, if a dash appears at the end of a rule entry and the next line contains a comment, the comment is assumed to be a continuation of the rule entry.

Use the following parameters to specify the access environment:

**operandmask**

Defines a unique combination of CP command operands. For example, when you enter a command, you can specify one or more command operands, such as IPL CMS. In this case, CMS is an operand and becomes part of your access environment. The UID keyword must follow the last operand in this mask. CA ACF2 for z/ VM treats any other rule entry keywords found before UID(uidmask) as operandmask operands. You can mask operands. For information about masking operands, see the Operand Masking Techniques section.

**UID(uidmask)**

Specifies the UID strings of users this rule entry applies to. This parameter is required and must follow the operandmask because it acts as the ending delimiter of the operandmask. For more information about this control statement, see the *Administrator Guide*.

**SHIFT(shift)**

Specifies the name of the shift record on the Infostorage database that applies to this rule entry. It defines days, dates, and times when access is allowed. If you do not specify this parameter, any access the rule indicates is appropriately allowed, logged, or prevented for all days, dates, and times. This parameter is optional.

**SOURCE(source)**

Specifies an input source or source group name where this rule should apply. For example, you can specify a terminal ID. The access is allowed only if the user is logged onto the specific terminal. If you do not specify a source, any input source is valid. Ask your Security Administrator for a list of valid group names. This parameter is optional.

**UNTIL(date)**

Specifies the last date this command limiting rule applies. For more information about this control statement, see the *Administrator Guide*.

**FOR(days)**

Specifies the number of days this command limiting rule applies. For more information about this control statement, see the *Administrator Guide*.

**DATA(userdata)**

Specifies any character string up to 64 characters. This string is retained with the rule entry. For more information about this control statement, see the *Administrator Guide*.

**NEXTKEY(nextkey)**

Specifies the rule ID of the next (or alternate) rule set that will be checked for this access. If CA ACF2 for z/ VM denies access to this command based on the rule set environment and access permissions in the original rule, CA ACF2 for z/ VM proceeds to the rule specified in the NEXTKEY operand for further checking.

## Access Permissions

A rule entry contains one parameter that specifies the action CA ACF2 for z/ VM takes when an access environment matches the environment defined in a rule entry. If your request to execute a CP command does not match any of the environments specified in the related rule entries, the execution is usually denied, depending on the value specified for the SYNERR field. For more information about this field, see the "Controlling Syntax Error Processing for Command Limiting" chapter. The possible access permission values are:

**ALLOW**

Specifies execution is allowed if the execution attempt matches the environment.

**LOG**

Specifies execution is allowed but logged if the execution attempt matches the environment. A System Management Facility (SMF) record is written to log the event for later reporting on the Command Limiting Journal (ACFRPTCL).

**PREVENT**

Specifies execution is denied if the execution attempt matches the environment. A System Management Facility (SMF) record is written to log the event for later reporting on the Command Limiting Journal (ACFRPTCL).

If you do not specify ALLOW or LOG, CA ACF2 for z/ VM assumes PREVENT.

# Operand Masking Techniques

Because most CP commands are free form, you can enter the command name, operand values, and keywords many different ways. Almost every command and keyword has a two- or three-character abbreviation. For most commands, you can enter operands and keywords in any order. In addition, there are various command formats associated with different CP privilege classes. In short, there is almost no end to the different ways you can enter a command.

To simplify your job as a command limiting rule writer, CA ACF2 for z/ VM always breaks down a command into a common format, described by the command model. This means you can write your rules in a defined format, even though a user can enter the command several different ways.

CA ACF2 for z/ VM lets you mask the values of UID strings and CP command operands. UID masking works the same in a command limiting rule as it does for access rules. If you are not familiar with UID masking, see the *Administrator* Guide for more information. You can also mask CP command operands. Effective operand masking is a critical element in every rule entry. CP command operand masking uses the dash (-), asterisk (*), and pseudo operand values. For information about pseudo operand values, see the Using Pseudo Operand Values in Rules section. You can combine the asterisk and dash, but you cannot mask the CP command name.

```
$KEY(SPOOL)
 PRINT - LOG
```

In the above command limiting rule, the dash (-) acts as a mask. The rule applies to the execution of any CP SPOOL command with the operand PRINT followed by zero or more valid operands. Of course, the CP SPOOL command syntax does not allow zero operands in this case.

The following tables illustrate how to use the dash (-) and asterisk (*) for masking keyword operands. For information about masking operands in a transposition routine, see the Using Pseudo Operand Values in Rules section.

Below are examples of masks for operands:

| Mask | Description |
|---|---|
| - | Masks all operands. Example:<br>- UID(*) ALLOW |
| - operand | Masks all operands before the specified operand. Example:<br>- CLASS A UID(*) |
| - operand - | Masks all operands, except for the one specified. Example:<br>- CLOSE - |

| Mask | Description |
|---|---|
| operand - | Masks all operands after the specified operand. Example:<br>T3380 - UID(*) ALLOW |
| *- | Operand mask for a single operand (recommended for leading masks that require at least one character)<br>Example:<br>TIMER *- UID(*) ALLOW |
| * | Mask for a single character operand. Example:<br>- CLASS * UID(*) ALLOW |

Below are examples of character masks in operands:

| Mask | Description |
|---|---|
| c*<br>c**<br>c**..* | Mass up to one character per asterisk (*) (can be less or none). Example:<br>abc***** = any three- to eight-character string beginning “abc” |
| c*-<br>c- | Mask any number of characters. Example:<br>abc*- = any length string beginning “abc” |
| *c<br>**c<br>*...*c | Mask one character per asterisk (*). Example:<br>*****abc = any eight-character string ending “abc” |
| c*c<br>c**c<br>c**..* | Masks one character per asterisk (*). Example:<br>***a***c = any eight-character string with “a” as the fourth character and “c” as the eighth character |
| **<br>***<br>*...* | Masks up to one character per asterisk (*) (can be less or none). Example:<br>** = Zero to two characters<br>******** = Zero to eight characters |

## Rule Masking Example

To effectively write rules, you must understand how operands relate to models and how to mask the operands. Below are two versions of the IPL command.

```
IPL 190 28 CL ATTN PARM AUTOCR

IPL 190 ATTN 28 CL PARM AUTOCR
```

Both of these commands perform the same function, even though the operands are in a different order in each command. To make rule writing easier in these instances, CA ACF2 for z/ VM sorts operands against a supplied IPL command model so they are always validated in a predictable order. To demonstrate this concept, the next three sections examine the components of command limiting:

- The syntax of a command (the human representation)
- The model (the machine representation)
- A sample rule, that CA ACF2 for z/ VM uses to govern actions.

## IPL Command Syntax

Shown below is the syntax of a typical CP command, in this case, the IPL command:

```
     { vaddr [cylno ] [CLear ]               }
     {       [number] [NOCLear]{STOP} [ATTN] }
IPL  {                         {PMA }        } [(PARM p1 [p2 [pn)]]
     {                         {PMAV}        }
     {                                       }
     { systemname                            }
```

## IPL Command Model

The following is a representation of the command model for the IPL command:

```
COMMAND IPL
  FORMAT CLASS=G
    OPERAND VCUU,4,TRAN=VCUU
    OPERAND GROUP=OPTIONS
    OPERAND GROUP=PLIST
 FORMAT END
```

```
 FORMAT CLASS=G
    OPERAND SYSNAME,8,TRAN=ANY
    OPERAND GROUP=PLIST
 FORMAT END

OPTIONS GROUP TYPE=OPTIONAL
        OPERAND LIST=((CYLNO,TRAN=HEX),        -
                      (BLOCKNO,TRAN=DECIMAL))
        OPERAND LIST=((CLEAR,2),               -
                      (NOCLEAR,3,TYPE=DEFAULT))
        OPERAND GROUP=MOREOPT
      GROUP END

MOREOPT GROUP TYPE=OPTIONAL
        OPERAND LIST=((GROUP=STOPATN),         -
                      (GROUP=PMAOPT))
      GROUP END

STOPATN GROUP TYPE=OPTIONAL
       OPERAND STOP,4
        OPERAND ATTN,4
      GROUP END

PMAOPT GROUP TYPE=OPTIONAL
       OPERAND LIST=(                          -
                    (PMA,3),                   -
                    (PMAV,4))
      GROUP END

PLIST  GROUP TYPE=OPTIONAL
           OPERAND GROUP=PARM
       GROUP END

PARM  GROUP TYPE=KEYWORD
           OPERAND PARM,4
           OPERAND GROUP=PARMOPTS
      GROUP END

PARMOPTS GROUP TYPE=OPTIONAL
           OPERAND AUTOCR,6
           OPERAND BATCH,5
           OPERAND NOSYSPROF,7
           OPERAND GROUP=CMSSEG
           OPERAND GROUP=INSTSEG
           OPERAND GROUP=SAVESYS
           OPERAND ANYTHING,236,TRAN=REST
      GROUP END
```

```
CMSSEG  GROUP TYPE=KEYWORD
          OPERAND SEG,3
          OPERAND LIST=(                      -
                        (NULL,4),             -
                        (SEGMNAME,8,TRAN=ANY) -
                       )
        GROUP END
INSTSEG GROUP TYPE=KEYWORD
          OPERAND INSTSEG,7
          OPERAND LIST=(                      -
                        (YES,3,TYPE=DEFAULT), -
                        (NO,2),               -
                        (NAME,8,TRAN=ANY)     -
                       )
        GROUP END

SAVESYS GROUP TYPE=KEYWORD
          OPERAND SAVESYS,7
          OPERAND SYSNAME,8,TRAN=ANY
        GROUP END

 COMMAND END
```

## Sample IPL Command Rule

The sample rule below was written based on the command model and IPL command syntax shown in the previous two sections:

```
$KEY(IPL)
 CMS PARM AUTOCR UID(*) ALLOW
 CMS10 - UID(QA) ALLOW
 SYSNAME - UID(MAINT) ALLOW
 *- - NOCLEAR - UID(*) PREVENT
 *- - CLEAR PARM SAVESYS SYSNAME UID(MAINT) ALLOW
 *- - CLEAR PARM SAVESYS CMS UID(HELPDESK) ALLOW
 *- - CLEAR PARM - UID(*) ALLOW
 *- - PARM BATCH UID(CMSBATCH) ALLOW
```

- In the first rule entry, anyone can issue the IPL CMS command if they specify the PARM AUTOCR parameter.
- In the second rule, only the QA user ID is allowed to IPL CMS10.
- In the third rule, user ID MAINT can IPL any system name. SYSNAME is a pseudo operand name because it is the same as the operand name specified in the model. When you use a pseudo operand name, it matches all values allowed by the transposition routine name.

- In the fourth rule, everything in all groups is masked except for NOCLEAR. Specifying NOCLEAR prevents command execution. NOCLEAR specifies the only way a user can IPL a virtual device is if they IPL with the CLEAR option.
- In the fifth rule, a user can perform all IPL functions if they specify CLEAR. Only user ID MAINT can issue SAVESYS to any SYSNAME.
- The sixth rule is very similar to the second one, except that only the user ID HELPDESK can save systems named CMS.
- The seventh rule entry allows CMSBATCH to IPL a device with any option and allows it to use the BATCH parameter.

## Rules for Operands that Have Numeric Values

Syntax Model Command Language (SMCL) is a facility that describes the syntax for a CP command. This section explains the syntax of the language so you can read a model to write command limiting rules. They also provide a more indepth knowledge of the language so you can modify the supplied models when you change the syntax of a command or need to create models and limit commands that you added to CP.

Some commands accept numeric values for operands. They also have special transposition routines that check to make sure that the numeric operand supplied is valid. Operands that fall into this category and their corresponding transposition routines are:

| Transposition Name | Operand |
|---|---|
| DECIMAL | *number* |
| HEX | *hexloc* |
| HHMM | *nnnn* |
| MMSS | *nnnn* |
| RCUU | *raddr* |
| SPOOL | *spoolid* |
| STORADDR | *hexloc* |
| STRSIZ | *nnnk* |
| VCUU | *vaddr* |
| VUR | *vaddr* |

You can mask these operands.

Following is a brief explanation for using masking as a value for operands that accept a numeric value.

- A single * means the operand is optional
- ** used for a decimal number means 0-99
- *** used for a decimal number means 0-999
- 0* used for a decimal number means 0-09

Let's use the SLEEP command as an example. To let someone sleep for 0 to 9 seconds, code the rule as 0* SEC UID(*) ALLOW. Had you just put a single *, CA ACF2 for z/ VM would interpret this as meaning any value can be present. Specify 0*, CA ACF2 for z/ VM to allow 0-09 seconds.

As another example, to let anyone in your site define 0-50 cylinders of TDISK, code the DEFINE rule as shown in the next example.

```
$KEY(DEFINE)
 T**** AS *- CYL 0* UID(*) ALLOW
 T**** AS *- CYL 1* UID(*) ALLOW
 T**** AS *- CYL 2* UID(*) ALLOW
 T**** AS *- CYL 3* UID(*) ALLOW
 T**** AS *- CYL 4* UID(*) ALLOW
 T**** AS *- CYL 50 UID(*) ALLOW
 T**** AS - - - UID(*) PREVENT
 - UID(*) ALLOW
```

Because you specified no MDLTYPE for this DEFINE rule, CA ACF2 for z/ VM uses the default MDLTYPE command model (as defined in the CMDLIM VMO record) for syntax checking. This rule specifies 0-50 cylinders of TDISK and any other DEFINEs.

## Rules for Operands That Have a Range

Some CP commands specify operands as single values or as value ranges. In addition, these commands usually have multiple formats. A good example of this is the DETACH command. Some common uses of DETACH are:

- Detach a single device, such as DETACH 0191
- Detach many devices, such as DETACH 0191 0492 0399
- Detach a range of devices, such as DETACH 0191-019F
- Detach devices from a user, such as DETACH 0191 FROM USER01.

The supplied command model for DETACH includes special indicators to handle any combination of operands, including those listed above. Before showing sample rules for operand ranges and addresses, you should understand that rule operand masking is slightly modified to handle device addresses. Consider the mask 019*. To ensure only valid addresses are used, a transposition routine modifies this standard masking technique. For device addresses, the * is a position holder and the operands going to it are numeric values. This means an operand mask of 019* is treated as 0190-019F (that is, 019G-019Z is invalid, as it should be).

Under normal masking conventions, the trailing * means zero or one character must be present to match the mask. In normal cases, this enables a singular address of 019 or an address range of 0190-019F or 019G-019Z. However, 019 and 019G-019Z are not valid because all valid device addresses are *nn*0 through *nn*F, where *nn* is any value between 00 and FF. Further, CA ACF2 for z/ VM considers an * in the low order portion of the range to be a low-value (0) and considers an * in the high order portion to be a high-value (F). For example, *19-1A* is transposed into a range of 019-1AF. For more information about how ranges are transposed, see the appendix “Transposition Routines for Command Limiting.” To demonstrate how this masking works, consider the following rule entries for the DETACH command.

```
$KEY(DETACH)
 0190 UID(*) ALLOW
 019* UID(*) ALLOW
 0190-01AF UID(*) ALLOW
 0190 0191 019D 01B0 UID(*) ALLOW
 *- UID(*) ALLOW
 -  UID(*) ALLOW
```

- In the first rule entry, one operand is allowed, but it must be 0190.
- In the second rule entry, one operand or a range of operands is allowed. The range for matching purposes is 0190-019F.
- In the third rule entry, one operand or a range of operands is also allowed. The range for matching purposes is 0190-01AF.
- In the fourth rule entry, you can specify 0190 0191 019D 01B0, but they must appear in that order. These singular values are treated as AND situations, meaning this operand and that operand must be present for a match.
- In the fifth rule entry, only one operand is allowed.
- In the last rule, zero or more operands are allowed.

The next series of examples shows how the ATTACH command uses device ranges. In this first example, only OPRLEAD1 can issue all forms of the ATTACH command.

```
$KEY(ATTACH)
 - UID(OPRLEAD1) ALLOW
```

In the next example, user MAINT can ATTACH tape drives 0581 and 0583 to any virtual machine. The TO is not required in the rule because it is a default in the command model. Here it clarifies the example.

```
$KEY(ATTACH)
 0581 TO - UID(MAINT) ALLOW
 0583 TO - UID(MAINT) ALLOW
```

In the next example, all system operators can ATTACH devices to the SYSTEM, but these commands are logged.

```
$KEY(ATTACH)
 - TO SYSTEM AS - UID(OPERATOR) LOG
```

In the next example, no users can ATTACH a volume as a 3300V device.

```
$KEY(ATTACH)
 - TO - AS 3330V UID(-) PREVENT
```

## Using Pseudo Operand Values in Rules

Another rule writing technique lets you specify a pseudo operand in a rule. This means you can specify a fixed name in a rule for operands that have variable values and let any operand match, as long as you specify a valid value.

You can use a pseudo operand name for any command operand that contains a variable. CA ACF2 for z/ VM provides transposition routines to allow you to use pseudo operands. You can determine the commands to have transposition routines by examining the command model. If the operand is defined with a TRANS=routine verb, then it is a variable. For more information, see the appendix “Transposition Routines for Command Limiting.”

We can use the READY command to illustrate a simple rule.

```
$KEY(READY)
 VCUU ALLOW
```

In the above rule, VCUU is the pseudo operand. It says that any virtual device address specified as an operand of the READY command matches the rule. Granted, this is a simple example where a dash rule also works, but consider how useful a pseudo operand can be in a SPOOL command rule.

For example, when writing a rule for the SPOOL command, you probably do not always know what address the user's reader is at because he could issue a SPOOL 00C class G command. It is difficult to account for all possible device addresses or for the possibility that he could issue a DEFINE command to change the address. But through the pseudo operand technique, you can determine whether the device address is RDR, PRT, PUN, or CON. So when writing a rule, you could simply say RDR, controlling all the different types of readers.

```
$KEY(SPOOL)
 RDR - ALLOW
```

The next example illustrates how pseudo operands can specify unit record type devices:

```
$KEY(BACKSPAC)
 PRT FILE UID(OPRLEAD) LOG
 PRT ***** *** UID(OPRLEAD) ALLOW
 PUN - UID(OPRLEAD) ALLOW
```

- $KEY(BACKSPAC) identifies the command.
- In the first rule entry, PRT indicates this rule applies to all real printers. This is an example of using a pseudo operand in a rule. In this example, PRT matches any real device (*raddr*) that is attached as a printer. UID(OPRLEAD) indicates lead operators (any ID that starts with OPRLEAD) can execute a BACKSPAC *raddr* FILE command, but the execution is logged. FILE indicates this rule applies when you use the FILE operand. This becomes part of the command environment. LOG allows the command, but logs the event.
- In the second rule entry, all lead operators are allowed to issue all other forms of BACKSPAC PRT. ****** *** indicates this rule applies for any operand combination and that these operands are optional. ALLOW allows the command.
- The third rule entry applies to punch devices. It is almost identical to the third rule entry, except we use the pseudo operand PUN.

## Rules for Commands with a Password

Some commands, like AUTOLOG, often require a password operand. Naturally, you do not want to put a clear text VM directory password in a rule. However, you might want the rule to force the user to enter the password.

The rule set below shows how to do this.

```
KEY(AUTOLOG)
SMAINT *- - UID(MAINT) ALLOW
- UID(OPRLEAD1) ALLOW
- UID(OPRLEAD2) ALLOW
- UID(OPRLEAD3) ALLOW
```

The first line identifies the command. If SMAINT is not defined in the VM directory, a syntax error occurs. In the second rule entry, MAINT can issue an AUTOLOG for the SMAINT machine. SMAINT indicates the first operand must be SMAINT. For the AUTOLOG command, the first operand is the user ID of the machine to be autologged. *- indicates a second operand is required. The second operand is the VM directory password of the machine being autologged. - indicates the third operand of the AUTOLOG command is optional and is for variable length data that is passed to the virtual machine.

The solitary dash (-) says that all operands for the command are masked. According to these rule entries, the lead operators can execute the AUTOLOG command.

Entering a password on the command line might conflict with a CP operating requirement that you must enter passwords on a separate line in nondisplay mode. For more information, contact your systems programmer.

## Rules for Storage Type Commands

The DCP, DMCP, and STCP commands are protected at the operand level. Be aware that special transposition routines are maintained because these commands are complex and do not follow the strict dependencies that are usually found in the other commands.

The STORE, DISPLAY, and PER commands are protected only at the command name level. These commands are not protected down to the operand level for the storage address type fields. These are virtual storage type commands that can easily be replicated by any assembler program or the CMS DEBUG facility.

## Rules for Repeating Operands

There are some commands whose formats can repeat themselves or are actually compound commands, such as DCP ml0.10 nl0.10. This is called a compound command. When CP executes a compound command, the repeating operands are actually treated as separate commands.

```
DCP ml0.10
DCP nl0.10
```

To make rule writing more consistent and easy to follow, CA ACF2 for z/ VM also treats this as two separate commands and passes the command through the rule set twice, as if they were separate. After separating the commands, only those commands that are authorized by a rule are allowed. For example, a user might be authorized to issue DCP m10.10, but not authorized to issue DCP n10.10.

You can determine if a command repeats by examining its command model. If the REPEATS verb is specified in the FORMAT or GROUP clause, it is a repeating command. For more information, see the “Using the Model Setting” chapter.

## Rules for Defaults from Other Operands (VALUEFOR)

Commands can obtain a default value for an operand when the command is executed, as in the ATTACH command, shown below.

```
ATTACH B8A TO USERA
ATTACH B8A TO USERA AS B8A
ATTACH B8A TO USERA AS BA8
```

- In the first line, real device B8A is attached to USERA as virtual device B8A. The virtual device address is obtained from the real device address since none was specified. You can think of the real device address as being a value for the virtual device address.
- The next example (second line) is the same as the first, except that the virtual device address is explicitly specified.
- The third example specifies a virtual device address that is different than the real device address.

When a command such as ATTACH is processed against an operand with a VALUEFOR clause, the command limiting interpreter fills in the default value. This is the way CP would behave, as in the first command example above. However, when you are writing rules, the VALUEFOR has no special meaning. You must specifically write rules to protect its object. If this was not true, the following rule would let TLCAMS issue the third sample command.

```
$KEY(ATTACH)
 B8A - UID(TLCAMS) PREVENT
 - UID(*) ALLOW
```

In the above rule, we want to prevent TLCAMS from issuing the ATTACH command against the real device B8A, regardless of his privileges (CP or CA ACF2 for z/ VM).

To protect specific values of an object of a VALUEFOR, you must write specific rules. To specifically control what virtual device address a user can specify for a particular device, examine the following rule.

```
$KEY(ATTACH)
 58* TO *- AS 18* UID(TLCPAM) ALLOW
 - UID(*) ALLOW
```

In the above example, user TLCPAM can issue the ATTACH command against devices 580-58F. He can attach them to anyone as long as they are attached as virtual addresses 180-18F.

# Using NEXTKEY

The NEXTKEY operand splits a very large rule set into several sets. Specify the rule ID of an alternate rule set in the NEXTKEY operand. If the environment and permissions of the current rule set prevents access to a command, CA ACF2 for z/ VM searches the alternate rule set specified through NEXTKEY.

When NEXTKEY specifies an alternate rule set, a security administrator must grant authority to whoever is responsible for writing and maintaining that command limiting rule set through the %CHANGE control statement. This control statement must reside in the alternate rule set that the security administrator must initially establish.

For a %CHANGE authorization to be active, you must specify the CHANGE operand of the RULEOPTS VMO record. If you specify NOCHANGE, all %CHANGE authorizations are inactive. The default is CHANGE. This default is required to use %CHANGE.

## Splitting Rule Sets

As stated before, the NEXTKEY operand splits a command limiting rule set. This may be necessary to selectively delegate rule maintenance (%CHANGE) authority. Or you may need to use it if a rule set is very large and exceeds the 4K physical storage size limit.

For example, you can have several entries for a rule set, all under the same CP command, ATTACH. The NEXTKEY feature can redirect or split the rule set for ATTACH into smaller sets as follows:

```
$KEY(ATTACH)
 580 TO USERID AS *- NEXTKEY(A)
 581 TO USERID AS *- NEXTKEY(B)
 582 TO USERID AS *- NEXTKEY(C)
```

In the above example, USERID is a pseudo-operand that matches any user ID. The first three rule entries specify the NEXTKEY rule sets to validate access to real devices 580, 581, and 582. You can then write three smaller rule sets as follows:

```
$KEY(A)
$MODEL(ATTACH)
%CHANGE SECDIR
 - UID(*) ALLOW

$KEY(B)
$MODEL(ATTACH)
%CHANGE OPSDIR
 - UID(OPR) ALLOW

$KEY(C)
$MODEL(ATTACH)
%CHANGE PRGDIR
 - UID(PRG) ALLOW
```

These rule sets let you delegate authority through the %CHANGE control statement, but they are smaller than a single rule set required for the ATTACH command. Specify the $MODEL control statement to indicate that CA ACF2 for z/ VM is to use this syntax model during compilation to syntax check the rule entries.

Examining the last three examples, the first rule set for device 580 specifies a %CHANGE control statement to allow the Director of Security (SECDIR) to change rule entries governing only device 580. The second rule set grants similar authority to the Director of Operations (OPSDIR) for device 581. The third rule set grants similar authority to the Director of Programming (PRGDIR) for device 582.

Computer operators (OPR) can only attach device 581. The second example rule set ($KEY(B)) allows this access. Similarly, programmers (PRG) can attach device 582 because of the third rule set.

The NEXTKEY operand directs CA ACF2 for z/ VM validation only when access based on the current rule set is prevented. You can have a chain of up to 25 NEXTKEY operands. If you specify more than 25, CA ACF2 for z/ VM denies access and writes a KEYEXCES violation record that appears on the ACFRPTCL report. You cannot reference the same rule set twice during a single validation. That is, the chain of NEXTKEY options cannot form a loop. If you reference the same rule set twice, CA ACF2 for z/ VM denies the access and writes a NKEYLOOP violation record to the ACFRPTCL report. For more information about ACFRPTCL and NEXTKEY reporting, see the *Reports and Utilities Guide*.

# Suggested Rules for Sensitive Commands

Implementing specific CA ACF2 for z/ VM features and carefully following certain procedures assures security requirements are met. Recommended settings for CP commands are described in the following sections.

## COUPLE Command

For two virtual machines to establish Virtual Channel-To-Channel Adapters (CTCAs) for data transfer, they both must issue the DEFINE command. (For information about the DEFINE command, see the DEFINE and DETACH Command section.) As soon as both machines establish virtual CTCAs with DEFINE, you can issue the CP COUPLE command from one of the machines to the other for CTCA data transfer.

With CA ACF2 for z/ VM command limiting, you can log and prevent the use of COUPLE to control the use of CTCAs for data transfer between two virtual machines. To implement command limiting for the CP COUPLE command, you must create a command limiting rule set with a $KEY of COUPLE. For example, to audit CTCA data transfers between virtual machines, LOG access permission is required.

```
$KEY(COUPLE)
 - UID(-) LOG
```

## DEFINE and DETACH Command

In a C2 environment, you should be able to restrict and log the introduction (and deletion) of objects into a user's address space. This means that you must be capable of both restricting objects to authorized users and auditing the use of the objects.

The DEFINE (privilege class A or B or G) and DETACH (privilege class B or G) commands control the introduction and deletion of objects into a user's address space by establishing and terminating virtual devices, such as T-DISKs (temporary minidisk storage areas).

DEFINE is a prerequisite for introducing objects through CTCAs; that is, when two virtual machines establish virtual CTCAs with DEFINE, data transfers can then take place by linking the CTCAs with the COUPLE command. (For information about the COUPLE command, see the COUPLE Command section.) When the data transfer is complete, you can issue DETACH to terminate the virtual CTCAs.

With CA ACF2 for z/ VM command limiting, you can log or prevent the use of DEFINE and DETACH to control how objects can be accessed. To implement command limiting for the CP DEFINE and DETACH commands, you must create command limiting rule sets with a $KEY of DEFINE and DETACH. For example, to audit all real and virtual devices being defined or detached, LOG access rules are required:

```
$KEY(DEFINE)
 - UID(-) LOG

$KEY(DETACH)
 - UID(-) LOG
```

## IPL Command

Preferred Machine Assist (PMA) is one of the performance enhancements available for production guest virtual machines. It lets these machines run in real supervisor state. Virtual machines that run in real supervisor state can examine or modify any storage on the machine. They can also execute I/O to any device unchecked. To activate real supervisor state, the virtual machine must have the V=R or V=F directory option. You must also IPL the system with the PMA or PMAV option.

Specify PMA as a parameter on the CP IPL command to enable PMA, such as IPL 440 PMA. This command performs an IPL from device 440 and requests PMA initialization. A PMAV option also exists that provides the same basic function of PMA with some extensions.

Absolutely no constraints apply to the OS/390 V=R guest when running through PMA; the guest has complete control of the machine until the microcode transfers control back to VM. Therefore, you might want to prevent a V=R untrusted guest from invoking PMA or PMAV.

To prevent IPL *ccu* PMA and IPL *ccu* PMAV while allowing other forms of the CP IPL command, use a rule such as the one shown below.

```
$KEY(IPL)
 - PM- - UID(*) PREVENT
 - PM- - UID(PRODSYS) ALLOW
 -     UID(-) ALLOW
```

In the above rule, only the OS/390 production user can IPL PMA or PMAV. All other users are prevented from IPLing PMA and PMAV. All other IPLs are allowed.

# LINK Command

The CP LINK command lets you link to another user's disks in various modes. These link access modes are shown below:

**R**

Primary read-only access. You can establish a read-only link only if no one else is linked to the disk in write mode. Otherwise, no link is established.

**RR**

Primary read-only access or alternate read-only access. You can only establish a read-only link, no matter what other links users have to the disk.

**W**

Primary write access. You can establish a write link only if no other user is linked to the disk. Otherwise, no link is established.

**WR**

Primary write access or alternate read-only access. You can establish a write link only if no other user is linked to the disk. Otherwise, this user is linked as read-only.

**M**

Primary multiple access. You can establish a write link unless some other user already is linked in write mode. Otherwise, no link is done.

**MR**

Primary multiple access or alternate read access. You can establish a write link unless some other user already is linked in write mode. If some other user is already linked MR, a read-only link is done.

**MW**

Primary multiple access or alternate write access. A user is established as a write link in all cases.

We suggest you limit the LINK command since it is not advisable to have more than one user updating a file at the same time.

Through command limiting, you can control who can link to your disk and the type of access mode that is valid for that user. The rule below lets TLCPJM link in primary write access mode or alternate read-only access mode to TLCAMS 191 disk. All other users whose logon ID begins with TLC can link to this disk in read-only or alternate read-only access mode:

```
$KEY(LINK)
 TLCAMS 191 - W* UID(TLCPJM) ALLOW
 TLCAMS 191 - R* UID(TLC) ALLOW
 DOC - - R* UID(-) ALLOW
 TLCAMS 191 - MW UID(-) PREVENT
 DOC - - MW UID(-) PREVENT
```

The third rule entry specifies that all users can access any DOC disk in read-only access mode or alternate read-only mode. The last two entries prevent anyone from linking multiple write to the TLCAMS 191 and any DOC disks.

Do not be confused. The LINK command only allows the link. If you have installed CMS protection, you still need to write access rules to allow users to update or read files on your disk.

## SET Command

VM has a system directory option that is designed to improve system performance. This option, known as V=F on VM systems, designates a virtual machine as the owner of real storage; it is specified in the system directory for a given virtual machine with OPTION VIRT=REAL (Virtual=Real) or OPTION VIRT=FIXED (Virtual=Fixed).

A guest operating system usually generates vast amounts of I/O operations for its own paging and application program processing. Not surprisingly, there is an accompaniment to the V=R option that shortens the path of Input/Output (I/O) processing in a V=R guest machine. This accompaniment is implemented with the CP command SET NOTRANS ON.

Normally, when a V=R guest requests some I/O operation, a string of commands is passed to the I/O device informing it what and how much data to read or write. CP scans, checks, and translates the string of commands, making sure the I/O operation request is valid for the guest. This validation checks to make certain that the return area for the data from a read I/O operation is in the virtual machine's storage. This check prevents one virtual machine from reading data into another virtual machine's storage.

This scanning and checking involves considerable overhead on the part of CP. Because a guest V=F operating system is generally considered a trusted guest (its integrity as a functional operating system can be trusted), it can avoid the translational overhead through the CP SET NOTRANS ON command. This command skips the translation process, trusting the guest to ensure that the I/O command strings are correct.

The CP SET NOTRANS ON command represents an integrity exposure. By manipulating the I/O request command string, the guest can alter the pages of real storage outside the realm of the V=R area (circumvent hardware and software storage protection mechanisms). Therefore, a V=R guest could be prevented from issuing a SET NOTRANS ON command. Obviously, you must consider the extra overhead that is incurred if you prevent SET NOTRANS ON.

You can use command limiting to prevent SET NOTRANS ON and allow other forms of the CP SET command. Create a CP SET command limiting rule with a $KEY of SET.

```
$KEY(SET)
 NOTRANS ON UID(-) PREVENT
 - UID(-) ALLOW
```

Here, all forms of the CP SET command are allowed except SET NOTRANS ON.

When the IBM SYSRES macro SYSCLR keyword is set to YES in HCPSYS, native VM CP code automatically clears temporary disk space when a user allocates it. IBM lets Class B users turn off TDISK clearing with the CP SET TDISKCLR OFF command. To avoid this integrity exposure, we strongly recommend you limit the use of this command:

```
$KEY(SET)
 TDISKCLR OFF UID(-) PREVENT
```

In the rule above, all users are prevented from executing the SET TDISKCLR OFF command.

## SHUTDOWN Command

The CP SHUTDOWN command lets a class A user end all VM system functions, disable communication lines, checkpoint the system for a warm start, save enabled virtual machines to be saved, and automatically do a warm start. We strongly recommend that you implement command limiting for the SHUTDOWN command on CPUs that do not support POWEROFF. Otherwise, an invalid SHUTDOWN command shuts down the service machine, but CP does not shut down, perhaps resulting in users or the system being hung.

To restrict use of the CP SHUTDOWN command, create a command limiting rule with a $KEY of SHUTDOWN.

```
$KEY(SHUTDOWN)
 - UID(operator) LOG
 POWEROFF UID(*) PREVENT
```

Here, only users with the OPERATOR UIDs are allowed SHUTDOWN access to the secured CPU. All such events are logged. Anyone else who attempts to SHUTDOWN this machine is prevented, with the event logged.

# Chapter 3: Using the ACF Command (CMDLIM Setting)

This chapter explains the ACF subcommands listed below. You should also review the "Rule Writing Guidelines" chapter to obtain a full understanding of how rules are interpreted. Use this chapter as a reference aid when you want to create, modify, display, test, or list command limiting rule sets.

The following commands are explained in this chapter:

**COMPILE**

Converts rule sets into the form needed by CA ACF2 for z/ VM.

**DECOMP**

Lists previously stored rule sets

**DELETE**

Deletes command limiting rule sets

**LIST**

Lists previously stored rule sets

**STORE**

Stores compiled rule sets on the Infostorage database

**TEST**

Tests the correctness of a rule set.

This section contains the following topics:

# Creating a Rule Set

You can create command limiting rule sets directly from the terminal or by first building the rule set in a file. The general procedure is:

- Build a command limiting rule set in a standard CMS file. Normally, the file name of the rule is the same as the command name. The file type is always RULE.
- From CMS, issue the ACF command.
- After issuing the ACF command, establish the CMDLIM setting (SET CMDLIM, SE CMDLIM, or T CMDLIM).

```
acf
ACF
set cmdlim
CMDLIM
```

- To interactively compile from the terminal, issue the COMPILE subcommand without a filename. The COMPILE subcommand lets you enter the control statements and rule entries at the terminal. To compile from a CMS file, issue the COMPILE subcommand with the name of the file that contains the rule set text.
- To test the rule set, issue the TEST subcommand. The TEST subcommand can give you an idea of whether the rule set does the validation of CP command execution as you intended.
- By default, the COMPILE subcommand automatically stores the rule set on the Infostorage database if the rule set was compiled from a CMS file. If rule input is entered directly from the terminal, you must issue the STORE subcommand.
- After you store a command limiting rule set, it is effective on the database and in the rule set cache.

# ACF Subcommands

You can process command limiting rules after establishing the CMDLIM setting of the ACF command.

```
acf
ACF
set cmdlim
CMDLIM
```

Under the CMDLIM setting, you can issue any of the following ACF subcommands:

- COMPILE
- CMS
- DECOMP

- DELETE
- END
- HELP
- LIST
- SET
- SHOW
- STORE
- TEST

For information about the common subcommands END, HELP, SET, and SHOW, see the *Administrator Guide*.

The other commands, specific to the CMDLIM setting, are explained in this chapter.

# COMPILE Subcommand

The COMPILE subcommand creates a set of command limiting rules. CA ACF2 for z/ VM provides two ways of compiling command limiting rule sets: directly at the terminal or using text in a CMS file as input to the compiler.

## Compiling Directly at the Terminal

You can enter a command limiting rule set from the terminal as follows:

- Enter the COMPILE subcommand without parameters**.**
- Enter the $KEY control statement on the first line.
- Enter the other control statements, each on a separate line**.**
- Enter all rule entries, each on a separate line.
- To end the rule set, press **Enter** to enter a blank line.
- The COMPILE subcommand automatically ends.
- Enter the STORE command to save the rule set on the Infostorage database.

A full example of this follows:

```
acf
ACF
set cmdlim
CMDLIM
compile
ACFpgm510I ACF compiler entered
$key(spool)
 con purge uid(****opr) log

 print copy - allow

 print rscs allow

ACFpgm551I Total record length='length' byte - 'percent' percent utilized
CMDLIM
store
ACFpgm769I Rule 'ruleid' stored
```

You should test a rule before actually storing it to determine if it performs as you intended. For examples of how to test rules, see the section Test Subcommand.

## Compiling from a CMS File

You can also enter the control statements and rule entries into a CMS file to create a command limiting rule set. The file must have a file type of RULE. Each control statement or rule entry must be on a separate line. The last line does not have to be a blank line.

```
xedit spool rule a
    (goes to edit screen)
$key(spool)
 con purge uid(****opr) log
 print copy - allow
 print rscs allow
 file (to save the file)
```

After you enter the control statements and rule entries into the file, you can invoke CA ACF2 for z/ VM and compile the command limiting rule set. Save the file. Issue the COMPILE subcommand with the name of the file.

```
acf
ACF
set cmdlim
CMDLIM
compile spool
ACFpgm510I ACF compiler entered
      ... (display of compiled rule set)
ACFpgm551I Total record length='length' byte - 'percent' percent utilized
ACFpgm768I Rule 'ruleid' replaced
```

You specify only the file name. The file type is always RULE. The rule set is compiled and, by default, stored.

## Syntax of the COMPILE Subcommand

The full syntax of the COMPILE subcommand is shown below.

```
COmpile   {*        } [ List |NOList ]
          { filename} [ Store|NOStore]
                      [ Force|NOForce]
```

Under the CMDLIM setting, the COMPILE subcommand takes the parameters listed below.

*****

Indicates that the text subsequently entered is input to the compiler. In an online environment, the system prompts you to enter the rule text directly from the terminal. Using the COMPILE subcommand without parameters is equivalent to specifying an asterisk.

**filename**

Specifies the CMS file that contains the command limiting rule text to be compiled. The file type is always RULE.

**List|NOList**

The LIST parameter displays the input to the compiler on your screen or printed on your listing during compilation of a rule set. NOLIST does not display or list. LIST is the default when you compile from a CMS file. Otherwise, NOLIST is the default.

**STore|NOStore**

The STORE parameter automatically stores the rule set at compilation time. NOSTORE does not store the rule set. STORE is the default if you compile the rule set from a CMS file. Otherwise, NOSTORE is the default.

**Force|NOForce**

The FORCE parameter stores the command limiting rule set regardless of whether it currently exists. NOFORCE stores the command limiting rule set only if it does not already exist. FORCE is the default.

When used as parameters of the COMPILE subcommand, FORCE|NOFORCE applies only to the COMPILE subcommand that you are currently issuing. When used as parameters of the SET subcommand, FORCE|NOFORCE is in effect until you change it or until you end the ACF command. It is not affected by changes in the ACF command setting.

# DECOMPILE Subcommand

The DECOMP subcommand retrieves a command limiting rule set that has been previously compiled and stored. This subcommand is useful for examining, updating, testing, or changing rule sets. You can decompile a rule set at the terminal or into a CMS file.

## Syntax of the DECOMP Subcommand

The DECOMP subcommand has the following syntax.

```
         { *             }
DEComp   { ruleid        } [ MDLTYPE(|mdltype|mdlmask)]
         { LIKE(rulemask)} [ INTO(filename)           ]
```

When you decompile the rule set into a CMS file, its record length is limited to 80 characters. Any existing rule exceeding 80 characters is broken into continuation lines (signified by a dash (-) at the end of the line). The compiler still accepts input files from previous releases that have record lengths up to 256 characters.

The DECOMPILE subcommand takes the parameters listed below.

*****

Decompiles the last rule set brought into storage since you established the CMDLIM setting.

**ruleid**

Specifies the key of an individual rule set to be decompiled or listed.

**LIKE(rulemask)**

Specifies a mask of rule IDs for decompiling a group of rule sets.

**MDLTYPE(|mdltype|mdlmask)**

Specifies the model type of the rules to be decompiled. If you used the LIKE(rulemask) parameter, you can mask the MDLTYPE, as in DEC LIKE(ATTACH-) MDLTYPE(-) INTO(RULE). If you specified MDLTYPE(), CA ACF2 for z/ VM decompiles the rule sets written under previous releases of CA ACF2 for z/ VM.

If you do not define a MDLTYPE, CA ACF2 for z/ VM uses the default as specified in the CMDLIM VMO record.

**INTO(filename)**

Specifies the name of a CMS file where the rule set is decompiled. You cannot specify the file type, because a file type of RULE is always assigned.

## How to Use the DECOMP Subcommand

Below are two examples that show how you can use the DECOMP subcommand. This first example shows how you can use DECOMP to display a rule set on your terminal.

```
acf
ACF
set cmdlim
CMDLIM
decomp spool mdltype(530)
ACFpgm762I 530 SPOOL stored by VMISO on 05/28/07-17:30
$KEY(SPOOL) MDLTYPE(530)
 CON PURGE UID(****OPR) LOG
 PRINT COPY - ALLOW
 PRINT RSCS ALLOW
ACFpgm551I Total record length=245 byte - 5 percent utilized
CMDLIM
```

The next example shows how DECOMP can write a rule set into a CMS file. Writing a rule set into a CMS file is very useful because it lets you easily change an existing rule set and then recompile it. It can also save time if you misspell a keyword.

```
acf
ACF
set cmdlim
CMDLIM
dec spool into(rulefile)
ACFpgm556I 'rule' rule 'key' stored by 'lid' on 'date'
ACFpgm551I Total record length='length' byte - 'percent' percent utilized
CMDLIM
cms xedit rulefile rule a
```

Because you did not specify MDLTYPE, this rule was decompiled with the default model as defined in your CMDLIM VMO record (in this case, SP6).

```
ACFpgm556I 'rule' rule 'key' stored by 'lid' on 'date'
$KEY(SPOOL) MDLTYPE(530)
 CON PURGE UID(****OPR) LOG
 PRINT COPY - ALLOW
 PRINT RSCS ALLOW
ACFpgm551I Total record length='length' byte - 'percent' percent utilized
```

To decompile all rules with a particular model type, enter the following command, where mdltype is the model type of the commands you want to decompile:

```
DECOMP MDLTYPE(mdltype)
```

To decompile all spool rules, use the command DECOMP SPOOL MDLTYPE( ).

# DELETE Subcommand

The DELETE subcommand removes a command limiting rule set and syntax models from the Infostorage database. To delete command syntax models, you must have set the ACF MODE to MODEL.

## Syntax of the DELETE Subcommand

The DELETE subcommand has the syntax shown below.

```
DELete   { ruleid    }     [MDLTYPE(mdltype|mdlmask)]
         { LIKE(rulemask)}
```

The DELETE subcommand takes the parameters listed below.

**ruleid**

Specifies the ID you want to delete. If you specify a MDLTYPE, only the rule for that model type is deleted.

**LIKE(rulemask)**

Deletes all rules that match the rule mask. If you do not specify MDLTYPE, only rules that match the current MDLTYPE are deleted.

**MDLTYPE(mdltype|mdlmask)**

Deletes the rule set under a specific operating system. Masking the MDLTYPE deletes rules for operating systems that match the mdlmask.

# STORE Subcommand

The STORE subcommand places a previously compiled set of command limiting rules onto the Infostorage database. You must have authority to store the rule. This authority is granted through the SECURITY privilege or through the %CHANGE or %RCHANGE control statements. If you are not authorized to store the rule, the operation is rejected.

## Syntax of the STORE Subcommand

The STORE subcommand has the following syntax:

```
STore   { Force  }
        { NOForce}
```

This subcommand accepts one parameter:

**Force**

The default of FORCE stores a rule set even it already exists. In this case, the new version of the rule set replaces the existing version.

**NOForce**

NOFORCE stores the rule set only if it does not already exist. If the rule does exist, then NOFORCE rejects the store operation.

There is a SET FORCE|NOFORCE subcommand that defaults to SET FORCE. You can use the FORCE|NOFORCE parameter of STORE to override the SET value, or use the SET subcommand to change the defaults for the STORE subcommand. For example, you could issue a SET NOFORCE that effectively changes the default of STORE FORCE to STORE NOFORCE.

## How to Use the STORE Subcommand

The example below shows how to use the STORE subcommand:

```
acf
ACF
set cmdlim
CMDLIM
compile
ACFpgm510I ACF compiler entered
$key(spool)
 con purge uid(****opr) log
 print copy - allow
 print rscs allow
 - uid(*) allow

ACFpgm551I Total record length='length' byte - 'percent' percent utilized
CMDLIM
store
ACFpgm769I Rule SPOOL stored
```

# TEST Subcommand

The TEST subcommand puts you in an environment where you can interactively test a compiled command limiting rule set. Testing can give you an idea of whether the rule set provides the protection you want.

## Syntax of the TEST Subcommand

The syntax of the TEST subcommand is shown below.

```
TEST   { *     }    [MLTYPE(mltype)]
       { ruleid}
```

The TEST subcommand takes the following parameters:

*****

Tests a previously compiled (but not necessarily stored) or decompiled command limiting rule set.

**(no parameter)**

When specified without a parameter, the TEST subcommand operates the same as when you specify an asterisk.

**ruleid**

Identifies the key of a command limiting rule set to be tested.

**MDLTYPE(mdltype)**

Specifies the operating system and release of the rule to be tested. If you do not specify a MDLTYPE, CA ACF2 for z/ VM uses your default model type (as defined in the CMDLIM VMO record).

## TEST Subcommand Keywords

After you have issued the TEST subcommand along with any of the parameters described, the TEST subcommand is active. You can enter any of the keywords and values described below to specify a test access environment. You must separate each keyword with a blank character. You can enter all keywords on a single line or use separate lines to enter each keyword.

**OPERANDS(operands)**

Specify the OPERANDS keyword with the CP command operands to be tested. These operand names must be separated by blank characters. You cannot mask the command operands. You do not need to specify the CP command name since it is taken from the rule ID. You can also use the abbreviation O for OPERANDS.

**UID(uidmask)**

Specify the UID keyword with the UID string identifying the user to be tested. You can mask this UID string. To specify this keyword, you do not need access to the corresponding logonid record. If you specify both LID and UID, CA ACF2 for z/ VM uses the last LID or UID value specified. For example, if you specify LID(TLCJJD) UID(TLCNLT), CA ACF2 for z/ VM uses only UID(TLCNLT). If you do not specify a CLASS, the testing function uses your current class.

**LID(lid)**

Specify the LID keyword with the logon ID identifying the user to be tested. You cannot mask this logon ID . To specify this keyword, you do not need access to the corresponding logon ID record. It will be obtained from the service machine for testing purposes. If you specify both LID and UID, CA ACF2 for z/ VM uses the last LID or UID value specified. For example, if you specify UID(***TLCNLT) LID(TLCJJD), CA ACF2 for z/ VM uses only LID(TLCJJD). If you do not specify a CLASS, the testing function obtains the CP privilege class for the indicated logon ID from the VM directory.

**CLASS(classes)**

Specify the CLASS keyword with the CP privilege classes that you want to test the command operands against. Valid classes are A through Z and 1 through 6. Only those classes that apply to the particular release of VM you are running match the classes on the command models. Using the CLASS keyword can be particularly useful if you are testing rules for groups of users. If you do not specify either an unmasked logon ID or classes, your current CP privilege class is used for testing purposes. If you do not specify a CLASS, the testing function obtains the CP privilege class for the indicated UID from the VM directory.

**DATE(date)**

Specify the DATE keyword with the date to be tested. This date must be in the format selected during installation of CA ACF2 for z/ VM (mm/dd/yy, dd/mm/yy, or yy/mm/dd). The current date is assumed as a default.

**SOURCE(sourceid)**

Specify the SOURCE keyword with the logical name of the input source or source group.

**TIME(hhmm)**

Specify the TIME keyword with the time, in hours (hh) and minutes (mm). CA ACF2 for z/ VM uses this time to test the access.

## How to Use the TEST Subcommand

Suppose you just compiled or decompiled a command limiting rule set with the key SPOOL:

```
acf
ACF
set cmdlim
CMDLIM
decomp spool
ACFpgm556I 'rule' rule 'key' stored by 'lid' on 'date'
$KEY(SPOOL) MDLTYPE(530)
$USERDATA (PROTECT THE SPOOL COMMAND)
 CON PURGE UID(****OPR) LOG
 PRINT COPY - ALLOW
 PRINT RSCS ALLOW
 - UID(*) ALLOW
ACFpgm551I Total record length='length' byte - 'percent' percent
      utilized
CMDLIM
```

The TEST subcommand could then be issued:

```
te
 ACFpgm742I CMD='cmd', MDLTYPE='model'
 ACFpgm734I USERDATA contents: 'text'
```

At this point, the TEST subcommand is active. Because we did not specify a MDLTYPE in the DECOMP SPOOL subcommand line, CA ACF2 for z/ VM uses E21 (the site default) for the MDLTYPE. You can now enter any of the TEST subcommand keywords to specify the particular environment that you want to test.

For example, the following keywords test whether the command limiting rule set named SPOOL lets TLCNOPRNAS execute the SPOOL command with the operands CON and PURGE.

```
operands(con purge) UID(tlcnoprnas)
ACFpgm740I The following parameters are in effect:
 UID=TLCNOPRNAS SOURCE=********
 DATE=11/28/00 TIME=********
 CLASS=********
 OPERANDS=CON PURGE

THE FOLLOWING WOULD APPLY: LOG
   (RELATIVE RULE ENTRY 00001)
 .(signals you can enter more TEST subcommand keywords)
end (signals the end of TEST subcommand)
CMDLIM
```

We recommend you decompile before a test so you can see how CA ACF2 for z/ VM sorted your rule entries. If you do not decompile first, the TEST command results can be deceiving (TEST may report a relative rule entry 0003 applied when you know that the first rule entry you wrote should apply).

While the TEST subcommand is active, only command limiting rule interpretation is done. This testing does not take into account any virtual machine configurations, such as virtual unit record devices. Your unit record configuration is used. This should not cause any problems if you used the pseudo operand rule writing technique described earlier (if your virtual console was at 009 and the user's console was at 00A.) You could specify CON or 009 to test the SPOOL command and have the CON rule apply. When the user whose virtual console was at 00A issued a SPOOL 00A..., the CON rule applies.

## How to Interpret TEST Results

As you can see in the previous example, CA ACF2 for z/ VM responds to the TEST request by displaying all of the current values that describe the environment being tested. At the bottom of the display is an indication of whether the command with the specified operand combination is allowed, logged, denied, or if there are any overrides, such as SYNERR or security. In the previous example, the result is that user TLCNOPRNAS is allowed to execute the CP command with the specified operand combination. The first rule entry in the rule set (as sorted by CA ACF2 for z/ VM) allowed the execution. Nearly all keyword values that are not specified are assumed to be completely masked, by default. (The values for the DATE, CLASS, and OPERANDS keywords are the only exceptions.) For instance, if you do not specify the UID keyword, the subcommand tests whether all UIDs are allowed access. For an explanation on how CA ACF2 for z/ VM obtains the CP privilege CLASS, see the LID and CLASS keywords in the TEST Subcommand Keywords section.

The results of the TEST subcommand show whether execution of the specified format of the CP command is allowed, logged, or prevented.

**ALLOW**

Access is allowed

**LOG**

Access is allowed but logged

**PREVENT**

Access is specifically prevented.

If no rule entry specifically applies to the test access environment, CA ACF2 for z/ VM displays the following message:

No rules apply, access would be denied

After a result is displayed, you can enter other keywords and values to specify another environment for testing. The END subcommand stops the TEST subcommand.

# Chapter 4: Using the ACF Command (DIAGLIM Setting)

Diagnose limiting rules validate a user's authority to issue a diagnose instruction with a particular diagnose code. This validation occurs after normal CP user class validation for diagnose instructions.

Diagnose limiting rules are stored on the Infostorage database in a storage class known as the limiting class. This class, identified by L, includes both diagnose limiting and command limiting rules. Rules for diagnose limiting have a type code of VRD in the L storage class.

Loggings and violations that occur during the execution of diagnose instructions are recorded in SMF records. You can analyze these records through the Command Limiting Journal (ACFRPTCL). For more information, see the chapter “Running the Command Limiting Report (CL)” in the *Reports and Utilities Guide*.

SECURITY, NON-CNCL, and READALL attributes do not grant any special privilege for diagnose limiting. These attributes bypass access validation only for data access rules and resource rules.

To implement diagnose instruction validation:

- Create the diagnose limiting rule sets. All diagnose limiting rule set processing is done under the DIAGLIM setting of the ACF command. You do not need to specify the type code for diagnose limiting rules when you establish the DIAGLIM setting. For the complete syntax for a diagnose limiting rule set, see Syntax of a Rule Set in the chapter “Rule Writing Guidelines.”
- Specify the diagnose codes to include or exclude from validation through the DIAGLIM VMO record.
- Establish the mode for diagnose limiting validation. This mode is independent of the CA ACF2 for z/ VM system mode and is specified through the DIAGLIM VMO record.
- Write diagnose limiting rules for each diagnose code that you want to control in #2 above.

Remember, you must define all diagnose codes that are or are not subject to diagnose limiting rule set validation. For each diagnose instruction specified in this operand, you must write a single diagnose limiting rule set.

The rest of this chapter provides information on creating and maintaining the diagnose limiting rule sets. For more information about establishing the diagnose code and mode, see the chapter "Defining Structured Infostorage Records" in the *Administrator Guide.*

This section contains the following topics:



# Internal Diagnose Codes

The ACF command SRF and most other CMS-based CA ACF2 for z/ VM programs use diagnose codes ACF2 and 0ACF to communicate with code running in CP. CA ACF2 for z/ VM has built-in protection for these codes. We recommend that you do not limit them. For example, SRF applications cannot perform validations or modify database records unless the user has the appropriate authorization. However, if you code an INCLUDE ALL in the DIAGLIM VMO record, be sure to write a rule allowing for both the ACF2 and 0ACF diagnoses.

# POSIX Diagnose Calls

In Portable Operating Systems Interface for Computing Environments (POSIX), CP lets users inquire on the contents of the POSIX database through new CP diagnose calls. These calls extract POSIX database runtime information or issue setid() requests.

These new CP diagnose calls are:

```
x'2A4' - get a PID from CP
x'29C' - set POSIX IDs (plist SPXBK)
           setuid()
           seteuid()
           setgid()
           setegid()
           newgrp - shell utility
x'2A0' - query POSIX IDs (plist QPXBK)
           query process attributes
           query user database
           query group database
           query SGIDs for process or user
           query POSIX config info
```

# Sample Diagnose Limiting Rule Set

A diagnose limiting rule set is identified through a $KEY value of DIAGnnnn, where nnnn is the four-digit hexadecimal code for the diagnose instruction. You could use the following rule set to validate a diagnose instruction with code 0004:

```
$KEY(DIAG0004)
 UID(TLCLLS) ALLOW
```

This rule set lets TLCLLS issue a diagnose instruction with the code X'0004'. Through this diagnose code, you can manipulate input spool files.

# Creating a Rule Set

You can create diagnose limiting rule sets directly from the terminal or by building the rule set in a CMS file. The general procedure is:

- Build the diagnose limiting rule set text in a file (explained later).
- Issue the ACF command from CMS.
- Enter SET DIAGLIM to establish the DIAGLIM setting.
- To compile from a CMS file, issue the COMPILE subcommand with the name of the file that contains the rule set text.

  To compile directly from the terminal, issue the COMPILE subcommand without a filename. You can enter the control statements and rule entries one at a time at the terminal when you issue the COMPILE subcommand.
- To test the rule set, enter the TEST subcommand. This gives you an idea of whether the rule set does the intended validation for execution of diagnose instructions.

- If you compile the rule set from a CMS file, the COMPILE subcommand automatically stores the rule set on the Infostorage database. If you compile the rule set directly from the terminal, you must issue the STORE subcommand.
- After you store a diagnose limiting rule set, it is effective on the database and in the running system if the particular diagnose code is being modified.

# ACF Subcommands

You create, display, and maintain diagnose limiting rules using the ACF subcommands. To process the rules, establish the DIAGLIM setting of the ACF subcommand.

```
acf
ACF
set diaglim
DIAGLIM
```

Again, you do not specify the type code for diagnose limiting rules when you establish the DIAGLIM setting. The following ACF subcommands are valid to process diagnose limiting rule sets under the DIAGLIM setting:

- COMPILE
- DECOMP
- DELETE
- END
- HELP
- LIST
- SHOW
- STORE
- TEST

The END, HELP, SET, and SHOW subcommands are valid under all ACF settings. For more information about these subcommands, see the *Administrator Guide*.

The other subcommands are explained in the following sections.

# COMPILE Subcommand

The COMPILE subcommand compiles the diagnose limiting rule set. CA ACF2 for z/ VM provides two ways of compiling diagnose limiting rule sets: directly at the terminal or using text in a CMS file as input to the compiler.

## Compiling Directly at the Terminal

To enter a diagnose limiting rule set directly from the terminal, enter the COMPILE subcommand without parameters:

```
COMPILE

ACFpgm510I CA-ACF2 compiler entered
```

To begin, enter the $KEY control statement on the first line. Enter the other control statements and then the rule entries, each on a separate line. After entering each line of information, press **Enter** or the return key.

```
DIAGLIM

COMPILE
ACFpgm510I CA-ACF2 compiler entered

$KEY(DIAG0004)
 UID(*****OPR) LOG
 UID(*****TEC) ALLOW

ACFpgm551I Total record length=194 bytes - 4 percent utilized

DIAGLIM
```

When you are finished adding entries to the rule set, press **Enter** or the return key without adding any text. The COMPILE subcommand automatically ends. After you compile the diagnose limiting rule set, you can use the TEST subcommand to see if the rule set performs as you intended.

## Compiling from a CMS File

To create a diagnose limiting rule set, enter the control statements and rule entries in a CMS file. The file must have a filetype of RULE. Enter each control statement or rule entry on a separate line.

Assume that the following text was input into a file named DIAGFILE with a filetype of RULE.

```
$KEY(DIAG0004)
 UID(OPR) LOG
 UID(TEC) ALLOW
```

After you enter the control statements and rule entries in the file, invoke CA ACF2 for z/VM and compile the diagnose limiting rule set using the COMPILE subcommand under the DIAGLIM setting. Specify the filename following the COMPILE subcommand:

```
DIAGLIM

compile diagfile
```

Indicate only the filename; the filetype is assumed to be RULE. The rule set is compiled and, by default, stored.

After you compile the diagnose limiting rule set, you can use the TEST subcommand to see if the rule set performs the intended validation of attempts to issue a diagnose instruction.

## Syntax of the COMPILE Subcommand

Under the DIAGLIM setting, the COMPILE subcommand has the following syntax.

```
COmpile        { *        } [ List|NOList  ]
               { filename } [ Store|NOStore]
                            [ Force|NOForce]
```

*****

Indicates that the text is input to the compiler. In an online environment, the system prompts you to enter the diagnose limiting rule text directly from the terminal.

**(no parameters)**

Using the COMPILE subcommand without parameters is the same as specifying an asterisk (*).

**filename**

Specifies the CMS filename that contains the diagnose limiting rule text to compile. The filetype is always RULE.

**LIST|NOLIST**

LIST displays the input to the compiler on your screen or printed on your listing when you compile the rule set. NOLIST does not display on your screen or listing. LIST is the default when compiling from a CMS file. Otherwise, NOLIST is the default.

**STORE|NOSTORE**

STORE stores the rule set after it is compiled. NOSTORE does not automatically store the rule set. STORE is the default if you are compiling the rule set from a CMS file. Otherwise, NOSTORE is the default.

**FORCE|NOFORCE**

FORCE (the default) stores the diagnose limiting rule set even if it currently exists. NOFORCE only stores the rule set if it does not already exist. When you use FORCE|NOFORCE as a parameter of the COMPILE subcommand, it only applies to the COMPILE subcommand you are currently issuing. FORCE|NOFORCE is in effect until you change it or until you end the ACF command. Changes in the ACF command setting do not affect this setting.

CA ACF2 for z/ VM provides two ways of compiling a diagnose limiting rule set: From a CMS file or directly at the terminal.

# DECOMPILE Subcommand

Under the DIAGLIM setting, the DECOMP subcommand decompiles a diagnose limiting rule set. Use it to examine, update, test, or change a diagnose limiting rule set. You can decompile a diagnose limiting rule set at the terminal or into a CMS file to keep for later reference.

## Syntax of the DECOMP and LIST Subcommands

Under the DIAGLIM setting, the LIST subcommand is a synonym for DECOMP and performs a similar function.

```
DEComp    { *             }
          { ruleid        }  [INTO(filename]
    List  { LIKE(rulemask)}
```

*****

Decompiles the diagnose limiting rule set as previously compiled and stored.

**ruleid**

Specifies the $KEY value of the diagnose limiting rule set to decompile. It is always in the form DIAGnnnn, where nnnn is the four-character code of the diagnose instruction.

**LIKE(rulemask)**

Specifies a mask of rule IDs for decompiling a group of diagnose limiting rule sets. The CMS file containing rules created by the DECOMP LIKE(-) function no longer have a record length greater than 80 characters. Any rule text exceeding 80 characters is changed into continuation lines. This makes editing and printing the file easier. However, the compiler continues to accept input files that exist from previous releases that have record lengths up to 256 characters.

**INTO(filename)**

Specifies the CMS file where the diagnose rule set is decompiled. You do not need to specify the file type, since a file type of RULE is always assumed.

```
DECOMP DIAG0006 INTO(DIAG0006)

ACFpgm762I DIAG0006 STORED BY VMSA ON 01/13/07
ACFpgm551I TOTAL RECORD LENGTH=128 BYTES - 25 PERCENT UTILIZE
```

This example subcommand decompiles the DIAG0006 rule set into a file named DIAG0006 RULE.

# DELETE Subcommand

Under the DIAGLIM setting, the DELETE subcommand deletes a diagnose limiting rule set.

## Syntax of the DELETE Subcommand

Listed below is the syntax of the DELETE subcommand under the DIAGLIM setting.

```
DELete       { ruleid        }
             { LIKE(rulemask)}
```

**ruleid**

Deletes the specified rule ID.

**LIKE(rulemask)**

Deletes all rule sets that match the mask.

The following DELETE subcommand deletes the diagnose limiting rule set for diagnose instruction X'0018'.

```
DELETE DIAG0018
```

CA ACF2 for z/ VM returns the following message when it deletes the rule set:

```
DELETED
```

# STORE Subcommand

The STORE subcommand, under the DIAGLIM setting, stores previously compiled diagnose limiting rule sets. CA ACF2 for z/ VM can reject this operation because of insufficient authority for storing the diagnose limiting rule set.

## Syntax of the STORE Subcommand

Under the DIAGLIM setting, the STORE subcommand has the following syntax.

```
STore      {  FORCE   }
           {  NOFORCE }
```

**FORCE**

Stores a rule set, even if it currently exists.

**NOFORCE**

Stores a rule set only if it does not currently exist.

**Note:** You can also use the SET subcommand to set the default for FORCE|NOFORCE.

# TEST Subcommand

The TEST subcommand lets you to interactively test a diagnose limiting rule set. Testing ensures the diagnose limiting rule set provides the intended validation of attempts to issue the diagnose instruction. When the TEST subcommand is active, CA ACF2 for z/ VM only interprets diagnose limiting rules. Testing does not take into account any site-specific system options or attributes of the logonids being tested.

## Syntax of the TEST Subcommand

Listed below is the syntax of the TEST subcommand under the MODEL setting.

```
TEST       { *        }
           { DIAGnnnn }
```

*****

Indicates you want to test the previously compiled diagnose limiting rule set.

**(no parameter)**

Operates the same as when you specify an asterisk.

**DIAGnnnn**

Identifies the key of the diagnose limiting rule set being tested. nnnn is the four-character hexadecimal code for the diagnose instruction.

# TEST Subcommand Keywords

After you have issued the TEST subcommand, a period (.) indicates it is active. You can enter any of the following keywords with appropriate values to a test access environment. You must separate each keyword with blank characters. You can specify one or more input lines.

**UID(uidmask)**

Identifies the user to be tested for execution of a diagnose instruction. To specify this keyword, you do not need access to the tested user's logon ID record. If you specify both the LID and UID keywords, CA ACF2 for z/ VM uses the last LID or UID value specified. For example, if LID(TLCJJD) and UID(TLCNLT) are specified, CA ACF2 for z/ VM uses only UID(TLCNLT).

**LID(lidmask)**

Specifies the logon ID of the user to be tested for execution of a diagnose instruction. Like UID, you can mask the value of LID. To specify this keyword, you do not need access to the corresponding logon ID record. If you specify both LID and UID, CA ACF2 for z/ VM uses the last LID or UID value specified (just like UID above).

**DATE(date)**

Specifies the date to be tested. This date must be in the format mm/dd/yy, dd/mm/yy, or yy/mm/dd, as specified in the OPTS VMO record. The current date is assumed as a default. For more information about the OPTS VMO record, see the chapter “Defining Structured Infostorage Records” in the *Administrator Guide*.

**TIME(hhmm)**

Specifies the time when execution of a diagnose instruction is tested for. This time is specified in hours and minutes (four digits).

**SOURCE(sourceid)**

Specifies the logical name of the input source or source group.

## Examples of TEST Subcommands

After you compile a diagnose limiting rule set, you can issue the TEST subcommand to test the rule set.

```
DIAGLIM

compile
ACFpgm510I CA-ACF2 compiler entered

$key(diag0006)
 uid(tlcopr) allow
 uid(tlctec) log

ACFpgm551I Total record length=194 bytes - 4 percent utilized

DIAGLIM

test *
DIAG=DIAG0006, MDLTYPE=H50
.
```

The TEST subcommand is active, as indicated by the period (.). You can enter any of the TEST subcommand keywords to specify the environment you want to test.

The UID keyword, for example, tests whether a diagnose limiting rule set allows a certain user to issue a diagnose instruction. Here, we are testing the previously compiled diagnose rule set for code X'0006' to see if user TLCNLT can issue the diagnose.

```
DIAGLIM

test *
DIAG=DIAG0006, MDLTYPE=H50
 .
UID(TLCNLT)
The following parameters are in effect:
 Date=11/08/00, time=*****, UID=TLCNLT, source=********
 Diagnose=DIAG0006

The following would apply: LOG (relative rule entry 2)
```

The system displays all of the current values of the environment being tested. At the bottom of the display is a message that indicates if execution of the diagnose instruction is allowed, logged, or prevented. From the previously compiled rule set, TLCNLT is allowed to execute the diagnose code X'0006', but CA ACF2 for z/ VM writes an SMF record to log the event.

After a result is displayed, you can make another entry of keywords to test another rule set environment. But remember, after you enter TEST command keywords, the values you specify remain in effect until you explicitly change them. Furthermore, as shown in the previous example, almost all values you do not specify are assumed to be completely masked, by default. The values for the DATE keyword and the DIAGNOSE value are the only exceptions. If you specify no UID keyword, the TEST subcommand tests all UIDs.

To terminate the TEST subcommand, enter END.

## TEST Subcommand Results

The results of the TEST subcommand show if execution of the diagnose instruction is allowed, logged, or prevented.

**ALLOW**

Access is allowed

**LOG**

Access is allowed but logged

**PREVENT**

Access is explicitly prevented.

If no rule entry specifically applies to the test access environment, CA ACF2 for z/ VM displays the following message:

```
ACFpgm74CI  No rule applies, access would be denied
```

# Chapter 5: Command Limiting the CP Spooling System

CP spooling facilities maintain input and output files for access by virtual machines through unit record devices (printers, readers, punches, and consoles). Spooling gives all users access to real unit devices that are under the control of the spooling system. Below is a list of spool file commands that class D users can normally execute. Most companies enforce strict access controls to data files. Often, they do not carry this level of protection far enough (printed output that sits in the spool queue waiting to be printed).

This chapter includes:

- Guidelines for using CA ACF2 for z/ VM to control spool-related commands
- Suggestions for protecting the spool files
- Examples of actual command limiting rules to control spool-related commands.

This section contains the following topics:



## CP Commands That Affect Spooling

There are several commands that control and manipulate spooling operations. To effectively protect your spool files, you should be aware of how each command affects spooling and spool files. These commands are divided into three categories:

- Class D for the spooling operator
- Class G for general users
- Commands that indirectly affect spooling.

### Class D Spool File Commands

Below is a list of spool file commands that class D users can normally execute:

**BAckspac**

Restarts or repositions the current spool file.

**CHange**

Alters the attributes of a closed spool file.

**DRain**

Stops spooling activity on a device after the current file is finished.

**FLush**

Immediately stops the current spool file.

**FRee**

Removes the hold attribute from a user's spool files.

**HOld**

Puts the system hold attribute on the specified user's spool files that defers processing of them.

**LOADBUF**

Loads a UCS or FCB on a real printer.

**ORDer**

Changes the order that closed spool files are to be processed.

**PURge**

Deletes a closed spool file from the spool queue.

**Query**

Displays the attributes of closed spool files on the spool queue.

**REPeat**

Changes the number of copies of the current file being processed, or changes it to hold status.

**SPAce**

Forces single spacing on the printer.

**SPTape**

Dumps (backup) and loads (restores) closed spool files to tape.

**STArt**

Starts the processing of spool files on the indicated device.

**TRANsfer**

Moves closed spool files from one user spool queue to another user spool queue.

## Class G Spool File Commands

Below is a list of spool file commands that class G users can normally execute:

**CHange**

Alters the attributes of a closed spool file.

**Close**

Terminates spooling operations on the indicated virtual device that closes the file so it appears (can be queried or altered) on the spool queue.

**LOADVFCB**

Specifies the forms control buffer image for a virtual spooled printer.

**ORDer**

Changes the order that closed spool files are to be processed.

**PURge**

Deletes a closed spool file from the spool queue.

**Query**

Displays the attributes of closed spool files on the spool queue.

**SPool**

Changes or sets spool options or attributes for the indicated device. Also directs spool output to a user's spool queue.

**TAG**

Associates descriptive information with a spool file.

## Commands That Indirectly Affect Spooling

Besides to the above commands, there is also a group of commands that have an impact on spooling operations. They are:

**DEFine**

Establishes virtual unit record devices used for spooling operations.

**LOGon**

Processes the SPOOL statements in a user VM directory entry. These statements define the virtual spooling devices available to a user and some of the attributes of those devices. CA ACF2 for z/ VM command limiting does not process SPOOL statements in a user's directory entry.

**LOGoff**

Automatically closes all open spool files.

# Using Command Limiting to Protect Spool Files

When writing rules to protect your spool queue and files, you must consider all of the spooling-related commands. Be aware that some commands, such as CHANGE, QUERY, PURGE, SPTAPE, and TRANSFER, alter the attributes of a spool file. Rules for these commands should ensure that a user cannot deliberately alter the attributes of a spool file and then obtain access to it. When one of these commands is issued, CA ACF2 for z/VM searches through the spool queue and matches all attributes of the spool file to the operands specified in the CP command. This process ensures that the object of the command (the affected spool file) is properly matched against the CA ACF2 for z/ VM rule.

Class C and E users, or a person that has access to the computer system console, can use the CP commands that display storage (DCP and DMCP) to determine the attributes of a spool file or spooling device. These users could also use the STCP command to alter the attributes of spool files. We recommend you use CA ACF2 for z/ VM command limiting to control these powerful CP commands and employ physical security measures to protect the computer system console.

## Spool File Attributes That Can Be Used in a Rule

You can specify most of the information that CP maintains about a spool file as part of a command limiting rule entry. You can explicitly or implicitly cite the following spool file attributes in a rule entry:

| Attribute | Description |
|---|---|
| Spool queue name | RDR (for reader), PRT (for printer), PUN (for punch), CON (for console), or ALL (for all the different types of spool files). |
| Spool file owner | User ID, SYSTEM, or * (self) (defaults to * for class G users) |
| Spoolid | Sequentially assigned four-digit numeric spool file number. We recommend you do not use the spool ID to protect spool files because it is a sequentially assigned number. |
| Spoolid2 | For SPTape command only, end spool file number, or END. |
| Spool file class | Such as A, B, ... Z, 0, 1, ... 9. |
| Output form name | Can be any one- to eight-character name, such as STANDARD, STD, and 2PART. |

## Choosing a Method of Spool File Protection

When deciding how to protect spool files, consider the output control procedures at your site.

- Do you have multiple remote printers (DEST)?
- Do you use spool file classes as a printing control (CLASS)?
- Do you have special forms (FORM)?

How you process output and the type of information processed dictates how you use CA ACF2 for z/ VM to protect your spool files.

There are six recommended methods to protect your spool queue:

| Protect by | Description |
|---|---|
| Class | Minimally, a rule entry includes the owner ID of the spool file, the class of spool files, the UID string of the users, and the access permission. |
| Form | Minimally, a rule entry includes the owner ID of the spool file, the form number of the spool files, the UID, and the access permission. |
| Class and form | |
| Spool file owner | Minimally, a rule entry includes the owner ID of the spool file, the UID, and the access permission. |
| Class and target | Minimally, a rule entry includes the owner ID of the spool file, the class and target of the spool files, the UID, and the access permission. |
| Destination | |

For example, user PAYOPER displays the spool queue through a QUERY command:

```
ORIGINID  FILE CLASS ... FORM

MAINT     4324 A PRT     STD
VSEIPO    4567 T CON     3HOLE
PAYROLL   4568 P PRT     REGISTER
PAYROLL   4569 P PRT     W2
PAYROLL   4790 P PRT     STANDARD
```

They then issues a TRANSFER command to place all of the output class P spool files into his spool queue:

```
TRAN PAYROLL PRT CLASS P TO *
```

CA ACF2 for z/ VM transposes this command to read TRANSFER PAYROLL PRT CLASS P TO PAYOPER.

Before CA ACF2 for z/ VM transposes the rule set, the TRANSFER rule set states:

```
$KEY(TRANSFER)
 *- PRT CLASS P - UID(PAYOPER) ALLOW
 *- PRT CLASS A - UID(TLCOPER) ALLOW
 *- PRT CLASS T - UID(TLCOPER) ALLOW
```

After transposition, CA ACF2 for z/ VM interprets the rule set as:

```
$KEY(TRANSFER)
*- PRT CLAS P TO PAYOPER UID(PAYOPER) ALLOW
*- PRT CLAS A TO PAYOPER UID(TLCOPER) ALLOW
*- PRT CLAS T TO PAYOPER UID(TLCOPER) ALLOW
```

Based on the transposition of the command and rule set, CA ACF2 for z/ VM selects the following files as eligible for the TRANSFER command:

- PAYROLL 4568 P PRT REGISTER
- PAYROLL 4569 P PRT W2
- PAYROLL 4790 P PRT STANDARD

CA ACF2 for z/ VM examines the spool queue for each occurrence of a file that is owned by PAYROLL with class P. Each file is then compared to all of the rule entries. All of the files must be allowed by one or more rule entries in the rule set for the command to be allowed to execute.

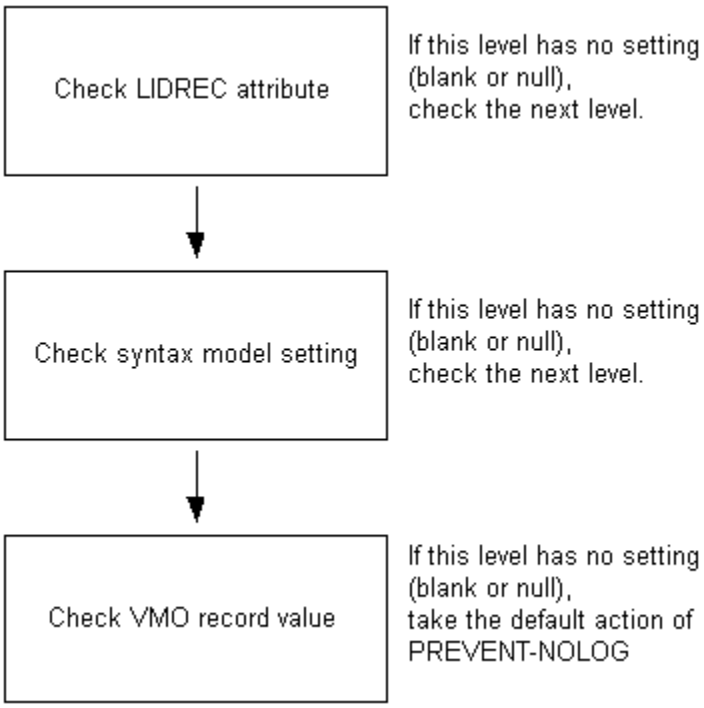
Mixing class and form in the same rule can lead to undesired results unless you explicitly specify both operands in all rule entries related to spool files. For more information, see Protection by Class and Form in this chapter.

CA ACF2 for z/ VM can encounter a spool file not found condition when there are no spool files that match the command. You can control the action CA ACF2 for z/ VM takes when a user issues a CP spool command for a spool file that does not exist. There are three ways to control NOSPOOL processing: the ACFFDR, command model, and the logon ID record.

For information about using the ACFFDR to control NOSPOOL processing, see the chapter “The CA-ACF2 Field Definition Record” in the *Installation Guide*.

For information about the logon ID record, see the chapter “About the Logonid Record” in the *Administrator Guide*.

## Hierarchy of Options




**Logonid Record Field**

You can specify how you want CA ACF2 for z/ VM to react for each individual user when the user enters a command for no spool files. This field is called NOSPOOL. This is the highest level of override. When a user has the NOSPOOL field, a spool file not found condition is handled as defined in the logon ID record (overrides both the COMMAND clause and the ACFFDR setting). If this level is null (or blank), CA ACF2 for z/ VM passes processing to the next level.

**Command Model Options**

For each CP command, you can specify how you want CA ACF2 for z/ VM to handle a spool file not found condition. You can do this using the NOSPOOL operand on the COMMAND clause. This is the second level of override. The action specified in the NOSPOOL operand on the COMMAND clause overrides the option specified in the ACFFDR, but only for that specific command. If this level is null (or blank), CA ACF2 for z/ VM passes processing to the next level.

**VMO Record**

In the CMDLIM VMO record, there is an operand named NOSPOOL that defines how CA ACF2 for z/ VM handles a spool file not found condition. You can override NOSPOOL using the LIDREC attribute or the syntax model setting previously described.

As previously stated, the command model controls NOSPOOL processing. There are four options available to indicate what action to take when a spool file not found condition occurs:

**ALLOW**

CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CP issues standard error messages to the user. CA ACF2 for z/ VM does not log the syntax error to SMF. You can abbreviate ALLOW with A in model setting.

**LOG**

CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CP issues the standard error messages to the user. CA ACF2 for z/ VM also writes an SMF record that appears on the Command Limiting Journal (ACFRPTCL). You can abbreviate LOG with L in the model setting.

**PREVENT**

When a spool file not found condition occurs, CA ACF2 for z/ VM displays the following message and rejects the command:

```
ACFpgm277I No spool files found
```

CA ACF2 for z/ VM does not charge the user with a violation (CA ACF2 for z/ VM does not increment the SEC-VIO count). PREVENT is the default for NOSPOOL processing. CA ACF2 for z/ VM does not write System Management Facility (SMF) records for spool file not found conditions (CA ACF2 for z/ VM does not log the event). You can abbreviate PREVENT with P in the model setting.

**PREVENT-LOG**

Works the same as PREVENT above, except the condition generates an SMF record that appears on the Command Limiting Journal (ACFRPTCL). This option does not cause a violation. You cannot abbreviate in the model setting.

# Using Command Limiting to Protect the Spool Queue

As previously explained, there are six ways to protect your spool queue, by class, form, class and form, spool file owner class and target, and destination. We have included sample rules here for each method to help you understand how you should write these rules. These rules are shown in the decompiled format.

When practicing writing rules, you must use actual user IDs. If you do not use real user IDs, test results can be incorrect. CA ACF2 for z/ VM checks the VM directory for user IDs and issues the following messages if you use an undefined user ID, even though the rule is correct:

```
ACFpgm751W  No applicable rule could be found
ACFpgm757W  Command syntax error (operand number <nn> is in error)
```

## Protection by Class

In the example below, the first line of the rule indicates that the CHANGE command is command limited. The second line prevents any user from changing a spool file in classes C or 2. All other spool changes are allowed (last line of rule set).

```
$KEY(CHANGE)
 -  CLASS C2 -  UID(*) PREVENT
 -  UID(*) ALLOW
```

In the next rule, user OPR can purge classes A, E, F, G, M, O, or R. User PAYOPR is allowed to purge classes P and W that belong to user IDs beginning with P only. Other users can only purge files in their own spool queue.

```
$KEY(PURGE)
 P***** - CL PW -  UID(PAYOPR) ALLOW
 - CL AEFGMOR -  UID(OPR) ALLOW
 ALL -  UID(*) ALLOW
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

The next rule set prevents users with the PAY, PER, GEN, MKT, and OPR user ID from spooling the classes listed in their particular rule. The last five lines of the rule set allow all other spooling for those classes not specifically prevented in the previous rules. The mask *- requires you enter at least one character for the device type before you specify other operands, including class. The - masks possible other operands, either before or after class.

```
$KEY(SPOOL)
 *- - CLASS BCEFGHIJKLMNQSTUVXYZ0123456789 - UID(PAY) PREVENT
 *- - CLASS ABDGHIKLMNOQRSTVWXYZ0123456789 - UID(PER) PREVENT
 *- - CLASS ABCDEFHIJKLMNOPRSTUVXYZ013456789 - UID(GEN) PREVENT
 *- - CLASS BCDEGHIJKNOPQRTUVWXYZ0123456789 - UID(MKT) PREVENT
 *- - CLASS ABCDEFGHIJKLMNQRSTVWXY012356789 - UID(OPR) PREVENT
 -  UID(PAY) ALLOW
 -  UID(PER) ALLOW
 -  UID(GEN) ALLOW
 -  UID(MKT) ALLOW
 -  UID(OPR) ALLOW
```

The next rule lets an OPR user ID dump any class A, B, or C file, but the occurrence is logged. CA ACF2 for z/ VM denies all attempts at dumping any other classes.

```
$KEY(SPTAPE)
 - CLASS ABC UID(OPR) LOG
```

Below, OPR user IDs can start up classes A, E, F, G, M, O, and R only. PAYOPR can only start up classes P and W.

```
$KEY(START)
 *- CL AEFGMOR - UID(OPR) ALLOW
 *- CL PW - UID(PAYOPR) ALLOW
```

The following rule lets PAYOPR send files in classes A, P, R, and W to and from Payroll and Personnel. The OPR user can send class E, F, G, M, and O files to anyone, and class A and R files only to Payroll and Personnel. PAY and PER users can transfer files in classes A, P, R, and W to or from others in Payroll and Personnel. PER users can also transfer files in classes E, F, G, M, and O to or from anyone. GEN and MKT users can transfer files to and from anyone.

```
$KEY(TRANSFER)
 SYSTEM *- CL APRW *- P***** -  UID(PAYOPR) ALLOW
 SYSTEM *- CL AR *- P***** -  UID(OPR) A
 *- CL APRW *- PER***  UID(PER) ALLOW
 *- CL APRW *- PAY*** -  UID(PAY) ALLOW
 SYSTEM *- CL EFGMO -  UID(OPR) ALLOW
 *- CL EFGMO -  UID(OPR) ALLOW
 *- CL EFGMO -  UID(PER) ALLOW
 -  UID(GEN) ALLOW
 -  UID(MKT) ALLOW
```

## Protection by Form

There can be instances when spooling is best controlled by form. Following are examples of how this could be accomplished. The second line of the rule set below prevents any user from changing a spool file on any SYSTEM where the value of form is STD (standard). Also, all users are prevented from changing a spool file on any device with XY form. All other spool changes are allowed (last line of rule set). Remember, the *- masks operands that you must specify (device type).

```
$KEY(CHANGE)
 SYSTEM *- *- ******** - FORM STD - PREVENT
 *- *- ******** - FORM XY - UID(*) PREVENT
 -  UID(*) ALLOW
```

In the rule below, user OPR cannot purge any files with form EXEC. PAYOPR can purge any spool files with a form of A. Other users can only purge files in their own spool queue.

```
$KEY(PURGE)
 - FORM A -  UID(PAYOPR) ALLOW
 - FORM EXEC - UID(OPR) PREVENT
 ALL -  UID(*) ALLOW
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

This rule set lets users with the PAY user ID spool any files with the EXEC form. PER users can spool any files with the STD (standard) form.

```
$KEY(SPOOL)
 - FORM EXEC -  UID(PAY) ALLOW
 - FORM STD - UID(PER) ALLOW
```

Next, a user with the OPR user ID can dump any spool with a form of ABC, but CA ACF2 for z/ VM logs the occurrence. CA ACF2 for z/ VM denies all attempts at dumping any other form.

```
$KEY(SPTAPE)
 *- *- *- FORM ABC - UID(OPR) L
```

According to the next rule, users with the OPR user ID can start up any spool files with the STANDARD form only. All other users can start their own files, no matter what their form.

```
$KEY(START)
 *- FORM STANDARD - UID(OPR) ALLOW
 ALL -  UID(*) PREVENT
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

The following rule lets PAYOPR transfer files in the form PAY to anyone in Payroll or Personnel. PER and PAY users can transfer files, in the standard (STD) form, to anyone in Payroll or Personnel. They cannot transfer any other form of files to anyone else. GEN and MKT can transfer any files in any form.

```
$KEY(TRANSFER)
 SYSTEM *- FORM PAY *- P***** - UID(PAYOPR) ALLOW
 *- FORM STD *- P*****  UID(PER) ALLOW
 *- FORM STD *- P***** -  UID(PAY) ALLOW
 *- FORM * -  UID(PAY) PREVENT
 *- FORM * -  UID(PER) PREVENT
 -  UID(GEN) ALLOW
 -  UID(MKT) ALLOW
```

## Protection by Class and Form

If class or form by themselves do not provide the desired protection, you can combine them. These rules can be more complicated because there can be a wide variety of combinations of classes and forms.

The first entry of the rule set below prevents any user from changing a spool file in class A or B in any SYSTEM where the value of form is STD (standard). Also, all users are prevented from changing a spool on any device in class C with a form of XY. This rule set allows all other spool changes (last line of rule set).

```
$KEY(CHANGE)
 *- CLASS AB CLASS * FORM STD -  UID(*) PREVENT
 *- CLASS C CLASS * FORM XY - UID(*) PREVENT
 -  UID(*) ALLOW
```

In the rule below, user OPR cannot purge any files in classes A or B, or with a form of EXEC. User PAYOPR can purge any spool files with a class of P and form of A. Other users can only purge files in their own spool queue, with any class.

```
$KEY(PURGE)
 *- *- CLASS P FORM A -  UID(PAYOPR) ALL
 *- *- CLASS ABC FORM EXEC - UID(OPR) PR
 ALL -  UID(*) ALLOW
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

The next rule set prevents users with the PAY user ID from spooling any files in class P with the EXEC form.  PER users cannot spool any files with the STD (standard) form and class of X. The last two lines of the rule set allow all other spooling for those classes and form not specifically prevented in the previous rules.

```
$KEY(SPOOL)
 *- *- CLASS P FORM EXEC -  UID(PAY) PRE
 *- *- CLASS X FORM STD - UID(PER) PREVE
 *- - UID(PAY) ALLOW
 *- - UID(PER) ALLOW
```

This rule set lets users with the OPR user ID start up any spool files in classes A, B, C, or D with the STANDARD form only. PAYOPR cannot start any spool files in class P, any form. All other users can start their own files, no matter what their class or form.

```
$KEY(START)
 *- CLASS ABCD FORM STANDARD - UID(OPR) ALLOW
 *- CLASS P FORM * - UID(PAYOPR) PREVENT
 ALL -  UID(*) ALLOW
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

## Protection by Spool File Owner

You can protect spool files by spool file owner.  This type of protection is useful when you need to protect group or departmental files. The first rule entry lets any PAY user change a spool that is owned by SYSTEM, but the event is logged. Also, no other users can change PAYOPR files. OPR users can change any other files (last line of rule set).

```
$KEY(CHANGE)
 SYSTEM *- -  UID(PAY) LOG
 PAYOPR *- -  UID(*) PREVENT
 -  UID(OPR) ALLOW
```

In the next rule, PAYOPR can purge PAY files. OPR users can purge any SYSTEM files. PEROPR can purge any files, but CA ACF2 for z/ VM logs the event. Other users can only purge files in their own spool queue.

```
$KEY(PURGE)
 PAY- *- -  UID(PAYOPR) ALLOW
 SYSTEM *- - UID(OPR) ALLOW
 *- - UID(PEROPR) LOG
 ALL -  UID(*) ALLOW
 PRT -  UID(*) ALLOW
 PUN -  UID(*) ALLOW
 RDR -  UID(*) ALLOW
```

The rule set below lets OPR users spool TO and FOR any files. PAYOPR can spool TO and FOR PAY users, but the event is logged. The last four lines of the rule set let users in a department spool TO and FOR users in the same department (any user in the Accounting department can spool TO or FOR any users in the Accounting department users, Marketing department users can spool TO or FOR any other users in the Marketing department).

```
$KEY(SPOOL)
 *- *- PAY- - UID(PAYOPR) LOG
 *- *- GEN- -  UID(GEN) ALLOW
 *- *- MKT- -  UID(MKT) ALLOW
 *- *- EXC- -  UID(EXC) ALLOW
 *- *- ACC- -  UID(ACC) ALLOW
 -  UID(OPR) ALLOW
```

This rule set lets OPR user IDs start up any files.  PAY users can only start up PAY files, but CA ACF2 for z/ VM logs the event.  CA ACF2 for z/ VM denies all other users start up privileges.

```
$KEY(START)
 PAY -  UID(PAY) LOG
 SYSTEM -  UID(*) PREVENT
 *- -  UID(OPR) ALLOW
```

The following rule set lets PAYOPR transfer any SYSTEM files to anyone. EXC users can transfer EXC*** files to anyone. All PER*** users can transfer any PERxxx files only to any other PER*** user. Users with the OPR*** user ID can transfer any files to anyone.

```
$KEY(TRANSFER)
 SYSTEM *- *- *- *- *-
UID(PAYOPR) ALLOW
 EXC- *- *- *- *-  *-  UID(EXC) ALLOW
 PER- *- *- *- PER-  *-  UID(PER) ALLOW
 -  UID(OPR) ALLOW
```

## Protection by Class and Target

Use protection by class and target when you must restrict files in certain classes to distribution to various users. The rule set below prevents users with the PAY, PER, GEN, MKT, and OPR user ID from spooling the classes listed in their particular rule. PAY users can spool TO or FOR Payroll and Personnel, but again, only for the classes they are allowed to spool as specified in the previous rule. PAY users can also spool files to themselves. They cannot send files to anyone else. This rule set allows all other spooling actions.

```
$KEY(SPOOL)
$NOSORT
 *- - CLASS BCEFGHIJKLMNQSTUVXYZ0123456789 - UID(PAY) PREVENT
 *- - CLASS ABDGHIKLMNOQRSTVWXYZ0123456789 - UID(PER) PREVENT
 *- - CLASS ABCDEFHIJKLMNOPRSTUVXYZ01345689 - UID(GEN) PREVENT
 *- - CLASS BCDEGHIJKNOPQRTUVWXYZ0123456789 - UID(MKT) PREVENT
 *- - CLASS ABCDEFGHIJKLMNQRSTVWXY012356789 - UID(OPR) PREVENT
 *- - TO P***** -  UID(PAY) ALLOW
 *- - FOR P***** -  UID(PAY) ALLOW
 *- - TO OWNER -  UID(PAY) ALLOW
 *- - FOR OWNER -  UID(PAY) ALLOW
 *- - TO *- -  UID(PAY) PREVENT
 *- - FOR *- -  UID(PAY) PREVENT
 -  UID(PAY) ALLOW
 -  UID(PER) ALLOW
 -  UID(GEN) ALLOW
 -  UID(MKT) ALLOW
 -  UID(OPR) ALLOW
```

This rule only lets PAYOPR transfer files in classes A, P, R, and W to Payroll. The OPR user can transfer files in classes A and R TO or FOR anyone in Payroll or Personnel. PAY and PER users can transfer files in classes A, P, R, and W to others in Payroll and Personnel. OPR users can also send their files on classes E, F, G, M, and O to anyone. GEN and MKT users can send their files to anyone.

```
$KEY(TRANSFER)
 SYSTEM *- CL APRW TO PAY*** -  UID(PAYOPR) ALLOW
 SYSTEM *- CL AR *- P***** -  UID(OPR) A
 *- CL APRW *- P***** -  UID(PER) ALLOW
 *- CL APRW *- P***** -  UID(PAY) ALLOW
 SYSTEM *- CL EFGMO -  UID(OPR) ALLOW
 *- CL EFGMO -  UID(OPR) ALLOW
 *- CL EFGMO -  UID(PER) ALLOW
 -  UID(GEN) ALLOW
 -  UID(MKT) ALLOW
```

## Protection by Destination

Use protection by destination when you want to restrict files to certain printer or punch destinations. The rule set below allows user IDs beginning with SEC to spool files with a DEST of TOPSEC only. All other users can specify any destination other than TOPSEC.

```
$KEY(SPOOL)
 *- DEST TOPSEC - UID(SEC) ALLOW
 *- DEST *- - UID(SEC) PREVENT
 *- DEST TOPSEC - UID(*) PREVENT
 *- DEST *- - UID(*) PREVENT
```

# Chapter 6: Command Limiting for Shared File System

Native Shared File System (SFS) administration provides for one or more administrators who have unrestricted access to everything in a filepool. This forces you to create separate filepools whenever you want to grant an SFS administrator authority over a subset of users. While you may want multiple filepools for performance reasons, they may not be the answer to providing greater ease of administrative authority. Even creating multiple filepools does not always solve the problem of splitting authority in a detailed and flexible manner.

CA ACF2 for z/ VM provides the option of validating SFS-related commands SFS administrators issue through the CA ACF2 for z/ VM command limiting feature. The command rule key is always SFS_ followed by up to eight bytes of command name. For example, the SFS QUERY command rule key is SFS_QUERY. You must include the rule key of any SFS command to be validated explicitly or implicitly in the COMMANDS keyword of the CMDLIM VMO record.

The SFS command syntax to be validated is not free form like CP commands. Instead, SFS reconstructs each command in a fixed format before passing it to CA ACF2 for z/ VM. CA ACF2 for z/ VM uses this reconstructed syntax to create command limiting models.

This section contains the following topics:



## Command Syntax

Listed below are the command syntaxes available for SFS. You should carefully review these commands to see whether any are appropriate for command limiting at your site. For the values of the variables for these syntaxes, see Variables in this chapter.

```
SFS_CONNECT fpid            [ USER userid ]

SFS_DATASPAC fpid           { ASSIGN } dirid
                            { RELEASE}

                            { LOCK fn ft dirid ( FROM userid      }

SFS_DELETE fpid             { PUBLIC                              }
                            { USER userid                         }
```

```
SFS_DIRATTR fpid dirid       {FILECONTROL                 }
                             {DIRCONTROL ( [FORCE|NOFORCE] }

SFS_DMSDISFS fpid userid     {SHARE     }
                             {EXCLUSIVE }

SFS_DMSDISSG fpid nnnnn      {SHARE     } { DETACH    }
                             {EXCLUSIVE } { NODETACH  }


SFS_DMSENAFS fpid userid

SFS_DMSENASG fpid nnnnn

                             {DIRECTORY  } { dirid 0  {READ|FILEATTR}        }
SFS_DMSOPCAT fpid            {FILESPACE  } { userid 0 {READ|WRITE|FILEATTR}  }
                             {           } {READ     }
                             {GROUP nnnnn} {WRITE    }
                                           {FILEATTR }


SFS_DMSQUSG fpid nnnnn

SFS_DMSRELBK fpid nnnnn

SFS_DMSWRACC fpid

SFS_ENROLL fpid              { PUBLIC                                          }
                             { USER userid ( BLOCKS nnnnnnnn  STORGROUP nnnnn   }


SFS_FILEPOOL fpid            { MINIDISK                          }
                             { CONTROL BACKUP [ DISK fn ft dirid ] }
                             {                [ TAPE vdev        ] }

SFS_MODIFY fpid USER         {+nnnnnnnnnn} FOR userid
                             {-nnnnnnnnnn}

                             {ACCESSORS  [dirid|( DATASPACE] }
                             {DATASPACE  [dirid]             }
SFS_QUERY fpid               {FILEPOOL  {CONFLICT userid    }}
                             {          {{STATUS|( CATALOG\}}}
                             {LIMITS {ALL|FOR userid}        }

SFS_RELOCATE fpid            {dirid1 TO dirid2       }
                             {fn ft dirid1 TO dirid2 }


SFS_RENAME fpid dirid1 dirid2

SFS_SMSCDRA fpid fn ft dirid  <dra1> <dra2> <dra3>

SFS_SMSERASE fpid fn ft dirid ACFONLY
```

```
                                  {WRITE  }
SFS_SMSOPEN fpid fn ft dirid      {NEW    } ACF
                                  {REPLACE}

SFS_SMSOPENX fpid fn ft dirid     {MIGRATE}
                                  {RECALL }
```

## Variables

The variables in the above syntax are:

**fpid**

Filepool ID

**dirid**

Directory ID

**fn**

CMS file name

**ft**

CMS file type

**userid**

User ID. Also shown as userid1 to userid15. The ESM is called multiple times when the command references more than 15 user IDs.

**dra1-3**

Internal operands DF/SMS uses

**nnnnn**

Five-digit number

**nnnnnnnnn**

Ten-digit number

# Chapter 7: Controlling Syntax Error Processing for Command Limiting

CA ACF2 for z/ VM command limiting checks each CP command to ensure that the command syntax is correct. By default, CA ACF2 for z/ VM takes the following actions when it detects a syntax error:

- Displays a message stating that it detected a syntax error.
- Rejects the command. It never reaches the IBM CP command processor. This avoids the overhead of rule validation for a syntax error and passing the command to CP, that would reject it.
- Does not write a logging or violation record regarding the syntax error.
- Does not update the user's violation count (SEC-VIO). This is important because most syntax errors are the result of honest typographical errors. Updating the user's violation count might cause a suspension of the user's logonid.

Remember, CA ACF2 for z/ VM only performs syntax checking for commands it validated. Specify these commands in the CMDLIM VMO record.

This section contains the following topics:



## Overriding the Defaults

CA ACF2 for z/ VM gives you three ways to handle a syntax error condition, described in hierarchical order:

**Logonid record field**

You can specify how you want CA ACF2 for z/ VM to react for each individual user when that user enters a syntactically incorrect command. This field is called SYNERR.

This is the highest level of override. When a user has the SYNERR field turned on, CA ACF2 for z/ VM processes the syntax error as defined in the logonid record, regardless of any other syntax error options (overrides both the COMMAND clause and the ACFFDR setting). If this level is null (or blank), CA ACF2 for z/ VM passes processing to the next level, command model options.

**Command model options**

For each CP command, you can use the SYNERR operand on the COMMAND clause to specify how you want CA ACF2 for z/ VM to react when it detects a syntax error.

This is the second level of override. The action specified in the SYNERR operand on the COMMAND clause overrides the option specified in the ACFFDR, but only for that specific command. If this level is null (or blank), CA ACF2 for z/ VM passes processing to the next level, VMO records.

**VMO records**

There is a field named SYNERR in the CMDLIM VMO record that sets the global syntax error option for the system. You can use one of the above options to override SYNERR.

If all of these options are null, CA ACF2 for z/ VM assumes PREVENT (no log).

# Syntax Error Options

As mentioned above, there are three ways to control syntax error processing: VMO records, command models, and logonid records. The options available for all three are basically the same. That is, they all accept an ALLOW, LOG, PREVENT, or PREVENT-LOG keyword or a null (blank):

**ALLOW**

When you set this option, CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CP issues standard error messages to the user. CA ACF2 for z/ VM does not log the syntax error to SMF.

**LOG**

When you set this option, CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CP issues standard error messages to the user. CA ACF2 for z/ VM also writes an SMF record that appears on the Command Limiting Journal (ACFRPTCL).

**PREVENT**

When CA ACF2 for z/ VM detects a syntax error, it sends the following message to the user and rejects the command:

```
ACFpgm274E  CA-ACF2 syntax error, operand number <operand>
- command <cmd> rejected
```

Also, CA ACF2 for z/ VM does not charge the user with a violation (CA ACF2 for z/ VM does not increment the SEC-VIO count because of the syntax error).

SYNERR=PREVENT is the default for CMDLIM VMO record and the CA ACF2 for z/ VM-supplied command models. CA ACF2 for z/ VM does not write a System Management Facility (SMF) record for syntax errors.

**PREVENT-LOG**

When you set this option, CA ACF2 for z/ VM prevents the execution of the command and logs the fact that the user entered the command incorrectly. This option does not cause a violation.

In the logonid record and command model, SYNERR( ) is the default setting. You can change this setting to SYNERR(LOG) or SYNERR(PREVENT-LOG) if you need SMF records for an individual user. These SMF records appear in the Command Limiting Journal (ACFRPTCL).

**null**

If you set the value of SYNERR= to null (blank), CA ACF2 for z/ VM passes error processing to the VMO record for checking.

# Logonids That Should Have the SYNERR Logonid Field

We recommend you assign SYNERR(LOG) or SYNERR(ALLOW) to the directory maintenance virtual machine (usually DirMaint) and the remote spooling communications subsystem (usually RSCS). These virtual machines rely on the standard CP syntax error return codes during their normal processing.

To include the SYNERR field in a user logonid record, use the CHANGE lid SYNERR(option) command, where option is ALLOW, LOG, PREVENT, or PREVENT-LOG. To remove the field from a user's logonid record, specify **SYNERR( )** .

If you do not change the value for SYNERR from the default of null (or blank), CA ACF2 for z/ VM passes error processing to the next hierarchical level for checking. The first level of checking is the logonid record attribute. If the SYNERR value is null, the next level of checking is the syntax model. If you did not specify a value for the SYNERR syntax model, CA ACF2 for z/ VM checks the VMO record. If CA ACF2 for z/ VM does not find a value for SYNERR there, it takes the default action of PREVENT (no logging).

# Chapter 8: VM Directory Command Limiting and Logging Support

The VM directory contains information for all authorized users on your system. These entries contain the user ID (user's identification), the password, virtual storage size, spool addresses, and the minidisk locations and sizes for each user. CA provides a directory maintenance program, CA-Director. IBM also provides a directory maintenance program, DirMaint. Both programs control access to the VM directory by restricting the additions of new minidisks, changing the size of an existing minidisk, deleting minidisks, and transferring ownership of minidisks.

CA ACF2 for z/ VM provides an optional interface to the DirMaint program product that controls:

- The execution of all DIRM commands by all users. All of these command executions are logged in the DirMaint Event Log (ACFRPTDL). For DirMaint logging records for the MDISK and MDPW functions, the user-specified password in the SMF records is Xed out unless you specified the ALL parameter for ACFRPTDL.
- All minidisk overlaps, to help guard against accidental disclosure, modification, or destruction of data on minidisks.
- The DirMaint command and its operands, so you can decentralize directory maintenance between multiple DirMaint privileged users.

DirMaint command limiting violations do not increment a user's violation count. CA ACF2 for z/ VM only increments this count for access violations. If access violations cause the user to reach the MAXVIO threshold, CA ACF2 for z/ VM forces him off.

This section contains the following topics:



## Important Installation Information

CA ACF2 for z/ VM supports DirMaint Version 1 Release 5 and Function Level 410 and above.

Support of the DirMaint command is optional. You must install it before validation occurs. If you decide to use DirMaint command limiting, you must turn the AUDIT bit of the DirMaint service machine logon ID record on immediately. If you do not turn AUDIT on, the DirMaint service machine issues error messages (prefixed by DirMaint) to the owner and operator, then logs itself off. For installation instructions, see the chapter "Installation Options" in the *Installation Guide*.

# Protecting the VM Directory in DirMaint

This section will help you control the VM directory using DirMaint. You should have the IBM DirMaint Command Reference that describes all the DirMaint commands handy for reference when writing rules.

## Rule Writing Guidelines for the DirMaint command

This section introduces the structure of the DirMaint command limiting rule sets. It also explains the scenario used in the sample rules in subsequent sections. CA ACF2 for z/ VM provides one $KEY (DirMaint) for all the DirMaint subcommands. CA ACF2 for z/ VM automatically logs the execution of any DirMaint command. CA ACF2 for z/ VM does not support the commands listed in this chapter for all DirMaint releases.

### Compiling DirMaint Models

Before you can write command limiting DirMaint rules, you must compile the models. To compile a syntax model for DirMaint commands, issue the following commands:

```
acf
ACF
set model
MODEL
compile DIRMxxxx
  .
```

where xxxx identifies your DirMaint release. Use DIRMR510 for Function Level 510 and DIRML530 for Function Level 530.

Rule sets with a key of DirMaint can prevent users from executing certain DirMaint commands on a DirMaint system:

```
$KEY(DirMaint) MDLTYPE(530)
```

You can now write rules to restrict or allow DirMaint commands without having to define individual users.

### Writing Initial Rules

As a very simple start, you could write the rules in this section when you install the CA ACF2 for z/ VM DirMaint logging feature. This rule set allows users do everything they normally do.

```
$KEY(DirMaint) MDLTYPE(530)
 FORUSER * - UID(*) ALLOW
```

The keyword DirMaint in the above rule lets all users execute all DirMaint commands.

## DirMaint Version 1 Release 5 and Above Command Syntax

DirMaint Version 1 Release 5 and above command structure follows:

```
DirMaint prefix-keywords command-string
```

**prefix-keywords**

Any of the several DirMaint options, such as FORUSER, ASUSER, and so on.

**command-string**

The actual DirMaint command string.

Most of the prefix-keywords affect how the command operates. However, the FORUSER keyword specifies who the command is issued for, which in many cases means what directory entry the command should actually affect. Because of this, the CA ACF2 for z/VM VM DirMaint Version 1 Release 5 interface constructs a command validation string as follows:

```
DirMaint FORUSER userid cmd-string
```

**DirMaint**

The command name and, therefore, the name of the rule ($KEY value) that is validated.

**FORUSER**

A constant value of FORUSER.

**userid**

The value that follows the FORUSER keyword. This value is normally * if the command issuer did not supply the FORUSER prefix-keyword, but the DirMaint default settings or the ASUSER prefix-keyword can override it.

**cmd-string**

The actual function to perform. This does not include the prefix-keywords.

Even if DirMaint Version 1 Release 5 is running with CMDLEVEL= 140A, CA ACF2 for z/VM validates the Version 1 Release 5 syntax because DirMaint converts the 140A compatibility syntax to Version 1 Release 5 syntax before calling the command validation exits.

For example, to allow MAINT to issue the command:

```
DirMaint FORUSER TESTUSER PURGE
```

The following rule entry applies:

```
$KEY(DirMaint)
 FORUSER TESTUSER PURGE UID(MAINT) ALLOW
```

This allows MAINT to PURGE the directory entry for ID TESTUSER. The following rule also applies:

```
$KEY(DirMaint)
 FORUSER * PURGE UID(MAINT) ALLOW
```

If you use the value of * in the rule for the FORUSER operand, any FORUSER value applies. In other words, the above rule allows MAINT to PURGE the directory entry for any user.

To allow a rule to only apply to the command issuer, such as when a command is issued without the FORUSER operand or a FORUSER value that matches the command issuer, then write the rule as:

```
$KEY(DirMaint)
 FORUSER OWNER MDPW *-  UID(*) ALLOW
```

The above rule allows any user to issue the MDPW command for its own minidisks, but not for another user's minidisks. OWNER is a special token that matches either * or the ID being validated.

## Commands with Special Rule Considerations

All DirMaint commands are validated as discussed previously, but there are some special considerations and multiple validations for some commands.

**ADD**

The ADD command is validated. Then CA ACF2 for z/ VM validates each MDISK you are adding with the user you are adding as an AMDISK command.

**CMDISK**

The CMDISK command is first validated as is and, if the command is allowed, then CA ACF2 for z/ VM validates a second time to verify the authority to delete the old extent. This second validation is a DMDISK validation using the VOLSER that the old minidisk is deleted from when the change is complete.

**DMDISK**

The DMDISK command is validated with one additional parameter. The VOLSER that the minidisk is on is added to the command. The VOLSER is inserted immediately after the virtual address of the minidisk you are deleting.

To allow ACC001 to delete any minidisk from the real DASD with VOLSER ACCPAK, the rule is:

```
FORUSER * DMDISK *- ACCPAK - UID(ACC001) ALLOW
```

**PURGE**

The PURGE command is validated. Then CA ACF2 for z/ VM validates each MDISK you are deleting with the user you are purging as a DMDISK command.

**REPLACE**

The REPLACE command is validated. Then CA ACF2 for z/ VM validates each MDISK that you are changing with both a DMDISK and AMDISK command. Any new minidisks are validated as an AMDISK command. Then, CA ACF2 for z/ VM validates any minidisk that will no longer exist in the replaced directory as a DMDISK command.

**RMDISK**

The RMDISK command is validated. Then CA ACF2 for z/ VM validates a DMDISK command for the old minidisk.

For each of these commands, all of the validations must be allowed for each command for DirMaint to actually process it.

## Adding Minidisks (AMDISK)

The sample rule set shown below implements the following controls:

- In the first rule entry, we allowed only ACCMGR to add minidisks to the directory on a group of volumes belonging to the group defined as ACC that are charged to the Accounting Department
- The second rule entry allows PAYMGR to add minidisks to the directory on a group of volumes belonging to the group defined as PAY that are charged to the Payroll Department.
- The third rule entry states the account manager can also add minidisks to any volume that begins with ACC00.
- The fourth rule entry states the payroll manager can add minidisks to any volume that begins with PAY00 as defined in the.
- The last rule entry lets SYSADM add minidisks to any volume serial.

```
$KEY(DirMaint)
$MODE(ABORT)
 FORUSER * AMDISK *- XXXX AUTOG *- ACC -  UID(ACCMGR) ALLOW
 FORUSER * AMDISK *- XXXX AUTOG *- PAY -  UID(PAYMGR) ALLOW
 FORUSER * AMDISK *- *- *- *- ACC00* - UID(ACCMGR) ALLOW
 FORUSER * AMDISK *- *- *- *- PAY00* - UID(PAYMGR) ALLOW
 FORUSER * AMDISK -  UID(SYSADM) ALLOW
```

The field XXXX is a place holder. When using a group value (AUTOG), do not define a volume serial. XXXX tells the system that we did not assign a volume serial. You cannot use fields other than XXXX as a placeholder because of the DirMaint command format.

## Changing Minidisks (CMDISK)

The sample rule set below illustrates how to control who can make changes to various minidisks.

- The first rule entry limits ACCMGR to changing only minidisks that reside on the ACC group of volumes.
- The second entry limits PAYMGR to changing only minidisks that reside on the PAY group of volumes.
- The third rule entry limits ACCMGR to changing minidisks to reside on any volume that starts with ACC00.
- The fourth rule entry limits PAYMGR to changing PAY00 minidisks to reside on any volume that starts with PAY00.
- The fifth entry lets SYSADM make changes using automatic allocation on any group of volumes.

```
$KEY(DirMaint)
 FORUSER * CMDISK *- DUMMY AUTOG *- ACC  UID(ACCMGR) ALLOW
 FORUSER * CMDISK *- XXXX AUTOG *- PAY UID(PAYMGR) ALLOW
 FORUSER * CMDISK *- *- *- *- ACC00*  UID(ACCMGR) ALLOW
 FORUSER * CMDISK *- *- *- *- PAY00* UID(PAYMGR) ALLOW
 FORUSER * CMDISK *- XXXX AUTOG *- -  UID(SYSADM) ALLOW
```

The field XXXX is a place holder. When using a group value (AUTOG), do not define a volume serial. XXXX tells the system that we did not assign a volume serial. You cannot use fields other than XXXX as a placeholder because of the DirMaint command format.

# Converting DirMaint Version 1 Release 4 Rules to Version 1 Release 5 and Above

For most cases, you should take your DirMaint Version 1 Release 4 rule and add the FORUSER keyword and value to the front of each rule entry. Then, if the rule is for one of the commands where “user to change” was a command operand, remove that operand from the rule. The FORUSER operand value replaces the “user to change” operand in the new DirMaint command syntax. ADD is an exception, since the user you are adding is still the first operand of ADD.

For example, to convert the following DirMaint Version 1 Release 4 rule:

```
$KEY(DIRMPRIV)
 ADD ACC***  UID(ACCMGR) ALLOW
 AMDISK *- *- DUMMY AUTOG *- ACC - UID(ACCMGR) ALLOW
 AMDISK *- *- *- AUTOV *- ACC00* - UID(ACCMGR) ALLOW
 CHVADDR ACC- *- TO - UID(ACCMGR) ALLOW
 CMDISK *- *- DUMMY AUTOG *- ACC UID(ACCMGR) ALLOW
 CMDISK *- *- *- AUTOV *- ACC00* - UID(ACCMGR) ALLOW
 DMDISK *- *- ACC00* - UID(ACCMGR) ALLOW
 LOCK ACC*** UID(ACCMGR) ALLOW
 TMDISK ACC- 019*- TO ACC- 019* UID(ACCMGR) ALLOW
 UNLOCK ACC***  UID(ACCMGR) ALLOW
```

Change the rule to:

```
$KEY(DirMaint)
 FORUSER * ADD ACC***  UID(ACCMGR) ALLOW
 FORUSER * AMDISK *- XXXX AUTOG *- ACC - UID(ACCMGR) ALLOW
 FORUSER * AMDISK *- *- AUTOV *- ACC00* - UID(ACCMGR) ALLOW
 FORUSER ACC- CHVADDR *- TO - UID(ACCMGR) ALLOW
 FORUSER * CMDISK *- XXXX AUTOG *- ACC UID(ACCMGR) ALLOW
 FORUSER * CMDISK *- *- AUTOV *- ACC00* - UID(ACCMGR) ALLOW
 FORUSER * DMDISK *- ACC00* - UID(ACCMGR) ALLOW
 FORUSER ACC*** LOCK UID(ACCMGR) ALLOW
 FORUSER ACC- TMDISK 019*- TO ACC- 019* UID(ACCMGR) ALLOW
 FORUSER ACC*** UNLOCK UID(ACCMGR) ALLOW
```

Notice that the $KEY has changed to DirMaint and XXXX is now used for the generic device type when you are allocating a minidisk with AUTOV or AUTOG instead of the DUMMY that was used in DirMaint Version 1 Release 4. For all of the commands except for ADD, the userid operand now follows FORUSER.

# Chapter 9: Syntax Model Command Language

Syntax Model Command Language (SMCL) is a facility that describes the valid command syntax for a CP command.

Before you can use the CA ACF2 for z/ VM command limiting feature, you must define one command model for each CP command that CA ACF2 for z/ VM will validate. Each model identifies a CP command to CA ACF2 for z/ VM, the various formats allowed for a command, and the attributes of every operand used in a particular command format. CA ACF2 for z/ VM uses a model during command limiting validation to determine the format of the command and to match the command to the appropriate command limiting rule.

Command models are supplied with the base CA ACF2 for z/ VM product for all standard CP commands. You do not have to alter the supplied command model unless you alter the format of a command. For example, if you add an operand to a standard CP command, you must modify the supplied model for that command to include the new operand.

A separate set of command models are supplied for each VM product level. These product levels and their file names are listed in the chapter “System Requirements” in the *Installation Guide*. CA ACF2 for z/ VM supports all levels of VM that IBM currently supports.

This section contains the following topics:



## Compiling Command Syntax Models

To compile a syntax model, issue the following commands:

```
acf
ACF
set model
MODEL
decompile fn
```

**fn**

The file name of the sample model file in the previous list.

**Note:** The default MDLTYPE from the CMDLIM VMO record is used unless the MDLTYPE is specified on each model in the file being compiled or the MDLTYPE parameter is included on the COMPILE command.

For more information on compiling command syntax models, see the *Installation Guide*.

Below is a sample compile of the syntax model.

```
*    ACFMOUNT COMMAND
 COMMAND ACFMOUNT
    FORMAT
       OPERAND VOLSER,6,TRAN=ANY
       OPERAND GROUP=OPTIONS
    FORMAT END
OPTIONS GROUP TYPE=OPTIONAL
          OPERAND VCUU,4,TRAN=VCUU
                              ADDRESS OF VIRTUAL TAPE DRIVE
          OPERAND LIST=((WRITE),             -
                        (READ,TYPE=DEFAULT))
         GROUP END
 COMMAND END
 .
 .
 .
 COMMAND WARNING
    FORMAT CLASS=AB
        OPERAND LIST=(                        -
                      (ALL,3),                -
                      (*,1,TRAN=SELF),        -
                      (OPERATOR,2),           -
                      (USERID,8,TRAN=USER)    -
                     )
        OPERAND MSGTEXT,231,TRAN=REST,OPTIONAL
    FORMAT END
 COMMAND END
```

For the sake of brevity, the above syntax model only shows the first and last CP commands in this model. For information about the names of the syntax model files, see the *Installation Guide*.

Many VM sites often develop complete CP commands locally. If this is the case at your site, be sure to define a command model for each locally-developed CP command. This lets CA ACF2 for z/ VM validate execution of the command using the standard command limiting feature.

# Components of a Model

Each command model describes a valid CP command, its formats, and its operands. Command models are created through the syntax model command language facility.

You use the following set of clauses to define a syntax model.

COMMAND clau**se**

Identifies a command by name. It signals the start of a command model definition and signals the end of a command model.

**FORMAT clause**

Identifies the VM privilege class required to use a particular format of a command. You can specify multiple FORMAT clauses for a single command to accommodate CP commands that have multiple formats.

**NEXTMDL clause**

Describes how you can break large command limiting models into smaller, more manageable syntax models.

**OPERAND clause**

Describes an individual command operand and its attributes.

**GROUP clause**

Describes groups of command operands that relate or depend on each other. Typically, a GROUP clause describes keyword operands, required operands, optional operands, and mutually exclusive operand groups.

**COMMENT clause**

Places comments in the command model.

**NULL clause**

Places blank (or null) lines in the command model to make reading a command model easier.

# Characteristics of a Command Model

Take a look at some of the CP commands that are documented in the IBM *CP Command Reference*. Notice that:

- Most CP commands can have multiple formats
- Different privilege classes of users have different operands available to them
- Optional operands can appear in any order or in a specified order
- Some commands have default operands
- Operands can be mutually exclusive
- Operands are sometimes preceded by a keyword descriptor
- Operands are sometimes repeated.

To illustrate some of these characteristics, review the standard syntax for the IPL command in the next figure.

```
Ipl       {vaddr [cylno ] {CLear  }              }
          {      [nnnnnn] {NOCLear} [STOP] [ATTN]}[PARM p1 [p2..[p32]]]

           systemname
```

While reviewing the IPL command syntax, notice:

- There are two command formats. You can IPL a vaddr or a systemname.
- When you IPL a vaddr, you can also specify one or more optional operands. There are two groups of optional operands. The first group includes cylno, nnnnnnnn, CLear, NOCLear, STOP, and ATTN. Both groups include (PARM p1 [p2..p32].
- Some operands have default values. For example, if you IPL a vaddr, then NOCLear is the default.
- Some operands are mutually exclusive, meaning you can only use one of them. For example, CLear and NOCLear are mutually exclusive.
- Notice how the PARM operand relates to p1. PARM is a keyword descriptor for p1, so that CA ACF2 for z/ VM does not confuse it with any other operand. When you specify PARM, you must also include at least one parameter (p1) following the keyword PARM. The parameters p2 through p32 are optional. In command model terminology, these parameters can occur 31 times.
- Some operands have constant values. We show them in caps. They include CLear, NOCLear, STOP, ATTN, and PARM.
- Operands that are constants do not have transposition routines. Other operands are variables. We show these in lower case. They include vaddr, systemname, cylno, nnnnnn, and p1...p32. Operands that are variables are always associated with a transposition routine.

The syntax model command language provides a relatively straightforward method for modelling a command. An example of how the IPL command is modelled follows:

```
Notes:    SMCL CLAUSES:

 1.       COMMAND IPL
             *
 2.              FORMAT CLASS=G
 3.                OPERAND vcuu,4,TRAN=VCUU
 4.                OPERAND GROUP=OPTIONS
 5.                OPERAND GROUP=PLIST
                  FORMAT END

 6.              FORMAT CLASS=G
 7.                OPERAND systemname,8,TRAN=ANY
 5.                OPERAND GROUP=PLIST
                  FORMAT END

 4.       OPTIONS GROUP TYPE=OPTIONAL
                    OPERAND LIST=((cylno,3,TRAN=HEX),     -
                                  (nnnnnn,6,TRAN=DECIMAL))
                    OPERAND LIST=((CLEAR,2),              -
                                  (NOCLEAR,4,TYPE=DEFAULT))
                    OPERAND STOP,4
                    OPERAND ATTN,4
                  GROUP END

 5.       PLIST    GROUP TYPE=KEYWORD
                    OPERAND PARM,4
                    OPERAND P1,8,TRAN=ANY
                    OPERAND P2,8,TRAN=ANY,OCCURS=31,OPTIONAL
                  GROUP END
 8.       COMMAND END
```

Most of the supplied models shown in this guide apply to VM/HPO Release 4.2 systems.

## Notes on SMCL Clauses

1. This is the SMCL for the IPL command. It signifies the beginning of the IPL command.
2. This is the first format of the IPL command. All the operands for this command format are described in the boundaries of the FORMAT clause (you must describe all operands before the FORMAT END clause). There is also a second format allowed for IPL, but it is described in its own FORMAT clause (see Notes 6 and 7).
3. The first operand in the first format is vcuu that has a maximum length of four characters. Because there is no way to know what value vcuu is when you enter the command, CA ACF2 for z/ VM uses a transposition routine (TRAN=VCUU) when you enter the IPL command. This verifies that the value entered is a valid hexadecimal value.
4. The group called OPTIONS describes several optional operands.

   ```
   OPTIONS GROUP TYPE=OPTIONAL
             OPERAND LIST=((cylno,3,TRAN=HEX),         -
                           (nnnnnn,6,TRAN=DECIMAL))
             OPERAND LIST=((CLEAR,2),                  -
                           (NOCLEAR,4,TYPE=DEFAULT))
             OPERAND STOP,4
             OPERAND ATTN,4
           GROUP END
   ```

   See how the CLEAR and NOCLEAR operands are described in the model. These are called mutually exclusive operands, meaning you can choose either CLEAR or NOCLEAR, but not both. The 2 following CLEAR and the 4 following NOCLEAR indicate the minimum length of these operands. NOCLEAR is identified as the default operand through a TYPE=DEFAULT verb. The GROUP END clause signals the end of this operand group.
5. There is another set of keyword operands that are described in a group called PLIST.

   ```
   PLIST  GROUP TYPE=KEYWORD
            OPERAND PARM,4
            OPERAND P1,8,TRAN=ANY
            OPERAND P2,8,TRAN=ANY,OCCURS=31,OPTIONAL
          GROUP END
   ```

   The first parm value (P1) means that you must enter at least one value whenever you specify the PARM keyword.

   If you specify PARM with nothing following it, CA ACF2 for z/ VM stops looking at the command model and signals a syntax error.
6. The OCCURS=31 verb indicates that you can enter up to 31 parm values (p2,...,p32). However, the OPTIONAL verb says that p2, p3, are optional and you do not need to specify them. The GROUP END clause signals the end of this operand group.

7. This is the second format of the IPL command. All the operands for this command format are described in the boundaries of the FORMAT clause (all operands are described before the FORMAT END clause).

   The first operand in the second format is systemname that has a maximum length of eight characters.

   Because there is no way to know what value systemname is when you enter the command, a transposition routine (TRAN=ANY) is used when you enter the IPL command. This verifies that the length of systemname is one to eight characters. This FORMAT clause also references the GROUP clause labeled PLIST (see Note 5).

8. The COMMAND END statement signals the end of this command model.

   The next section describes each syntax model command language clause and its verbs in detail.

# Elements of a Command Model

## COMMAND Clause

The COMMAND clause signals both the start and end of a command model. We show the full syntax of the COMMAND clause. Explanations of the verbs for the COMMAND clause follow the syntax.

### COMMAND Start Syntax

```
COMMAND command name,                                    {-}
       [{OPERAND=OPTIONAL}],                             {-}
       [{REPEATS}],                                      {-}
       [{(MDLTYPE=(ccc}],                                {-}
       [{NOSPOOL=ALLOW|LOG|PREVENT|PREVENT-LOG}],-       {-}
       [{SYNERR=ALLOW|LOG|PREVENT|PREVENT-LOG}]
```

### COMMAND End Syntax

```
COMMAND END
```

## Verb Descriptions (COMMAND Clause)

**command name**

Specifies the name of the CP command this command model applies to. This must be the full name of the command.

**OPERAND=OPTIONAL**

Indicates you can enter this command without an operand and there are no defaults for any optional operands. If you do not specify OPERANDS=OPTIONAL, CA ACF2 for z/ VM assumes that you must specify at least one operand with the command or that there is a default value for an optional operand. For example, the BEGIN command does not require an operand, nor does it have any default. Therefore, the command model for BEGIN includes:

```
COMMAND BEGIN,OPERAND=OPTIONAL

    FORMAT CLASS=G
       OPERAND HEXLOC,6,TRAN=HEX,OPTIONAL
    FORMAT END

COMMAND END
```

Although ECHO does not require you enter an operand with the command, it does provide a default for an optional operand. Therefore, ECHO does not need an OPERAND=OPTIONAL subparameter.

```
COMMAND ECHO

    FORMAT CLASS=G
       OPERAND LIST=(NN,2,TRAN=DECIMAL),  -
                    (1,1,TYPE=DEFAULT,    -
                     TRAN=DECIMAL))
    FORMAT END

COMMAND END
```

**REPEATS**

Indicates you can repeat all the formats of this command. To illustrate how this works, review the following excerpt from the TERMINAL model and some example commands:

```
COMMAND TERMINAL,REPEATS
```

For example, in the TERMINAL CHARDEL OFF LINESIZE 20 MODE VM command, CP and CA ACF2 for z/ VM treat this as three separate commands:

```
TERMINAL CHARDEL OFF
TERMINAL LINESIZE 20
TERMINAL MODE VM
```

Use REPEATS on the COMMAND clause when you can repeat all the formats for a command. The only COMMAND clause that currently uses REPEATS is the TERMINAL command. If only some formats repeat, then specify REPEATS in the FORMAT clause as described in the next section of this guide.

**MDLTYPE=ccc**

Compiles a model with an MDLTYPE that is different from the default MDLTYPE. The default is set in the MDLTYPE operand of the CMDLIM VMO record.

**NOSPOOL=ALLOW|LOG|PREVENT|PREVENT-LOG**

Specifies how CA ACF2 for z/ VM handles a no spool file found condition.

**ALLOW**

CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CA ACF2 for z/ VM does not write an SMF record.

**LOG**

CA ACF2 for z/ VM logs the error to SMF and then passes the command to CP to check for CP syntax. CP returns standard CP error messages to the user if it also detects a no spool error. You can use this option to test command models to determine if they are correct.

**PREVENT**

CA ACF2 for z/ VM rejects the command and returns a nospool message to the user. PREVENT is the default.

**PREVENT-LOG**

CA ACF2 for z/ VM rejects the command and logs the action. Does not cause a violation.

**SYNERR=ALLOW|LOG|PREVENT|PREVENT-LOG**

Specifies how CA ACF2 for z/ VM handles a command syntax error condition.

**ALLOW**

CA ACF2 for z/ VM passes the command to CP for normal syntax checking. CA ACF2 for z/ VM does not write an SMF record. You can abbreviate this parameter with A.

**LOG**

CA ACF2 for z/ VM logs the syntax error to SMF and then passes the command to CP, where it goes through normal CP syntax checking. You can abbreviate this parameter with L.

Standard CP error messages are returned to the user if CP also detects a syntax error. You can use this option to test command models to determine if they are correct.

**PREVENT**

CA ACF2 for z/ VM rejects the command and returns a syntax error message to the user. This is the default. You can abbreviate this parameter with P.

**PREVENT-LOG**

CA ACF2 for z/ VM rejects the command and logs the action. Does not cause a violation. You can also set a global system default value for this option. For more information, see the chapter “Controlling Syntax Error Processing for Command Limiting.”

**END**

Signals the end of the command model.

```
COMMAND command-name
          .
          .
          .
COMMAND END
```

# FORMAT Clause

The FORMAT clause signals both the start and end of a single format of a command. Most CP commands have multiple formats. We show the full syntax of the FORMAT clause below.

## FORMAT Start Syntax

```
FORMAT [{CLASS=cccccccc,}]              {-}
       [{REPEATS}]
```

## FORMAT End Syntax

```
FORMAT END
```

# Verb Descriptions (FORMAT Clause)

**CLASS=cccccccc**

Identifies the VM privilege class of this command. You can specify up to 32 classes. This is useful for commands, such as QUERY, that have the same format for several privilege classes.

CLASS defaults to A-Z and 0-6. Therefore, do not specify CLASS if any privilege class user (all users) can use the command. Also remember not to accidentally specify CLASS=ALL or CLASS=ANY as these are interpreted literally as classes A, L, L or A, N, Y. See the supplied DIAL model for an example of a command format that applies to all privilege classes.

**REPEATS**

Indicates that this format can occur more than once. Use REPEATS on the FORMAT clause when only a particular command format repeats. For example, following is an excerpt from the supplied CPTRAP model:

```
COMMAND CPTRAP

   FORMAT CLASS=C,REPEATS
      OPERAND TYPENUM,2,TRAN=HEX
      OPERAND GROUP=ENTRYTYP
   FORMAT END

ENTRYTYP GROUP TYPE=OPTIONAL
               .
               .
               .
         GROUP END
COMMAND END
```

In the above example, REPEATS indicates this format of the CPTRAP command repeats. Use REPEATS on the COMMAND clause when you can repeat all the formats for a command.

**END**

Signals the end of this format for the command.

```
FORMAT CLASS=AG
   .
   .
   .
FORMAT END
```

## NEXTMDL Clause

The NEXTMDL clause lets you break large command models into smaller, more manageable models. The full syntax of the NEXTMDL clause is NEXTMDL model_name.

## Verb Descriptions (NEXTMDL Clause)

The model_name clause identifies the name of the model to start using. Most special characters (such as a period), underscores, and bars are valid. The model name cannot contain a comma, blank, or parentheses as they conflict with the compiler's delimiting characters. The model name can be up to 12 characters long and cannot be NEXTMDL.

When the CMDLIM interpreter encounters a NEXTMDL clause while scanning the FORMATS for a match, it releases the current model and brings in a model of the same MDLTYPE as the current model and with the model_name specified in the NEXTMDL clause. It then begins using the new model. The CMDLIM interpreter ignores the rest of the current model. It enters NEXTMDL on the same level as a FORMAT clause. NEXTMDL must appear after the end of the last FORMAT and the start of any GROUPs.

## OPERAND Clause

CP commands use many different types of operands. The syntax model command language provides a number of ways to define the attributes of a command operand. It is very important that you define operands correctly in the command models. Therefore, knowing the attributes of the operand is essential.

We explain eight formats of the OPERAND clause in this section. You can use each format to describe a different type of operand.

**Constants (format 1)**

Describes an operand that always has a constant value. For example, all keyword operands are constants.

**Variables (format 2)**

Describes an operand where you enter a value. For example, all device address operands are variables.

**Groups (format 3)**

Describes a group of related operands by calling a GROUP clause. Each GROUP clause contains one or more OPERAND clauses.

**Spool Related (format 4)**

Describes operands that are related to spool files or unit record device names.

**Another Operand's Default (format 5)**

Signals that the value specified for an operand is also the default value for another operand. At present, only the supplied ATTACH model uses this format.

**Effector Operands (format 6)**

Describes operands that affect the way CP processes other operands. For example, many supplied models use this format to describe operands that accept a device address range (DETACH 0180-0188).

**Mutually Exclusive (format 7)**

Describes two or more mutually exclusive operands.

**Interpretation Control (format 8)**

Overrides the normal flow of a command as it goes through the CA ACF2 for z/ VM model interpreter. This is very useful for describing a command that has many similar formats.

We describe each of these formats individually with examples of how to use them in the CA ACF2 for z/ VM-supplied models.

## Constant Operands (Format 1)

Use this form of the OPERAND clause to describe an operand that has a constant value.

```
[{label} OPERAND value[{,minimum-length}][{,OPTIONAL}]       {-}
                      [{,TYPE=DEFAULT}]
```

In the sample IPL model shown in Characteristics of a Command Model in this chapter, the operand STOP was described as OPERAND STOP,4. This operand statement describes the STOP operand with a minimum abbreviation of four characters. Had you omitted the 4, the minimum length defaults to the number of characters in STOP, which is four.

Consider the following portion of the VARY model: OPERAND ONLINE,2. Here the minimum abbreviation is two characters, and acceptable specifications are: ON, ONL, ONLI, ONLIN, and ONLINE. The label verb is optional and not really needed in the above examples. For more information, see the VALUEFOR verb and Format 5 in this chapter.

## Variable Operands (Format 2)

This form of the OPERAND clause describes an operand whose value is a variable term. The transposition routine validates the value you enter and, in some cases, converts it to a common value so that differences between a command and a rule can be resolved. You can specify the value for any OPERAND clause that includes a transposition routine as a pseudo operand in a command limiting rule.

```
[{label} OPERAND value[{,maximum-length}],TRAN=routine     {-}
                      [{,OPTIONAL}][{,TYPE=DEFAULT}]
```

In the sample IPL model shown earlier, the operand vcuu was described as OPERAND vcuu,4,TRAN=VCUU. This operand statement describes the operand VCUU with a maximum length of four positions. CA ACF2 for z/ VM calls the transposition routine VCUU (TRAN=VCUU) at command interpret time to see if the operand being interpreted is a valid hexadecimal number. If it is, the transposition routine returns a binary value. If it is not, CA ACF2 for z/ VM returns a nomatch condition. Had we omitted the 4, the maximum length defaults to 4, the number of characters in VCUU.

If you use the TYPE=DEFAULT verb, the value you specify must correspond to the transposition routine you specify. For example, in the supplied BACKSPAC model, the default number of pages to be backspaced is decimal 1, as defined in OPERAND 1,1,TYPE=DEFAULT,TRAN=DECIMAL.

For a complete list of all the transposition routine names, see the section Transposition Routines for Command Limiting.

## Call GROUP Clause (Format 3)

This OPERAND clause calls a set of logically related operands or operands that can occur in multiple formats.

```
[{label}] OPERAND GROUP=groupname
```

The groupname corresponds to the label that is on a GROUP clause contained in the limits of the command description (between the COMMAND BEGIN and COMMAND END clauses). When this clause is found by the syntax interpreter, it operates as you called a subroutine. For example, in the sample IPL model shown earlier, the PARM options are described as OPERAND GROUP=PLIST, where PLIST is the label on the PLIST group that describes a logical set of operands that belong together as:

```
PLIST GROUP TYPE=KEYWORD
       OPERAND ...
         .
         .
         .
      GROUP END
```

When the OPERAND GROUP=PLIST clause is found, it performs the PLIST group. When the GROUP END statement is found in the PLIST group, control returns to the statement immediately following the OPERAND GROUP=PLIST clause.

## Spool Related Operands (Format 4)

This OPERAND clause describes an operand that selects spool files and the unit record devices available to the operand.

```
[{label}] OPERAND value [{,maximum-length}],TRAN=routine,    {-}
               SPOOLOPT=({ALL|,CON,|{PRT|PUN,RDR}}          {-}
               [SELECT] [{,DEVNONLY,|RADRONLY}])
```

**SPOOLOPT=(PRT,RADRONLY)**

You can find one example of how you can use this format in the BACKSPAC model supplied with CA ACF2 for z/ VM: OPERAND PRT,4,TRAN=RUR,SPOOLOPT=(PRT,RADRONLY). In this example, you can specify a printer. The transposition routine TRAN=RUR examines the SPOOLOPT specification PRT,RADRONLY. The operand passes the syntax check if it is a real printer (PRT) that is described to CP and only if you use a real device address (RADRONLY). CA ACF2 for z/ VM checks the RDEVBLOK to ensure the device address cites a real device and that it is a printer.

**SPOOLOPT=(PUN,RADRONLY)**

The example OPERAND PUN,3,TRAN=RUR,SPOOLOPT=(PUN,RADRONLY) is the same as the previous one, except that you can only specify a punch (PUN) device, provided you refer to its real device address.

**SPOOLOPT=RDR**

The example OPERAND RDR,6,TRAN=VUR,SPOOLOPT=RDR is taken directly from the CLOSE model supplied with CA ACF2 for z/ VM. Here, you can specify a virtual reader. Because you specified neither RADRONLY nor DEVNONLY, you can also enter the reader name. For example, you can specify the virtual reader as R, Re, Rea, Read, Reade, Reader, RDR, or any virtual address that is a reader (usually 000C).

**SPOOLOPT=(CON,PRT,PUN)**

The example OPERAND VUR,7,TRAN=VUR,SPOOLOPT=(CON,PRT,PUN) is similar to the previous one, except that you can only specify a console, printer, or punch device.

**SPOOLOPT=(PRT,PUN,RDR,DEVNONLY,SELECT)**

The example OPERAND RUR,7,TRAN=RUR,SPOOLOPT=(PRT,PUN,RDR,DEVNONLY,SELECT) is taken directly from the CHANGE model supplied with CA ACF2 for z/ VM. Here, you can specify Printer, PRT, PTR, Punch, PCH, Reader, or RDR, provided they are cited by device name. You can also use this operand to select items from the spool queue. The SELECT suboperand of the SPOOLOPT verb is explained in the next section.

**SPOOLOPT=SELECT**

You can only use SPOOLOPT=SELECT with the following transposition routines:

- ALLSPFIL
- ALLURDEV
- CLASS
- CPSYSTEM
- DEST
- FORM
- RUR

- SELF
- SPOOL
- SPOOLTO
- VUR
- USER

You can only use SPOOLOPT=({,CON{,PRT{,PUN{,RDR}}}}), {,DEVNONLY| ,RADRONLY}) with the RUR and VUR transposition routines.

You can only use SPOOLOPT=({ALL}{,DEVNONLY| ,RADRONLY}) with the ALLURDEV transposition routine.

## Device Address Default (Format 5)

This OPERAND clause describes a variable term operand and specifies that the value selected for this operand is the default setting for another operand. The transposition routine validates the value supplied for this operand and converts it to a common value.

```
[{label}] OPERAND  value                                    {-}
                  [{,maximum-length}]                       {-}
                  ,TRAN=routine                             {-}
                  [{,OPTIONAL}]                             {-}
                  [{,VALUEFOR=label}]
```

To illustrate how you use this format, review the ATTACH model we ship with CA ACF2 for z/ VM.

```
        FORMAT CLASS=B
          OPERAND RCUU,4,TRAN=RCUU,VALUEFOR=ATTACHAS
              .
              .
              .
ATTACHAS OPERAND VCUU,4,TRAN=VCUU,OPTIONAL
              .
              .
              .
        FORMAT END
```

In this example, the operand value supplied for the first operand (RCUU) is the default VALUEFOR the operand labeled ATTACHAS.

To further explain how this works, review this sample ATTACH command: ATTACH 0580 TO MAINT. When this command is processed by CP, real device 0580 is attached to MAINT as virtual device 0580. The CA ACF2 for z/ VM interpreter processes operand 0580 (the RCUU OPERAND clause) and uses 0580 as the default value for the VCUU OPERAND clause.

In many cases though, the operator probably issues ATTACH 0580 TO MAINT AS 0181. When this command is processed by CP, real device 0580 attaches to MAINT as virtual device 0181. The CA ACF2 for z/ VM interpreter processes operand 0580 against the RCUU OPERAND clause and uses 0580 as the default value for the VCUU OPERAND clause. When the 0181 operand is processed against the VCUU OPERAND clause, the 0181 supersedes the default value of 0580 in that position.

## Effector Operands (Format 6)

This format of the OPERAND clause describes operands that affect the way CA ACF2 for z/ VM processes other operands.

```
label OPERAND  value                                          {-}
               [{,maximum-length}]                            {-}
               [{,OCCURS=nnn}]                                {-}
               [{,TRAN=routine}]                              {-}
               {,TYPE=APREVADR|NONEXCL|NXTOPDEF|RANGE|SINGULAR|STOR}
```

**TYPE=STORADDR and TYPE=APREVADR**

To illustrate how the TYPE verb works, review the STCP model we ship with CA ACF2 for z/ VM. Here you can see a use for TYPE=APREVADR and TYPE=STORADDR. We use them in the STCP model as follows:

```
OPERAND HEXLOC,8,TRAN=STORADDR,TYPE=STORADDR

OPERAND HEXWORD,16,TRAN=HEX,TYPE=APREVADR
```

If you issue the STCP 20000 07000700 command, the 20000 is a storage address. CA ACF2 for z/ VM interprets the TYPE=STORADDR verb and saves the storage address so that, when CA ACF2 for z/ VM processes the data to be stored (HEXWORD), it updates the affected storage address.

When CA ACF2 for z/ VM processes the 07000700, the TYPE=APREVADR verb tells CA ACF2 for z/ VM to add that the number of bytes in this hexword to the previous storage address. After interpretation, as a result of this command, CA ACF2 for z/ VM stores 07000700 in locations 20000 through 20003. TYPE=STORADDR is only effective when you use it with TYPE=APREVADR, as shown in the above example.

**TYPE=NONEXCL and TYPE=NXTOPDEF**

To illustrate how to use the TYPE=NONEXCL and TYPE=NXTOPDEF, review the following excerpts from the SPOOL model.

```
CHAR      GROUP=KEYWORD
             OPERAND CHARS,2
             OPERAND NAME,4,TRAN=ANY
             OPERAND GROUP=MORECHAR
          GROUP END

MORECHAR GROUP TYPE=KEYWORD,OCCURS=3
             OPERAND CHARS,2,TYPE=NXTOPDEF
             OPERAND NAME,4,TRAN=ANY,TYPE=NONEXCL
          GROUP END
```

If you issue the SPOOL PRT CHARS nam1 CHARS nam2 nam3 FORM STD command, CA ACF2 for z/ VM processes the command as:

**nam1**

Fills in the CHARS and NAME operand clauses in the CHAR GROUP clause.

**nam2**

Fills in the first occurrence of the CHARS and NAME operand clauses in the MORECHAR GROUP clause.

**nam3**

Compared against the second occurrence of the CHARS operand clause and gets a nomatch condition. Because you specified the TYPE=NXTOPDEF (next operand's default), CA ACF2 for z/ VM saves the location of this clause. The nam3 is then compared to the NAME operand clause. Because you specified TYPE=NONEXCL (nonexclusive) verb, and it was not preceded by the keyword CHARS, CA ACF2 for z/ VM examines the rest of the format to see if nam3 fits anywhere else. Because it does not, nam3 is plugged into the NAME clause and, because there is an outstanding NXTOPDEF, replaces chars in the NAME clause.

**FORM**

When processed, it does not match to CHARS and does not fit the NAME operand clause because it matched another operand elsewhere in the format.

TYPE=NONEXCL is only effective when used with TYPE=NXTOPDEF, as shown in the above example. However, you can use TYPE=NXTOPDEF by itself.

**TYPE=RANGE and TYPE=SINGULAR**

The TYPE=RANGE verb says that this operand clause can accept an operand that is a range and contains both a low and high number. The DETACH model contains an example of this: OPERAND RCUU,7,TRAN=RCUU,TYPE=RANGE. You can only specify TYPE=RANGE with the TRAN=RCUU, TRAN=VCUU, TRAN=LDEVXA, and TRAN=LDEV routines. You do not have to specify TYPE=SINGULAR because it is the default setting for all operand clauses.

**OCCURS verb**

The OCCURS=nnn verb indicates you can specify this particular operand more than once. The IPL model shows how the OCCURS=nnn works in an OPERAND clause: OPERAND,P2,8,TRAN=ANY,OCCURS=31,OPTIONAL. In this example, you can specify up to 31 separate PARM operand values.

You can write only one rule operand to cover the entire operand clause. This is a permanent restriction. However, if this does not suit your needs, break the syntax model down to be more finite:

```
OPERAND P2,8,TRAN=ANY,OPTIONAL
OPERAND P3,8,TRAN=ANY,OPTIONAL
     .
     .
     .
OPERAND P32,8,TRAN=ANY,OPTIONAL
```

Operand clauses, like the one above, are for variable data that is usually application-specific. The TAG command contains a good example.

The tagtext in the command tells an application to do a specific function. For example, TAG DEV PRT CHICAGO tells the application to print on the device named Chicago. RSCS uses the tagtext to control routing of spool files. You define to RSCS all of the link IDs that RSCS can talk to. To control the IDs a user could send to, modify the syntax model, then write rules to cover it. For example, if you had nodes of New York, Chicago, Las Vegas, and San Diego in your network, you could make your own version of the model by adding the following format first in the TAG model:

```
OPERAND DEV,2
OPERAND VUR,7,TRAN=VUR,SPOOLOPT=(CON,PRT,PUN,RDR)
OPERAND LIST=((NEWYORK,8),  -
              (CHICAGO,7),  -
              (LASVEGAS,8), -
              (SANDIEGO,8), -
             )
FORMAT END
```

With the above format, you can easily write rules that apply to your needs. To limit Joe to only send to New York, Sue to Chicago, and Ann anywhere, you can write a rule as:

```
$KEY(TAG) MDLTYPE(-)
 DEV VUR NEWYORK UID(JOE) ALLOW
 DEV VUR CHICAGO UID(SUE) ALLOW
 DEV VUR - UID(ANN) ALLOW
```

## Mutually Exclusive Operands (Format 7)

Format 7 of the OPERAND clause defines mutually exclusive operands. Mutually exclusive means that you can choose one operand from the list, but not two. In this inline list format, the value-list specification can include any of the verbs available in the OPERAND clause. For example, you can use all of the verbs described in formats one through six and eight in format seven.

```
[{label}] OPERAND  LIST=((value-list),                    {-}

                  .          {-}
                  .          {-}
                  .          {-}
          (value-list))
```

There are many examples of the format 7 OPERAND clause in the models supplied with CA ACF2 for z/ VM. We took the example that follows from the IPL model. It shows how we defined the mutually exclusive operands CLEAR and NOCLEAR.

```
OPERAND LIST =((CLEAR,2),         -
               (NOCLEAR,4))
```

The BACKSPAC model contains a more intricate example of a format 7 OPERAND clause:

```
OPERAND LIST=((FILE,1),        -
              (GROUP=PAGES),   -
              (1,1,TYPE=DEFAULT,TRAN=DECIMAL))
```

There is no limit on the number of operands you can specify in a mutually exclusive list. However, there is a size limit of 4088 bytes for a compiled command model. See the NEXTMDL clause for information on using more than one model to describe a command.

## Interpretation Control (Format 8)

This format OPERAND clause includes additional verbs that alter the flow of command interpretation as it proceeds through a command model. You can use all of the verbs shown in format eight in formats one through seven.

```
[{label}]       OPERAND value-clause,                          {-}
                    { [,MATCH=CONTINUE|EXIT,]                } {-}
                    { [NOMATCH=CONTINUE|EXIT|EXITERR|NEXTFMT] }
                      [OPTIONAL]
```

The MATCH and NOMATCH verbs alter the flow a command follows during syntax interpretation. The default value for these verbs are globally set and managed by the CA ACF2 for z/ VM command model compiler. In most cases, you do not have to specify a MATCH or NOMATCH verb. However, some of the supplied models use the NOMATCH verb.

The ATTACH model contains a good example of how to use the NOMATCH=NEXTFMT verb. The ATTACH command has three very similar formats as shown in the following figure.

```
 COMMAND ATTACH
    FORMAT CLASS=B
        OPERAND RCUU,4,TRAN=RCUU
        OPERAND TO,2,TYPE=DEFAULT
        OPERAND SYSTEM,6,NOMATCH=NEXTFMT
        OPERAND AS,2,TYPE=DEFAULT
        OPERAND VOLID,6,TRAN=ANY
        OPERAND 3330V,S,OPTIONAL
        OPERAND VOLID,6,TRAN=ANY,OPTIONAL
    FORMAT END

    FORMAT CLASS=B
        OPERAND RCUU,4,TRAN=RCUU,VALUEFOR=ATTAS
        OPERAND TO,2,TYPE=DEFAULT
        OPERAND LIST=(                  -
                      (*,1,TRAN=SELF), -
                      (USERID,8,TRAN=USER,NOMATCH=NEXTFMT) -
                     )
        OPERAND AS,2,TYPE=DEFAULT
ATTAS   OPERAND VCUU,4,TRAN=VCUU,OPTIONAL
        OPERAND R/O,1,OPTIONAL
        OPERAND 3330V,S,OPTIONAL
        OPERAND VOLID,6,TRAN=ANY,OPTIONAL
    FORMAT END
   FORMAT CLASS=B
       OPERAND LIST=((RCUU,7,TRAN=RCUU,TYPE=RANGE),      -
                     (RCUU,4,TRAN=RCUU,OCCURS=48))
       OPERANDTO,2,TYPE=DEFAULT
       OPERAND LIST=(                                    -
                     (*,1,TRAN=SELF),                    -
                     (USERID,8,TRAN=USER)                -
                    )
       OPERAND R/O,1,OPTIONAL
       OPERAND 3330V,S,OPTIONAL
   FORMAT END
COMMAND END
```

As you can see, all three formats begin with the RCUU operand. The NOMATCH=NEXTFMT properly interprets the ATTACH command. For example, if CA ACF2 for z/ VM determines that the command does not match the first FORMAT clause, it checks the next FORMAT clause for a match, and so on until it finds the proper command format.

## Verb Descriptions (OPERAND Clause)

The verbs you can specify in an OPERAND clause are:

**OPERAND**

Starts an OPERAND clause.

**value-clause**

Specifies the expanded value of a token. Basically there are two types:

**constants**
Specify the full name of the constant. For example PARM, STOP, and ATTN, as shown in the IPL command. See the format 1 OPERAND clause for an example.

**variables**
Specify the same name IBM uses in the command syntax. For example, the IBM syntax for the IPL command uses systemname, vcuu, cylno, and nnnnnn to show that you can IPL a named system, a virtual device, a specific cylinder address, or a virtual block address, respectively. See the format 2 operands for an example.

**minimum-length**

Specify the CP-defined minimum acceptable length (abbreviation) of the operand. Only constant values use this minimum length. See the format 1 operands for an example.

**maximum-length**

Specify the maximum length of the operand. Only variable values (OPERAND clauses that specify a TRANS routine) use the maximum length. See the format 2 OPERAND clause for an example.

**OCCURS=nnn**

Specify the number of times this operand can repeat. For example, in the PARM p1(p2,...,p32) option of the IPL command, the parameters (p2,...,p32) can occur thirty-two times. The default is OCCURS=1 and the maximum is OCCURS=119. See the format 6 OPERAND clause for an example.

**MATCH=CONT|EXIT**

Informs the command interpreter what action to take when the operand matches the criteria described by this OPERAND clause.

**CONT**

Specifies to continue going through this command format when this operand matches this OPERAND clause.

**EXIT**

Stops the interpreter from looking at this OPERAND clause and returns the interpreter to the next higher level of the model. Checks other specified OPERAND clauses.The GROUP and FORMAT clauses globally set the default settings for these options. See the format 8 OPERAND clause for an example.

**NOMATCH=(CONT|EXIT|EXITERR|NEXTFMT)**

Informs the command interpreter what action to take when the operand does not match the criteria described for this OPERAND clause. If a default value is assigned for this operand, it is used. The command model compiler and interpreter globally set the defaults for these options. See the format 8 OPERAND clause for an example.

**CONT**

Specifies to continue going through this command format. If you assigned a default value for this operand, CA ACF2 for z/ VM uses it.

**EXIT**

Stops the interpreter from looking at this OPERAND clause and returns the interpreter to the next higher level of the model. Checks other specified OPERAND clauses.

**EXITERR**

Specifies the interpreter is to exit from the model and indicates a syntax error occurred. Either CP or CA ACF2 for z/ VM sends a syntax error message to the user. CP issues the error message if you specified SYNERR=LOG or SYNERR=PREVENT in the COMMAND clause for this model or specified globally in the CMDLIM VMO record. By default, CA ACF2 for z/ VM issues the syntax error message because SYNERR=PREVENT is the global system default and the default for the COMMAND clause.

**NEXTFMT**

Specifies the interpreter is to exit from this particular command format and use the next format, if one is available. This is useful for commands that have similar formats to one command, such as CHANGE.

**OPTIONAL**

OPTIONAL means that you do not need to enter this operand in the command. The interpreter continues with the next OPERAND.

**TRAN=routine**

Names a transposition routine that validates the value specified for this operand. In some cases, the routine also transposes the value entered for the operand into a common value used during CA ACF2 for z/ VM rule checking. For additional information about these transposition routines, see the appendix “Transposition Routines for Command Limiting.” See the format 2 OPERAND clause for an example.

**TYPE=operand-type**

Identifies an operand type. Most of these operand types are similar to the operand names used in commands. You can describe operand types you can use in your locally-written CP commands. Valid operand types are:

**APREVADR**

This operand affects the previous storage address. For example the 0102 (hexdata) in STCP S20000 0102 means the range is 20000-20001. Refer to the format 6 OPERAND clause for an example.

**DEFAULT**

This operand is a default for this token slot. CA ACF2 for z/ VM uses it i you did not choose an overriding operand (for example, the TO in SPOOL PRT TO userid). Refer to the format 4 OPERAND clause for an example.

**NONEXCL**

This is a nonexclusive operand if not proceeded by the previous operand (for example, the namen in CHANGE spoolid CHARS namen). Refer to the format 6 OPERAND clause for an example.

**NXTOPDEF**

This operand is the next operand default if another operand is present (for example, the CHARS in CHANGE spoolid CHARS namen). Refer to the format 6 OPERAND clause for an example.

**RANGE**

This operand contains a range of values (for example, DET 190-192 and DCP M20000:20200, 20000.200). Refer to the format 6 OPERAND clause for an example.

**SINGULAR**

Operand contains one value (for example, 0190 in DET 0190). Refer to the format 6 OPERAND clause for an example.

**STORADDR**

This operand is a storage address whose range can be affected by a following operand (for example, the 0A0B0C0D in STCP 20000 0A0B0C0D affects the storage range, which is really 20000-20003). Refer to the format 6 OPERAND clause for an example.

**SPOOLOPT=spool-opt**

Describes operands that are related to spool files. For examples, see the format 4 OPERAND clause.

Valid spool-opts are:

**ALL**

Specifies any unit record device.

**CON**

Lets you enter CONSOLE, CON, or a console device address for this operand.

**DEVNONLY**

Requires you to refer to unit record devices by name (for example, PUNCH, PUN, PU, PUNC, or PUN for PUNCH). Refer to the supplied TRANSFER model. Omit both DEVNONLY and RADRONLY to specify unit record devices using a device name or a device address.

**PRT**

Specifies a valid abbreviation for a PRINTER (PRT or PTR) or a printer device address for this operand. Refer to the supplied BACKSPAC model.

**PUN**

Specifies a valid abbreviation for a PUNCH (PUN or PCH) or a punch device address for this operand. Refer to the supplied BACKSPAC model.

**RADRONLY**

Requires you to refer to unit record devices by address only. Refer to the supplied BACKSPAC or DRAIN models. To specify unit record devices using a device name or a device address, omit both RADRONLY and DEVNONLY.

**RDR**

Specifies a valid abbreviation for a READER (RDR) or a reader device address for this operand. Refer to the supplied SPOOL model.

**SELECT**

Describes operands that select spool files on the spool queue. It invokes special processing that protects the object of the SPOOL command, including checking the SFBLOKS in CP to obtain all information about a spool file. For example, if you include the rule entry RDR CLASS A - UID(*) PREVENT in your CHANGE rule and a user enters CHANGE RDR FORM STD TO FORM MINE, the SELECT parameter examines all your spool files (FORM STD) and signals an error if any of the spool files have a class of A.

In the supplied models, the CHANGE, QUERY (for spool files), START, SPTAPE, and TRANSFER commands use the SELECT verb because they let you use an alternate operand to manipulate a spool file.

In the previous example of the CHANGE command, the FORM STD is a selection criterion. Therefore, you would code the FORMNAME operand with the SELECT verb. The FORM MINE does not need the SELECT operand because it is not used to choose a spool file. For these operands, your model would resemble:

```
    FORMAT CLASS=G
        OPERAND VUR,7,TRAN=VUR,SPOOLOPT=(PRT,PUN,RDR,DEVNONLY,SELECT
        OPERAND LIST=((GROUP=CLASSFR),                          -
                      (SPOOLID,4,TRAN=SPOOL,SPOOLOPT=SELECT),   -
                      (GROUP=FORMFR),                           -
                      (GROUP=DESTFR),                           -
                      (ALL,3,TRAN=ALLSPFIL,SPOOLOPT=SELECT))
        OPERAND GROUP=TOWHAT
        OPERAND GROUP=NAME
    FORMAT END

FORMFR   GROUP TYPE=KEYWORD
            OPERAND FORM,4
            OPERAND FORMNAME,8,TRAN=FORM,SPOOLOPT=SELECT
         GROUP END

TOWHAT   GROUP TYPE=OPTIONAL,REPEATS
            OPERAND GROUP=CLASS
            OPERAND GROUP=FORM
            .
            .
            .
         GROUP END

FORM     GROUP TYPE=KEYWORD
            OPERAND FORM,4
            OPERAND FORMNAME,8,TRAN=FORM
         GROUP END
```

You only need to code the SELECT verb in commands where a user could perform a set of actions to bypass security. For example, a privileged user could TRANSFER reader files to himself, PEEK them, then TRANSFER them back. The supplied models and recommended methods of protecting the spool file insulate you from this. On the other hand, the ORDER command does not need the SELECT verb because this command cannot gain access to spool files.

**VALUEFOR=label**

This value is the default for the operand identified by the label. For examples, see the format 5 OPERAND clause in Device Address Default (Format 5) in this chapter. For more information about VALUEFOR, see Rules for Defaults from Other Operands (VALUEFOR) in the chapter “Rule Writing Guidelines.”

# GROUP Clause

The GROUP clause simplifies the task of describing operands that are in some way dependent on the specification of other operands. We provide four formats for the GROUP clause so you can easily describe related operand groups.

**Optional Operands (Format 1)**

Describes a group of optional operands you can enter in any order.

**Keyword Operand (Format 2)**

Describes a group of keyword operands you must enter in a predefined order.

**Required Operands (Format 3)**

Describes a group of operands you can enter in any order, but you must enter at least one operand in the group.

**End GROUP (Format 4)**

Signals the end of a GROUP clause.

Examples of using each format of the GROUP clause follow.

## Optional Operands (Format 1)

Use this GROUP clause to describe a group of optional operands you can specify in any order.

```
label GROUP TYPE=OPTIONAL                          {-}
            [{,REPEATS}]
```

In this format, REPEATS indicates that the group of operands can occur more than once in a single command. The supplied ORDER model, shown in the figure below, contains an example of using the REPEATS verb in a GROUP clause.

```
           COMMAND ORDER

              FORMAT CLASS=G
->                OPERAND VUR,7,TRAN=VUR,                    -
->                         SPOOLOPT=(PRT,PUN,RDR,DEVNONLY)
->                OPERAND GROUP=WHAT
              FORMAT END

->    WHAT     GROUP TYPE=OPTIONAL,REPEATS
->                OPERAND GROUP=CLASS
->                OPERAND GROUP=FORM
->                OPERAND GROUP=SPOOL
->             GROUP END

     CLASS    GROUP TYPE=KEYWORD
                 OPERAND CLASS,2
                 OPERAND C,1,TRAN=CLASS
              GROUP END
FORM      GROUP TYPE=KEYWORD
             OPERAND FORM,4
             OPERAND FORMNUM,8,TRAN=FORM
          GROUP END

SPOOL     GROUP TYPE=KEYWORD
             OPERAND SPOOLID,4,TRAN=SPOOL
          GROUP END

      COMMAND END
```

The VUR operand is described in the ORDER model. Here, there is no REPEATS verb. However, we describe the other operands you can enter when you specify the ORDER RDR ... command in a GROUP clause labeled WHAT.

The WHAT group clause includes REPEATS indicating the operands in the group can repeat. In effect, the RDR operand is constant, while the operands after RDR repeat. For example, if you issue the command ORDER RDR CLASS A FORM STD 9999, CA ACF2 for z/VM and CP treat the command as three separate commands, such as:

```
ORDER RDR CLASS A
ORDER RDR FORM STD
ORDER RDR 9999
```

## Keyword Operands (Format 2)

Use this GROUP clause format to describe a group of keyword-type operands that you must specify in a predefined order.

```
label GROUP TYPE=KEYWORD                                {-}
            [{,OCCURS=nnn|REPEATS}]
```

The TYPE=KEYWORD verbs in the model require you to enter the appropriate keyword followed by the keyword value.

For an example of how to use the format 2 GROUP clause, see the PURGE model. A portion of the PURGE model follows:

```
      COMMAND PURGE

         FORMAT CLASS=G
            OPERAND LIST=((VUR,7,TRAN=VUR,            -
                           SPOOLOPT=(RDR,PRT,PUN,     -
                           DEVNONLY,SELECT)),         -
                          (ALL,3,TRAN=ALLURDEV,       -
                           SPOOLOPT=(SELECT,ALL)))
            OPERAND GROUP=WHAT
         FORMAT END

     WHAT   GROUP TYPE=OPTIONAL,REPEATS
               OPERAND GROUP=CLASS
               OPERAND GROUP=FORM
               OPERAND GROUP=SPOOL
               OPERAND GROUP=ALL
            GROUP END

->     CLASS  GROUP TYPE=KEYWORD
               OPERAND CLASS,2
               OPERAND C,1,TRAN=CLASS,SPOOLOPT=SELECT
            GROUP END

->     FORM   GROUP TYPE=KEYWORD
               OPERAND FORM,4
               OPERAND FORMNUM,8,TRAN=FORM,SPOOLOPT=SELECT
            GROUP END

     SPOOL  GROUP TYPE=OPTIONAL
               OPERAND SPOOLID,4,TRAN=SPOOL,SPOOLOPT=SELECT
            GROUP END

     ALL    GROUP TYPE=OPTIONAL
               OPERAND ALL,3,TRAN=ALLSPFIL,           -
                  SPOOLOPT=SELECT,TYPE=DEFAULT
            GROUP END

      COMMAND END
```

Examine the GROUP clauses for the CLASS and FORM groups in the above example. According to this model, a class G user can PURGE files that are assigned to a particular CLASS, FORM, or a combination of CLASS and FORM.

For example:

**PURGE RDR CLASS A**

Purges all RDR files that have CLASS=A

**PURGE RDR FORM STD1**

Purges all RDR files that have FORM=STD1

**PURGE RDR FORM STD1 CLASS A**

Purges all RDR files that have CLASS=A and FORM=STD1.

## Differences Between REPEATS and OCCURS

The GROUP TYPE=KEYWORD clause accepts the OCCURS verb that indicates you can specify the entire group of operands a definite number of times in a single command. Refer to the supplied SPOOL model for an example of how to use OCCURS.

The difference between REPEATS and OCCURS is that REPEATS means operands in this group can replicate themselves an infinite number of times, limited only by the actual size of the command buffer. OCCURS means operands in the group replicate themselves a fixed number of times.

Command limiting treats iterations of a command with a REPEATS definition as if they were separate commands. For example, assume you have four files in your virtual printer queue with spool IDs of 0091, 0092, 0093, and 0094. If you issue the PUR PRT 91 92 93 94 command, the spool ID is defined as a repeating operand. CP and command limiting would process the command as:

```
PUR PRT 91
PUR PRT 92
PUR PRT 93
PUR PRT 94
```

Command limiting does not treat iterations of OCCURS as multiple commands. For example, you can specify the CHARS operand on the SPOOL command up to four times:

```
SP PRT TO PSF CL P FO 010110 CHARS MONO CHARS TEXT CHARS CE12
CHARS PT24 646X
```

In the PURGE command above, each iteration is treated as a separate command and each iteration is compared against the entire rule set. The SPOOL command is only compared against the rule set once, but all combinations of each iteration of the CHARS operand are used against the qualifying rule entry.

## Required Operands (Format 3)

The format 3 GROUP clause describes a group of operands you can specify in any order, but you must code at least one operand in the group.

```
label GROUP TYPE=REQUIRED                                {-}
            [{,REPEATS}]
```

Following is an excerpt from the supplied SPOOL model that shows how to use the format 3 GROUP clause TYPE=REQUIRED.

```
      COMMAND SPOOL

         FORMAT CLASS=G
            OPERAND LIST=((RDR,6,TRAN=VUR,           -
                          SPOOLOPT=(RDR,DEVNONLY)), -
                        (RDR,4,TRAN=VUR,             -
                          SPOOLOPT=(RDR,RADRONLY)))
            OPERAND GROUP=RDR
         FORMAT END

->    RDR    GROUP TYPE=REQUIRED
               OPERAND GROUP=CLASS
               OPERAND LIST=((CONT,3),               -
                             (NOCONT,3))
               OPERAND LIST=((EOF,3),                -
                             (NOEOF,3))
               OPERAND LIST=((HOLD,2),               -
                             (NOHOLD,3))
            GROUP END
```

The RDR group includes TYPE=REQUIRED. You must enter one of the operands described in the RDR group whenever you issue the SPOOL RDR CLASS * command.

## End GROUP (Format 4)

The format 4 GROUP clause signals the end of an operand group.

```
GROUP END
```

# Verb Descriptions (GROUP Clause)

The verbs you can use in a GROUP clause are:

**label**

Specifies a one- to eight-character name of the GROUP clause.

**TYPE=OPTIONAL**

None. You can enter one or more of the operands in this group.

**TYPE=KEYWORD**

Identifies the beginning of a group of operands that you must enter in a predefined order. The operands are required. For an example, see the format 2 GROUP clause in Keyword Operands (Format 2) in this chapter.

**TYPE=REQUIRED**

Identifies the beginning of a group of operands where you must enter at least one operand in any order. For an example, see the format 3 GROUP clause in Required Operands (Format 3) in this chapter.

**REPEATS**

Indicates this group of operands can occur more than once in a single command. For an example, see the format 4 GROUP clause in End GROUP (Format 4) in this chapter.

**OCCURS=nnn**

Replicates all the control blocks in the group nnn times.

As explained before, the difference between REPEATS and OCCURS is that REPEATS means operands in this group can replicate themselves an infinite number of times, limited only by the actual size of the command buffer. OCCURS means operands in the group replicate themselves a fixed number of times.

**END**

Signals the end of a group of operands.

## Differences Between REPEATS and OCCURS

As explained before, the difference between REPEATS and OCCURS is that REPEATS means operands in this group can replicate themselves an infinite number of times, limited only by the actual size of the command buffer. OCCURS means operands in the group replicate themselves a fixed number of times.

**END**

Signals the end of a group of operands.

# COMMENT Clause

The COMMENT clause places comments directly in the command model. You might include comments if you change a supplied command model.

CA ACF2 for z/ VM treats any line starting with an asterisk (*) in column one as a comment. All of the supplied model files include comments to indicate when we added or changed clauses.

## NULL Clause

The NULL clause places blank lines directly in the command model. You might include blank lines to make reading a command model easier.

# Chapter 10: Using the Model Setting

If you have added or changed CP commands and you want to control the command execution or log their use, you must create a syntax model or change the distributed model.

This chapter explains the ACF subcommands you need to create or change syntax models. You should also review the chapter “Rule Writing Guidelines” to obtain a full understanding of how rules are interpreted. Use this chapter as a reference aid when you want to create, modify, display, test, or list models.

The following commands are explained in this chapter:

**COMPILE**

Converts rule sets into the form needed by CA ACF2 for z/ VM.

**DECOMP**

Lists previously stored rule sets.

**DELETE**

Deletes command limiting rule sets.

**LIST**

Lists previously stored rule sets.

**STORE**

Stores compiled rule sets on the Infostorage database.

**TEST**

Tests the correctness of a rule set.

This section contains the following topics:

# Creating a Model

To create a syntax model, follow these six steps:

1. Determine the syntax of the command.
2. Create a syntax model.
3. Compile the model.
4. Create a test rule.
5. Test the model.
6. Activate command limiting.

## Determine the Syntax of the Command

Before you create a syntax model, examine the syntax of the command and decide exactly what you want command limiting to do for you. If you want to log the execution of a command or control who can use it, you can write a simple model. If you want to control who can who can use certain operands or the value of an operand, you must write a complete model.

Listed below is the VSNAP command for the V/SNAP product from VM Systems Group. We use it here as a sample of adding a CP command. For more information about the CA ACF2 for z/ VM interface for V/SNAP, see the *Other Products Guide*.

The syntax of the VSNAP command is:

```
VSNAP      USER userid1 [ hexloc1 ][{ -  }  [ hexloc2  ]][ TO*           ]
                        [ 0       ][{ :  }  [ END      ]][ TO userid2    ]
                                    [                    ][ SYSTEM        ]
                                    [ {.{    [ bytecount]][                ]
                                    [        [ END      ]][ FORMAT vmtype ]
                                                          [ DSS           ]
                                                          [ *dumpid       ]
                                                          [ CP            ]
                                                          [ ALL           ]
                                                          [ V=R           ]
```

0-END TO * are the defaults for the first format. If you do not specify any operands, CP is the default.

## Create a Test Syntax Model

The next step in creating or changing a syntax model is to create a CMS file that contains the model control statements. The filename can be any legal CMS filename, but the file type must be MODEL.

If you only want to control who can execute the command or log the use of the command, create a simple model. A simple model for the VSNAP command would contain the following statements:

```
COMMAND VSNAP

   FORMAT CLASS=ACE
       OPERAND ANYOPERANDS,240,TRAN=REST
   FORMAT END

COMMAND END
```

A complete model for the VSNAP command contains the following statements:

```
COMMAND VSNAP

   FORMAT CLASS=ACE
       OPERAND USER,4
       OPERAND USERID,TRAN=USER
       OPERAND LIST=((HEXLOC,13,TRAN=STVDMP),  -
                     ('0-END',5,TRAN=STVDMP,TYPE=DEFAULT))
       OPERAND GROUP=OPTIONS
       OPERAND DUMPID,100,TRAN=REST,OPTIONAL
   FORMAT END

   FORMAT CLASS=ACE
       OPERAND LIST=((ALL,3),  -
                     ('V=R',3),    -
                     (CP,2,TYPE=DEFAULT))
   FORMAT END

OPTIONS  GROUP TYPE=OPTIONAL
           OPERAND LIST=((SYSTEM,5),             -
                         (GROUP=TO))
           OPERAND GROUP=FORMAT
           OPERAND DSS,3
         GROUP END

GO       GROUP TYPE=KEYWORD
           OPERAND TO,2,TYPE=NXTOPDEF
           OPERAND LIST=((*,1,TRAN=SELF,TYPE=DEFAULT), -
                         (USERID,8,TRAN=USER))
         GROUP END
FORMAT   GROUP TYPE=KEYWORD
            OPERAND FORMAT,6
            OPERAND VMTYPE,8,TRAN=ANY
         GROUP END

 COMMAND END
```

## Guidelines for Writing Models

When writing models, follow these guidelines:

- The file type must be MODEL.
- Where possible, arrange the formats in the model by privilege class. Classes A through Z, class ALL, then class ANY. If a format applies to multiple classes, use the lowest value class in the format (A is the lowest class, B is greater than A, and so on).

- For performance purposes, list the most specific formats first in a class group. An operand that has a single constant value is more specific than one that is a variable. Variables have transposition routines. A single constant value is more specific than constants in mutually exclusive list.

  Weigh the previous two guidelines against the complexity of any command. For example, the IBM SET and QUERY commands are so complex, it is much easier to code the model as documented.

- If you can issue the command with no operands and it has a default operand, that format must be the last one.

- Only use transposition routines when you want to control differing values of an operand position or to differentiate between two similar formats.

- Some of the transposition routines are very specific, and may only apply to a specific CPU. For example, to qualify an operand in the USER transposition routine, the user ID must exist in the VM directory. Therefore, you must thoroughly understand the qualifications of a transposition routine before using it.

- Where possible, use the generic ANY or REST transposition routines. The limitation of the ANY transposition routine is that you cannot effectively control different variable values through rules. You can only use the REST transposition routine on the last operand in a format.

  Where practical, all of the distributed models let you write rules to give you the most control over a command. We coded them using specific transposition routines. Where appropriate, you can change models to meet your needs and improve the performance of command limiting.

- If the operand value contains characters other than alphanumeric or an asterisk (*), enclose the value in single quote marks.

- Once a command matches the CLASS criteria and matches the first operand in a format, the command continues to be validated against the format until CA ACF2 for z/ VM finds a NOMATCH=NEXTFORMAT condition.

- All OPERAND clauses coded in the FORMAT area are required unless you code them with the OPTIONAL operand or with a default value.

- Some CP commands accept more operands than are documented and, in many cases, the command processors ignore them. Set an appropriate SYNERR option to deal with them or code an operand clause to soak up the extra operands. For example, OPERAND JUNK,140,TRAN=REST.

- CA ACF2 for z/ VM command limiting is not aware of all of the finer points of a command; it can only deal with commands in a general fashion. When you execute a command with a bad syntax, command limiting can only tell which operand was bad, but not why. In these cases, set a systemwide SYNERR option, put it on the rule, or set it for the individual having problems.

## Analysis of the VSNAP Model

Users with privilege classes of A, C, or E can execute both formats of the VSNAP command, so guideline number two does not apply.

The first format, USER userid is more specific than the second format, so it is put first in the model.

You can execute the VSNAP command without any operands. The default operand is CP, so that format must be last.

If you do not want to control what user IDs can be forced or to what user IDs the dump is spooled, change the transposition routines on the USERID operands to ANY. Notice that the VMTYPE operand has a transposition routine of ANY and that the DUMPID operand is coded with the REST transposition routine. To control which DUMPIDs can be used, you must code all of the possibilities for the DUMPID in a mutually exclusive list.

The 0-END and V=R operands contain nonalphanumeric characters and are enclosed in single quotes.

Notice where we coded the OPTIONAL and TYPE=DEFAULT operand values. To control the VSNAP command based on the first two operands of the command, you can simplify the model and end up with the following command model:

```
COMMAND VSNAP

   FORMAT CLASS=ACE
       OPERAND USER,4
       OPERAND USERID,TRAN=USER
       OPERAND OTHEROPS,180,TRAN=REST,OPTIONAL
   FORMAT END

   FORMAT CLASS=ACE
       OPERAND LIST=((ALL,3),  -
                     ('V=R',3),    -
                     (CP,2,TYPE=DEFAULT))
   FORMAT END

COMMAND END
```

## Compile the Model

You can compile your syntax model from the MODEL mode of the ACF command. If a command model already exists with the same command name, or if you just want to test your syntax model, we recommend that you do not compile the model with the current MDLTYPE. For testing purposes and to isolate your work from the live models and rules, you might use a MDLTYPE of TST.

```
acf
ACF
set model
MODEL
compile vsnap mdltype(tst)
```

## Create a Test Rule

To determine if your syntax model performs as it should, you must test it. To test a model, write a dummy rule. Using the above example and recommendations, the rule should contain the following statements in a CMS file called VSNAP RULE.

```
$KEY(VSNAP) MDLTYPE(TST)
 - UID(*) ALLOW
```

From the CMDLIM mode, enter the following to compile and store the rule:

```
acf
ACF
set cmdlim
CMDLIM
compile vsnap nolist
ACFxxx510I ACF compiler entered

ACFxxx551I Total record length=165 bytes - 4 percent utilized
ACFxxx769I Rule VSNAP, MDLTYPE TST, stored


CMDLIM
```

## Test the Model

To be sure your model behaves as intended in a production environment, thoroughly test the syntax model. You should test both valid and invalid operand combinations. Before you begin testing, you should probably have a hardcopy of both the syntax of the command and your syntax model. If your command and model are complex, you may also want to write an exec to drive the test. This exec can also come in handy when you need to write rules and want to test the correctness of them.

You can use the following rudimentary exec to drive the test.

```
/* THIS EXEC TESTS VARIOUS COMBINATIONS OF THE VSNAP COMMAND */
'ID (STACK'
pull userid .;
say ' '
say 'VALID VSNAP COMBINATIONS'
say ' '

queue 'SET CMDLIM'
queue 'TEST VSNAP MDLTYPE(TST)'
queue 'CLASS(A)'

queue 'O(USER MAINT)'
queue 'O(USER MAINT 0-END)'
queue 'O(USER MAINT 0-END SYSTEM)'
queue 'O(USER MAINT 0-END TO *)'
queue 'O(USER MAINT 0-END TO ' userid ')'
queue 'O(USER MAINT 0-END TO ' userid 'FORMAT xxxx)'
queue 'O(USER MAINT 0-END TO ' userid 'FORMAT xxxx DUMPID xxxx)'
queue 'O(USER MAINT 0-END TO ' userid 'FORMAT xxxx DSS DUMPID xxxx)
queue 'O(ALL)'
queue 'O(V=R)'
queue 'O(CP)'
queue 'END'
queue 'END'
ACF
say ' '
say 'SOME INVALID VSNAP OPERAND COMBINATIONS'
say ' '

queue 'SET CMDLIM'
queue 'TEST VSNAP MDLTYPE(TST)'
queue 'CLASS(A)'

queue 'O(USERMAINT)'
queue 'O(USER XMAINT)'
queue 'O(USER GEORGEIOUS)'
queue 'O(ALL WHET)'
queue 'O(ALT)'
queue 'O(V-R)'
queue 'O(CQ)'
queue 'END'
queue 'END'
ACF

exit
```

You can issue the exec from CMS and observe the following:

```
Ready; T=0.01/0.01 18:33:11
vsnap
valid vsnap combinations
ACF
CMDLIM
cmd=vsnap, mdltype=tst
 .
THE FOLLOWING PARAMETERS ARE IN EFFECT:
 DATE=01/04/00, TIME=*****, UID=******, SOURCE=********
 CLASS=A
 OPERANDS=

THE FOLLOWING WOULD APPLY: ALLOW (RELATIVE RULE ENTRY 1)
 .
THE FOLLOWING PARAMETERS ARE IN EFFECT:
 DATE=01/04/00, TIME=*****, UID=******, SOURCE=********
 CLASS=A
 OPERANDS=USER MAINT
 .
 .
 .
CMDLIM
some invalid vsnap operand combinations
ACF
CMDLIM
cmd=vsnap, mdltype=tst
 .
THE FOLLOWING PARAMETERS ARE IN EFFECT:
 DATE=01/04/00, TIME=*****, UID=******, SOURCE=********
 CLASS=A
 OPERANDS=

THE FOLLOWING WOULD APPLY: ALLOW (RELATIVE RULE ENTRY 1)
 .
THE FOLLOWING PARAMETERS ARE IN EFFECT:
 DATE=01/04/00, TIME=*****, UID=******, SOURCE=********
 CLASS=A
 OPERANDS=USERMAINT

COMMAND SYNTAX ERROR, LID SYNERR=LOG (OPERAND NUMBER 1 IS IN ERROR)
 .
 .
 .
CMDLIM
Ready; T=0.16/0.69 18:33:41
```

### Model Problem Support

If your results do not work as expected, fix the model, recompile it, and execute the test again. If you cannot determine what the problem is, contact your local CA ACF2 for z/ VM support representative for assistance. Before contacting CA ACF2 for z/ VM support, have the following ready:

- The syntax of the command and any related documentation
- A hardcopy of the model
- The exec that drove the test (if you used one)
- Console log output from the model and rule compile
- The output from the test command.

If the support personnel cannot solve your problem on the phone, you may need to send the above information to them.

## Activate Command Limiting

Your command model is almost ready to use. But before going into production, you should perform the following steps:

- Modify the command limiting rule to perform as needed.
- Use the test exec to ensure that the rule is performing as expected.
- Compile the model under the current MDLTYPE. To determine the MDLTYPE, display the CMDLIM VMO record or issue the ACF SHOW CMDLIM command.
- Compile the rule under the current MDLTYPE.
- Change the CMDLIM VMO record to limit your command.
- Delete the test model and rule you created earlier.

# Modifying a Model

To add operands or new formats to an existing CP command, follow these steps:

- Determine the syntax of the command.
- Decompile the existing syntax model into a CMS file or extract the model of the command from the distributed model file
- Modify the syntax model.
- Compile the model under a test MDLTYPE.
- Create a test rule.

- Test the model.
- Activate command limiting.

Continuing with the previous VSNAP example, V/SNAP adds a format to the QUERY command. Add this new format to the QUERY command model:

```
FORMAT CLASS=ALL
    OPERAND VSNAP,5
FORMAT END
```

After establishing the MODEL setting of the ACF command, you can compile command models.

```
acf
ACF
set model
MODEL
```

Under the MODEL setting, you can use any of the following ACF subcommands:

- COMPILE
- DECOMPILE
- DELETE
- END
- HELP
- SET
- SHOW

The common subcommands END, HELP, SET, and SHOW are explained in the *Administrator Guide*. The other commands, specific to the MODEL setting, are explained in this chapter.

# COMPILE Subcommand

Under the MODEL setting, the syntax of the COMPILE subcommand is:

```
                      [ MDLTYPE(mdltype) ]
COMPILE filename      [ LIST|NOLIST      ]
                      [ STORE|NOSTORE    ]
                      [ FORCE|NOFORCE    ]
```

Under the MODEL setting, COMPILE takes the following parameters:

**filename**

Specifies the CMS file that contains the Syntax Model Command Language (SMCL) statements. You must create this file before you use the COMPILE subcommand. The file type must be MODEL.

**MDLTYPE(mdltype)**

Specifies the type of command model to compile. By default, CA ACF2 for z/ VM uses the value specified in the CMDLIM VMO record. Normally, you do not code this parameter unless you want to change the model type from the default.

**LIST|NOLIST**

Displays the input to the compiler on your screen. NOLIST prevents the display. LIST is the default. If you compile with the LIST option, you see several comments displayed on your screen after all the models are compiled. This is normal. They are displayed because this is a maintenance log.

**STORE|NOSTORE**

Stores the command model automatically at compilation time. NOSTORE does not automatically store the command model. You can use NOSTORE to test for errors in the syntax model statements. STORE is the default.

**FORCE|NOFORCE**

Stores the command model, regardless of whether it currently exists. NOFORCE stores the command model only if it did not already exist. FORCE is the default.

There is a SET FORCE|NOFORCE subcommand that defaults to SET FORCE. Use the FORCE|NOFORCE parameter of COMPILE to override the SET value, or use the SET subcommand to change the defaults for the COMPILE subcommand. For example, you can issue a SET NOFORCE to change the default for COMPILE to NOFORCE.

## Modifying a Command Model

Before you can modify a command model, you must decompile it. We recommend you decompile the model into a CMS file for easier modification and testing. Decompile the model under the MODEL setting of ACF as shown below:

```
acf
ACF
set model
MODEL
```

# The DECOMPILE Subcommand

Under the MODEL setting, the syntax of the DECOMPILE subcommand is:

```
              {  *           } [  INTO(filename)               ]
DECOMP        { model        } [  MDLTYPE()|(mdltype|mdlmask)  ]
              { LIKE(mdlmask)}
```

*** (asterisk)**

Decompiles the model you previously worked on in this session.

**model**

Decompiles the specific model.

**LIKE(mdlmask)**

Decompiles all models that match the specified mask.

**INTO(filename)**

The CMS filename where the model is decompiled into. The file type is always MODEL.

**MDLTYPE()|(mdltype|mdlmask)**

Decompiles this type of command model. Valid options are () (null), mdltype (the three-character model type), or mdlmask (a mask for the model type). By default, CA ACF2 for z/ VM uses the value specified in the CMDLIM VMO record.

# The DELETE Subcommand

Under the MODEL setting, the syntax of the DELETE subcommand is:

```
              {  *           }
DELETE        { model        }   [MDLTYPE(mdltype)]
              { LIKE(mdlmask)}
```

*****

Deletes the model previously worked on in this session.

**model**

Deletes this specific model.

**LIKE(mdlmask)**

Deletes all models that match the specified mask.

**MDLTYPE(mdltype)**

Deletes this type of command model. By default, CA ACF2 for z/ VM uses the value specified in the CMDLIM VMO record.

# Chapter 11: Transposition Routines for Command Limiting

The chapter “Rule Writing Guidelines” explained the basic concept of transposition routines. This appendix explains these routines in more detail. As a command limiting rule writer, you should be familiar with these routines.

Transposition routines transpose rule entries and operand values to common values to ensure proper matching between a rule and a command. For instance, the transposition routine VCUU checks to see that the entered value is a valid hexadecimal number. If it is, the routine converts the hexadecimal value to a binary fullword.

When properly understood, transposition routines make rule writing easier. This is because most of the routines change variable operand values into constant values that you can specify in a rule entry. For example, the routine transposes the ATTACH 381 TO * AS 299 command to ATTACH 381 TO OWNER AS 299. The routine replaces the asterisk with the constant OWNER.

This section contains the following topics:



## Transposition Routines

**ALLSPFIL**

ALLSPFIL accepts all spool files. This routine validates that the data specified in the command is valid decimal data. When used with SPOOLOPT=SELECT, the operand selects files from the spool queue and is interpreted as the highest numbered spool file selected.

**ALLSYS**

ALLSYS accepts the keyword ALL on a CSE command and the pseudo operand ALL on a rule. The pseudo operand ALL in a rule matches each system name, not just the keyword ALL on a command line. On a rule with the ALL pseudoname, each system name in the CSESYS table is matched to the system name on the command. If any one matches, this rule matches the command.

If a rule has a specific system name and the command has the keyword ALL in it, conditional processing is done to match this rule to the command. If the rule specifies ALLOW, the rule only matches the keyword ALL when it matches every name in the CSE name list. This means that if you allow commands for one system, you do not necessarily allow the command for all systems.

If, however, the rule is a PREVENT rule, the rule matches the keyword ALL if it matches any name in the CSE name list. In other words, if a rule prevents any single system and allows the rest, the routine prevents the ALL keyword on the command.

ALLSYS works like an ANY transposition routine on non-XA systems. Test these transposition routines on XA systems.

**ALLURDEV**

ALLURDEV accepts spool queues and converts the words PRINTER, PUNCH, READER, and CONSOLE to PRT, PUN, RDR, and CON, respectively. When you use this routine with SPOOLOPT=SELECT, it selects files from the spool queue. The routine also checks to ensure that the specified device is a real unit record device.

**ANY**

ANY accepts any operand as long as it is not greater than the maximum allowed length of the operand. The value of the operand is accepted as entered in the command syntax. For example, the password operand of the AUTOLOG command uses the ANY routine.

**CLASS**

CLASS validates that the operand is a valid class character. Possible classes are A-Z, 0-9, or * that means all classes. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. For example, the CLASS operand of the CHANGE command uses the CLASS routine.

**COPY**

COPY accepts any decimal operand as long as it is not greater than the maximum allowed length of the operand. The value of the operand is accepted as entered in the command syntax. When used with SPOOLOPT=SELECT, it selects files from the spool queue.

Below is an example using no masking. The number portion must be a valid decimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 0000000000 |
| 1 | 0000000001 |
| 0000 | 0000000000 |
| 9 | 0000000009 |
| 0009 | 0000000009 |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to nines.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 0000000000-0000000009 |
| 1* | 0000000010-0000000019 |
| 5* | 0000000050-0000000059 |
| *1* | 0000000010-0000000919 |
| **1 | 0000000001-0000000991 |
| 6** | 0000000600-0000000699 |

Following is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value that the transposition routine can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 0000000004-2147483647 |
| 400- | 0000000400-2147483647 |
| 12345- | 0000012345-2147483647 |
| 1-100 | 0000000001-0000000100 |
| 256-512 | 0000000256-0000000512 |
| 40000-80000 | 0000040000-0000080000 |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 0000000005-2147483647 |
| 5*- | 0000000050-2147483647 |
| 5**- | 0000000500-2147483647 |
| 5****- | 0000050000-2147483647 |
| 1**-2** | 0000000100-0000000299 |
| *6*-**4** | 0000000060-0000099499 |

**CPSYSTEM**

CPSYSTEM accepts the SYSTEM keyword that means the system SPOOL Q. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue.

**DECIMAL**

For a command request, DECIMAL checks to be sure the data specified in the command is really decimal data. For a rule validation request, DECIMAL checks for masking. If there is no masking in the rule, the routine treats the rule like a command; otherwise, the routine builds an upper and lower range and checks both for decimal data. Masking in this case is treated as numeric, with each asterisk (*) occupying a decimal position.

You can specify TYPE=RANGE to denote a decimal range.

Below is an example using no masking. The number portion must be a valid decimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 0000000000 |
| 1 | 0000000001 |
| 0000 | 0000000000 |
| 9 | 0000000009 |
| 0009 | 0000000009 |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to nines.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 0000000000-0000000009 |
| 1* | 0000000010-0000000019 |
| 5* | 0000000050-0000000059 |
| *1* | 0000000010-0000000919 |
| **1 | 0000000001-0000000991 |
| 6** | 0000000600-0000000699 |

Following is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value the transposition routine can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 0000000004-2147483647 |
| 400- | 0000000400-2147483647 |
| 12345- | 0000012345-2147483647 |
| 1-100 | 0000000001-0000000100 |
| 256-512 | 0000000256-0000000512 |
| 40000-80000 | 0000040000-0000080000 |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 0000000005-2147483647 |
| 5*- | 0000000050-2147483647 |
| 5**- | 0000000500-2147483647 |
| 5****- | 0000050000-2147483647 |
| 1**-2** | 0000000100-0000000299 |
| *6*-**4** | 0000000060-0000099499 |

In the sample rule below, user OPR1 can issue the ECHO command 001 to 991 times. The second rule entry lets user OPR2 issue the ECHO command any number of times (NN is the pseudo operand). The third entry lets user OPR3 issue the ECHO command once.

```
$KEY(ECHO)
 **1 UID(OPR1) ALLOW
 NN UID(OPR2) ALLOW
 1 UID(OPR3) ALLOW
```

**DEST**

DEST accepts any operand as long as it is not greater than the maximum allowed length of the operand. The routine accepts the value of the operand as entered in the command syntax. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. For example, the value for the DEST operand in the CHANGE command uses the DEST routine.

**DSNAME**

DSNAME validates that:

- The data set name is not more than eight characters between the periods (.), which are delimiters.
- The data set name is not less than one character.
- There are no embedded blanks in the string.
- The total length of the string is not greater than 24 characters.

For example, the dsname option for the NAME operand of the CHANGE command uses the DSNAME routine.

**FORM**

FORM accepts any operand as long as it is not greater than the maximum allowed length of the operand. The routine accepts the value of the operand as entered in the command syntax. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. For example, the value for the FORM operand in the CHANGE command uses the FORM routine.

**HEX**

HEX is the same as DECIMAL except that it checks for valid hexadecimal arithmetic values. The input is right-justified and hex zero filled and then converted to a binary number. Refer to the HEXDATA transposition routine below for operands with hexadecimal character strings. You can specify TYPE=RANGE with HEX to denote a hexadecimal range.

For a command request, HEX checks to be sure the data specified in the command is really hexadecimal data. For a rule validation request, HEX checks for masking. If there is no masking in the rule, the routine treats the rule like a command; otherwise, the routine builds an upper and lower range and checks both for hexadecimal data. The routine treats masking in this case as numeric, with each asterisk (*) occupying a decimal position.

Below is an example using no masking. The number portion must be a valid hexadecimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 00000000 |
| 1 | 00000001 |
| 0000 | 00000000 |
| F | 0000000F |
| 000F | 0000000F |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to Fs.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 00000000-0000000F |
| 1* | 00000010-0000001F |
| 5* | 00000050-0000005F |
| *1* | 00000010-00000F1F |
| **1 | 00000001-00000FF1 |
| 6** | 00000600-000006FF |

Below is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value that the transposition routine can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 00000004-FFFFFFFF |
| 400- | 00000400-FFFFFFFF |
| 14A000- | 0014A000-FFFFFFFF |
| 1F0-A00 | 000001F0-00000A00 |
| 191-19F | 00000191-0000019F |
| E00000-FFFFFF | 00E00000-00FFFFFF |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 0000000005-FFFFFFFF |
| 5*- | 0000000050-FFFFFFFF |
| 5**- | 0000000500-FFFFFFFF |
| 5****- | 0000050000-FFFFFFFF |
| 1**-2** | 0000000100-000002FF |
| *6*-**4** | 0000000060-000FF4FF |

In the rule below, the first rule entry keeps anyone from setting an ADSTOP between address 0 - FFFF. (Each * is a place holder.) The second rule entry keeps anyone from setting an ADSTOP between 1000 and 1FFF. The first and second rule entries effectively prevent a user from setting an ADSTOP below address 2000.

```
$KEY(ADSTOP)
 **** UID(*) PREVENT
 1000-1FFF UID(*) PREVENT
 - UID(*) ALLOW
```

For example, the hexloc operand of the ADSTOP command uses HEX routine.

**HEXDATA**

Processes hexadecimal input as character type data.

Refer to the HEX transposition routine above for operands with hexadecimal arithmetic values.

The input string is validated for hexadecimal characters, then left-justified and right-filled with hex zeros, if required. This string can be masked in a rule. Masked strings are not allowed in commands. The string consists of an even number of characters, each representing half a hexadecimal character. A trailing mask of a dash fills the input string to an even number of characters, if required.

Below are examples of valid and invalid HEXDATA strings.

| Rule Operand | Interpreted As |
|---|---|
| C1C2C3 | Valid, no masking |
| C1C*C3 | Valid, with masking |
| C1C2- | Valid, with trailing "-" mask |
| C1C2C- | Valid, with trailing "-" mask |
| C1C2C | Invalid, odd number of characters |
| C1G2C3 | Invalid, "G" is not valid |

**HHMM**

During command syntax checking, HHMM converts hours and minutes to seconds. It locates the delimiter (:) between the HH and MM, right justifies both hours and minutes, then changes the value to seconds.

The routine checks for a mask that represents the valid time value during rules validation. If you do not use masking, the routine treats the rule like a command. If you do use masking, HHMM tries to find a delimiter in the mask (:). After finding the delimiter, the routine builds an upper and lower range of seconds for both the hours and minutes. If there is no delimiter, it converts the string to a seconds value.

**HOSTSTRG**

HOSTSTRG applies to XA operating systems only. It validates whether a STORE command has a host or guest operand specified. The default is a guest machine. If you did not specify a machine, the routine assumes guest.

```
STORE S20010 0000 HU00 THIS IS A TEST
```

In the above example, the routine returns HOST to the command interpreter (denoted by H in the HU00 operand).

**LDEV**

LDEV validates that the first character of a logical device is always L and the rest of the operand is a readable hexadecimal number. You can specify TYPE=RANGE with LDEV to denote a range of logical devices.

For a command syntax check, the routine checks only the first character to ensure it is an L. If not, then a syntax error occurs. If it is L, the HEX routine translates the hexadecimal number. For rules validation, you can specify the first character of a logical device address as a mask (*). For example, the LINES operand of the QUERY command uses the LDEV routine.

**LDEVXA**

.LDEVXA ensures that operands begin with L. It allows as many positions in the operand as specified in the syntax model, except that it limits the maximum number of numeric positions in a segment to eight digits. You can use masking, however, you cannot mask in ranges. The routine converts all values to binary for comparison purposes.

Below is an example using no masking. You must prefix the operand with L. The number portion must be a valid hex digit.

| Rule Operand | Interpreted As |
|---|---|
| L0 | 0000000 |
| L1 | 0000001 |
| L0000 | 0000000 |
| LA | 000000A |
| L000A | 000000A |

This is an example of using an asterisk (*) for masking:

| Rule Operand | Interpreted As |
|---|---|
| L*1 | 0000001-00000F1 |
| L1* | 0000010-000001F |

| Rule Operand | Interpreted As |
|---|---|
| L**1 | 0000001-0000FF1 |
| L*1* | 0000010-0000F1F |
| L1** | 0000100-00001FF |
| L1*** | 0001000-0001FFF |
| L***5 | 0000005-000FFF5 |
| L**5* | 0000050-000FF5F |
| L**55 | 0000055-00FF55 |
| L*5** | 0000500-00F5FF |

Following is an example of using a dash (-) for masking. If you use a dash, it must be the last byte.

| Rule Operand | Interpreted As |
|---|---|
| L- | 0000000-FFFFFFF |
| L5- | 0000005-FFFFFFF |
| L50- | 0000050-FFFFFFF |

Below is an example of using a combination of asterisks and dashes for masking. If you use a dash, it must be the last byte.

| Rule Operand | Interpreted As |
|---|---|
| L*5- | 0000005-FFFFFFF |
| L5*- | 0000050-FFFFFFF |
| L5**- | 0000500-FFFFFFF |
| L5****- | 0050000-FFFFFFF |

**LDEV Ranges:**

The LDEV range technique provides more flexibility. You can use a dash as a range character, not a masking character. There are two parts to a range: the low segment and the high segment.

When using LDEV ranges, remember:

- You cannot mask in ranges.
- Each segment must start with L.
- The rest of the segment must contain valid hexadecimal numbers.
- The high segment must be greater in value than the low segment.

An example of LDEV ranges follows.

| Rule Operand | Interpreted As |
|---|---|
| L0-LA | 0000000-000000A |
| L0000-L0000A | 0000000-000000A |
| L10-L1F | 0000010-000001F |
| L10-L12 | 0000010-0000012 |
| L12-L10 | (illogical range) |
| L100-L1FF | 0000100-00001FF |
| L100-L1FG | (illegal character) |
| L*00-L1FF | (masking not allowed) |
| L100-1FF | (L missing from second segment) |
| L00-LFF | 0000000-00000FF |
| L000-LFFF | 0000000-0000FFF |
| L0000-LFFFF | 0000000-000FFFF |
| L1000-LFFFF | 0001000-000FFFF |

**LOCALSYS**

LOCALSYS converts an asterisk in a CSE-related command to the local system name defined in CSESYS in HCPSYS (the name of the system the user is on). The routine does not interpret an asterisk in the rule as the local system because it considers an asterisk to be a single character mask. The routine translates the asterisk in a command to the local system name before processing it against a command limiting rule. This lets you write a rule for a specific system.

LOCALSYS works like an ANY transposition routine on non-XA systems. Test these transposition routines on XA systems.

**LPRT**

LPRT accepts any operand as long as it is not greater than the maximum allowed length of the operand. The value of the operand is accepted as entered in the command syntax. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue.

For example, the value for the LPRT operand in the CHANGE command uses the LPRT routine.

**MINSIZE**

During command syntax checking, MINSIZE validates that the operand conforms to the syntax of MINSIZE=nnnK or MINSIZE=nnnM, where nnn is a valid storage size. This transposition routine is identical to STRSIZE except that you must precede the storage size with MINSIZE=. For more rule masking examples, see STRSIZE in this appendix. All values are normalized to K-bytes before comparison.

| Rule Operand | Interpreted As |
|---|---|
| MINSIZE=512k | 00000512 |
| MINSIZE=512m | 05242888 (k) |
| MINSIZE=1**k | 00000100-00000199 |

**MMSS**

During command syntax checking, MMSS converts minutes and seconds to seconds. It locates the delimiter (:) between the MM and SS, right justifies both minutes and seconds, then changes the value to seconds.

The routine checks for a mask that represents the valid time value during rules validation. If you do not use masking, the routine treats the rule like a command. If you do use masking, MMSS tries to find a delimiter in the mask (:). After finding the delimiter, the routine builds an upper and lower range of seconds for both the minutes and seconds. If there is no delimiter, the routine converts the string to a seconds value.

| Rule Operand | Interpreted As |
|---|---|
| >00:** | 000 to 5400 sec |
| 01:00 | 360 sec |
| 0- | 000 to 5400 sec |
| 0*10 | 600 se |

If you enter unmasked times in rules, the routine will not find a match.

**PARMREGS**

During command syntax checking, PARMREGS validates that the operand conforms to the syntax of PARMREGS=n-n or PARMREGS=n, where n is a valid decimal value with a maximum a two digits.

| Rule Operand | Interpreted As |
|---|---|
| PARMREGS=0 | 00 |
| PARMREGS=6 | 06 |
| PARMREGS=* | 00-09 |
| PARMREGS=4-8 | 04-08 |
| PARMREGS=0-15 | 00-15 |

**PERCENT**

PERCENT validates and checks the syntax to ensure that the last character of the operand is a percent sign. The routine transposes the percent sign to a null and normal processing continues (for example, SET SRM IABIAS 10%).

When writing the rule, do not include the % character in the rule entry.

Below is an example using no masking. The number portion must be a valid decimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 0000000000 |
| 1 | 0000000001 |
| 0000 | 0000000000 |
| 9 | 0000000009 |
| 0009 | 0000000009 |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to nines.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 0000000000-0000000009 |
| 1* | 0000000010-0000000019 |
| 5* | 0000000050-0000000059 |

| Rule Operand | Interpreted As |
|---|---|
| *1* | 0000000010-0000000919 |
| **1 | 0000000001-0000000991 |
| 6** | 0000000600-0000000699 |

Following is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value the transposition can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 0000000004-2147483647 |
| 400- | 0000000400-2147483647 |
| 12345- | 0000012345-2147483647 |
| 1-100 | 0000000001-0000000100 |
| 256-512 | 0000000256-0000000512 |
| 40000-80000 | 0000040000-0000080000 |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 0000000005-2147483647 |
| 5*- | 0000000050-2147483647 |
| 5**- | 0000000500-2147483647 |
| 5****- | 0000050000-2147483647 |
| 1**-2** | 0000000100-0000000299 |
| *6*-**4** | 0000000060-0000099499 |

**PFKEY**

PFKEY validates that the PF key number is between 01 and 24. For rule validation, you can mask the PF key number. Examples of this are shown below. If you do not mask the PF key number in the rule, the routine treats it like a command.

| Rule Operand | Interpreted As |
|---|---|
| pf1 | pf1 |

| Rule Operand | Interpreted As |
|---|---|
| pf* | pf01-pf09 |
| pf** | pf01-pf24 |
| pf- | pf01-pf24 |
| pf-* | invalid |
| pf*8 | pf08-pf24 |

For example, the PF nn operand of the QUERY command uses the PFKEY routine.

**RCUU**

For a command request, RCUU validates that the real device address specified in the command is really a valid hexadecimal number. You can specify TYPE=RANGE with RCUU to denote a range of real device addresses.

For a rule validation request, RCUU checks for masking. If there is no masking in the rule, the routine treats it like a command. Otherwise, it builds an upper and lower range and checks both for hexadecimal data. Below is an example using no masking. The number portion must be a valid hexadecimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 00000000 |
| 1 | 00000001 |
| 0000 | 00000000 |
| F | 0000000F |
| 000F | 0000000F |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to Fs.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 00000000-0000000F |
| 1* | 00000010-0000001F |
| 5* | 00000050-0000005F |
| *1* | 00000010-00000F1F |
| **1 | 00000001-00000FF1 |
| 6** | 00000600-000006FF |

Below is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value that the transposition routine can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 00000004-FFFFFFFF |
| 400- | 00000400-FFFFFFFF |
| 14A000- | 0014A000-FFFFFFFF |
| 1F0-A00 | 000001F0-00000A00 |
| 191-19F | 00000191-0000019F |
| E00000-FFFFFF | 00E00000-00FFFFFF |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 0000000005-FFFFFFFF |
| 5*- | 0000000050-FFFFFFFF |
| 5**- | 0000000500-FFFFFFFF |
| 5****- | 0000050000-FFFFFFFF |
| 1**-2** | 0000000100-000002FF |
| *6*-**4** | 0000000060-000FF4FF |

For example, the rcuu operand of the ATTACH command, when specified a single device address, uses the RCUU routine.

**REST**

REST validates that the length of all remaining command operands is not greater than the allowed maximum. For example, the variable data operand of the AUTOLOG command uses the REST routine.

**RUR (REAL)**

RUR is the same as VUR (VIRTUAL) except that the routine scans the real device block for the device class and type to make sure it is a real device. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. Specify the generic names of RDR, PRT, or PUN in the rule entry. When writing rules for operands that use this transposition routine, use the following:

| Rule Operand | Interpreted As |
|---|---|
| *- | any VUR device |
| P** | any print or punch device |
| CON | a console device |
| PRT\|PTR | a print device |
| PUN\|PCH | a punch device |
| RDR | a reader device |
| ALL | any or all devices |

Do not use real device addresses, such as 00C (commonly a reader) in rules, instead use RDR. The SPOOLOPT operand further restricts the use of any or all of these devices. During rules validation, the operand mask indicates PRT, PUN, RDR, or CON represents a generic type of unit record. We show some examples of this in the next example.

| Rule Operand | Interpreted As |
|---|---|
| printer | prt |
| reader | rdr |
| 00e (prt) | prt |
| p- | p- |

**SELF**

In a command syntax check, if the operand is an asterisk (*), the routine translates it to OWNER for rules validation. For rules validation, use the word OWNER to indicate the user can specify the asterisk (*) operand in the command. When you use SELF with SPOOLOPT=SELECT, it selects files from the spool queue. For example, the * operand of the ATTACH command uses the SELF routine.

**SPOOL**

For a command syntax check, SPOOL validates that the data specified in the command is really decimal data. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue.

For a rule validation request, SPOOL checks for masking. If there is no masking in the rule, the routine treats the rule like a command. Otherwise, the routine builds an upper and lower range and checks both for decimal data. For example, the spoolid operand of the CHANGE command uses the SPOOL routine.

**SPOOLTO**

SPOOLTO validates that the data specified in the command is decimal data or the END keyword. If you specify END, the routine converts it to 9999. When used with SPOOLOPT=SELECT, it selects files from the spool queue.

**STORADDR**

For a command syntax check, STORADDR locates one of the three possible delimiters: colon (:), dash (-), or period (.). Both the colon and dash indicate that the value is a fixed range. The period delimiter means the routine adds the value after the delimiter to the value before the delimiter. This results in a low-storage to high-storage address range for the command.

For a rules validation, the routine locates the delimiter. When the command delimiter is a dash, the routine validates that the dash is not the last character in the rule entry. If it is the last character, the routine treats it as a masking character for multiple characters during rule validation.

If the last character in the rule mask is not a dash, the routine checks the mask for one of the other delimiters. After the routine locates the real command delimiter, it converts the value before the delimiter to the low address and the value after the delimiter to the high address to build an address range. It then matches this range against the rules to validate execution of a command.

| Rule Operand | Interpreted As |
|---|---|
| 0512.10 | 512 to FFFF |
| *512:**** | 512 to FFFF |
| 0*F | 000 to 00F |
| 0- | 000 to 7FFFFFFF |

For example, the hexloc options of the DCP command use the STORADDR routine.

**STORDISP**

STORDISP is the same as STORADDR, except that the routine justifies the upper and lower address ranges to a 4-byte boundary. However, if you prefix the storage address with a T, the routine justifies the range to a 16-byte boundary.

**STORDUMP**

STORDUMP is the same as STORADDR, except that the routine justifies the upper and lower address ranges to a 32-byte boundary.

**STORVDMP**

STORVDMP is the same as STORADDR, except that the routine justifies the upper and lower address ranges to a 4K-byte boundary.

**STRSIZE**

During command syntax checking, STRSIZE validates that the operand is between 0K and 999M and checks for masking (* and -) and range values. If you use range values, the last position must be either K or M, and the numeric part must be a valid decimal number. You cannot mask range operands. The routine normalizes all values to K-bytes.

For rules validation, the routine converts the storage size to a nnnnnK value. When comparing to a rule mask, the routine uses a lower and upper range value. We show some examples of this conversion below. First is an example with no masking used.

| Rule Operand | Interpreted As |
|---|---|
| 512k | 0000512 |
| 512m | 0000512 (Mb)<br>05242888 (converted to K) |
| 740K | 0000740 |

Some examples of this conversion with masking using an asterisk (*) are:

| Rule Operand | Interpreted As |
|---|---|
| 1*k | 0000010-0000019 |
| 1*M | 0000010-0000019 (Mb)<br>0010240-0019456 (converted to K) |
| 5*K | 0000050-0000059 |
| 5*M | 0000050-0000059 (Mb)<br>0051200-0060416 (converted to K) |
| 5** | 0000050-0000059 (low in K, high in Mb)<br>0000050-0060416 (converted to all K) |
| **1K | 0000001-0000991 |
| **1M | 0000001-0000991 (Mb)<br>0001024-1014784 (converted to K) |

| Rule Operand | Interpreted As |
|---|---|
| *1*K | 0000010-0000919 |
| *1*M | 0000010-0000919 (Mb)<br>0010240-0941056 (converted to K) |
| *1** | 0000010-0000919 (low in K, high in Mb)<br>0000010-0941056 (converted to K) |
| *5*K | 0000050-0000959 |
| *5*M | 0000050-0000959 (Mb)<br>0051200-0982016 (converted to K) |
| *5** | 0000050-0000959 (low in K, high in Mb)<br>0000050-0982016 (converted to K) |
| 1**K | 0000100-0000199 |
| 1**M | 0000100-0000199 (Mb)<br>0102400-0203776 (converted to K) |
| 1***K | 0001000-0001999 |
| 1***M | (error) |

We show some examples of this conversion with masking using a dash (-) below. The dash must be the last byte.

| Rule Operand | Interpreted As |
|---|---|
| - | 0000000-1022976 |
| 5- | 0000005-0000999 (low in K, high in Mb)<br>0000005-1022976 (converted to K) |
| 50- | 0000050-0000999 (low in K, high in Mb)<br>0000050-1022976 (converted to K) |

We show some examples of this conversion with masking using asterisk (*) and dash (-) below. The dash must be the last byte.

| Rule Operand | Interpreted As |
|---|---|
| 5*- | 0000050-0000999 (low in K, high in Mb)<br>0000050-1022976 (converted to K) |
| 5**- | 0000500-0000999 (low in K, high in Mb)<br>0000500-1022976 (converted to K) |
| 5****- | 0050000-0000999 (low in K, high in Mb)<br>0050000-1022976 (converted to K) |

Listed below are some examples of storage ranges. This preferred technique specifies ranges of storage more easily. It lets you use a dash (-) as a range character, not a masking character. You cannot mask in a range.

| Rule Operand | Interpreted As |
|---|---|
| 512-740K | 0000512-0000740 |
| 512-740M | 0000512-0000740<br>0524288-0757760 (converted to K) |
| 5*2-740K | (error, no asterisk mask permitted) |
| 740-512K | (error, illogical range) |
| 3-5K | 0000003-0000005 |
| 3-5M | 0000003-0000005 (Mb)<br>0003072-0005120 (converted to K) |
| -M | (error) |
| -K | (error) |

**SYSNAME**

SYSNAME scans the CSESYS table in HCPSYS to validate that a system name is a valid CSE system name. A command fails with an CA ACF2 for z/ VM syntax error if the system name or alias specified on a command is not in this list. When you specify a valid system name or alias in the command, this transposition routine converts the alias name into the system name. This lets you write rules for the system name while protecting the alias name.

When you specify a valid system name or alias in a rule, this routine also converts the alias name into the system name for the rule. If you write the rule for the alias name, you also protect the system name. You can use a new pseudoname, LOCAL, to write rules that apply only to the system the user is on. This is similar to using OWNER when dealing with spool-related commands.

SYSNAME works like an ANY transposition routine on non-XA systems. Test these transposition routines on XA systems.

**USER**

During a command syntax check, USER verifies that the user ID is greater than one and less than (or equal to) eight characters. Except for CMDLIM rule compiles, it also checks to ensure the specified user ID is in the VM directory. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. For example, the user ID operand of the QUERY command uses the USER routine. The routine checks the VM directory to help determine the command format the user is using so that CA ACF2 for z/ VM uses the correct FORMAT clause in the command model during validation.

**VCUU**

For a command request, VCUU validates that the data specified in the command is really hexadecimal data. You can specify TYPE=RANGE with VCUU to denote a range of hexadecimal data.

For a rule validation request, VCUU checks for masking. If there is no masking in the rule, the routine treats the rule like a command. Otherwise, the routine builds an upper and lower range and checks both for hexadecimal data.

Below is an example using no masking. The number portion must be a valid hexadecimal digit.

| Rule Operand | Interpreted As |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 0000 | 0000 |
| F | 000F |
| 000F | 000F |

Below is an example of using an asterisk (*) for masking. The transposition routine converts any operand with an asterisk to a low and a high range. It converts character positions with an asterisk in the low range to zeros and converts character positions with an asterisk in the high range to Fs.

| Rule Operand | Interpreted As |
|---|---|
| 0* | 0000-000F |
| 1* | 0010-001F |
| 5* | 0050-005F |
| *1* | 0010-0F1F |
| **1 | 0001-0FF1 |
| 6** | 0600-06FF |

Below is an example of using a dash (-) for masking. The transposition routine converts any operand with a dash to a low and a high range. The value before the dash becomes the low range. The value after the dash becomes the high range. If you omit the high range, then the high range defaults to the largest value that the transposition routine can convert to a fullword.

| Rule Operand | Interpreted As |
|---|---|
| 4- | 0004-FFFF |

| Rule Operand | Interpreted As |
|---|---|
| 400- | 0400-FFFF |
| 14A0- | 14A0-FFFF |
| 1F0-A00 | 01F0-0A00 |
| 191-19F | 0191-019F |

Below is an example of using a combination of asterisks and dashes for masking.

| Rule Operand | Interpreted As |
|---|---|
| *5- | 000005-FFFF |
| 5*- | 000050-FFFF |
| 5**- | 000500-FFFF |
| 1**-2** | 000100-02FF |
| *6*-**4* | 000060-FF4F |

For a rule validation request, VCUU checks for masking. If there is no masking in the rule, the routine treats the rule like a command. Otherwise, the routine builds an upper and lower range and checks both for hexadecimal data. For example, the vaddr operand of the ATTACH command uses the VCUU routine.

**VUR (VIRTUAL)**

During command syntax checking, VUR converts the words (or abbreviation) PRINTER, PUNCH, READER, or CONSOLE to PRT, PUN, RDR, or CON, respectively. It also checks to ensure that the device is really a PRT or PUN. When you use this operand with SPOOLOPT=SELECT, it selects files from the spool queue. For examples and restrictions, see Spool Related Operands (Format 4) in the chapter “Using the Model Setting.”

When writing rules for operands that use this transposition routine, use the following:

| Rule Operand | Interpreted As |
|---|---|
| *- | any VUR device |
| P** | any print or punch device |
| CON | a console device |
| PRT\|PTR | a print device |
| PUN\|PCH | a punch device |
| RDR | a reader device |

| Rule Operand | Interpreted As |
|---|---|
| ALL | any or all devices |

Do not use real device addresses, such as 00C (commonly a reader) in rules, instead use RDR. The SPOOLOPT operand further restricts the use of any or all of these devices.

During rules validation, the operand mask indicates PRT, PUN, RDR, or CON represents a generic type of unit record. We show some examples of this in the next example.

| Rule Operand | Interpreted As |
|---|---|
| printer | prt |
| ptr | prt |
| 00e (prt) | prt |
| p- | p- |

**XSIZE**

This transposition routine looks for a trailing M to normalize the storage size operand of the XSIZE command. If it finds one, it converts the decimal part of the operand into a binary word so that command limiting can match everything up (for example, ATTACH XSIZE 128000M).

| Rule Operand | Interpreted As |
|---|---|
| * | any |
| 0* | 0M-9M |
| 1* | 10M-19M |
| 5* | 50M-59M |
| *1* | 010M-919M |

# Index

## A



## C

## D



## E



## F

## G



## H



## I



## J



## K



## L



## M

## N



## O

## P



## R

## S

## T



## U



## V



## X