HP OpenView GlancePlus

(for HP-UX 10.20 and 11.x)

Adviser

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD PROVIDES THIS MATERIAL "AS IS" AND MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE OR USE OF THIS MATERIAL WHETHER BASED ON WARRANTY, CONTRACT, OR OTHER LEGAL THEORY.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard. This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Hewlett-Packard Company

Application & System Management Division

8000 Foothills Blvd.

Roseville, CA 95747-5726, USA

Conventions

boldface	Words in boldface represent the names of programs and commands.
`computer font`	Words in computer font represent file names, syntax, directory path names, or text as you should enter it on your workstation or terminal, and text that appears on the screen.
italics	Italics are used to emphasize words, phrases, or characters in the text, or indicate variables in syntax strings.
Return or Enter keys	Depending on your keyboard, one or the other represents the key used to execute a command.

Contents

Introduction

The GlancePlus Adviser monitors your system and helps you find potential problems. Think of the Adviser as a set of rules based on performance metrics; the rules can be used to take different actions. You can use the Adviser to communicate situations in a number of different ways. For example, you can edit the Adviser to:

- Display information to `stdout`.
- Execute UNIX commands, such as `mailx,` to send yourself a message.
- Make the ALARM button on the main GlancePlus window yellow or red, depending on the severity of the alarm.
- Make the border on the GlancePlus icon yellow or red when you are running in iconified mode.
- Display a specific GlancePlus window to help analyze problems.

A good way to learn the Adviser is to experiment with adding simple syntax to the default syntax that GlancePlus provides. As you gain knowledge about the Adviser and begin to understand the default syntax, you can modify the default syntax to make it more useful for your own environment.

Alarms and Symptoms

Alarms are simply a way to highlight metric conditions in GlancePlus. A symptom is a combination of conditions that occurs during an interval and contributes to a bottleneck on your system.

NOTE: An interval is the period of time since the last measurement. GlancePlus evaluates the Adviser SYMPTOMS and ALARMS at each interval. The default interval is 15 seconds. To change the default interval, use the Configure Measurement window.

Check Related Topics below for a list of topics with detailed discussions and examples of how to create alarms and symptoms using the Adviser syntax.

What is an Alarm?

An alarm can trigger whenever conditions that you specify are met. Alarms are based on any period of time you specify, which can be one interval or longer. Conditions or events that you might want to set as Adviser alarms include:

- when global swap space is nearly full
- when the page in rate is too high
- when your process table is near capacity
- when your CPU has been running at 75% utilization for the last two minutes

Several screens let you look at alarm status and history. The status of alarm conditions determines the color of the main window's Alarm button. Several alarms are defined in the GlancePlus default Adviser syntax. (To see the default syntax, open the Edit Adviser Syntax window in GlancePlus.)

What is a Symptom?

Complex alarms can be built based on symptoms. The GlancePlus default Adviser syntax defines four bottleneck symptoms for you, then defines alarms based on those symptoms. (Open the Edit Adviser Syntax window in GlancePlus to see the default syntax.)

By observing different metrics with corresponding thresholds and adding values to the probability that these metrics contribute to a bottleneck, the Adviser calculates one value that represents the combined probability that a bottleneck is present.

Unlike the ALARM statement that monitors conditions over a period of time normally longer than one interval, the SYMPTOM statement is evaluated and updated every interval. This is why you might see the CPU Bottleneck Symptom indication prior to a CPU Bottleneck Alarm. Symptoms change rapidly and can become yellow, then red, then go back to green. An alarm remains yellow or red until it is reviewed or reset.

You can also use the variables you defined in the SYMPTOM statements in the Alarm section. And you can link the symptoms to the CPU, Disk, Network, and Memory buttons on the main GlancePlus window to notify you of possible bottlenecks.

For every symptom that you define in the Adviser Syntax window, a graph appears on the Symptom History window to show that particular symptom's probability over time.

Editing Adviser Syntax

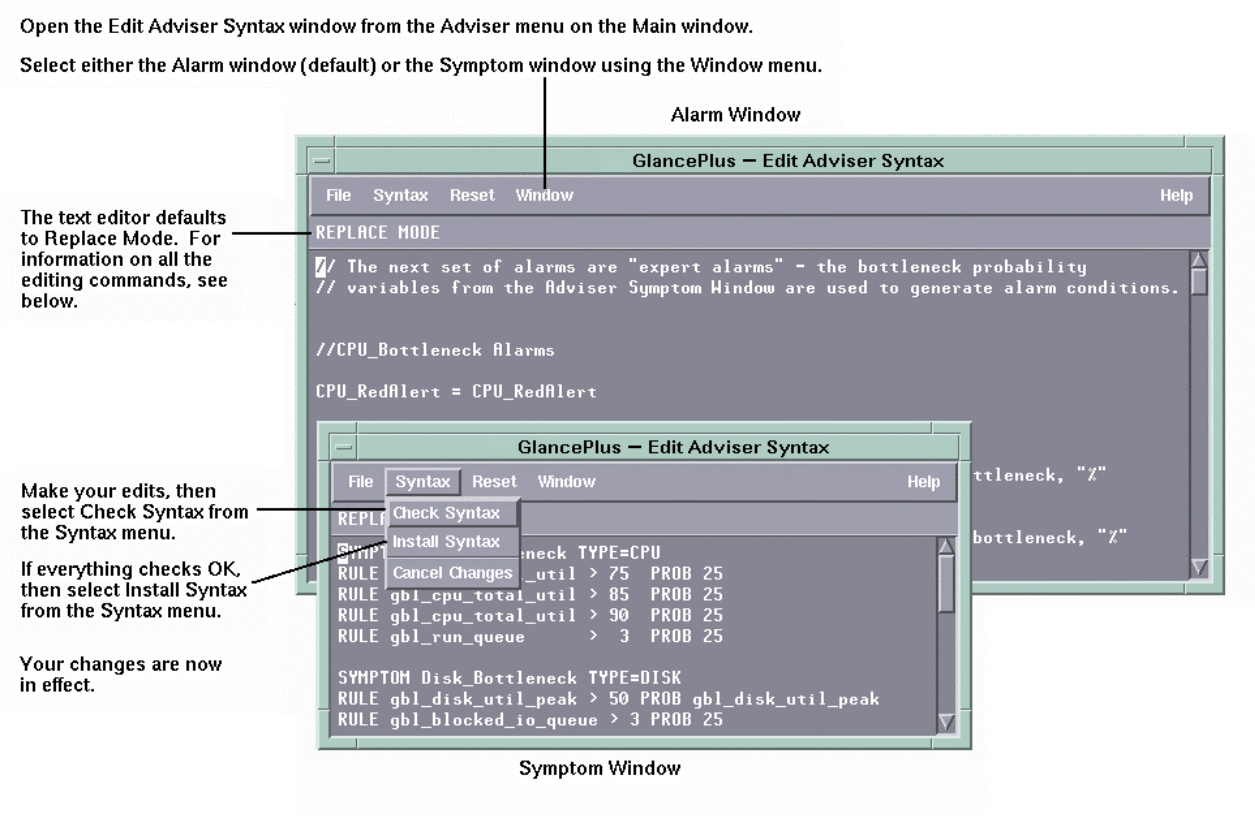
Don't worry too much about making mistakes; you can always go back to the default Adviser syntax by selecting the Default Syntax option from the Reset menu in the Edit Adviser Syntax Window.

You can edit the syntax in two ways:

- Using the GlancePlus Text Editor
- Using your own text editor

Using the GlancePlus Text Editor

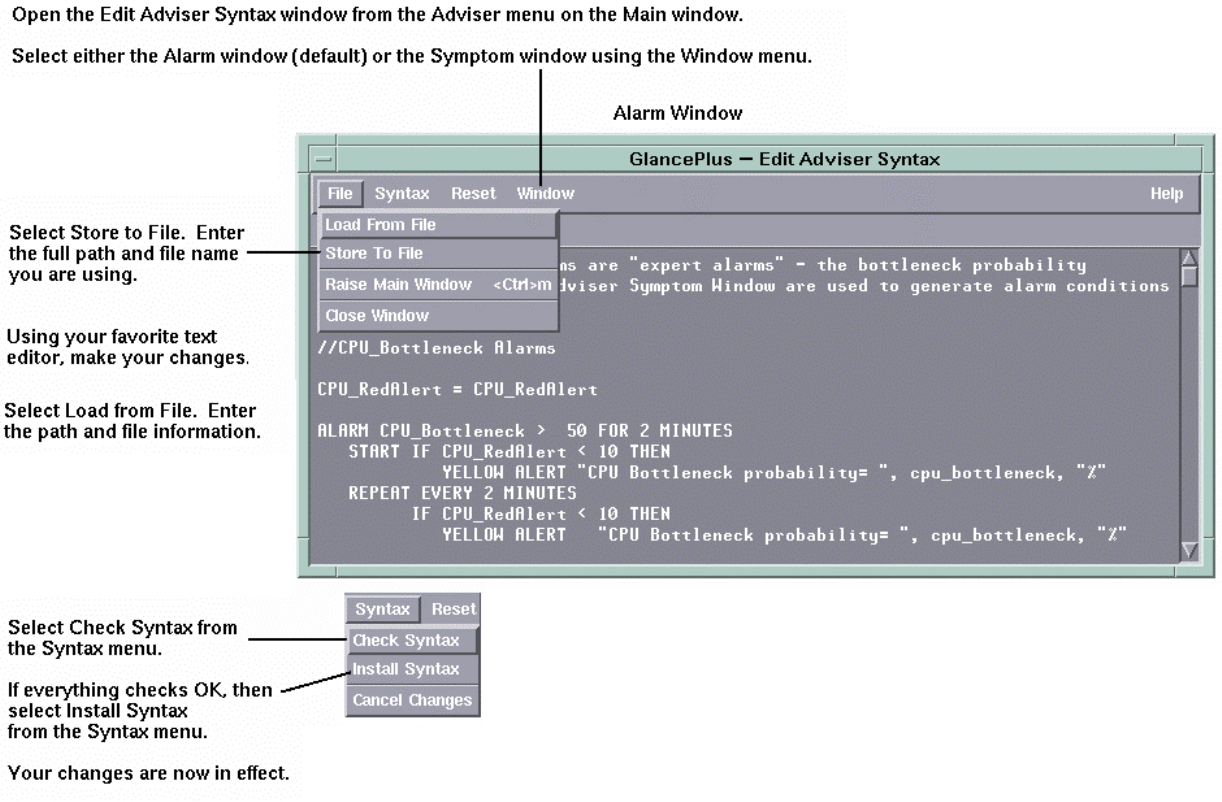
You can edit the adviser syntax from within GlancePlus. Here's how you do it.



See Syntax Editing Commands for instructions on using the GlancePlus text editor. If errors display after you select Check Syntax, see GlancePlus Messages for more information.

Using Your Own Text Editor

You can edit the adviser syntax using your favorite text editor. Here's how to do it.



See Syntax Editing Commands for instructions on using the GlancePlus text editor.

If errors display after you select Check Syntax, see GlancePlus Messages for more information.

Syntax Editing Commands

To edit text in the Adviser Syntax Window, you use various editing commands. You cannot use the mouse to move the cursor in the Adviser Syntax window.

To replace text:

Each time you open the Adviser Syntax window, the editing function is in REPLACE MODE. You can overtype the syntax with characters or blanks using the Replace Mode. To return to REPLACE MODE after inserting text, press the **Insert char** key.

To insert text:

To insert information in the Adviser Syntax window, press the **Insert char** key. The message at the top of the Adviser Syntax window changes to INSERT MODE. To insert lines or characters, use the **Insert line** or **Insert char** key.

To delete text:

To delete lines or characters, use the **Delete line** key or the **Delete** char key.

Moving the cursor in the Adviser Syntax window:

To move the cursor one character at a time, use your keyboard arrow keys. To page through text, use the **prev** and **next** keys, or use the vertical scroll bar on the right. To scroll through text horizontally, use the shift key and the left or right arrow keys.

Displaying GlancePlus Data

Print CPU Total Utilization

Follow these steps to print metric values to the gpm window:

1. From the GlancePlus main window, select Edit Adviser Syntax from the Adviser menu to open the Edit Adviser Syntax window.
2. In the Edit Adviser Syntax window, press the Insert (or Insert Char) key and then press Return a few times to insert several blank lines at the top of the file.
3. Insert the following text in the space you just created at the top of the syntax:
   ```
   print gbl_cpu_total_util
   ```
4. From the Syntax menu, select Install Syntax. The Edit Adviser Syntax window closes and the print statement executes the next time GlancePlus updates its data.

   When you select Install Syntax, GlancePlus checks your syntax for correctness. If an error is found, an error message is displayed at the top of the window. For an explanation of any syntax error messages, see GlancePlus Messages.
5. Look at the window from which you started GlancePlus. The numbers appearing in that window result from GlancePlus printing the value of a global GlancePlus metric (your global CPU utilization) every update interval.

   Refer to Printing CPU Utilization During High CPU Usage to see how you can print CPU utilization to stdlist only when your CPU is very busy.

Printing CPU Utilization During High CPU Usage

Perhaps you want to print CPU utilization only when usage exceeds 90% busy.

1. Go back to the Edit Adviser Syntax Window and replace the line you typed with the following:

```
if gbl_cpu_total_util > 90 then
    print "total cpu utilization is high: ", gbl_cpu_total_util
```

2. From the Syntax menu, select Install Syntax. The Edit Adviser Syntax window closes, and the print statement executes the next time GlancePlus updates its data.

   When you select Install Syntax, GlancePlus checks your syntax for correctness. If an error is found, an error message is displayed at the top of the window. For an explanation of any syntax error messages, see GlancePlus Messages.

3. Look at the window from which you started GlancePlus. You may not see any numbers because data only displays when your CPU is more than 90% busy.

4. To start a program that uses a lot of CPU and view what happens, type the following at a shell prompt (sh or ksh) to cause a loop:

```
while true
do
    A=1
done
```

   This makes the shell loop until you interrupt it with control-c. When the loop starts, the Adviser starts printing out information.

Sending Email Messages

You can use metrics that are shown in different GlancePlus windows in your Adviser syntax. Rather than printing metrics to `stdout`, you can send the same information to yourself in an email message.

1. Go to the Edit Adviser Syntax Window, and replace the line you typed with the following:

   ```
   if gbl_cpu_total_util > 90 then
     exec "echo 'cpu is too high ", gbl_cpu_total_util, "% ' | mail root"
   ```

2. From the Syntax menu, select Install Syntax. The Edit Adviser Syntax window closes.

   When you select Install Syntax, GlancePlus checks your syntax for correctness. If an error is found, an error message is displayed at the top of the window. For an explanation of any syntax error messages, see GlancePlus Messages.

Printing Process Information Within a Loop

To customize your syntax further, you can combine metrics, define variables, and use looping constructs. This example shows how you can:

- construct loops inside conditions which only execute when a potential problem situation arises
- use variables inside the adviser syntax to keep track of things inside loops. You could change the thresholds in this example to isolate problems unique to your environment.

This example tests for an overall high global system mode CPU utilization. When GlancePlus encounters this situation, it loops through all the active processes, printing out information about the process with the highest percentage of time spent in system mode.

1. Go back to the Edit Adviser Syntax Window, and replace the line you typed with the following:

```
// check for high system-mode cpu utilization, and when it is high,
// print the highest sys cpu consuming process:
if gbl_cpu_sys_mode_util > 50 then  {
  highestsys = 0
  process loop  {
    if proc_cpu_sys_mode_util > highestsys then  {
      highestpid = proc_proc_id
      highestname = proc_proc_name
      highestsys = proc_cpu_sys_mode_util
    }
  }
  print "--- High system cpu rate = ", gbl_cpu_sys_mode_util, " at ",
    gbl_stattime, " ---"
  print "    Process with highest system cpu was pid ", highestpid,
    ", name: ", highestname
  print "    which had", highestsys, " percent system mode cpu ",
    "utilization"
}
```

2. From the Syntax menu, select Install Syntax. The Edit Adviser Syntax window closes, and the print statement executes the next time GlancePlus updates its data.

Print to a File

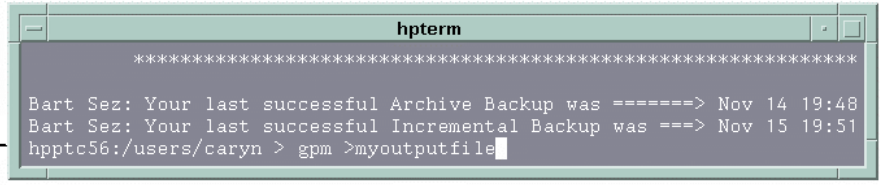
You can print information to a file by using the PRINT statement in the Adviser Syntax and by rerouting `stdout` to a file.

By using the PRINT statement, which sends its output to the defined `stdout` of GlancePlus, you can format metrics with literal constants and user-defined variables. To reroute the `stdout`, start GlancePlus by appending `>filename` to the command line. This causes all output destined for `stdout` to be placed in the file specified by `filename`.

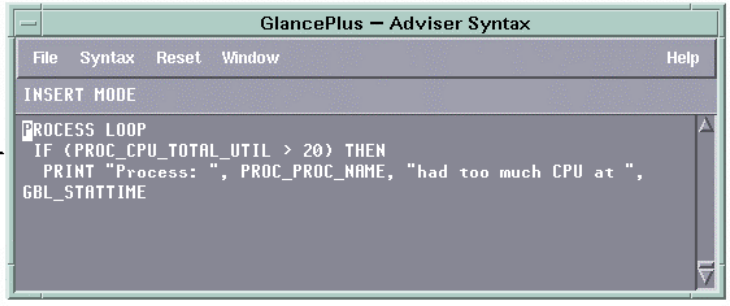
The following example shows how to print global and process metrics to a file:

Start GlancePlus at the command line.

All output will be routed to the file "myoutputfile".



To print a message for every process that had CPU greater than 20% for any interval, add the syntax shown here.



The output in your "myoutputfile" will look similar to this:

```
Process: cpuhog had too much CPU at 15:20:20
```

Adviser Syntax Structure

Alarm Syntax

```
ALARM condition [FOR duration{SECONDS, MINUTES, INTERVALS}]
  [condition [FOR duration[SECONDS, MINUTES, INTERVALS]] ...
  [START statement]
  [REPEAT [EVERY duration [SECONDS, MINUTES, INTERVALS]]
    statement]
  [END statement]
  [(RED or CRITICAL), (YELLOW or WARNING), RESET] ALERT
    statement
ALIAS variable = alias name
[VAR] variable = expression
{
    compound statements
}
EXEC printlist
GPM -rpt reportlist
IF condition
  THEN statement
  [ELSE statement]
{APPLICATION, APP, CPU, DISK, DISK_DETAIL, FILESYSTEM, FS, FS_DETAIL,
LAN, LOGICALVOLUME, LV, LV_DETAIL, NETIF, NFS, NFS_BYSYS_OPS, NFS_OP,
PRM, PROCESS, PROC, PROCDCEIF, PROCDCEOP, PROC_FILE, PROC_REGION,
PROC_SYSCALL, SWAP, SYSTEMCALL, SC, THREAD, TRANSACTION, TT, TTBIN,
TT_CLIENT, TT_INSTANCE, TT_UDM, TT_RESOURCE, TT_INSTANCE_CLIENT,
TT_INSTANCE_UDM, TT_CLIENT_UDM} LOOP statement
PRINT printlist
```

Symptom Syntax

```
SYMPTOM variable [ TYPE = {CPU, DISK, MEMORY, NETWORK}]
RULE measurement {>, <, <=, >=, ==, !=} value PROB probability
[RULE measurement {>, <, <=, >=, ==, !=} value PROB probability]
```

Adviser Syntax Reference

Syntax Conventions

- Braces ({}) indicate that one of the choices is required.
- Brackets ([]) indicate an optional item.
- Items separated by commas within brackets or braces are options. Choose only one.
- Italics indicate a variable name that you will replace.
- All CAPS are Adviser syntax keywords.

Comments

Syntax:

```
# [any text or characters]
```

or

```
// [any text or characters]
```

You can precede comments either by double forward slashes (//) or the pound sign (#). In both cases, the comment ends at the end of the line.

Conditions

A condition is defined as a comparison between two metric names, user variables, or numeric constants.

```
item1 {>, <, >=, <=, ==, !=} item2 [OR item3 {>, <, >=, <=, ==, !=} item4]
```

or:

```
item1 {>, <, >=, <=, ==, !=} item2 [AND item3 {>, <, >=, <=, ==, !=} item4]
```

("==" means "equal", and "!=" means "not equal".)

Conditions are used in the ALARM statement and the IF statement. They can be used to compare two numeric metrics, variables or constants, and they can also be used between two string metric names, user variables or string constants. For string conditions, only == or != can be used as operators.

You can use compound conditions by specifying the OR or AND operator between subconditions.

Examples:

```
gbl_swap_space_reserved_util > 95

proc_proc_name == "test" OR proc_user_name == "tester"

proc_proc_name != "test" AND
                  proc_cpu_sys_mode_util > highest_proc_so_far
```

Constants

Constants can be either numeric or alphanumeric. An alphanumeric constant must be enclosed in double quotes. Constants are useful in expressions, conditions, and statements. For example, you may want to compare a metric against a constant numeric value inside a condition to generate an alarm if it is too high.

Examples:

345	Numeric integer
345.2	Numeric real
"Time is"	Alphanumeric literal

Expressions

Use expressions to evaluate numerical values. An expression can be used in a condition or an action.

An expression can contain:

- numeric constants
- numeric metric names
- numeric variables
- an arithmetic combination of the above
- a combination of the above grouped together using parentheses

Examples:

```
Iteration + 1

3.1416

gbl_cpu_total_util - gbl_cpu_user_mode_util

( 100 - gbl_cpu_total_util ) / 100.0
```

Metric Names

You can use the following types of metrics in the Adviser syntax:

- global metrics (prefixed with `gbl_` or `tbl_`)
- application metrics (prefixed with `app_`)
- process metrics (prefixed with `proc_`)
- disk metrics (prefixed with `bydsk_`)
- file system metrics (prefixed with `fs_`)
- logical volume metrics (prefixed with `lv_`)
- network interface metrics (prefixed with `bynetif_`)
- swap metrics (prefixed with `byswp_`)
- ARM metrics (prefixed with `tt_` or `ttbin_`)
- PRM metrics (prefixed with `prm_`)

Metrics can contain alphanumeric (for example, `gbl_machine` or `app_name`) or numeric data and can reflect several different kinds of measurement. For example, the metric ending of a metric name indicates what is being measured:

- `_util` measures utilization in percentages
- `_rate` measures units per second
- `_queue` measures the number of processes or threads waiting for a resource

You can use a global metric anywhere in the Adviser syntax, but you can only use process, logical volume, disk, file system, LAN, and swap metrics within the context of a LOOP statement.

You must associate an application metric with a specific application, except when using the LOOP Statement. To do this, specify the application name followed by a colon, and then the metric name. For example, `other_apps:app_cpu_total_util` specifies the total CPU utilization for the application `other_apps`. Refer to the ALIAS Statement for more information on using application metrics in the syntax.

Printlist

The printlist is any combination of properly formatted expressions, Metric Names, user variables, or constants. See the examples for the proper formatting.

Expression Examples:

```
expression [|width[|decimals]]
```

Metric Names or User Variable Examples:

```
metric names [|width[|decimals]]
```

or

```
user variables [|width[|decimals]]
```

The metric names or user variables must be alphanumeric.

Constant Examples:

No formatting is necessary for constants.

Formatted Examples:

```
gbl_cpu_total_util|6|2    formats as     '100.00'
(100.32 + 20)|6    formats as     '   120'
gbl_machine|5      formats as     '7013/'
"User Label"       formats as     "User Label"
```

Variables

Variables must begin with a letter and can include letters, digits, and the underscore character. Variables are not case-sensitive.

Define a variable by assigning something to it. The following example defines the numeric variable `highest_CPU_value` by assigning it a value of zero.

```
highest_CPU_value = 0
```

The following example defines the alphanumeric variable `my_name` by assigning it a null string value.

```
my_name = ""
```

Adviser Syntax Statements

ALARM Statement

Use the ALARM statement to notify you when certain events, which you define, occur on your system. Using the ALARM statement, the Adviser can notify you in a number of different ways:

- through messages to the Alarm History window
- through messages sent to your originating shell
- by automatically opening a GlancePlus window

Syntax:

```
ALARM   condition   [FOR duration {SECONDS, MINUTES, INTERVALS}]
  [condition [FOR duration {SECONDS, MINUTES, INTERVALS}]  ] ...

[START statement]

[REPEAT [EVERY duration [SECONDS, MINUTES, INTERVAL, INTERVALS]]
  statement]

[END statement]
```

The ALARM statement must be a top-level statement. It cannot be nested within any other statement.

However, you can include several ALARM conditions in a single ALARM statement, in which case all conditions must be true for the alarm to trigger. And you can also use a COMPOUND Statement, which is executed at the appropriate time during the alarm cycle.

START, REPEAT, and END are ALARM statement keywords. Each of these keywords specifies a *statement*. You must have a START, REPEAT, or END in an ALARM statement, and they must be listed in correct order.

The alarm cycle begins on the first interval that all of the alarm conditions have been true for at least the specified duration. At that time, the Adviser executes the START *statement*, and on each subsequent interval checks the REPEAT condition. If enough time has transpired, the *statement* for the REPEAT clause is executed. This continues until one or more of the alarm conditions becomes false. This completes the alarm cycle and the END *statement* is executed.

If you omit the EVERY specification from the REPEAT statement, the Adviser executes the REPEAT statement at each interval.

ALARM Example: Typical ALARM Statement

The following ALARM example sets a red alert when the semaphore table is almost full. It is similar to a predefined Alarm in the default syntax. Do not add this to your syntax without removing the default, or your subsequent alert messages may be confusing.

```
ALARM tbl_sem_table_util > 90 FOR 1 MINUTE
  START RED ALERT "Semaphore Table is nearly full"
  REPEAT EVERY 30 SECONDS
    RED ALERT "Semaphore Table still nearly full"
  END RESET ALERT "End of Semaphore Table full condition"
```

This ALARM example tests the metric tbl_sem_table_util to see if it is greater than 90. If it is, the RED ALERT statement changes the ALARM button label on the Main window (or on the GlancePlus icon if you are running in iconified mode) to red and places the "Semaphore Table is nearly full" message in the Alarm History window.

The REPEAT statement checks for the tbl_sem_table_util condition every 30 seconds. As long as the condition is greater than 90, the REPEAT tells the Adviser to maintain a RED ALERT condition and sends the "Semaphore Table still nearly full" message to the Alarm History window.

When the tbl_sem_table_util condition goes below 90, the RESET ALERT statement turns off the alert color and logs the " End of Semaphore Table full condition" message in the Alarm History window.

ALARM Example: Using COMPOUND Statements

Use the following example to use a COMPOUND statement within the ALARM statement. This example shows you how to make the Adviser open a window when an event occurs and how to print a statement to your originating GlancePlus shell:

```
ALARM cpu_bottleneck > 90 FOR 1 MINUTE
  START  {
    RED ALERT "Your CPU is bottlenecked."
    GPM -rpt cpugraph
    PRINT "CPU is running at:  ", gbl_cpu_total_util
  }
  END
    RESET ALERT "CPU crisis is over."
```

ALARM Example: Using Multiple Conditions

You can have more than one test condition in the ALARM statement. In this case, each statement must be true for the alarm button to activate. For example:

```
ALARM gbl_cpu_total_util > 90 FOR 2 MINUTES
  gbl_cpu_sys_mode_util > 50 FOR 1 MINUTES
    START RED ALERT
      "The CPU is busy and System Mode CPU utilization is high."
    END RESET ALERT "The CPU alert is now over."
```

This ALARM example tests the metric gbl_cpu_total_util and CPU_Bottleneck. If both conditions are true, the RED ALERT statement sets a critical alert. When either test condition becomes false, the RESET is executed.

ALARM Example: Process Table

```
ALARM tbl_proc_table_util > 90 FOR 1 MINUTES
  START RED ALERT "Proc table is nearly full"
  END RESET ALERT "End of Proc table full condition"
```

This alarm turns the Alarm button red when the process table is full. This red alert alarm also shows up in the Alarm History window.

ALARM Example: Swap Space

```
//GLOBAL SWAP ALARM

symp_swap_util = gbl_swap_space_used / gbl_swap_space_avail

ALARM symp_swap_util > 0.9

  START

    RED ALERT "GLOBAL SWAP space is nearly full"

  END RESET ALERT "GLOBAL SWAP space crisis is over"
```

This example shows computing a new variable, `symp_swap_util`, which represents swap utilization. The Adviser will send an alarm when the swap utilization exceeds 90%. On the next interval that `symp_swap_util` falls below 90%, the alarm condition becomes false, and the ALARM is reset.

ALARM Example: Yellow Alert

```
ALARM Symp_Global_Cpu_Bottleneck >  50 FOR 2 MINUTES

  START YELLOW ALERT "CPU Bottleneck probability= ",
    Symp_Global_Cpu_Bottleneck, "% for the last 2 minutes"

  REPEAT every 2 minutes
    YELLOW ALERT "CPU Bottleneck probability= ",
      Symp_Global_Cpu_Bottleneck, "% for the last 2 minutes"

  END
    RESET ALERT " CPU Bottleneck Yellow Alert over, probability=",
      Symp_Global_Cpu_Bottleneck, "%"
```

The ALARM tests the SYMPTOM variable, which is defined in the SYMPTOM Statement `Symp_Global_Cpu_Bottleneck`. If the SYMPTOM variable is greater than 50 for 2 minutes the ALARM notifies you with a YELLOW ALERT to your main GlancePlus window. The `CPU Bottleneck probability` message is recorded in the Alarm History window.

The ALARM will REPEAT every 2 minutes until the ALARM condition is false, at which time END will RESET the ALERT and post the corresponding message to the Alarm History window.

ALARM Example: CPU Problem

```
ALARM
  gbl_cpu_total_util > 90 FOR 30 SECONDS
  gbl_run_queue > 3 FOR 30 SECONDS
  START YELLOW ALERT "CPU AT ", gbl_cpu_total_util,
    "% at ", gbl_stattime
  REPEAT EVERY 300 SECONDS {
    RED ALERT "CPU AT ", gbl_cpu_total_util
    exec "/usr/bin/pager -n 555-3456"
  }
  END ALERT "CPU at ", gbl_cpu_total_util, "% at ",
    gbl_stattime, " - RELAX"
```

This example lights a yellow alert on the ALARM button or icon whenever CPU utilization exceeds 90% for 30 seconds and the CPU run queue exceeds 3 for 30 seconds, and writes a message to the Alarm History window.

If both conditions continue to hold true, a red alert is generated, and another message is written to the Alarm History window, as well as a program (a fictitious example) run to page the system administrator.

When either one of the alarm conditions fails to be true, the ALARM BUTTON or icon resumes its normal color and a message is written to the Alarm History window giving the global CPU utilization, the time the alert ended, and a note to RELAX.

ALERT Statement

The ALERT statement is used to place a message in the Alarm History Window. Whenever an ALARM detects a problem it can execute an ALERT statement to activate the ALARM button label on the Main window or the icon border to notify you of a problem. A user-customized message, specified by *printlist*, records the event in the Alarm History window. You can use the ALERT statement in conjunction with an ALARM statement.

Syntax:

```
[(RED or CRITICAL), (YELLOW or WARNING), RESET] ALERT printlist
```

RED and YELLOW, are synonymous with CRITICAL and WARNING. These keywords place the *printlist* in the Alarm History window, along with the

time and alarm level, in red or yellow characters. They also change the text color of the ALARM button on the Main window to red or yellow, or if iconified, set the icon border to a flashing red or yellow color. If you prefer, you can set a no priority alert (not red or yellow, just information to the Alarm History Window).

RESET records the *printlist* in the Alarm History window and resets any colors on the icon or ALARM button to their normal color.

ALERT Example

An example an ALERT statement is:

```
RED ALERT "CPU utilization = ", gbl_cpu_total_util,
  " at ", gbl_stattime
```

When executed this statement turns the ALARM button label red or, if GlancePlus is iconified, puts a flashing red border in the icon and writes a message in the Alarm History window that reads, for example:

```
CPU utilization = 85.6 at 14:43:10
```

ALIAS Statement

Use the ALIAS statement to assign a variable to an application name that contains special characters or imbedded blanks.

Syntax:

```
ALIAS variable = "alias name"
```

ALIAS Example

Because you cannot use special characters or imbedded blanks in the syntax, using the application name "`other user root`" in the PRINT statement below would have caused an error. Using ALIAS, you can still use `"other user root"` or other strings with blanks and special characters within the syntax.

```
ALIAS otherapp = "other user root"

PRINT "CPU for other root login processes is: ",
  otherapp:app_cpu_total_util
```

ASSIGNMENT Statement

Use the ASSIGNMENT statement to assign a numeric or alphanumeric value, *expression*, to the user variable.

Syntax:

```
[VAR] variable = expression
[VAR] variable = alphaitem
[VAR] variable = alphaitem
```

ASSIGNMENT Examples

A user variable is determined to be numeric or alphanumeric at the first assignment. You cannot mix variables of different types in an assignment statement.

1. This example assigns an alphanumeric application name to a new user variable:

   ```
   myapp_name = other:app_name
   ```

2. This example is incorrect because it assigns a numeric value to a user variable that was previously defined as alphanumeric (in example 1):

   ```
   myapp_name = 14
   ```

3. This example assigns a numeric value to a new user variable:

   ```
   highest_cpu = gbl_cpu_total_util
   ```

4. This example is incorrect because it assigns an alphanumeric literal to a user variable that was previously defined as numeric (in example 3):

   ```
   highest_cpu = "Time is"
   ```

COMPOUND Statement

Use the COMPOUND statement with the IF Statement, the LOOP Statement, and the START, REPEAT, and END clauses of the ALARM Statement. By using a COMPOUND statement, a list of statements can be executed.

Syntax:

```
{
statement
statement
}
```

Construct compound statements by grouping a list of statements inside braces ({}). The compound statement can then be treated as a single statement within the syntax.

Compound statements cannot include ALARM and SYMPTOM statements. (Compound is a type of statement and not a keyword.)

COMPOUND Example

```
highest_cpu = highest_cpu
IF gbl_cpu_total_util > highest_cpu  THEN
  // Begin compound statement
  {
    highest_cpu = gbl_cpu_total_util
    PRINT "Our new high CPU value is ", highest_cpu, "%"
  }
  // End compound statement
```

In this example, `highest_cpu = highest_cpu` defines a variable called `highest_cpu`. The Adviser saves the `highest_cpu` value and notifies you only when that `highest_cpu` value is exceeded by a higher `highest_cpu` value.

In the example, if you replaced `highest_cpu = highest_cpu` with `highest_cpu = 0`, then the `highest_cpu` value would be reset to zero at each interval.

You would be notified at each interval what your `highest_cpu` value is.

EXEC Statement

Use the EXEC statement to execute a UNIX command from within your Adviser syntax. You could use the EXEC command, for example, if you wanted to send a mail message to the MIS staff each time a certain condition is met.

Syntax:

```
EXEC printlist
```

The resulting *printlist* is submitted to your operating system for execution.

Because the EXEC command you specify may execute once every update interval, be careful when using the EXEC statement with UNIX commands or scripts that have high overhead. For example, you would not want to rebuild the kernel inside a gpm EXEC statement.

EXEC Examples

In the following example, EXEC executes the UNIX `mailx` command at every interval.

```
EXEC "echo 'gpm mailed you a message' | mailx root"
```

In the following example, EXEC executes the UNIX `mailx` command only when the gbl_disk_util_peak metric exceeds 20.

```
IF gbl_disk_util_peak > 20 THEN
  EXEC "echo 'gpm detects high disk utilization' | mailx root"
```

GPM Statement

Use the GPM command to have selected GlancePlus windows display whenever conditions that you specify are met.

Syntax:

```
GPM -rpt reportlist
```

The *reportlist* contains the GlancePlus window names for the windows you want to display. In *reportlist*, the window names should be separated by commas. Refer to the Windows List for GlancePlus windows.

GPM Example

```
IF gbl_run_queue > 3 THEN
  GPM -rpt CpuGraph
```

IF Statement

Use the IF statement to test *conditions* you define in the Adviser syntax.

Syntax:

IF condition THEN statement [ELSE statement]

The IF statement tests the *condition*. If true, the *statement* after the THEN is executed. If the *condition* is false, then the action depends on the optional ELSE clause.

If an ELSE clause has been specified, the *statement* following it is executed. Otherwise, the IF statement does nothing. The *statement* can be a COMPOUND Statement which tells the Adviser to execute multiple statements.

IF Example

```
IF gbl_cpu_total_util > 90 THEN
  PRINT "The CPU is running hot at: ", gbl_cpu_total_util
ELSE IF gbl_cpu_total_util < 20 THEN
  PRINT "The CPU is idling at: ", gbl_cpu_total_util
```

In this example, the IF statement is checking the condition (gbl_cpu_total_util > 90). If the condition is true, then "The CPU is running hot at: " is displayed on `stdout` along with the % of CPU used.

If the (gbl_cpu_total_util > 90) condition is false, ELSE IF goes to the next line and checks the condition (gbl_cpu_total_util < 20). If that condition is true, then "The CPU is idling at: " is displayed on `stdout` along with the % of CPU used.

LOOP Statement

Use LOOP statements to find information about your system. For example, you can find the process that uses the highest percentage of CPU or the swap area that is being utilized most. You find this information with the LOOP statement and with corresponding statements that use metric names for the system conditions on which you are gathering information.

Syntax:

```
{APPLICATION, APP, CPU, DISK, DISK_DETAIL, FILESYSTEM, FS, FS_DETAIL,
LAN, LOGICALVOLUME, LV, LV_DETAIL, NETIF, NFS, NFS_BYSYS_OPS, NFS_OP,
PRM, PROCESS, PROC, PROCDCEIF, PROCDCEOP, PROC_FILE, PROC_REGION,
PROC_SYSCALL, SWAP, SYSTEMCALL, SC, THREAD, TRANSACTION, TT, TTBIN,
TT_CLIENT, TT_INSTANCE, TT_UDM, TT_RESOURCE, TT_INSTANCE_CLIENT,
TT_INSTANCE_UDM, TT_CLIENT_UDM}
LOOP statement
```

If you have a LOOP statement in your syntax for collecting specific data and there is no corresponding metric data on your system, the Adviser skips that LOOP and continues to the next syntax statement or instruction. For example, if you have defined a LOGICAL VOLUME LOOP, but have no logical volumes on your system, the Adviser skips that LOGICAL VOLUME LOOP and continues to the next syntax statement.

Loops that do not exist on your platform will generate a syntax error.

As LOOP statements iterate through each interval, the values for the metric used in the statement change. For instance, the following LOOP statement executes the PRINT Statement once for each active application on the system, causing the name of each application to be printed.

```
APP LOOP
  PRINT app_name
```

A LOOP can be nested within other syntax statements, but you can only nest up to five levels. The *statement* may be a COMPOUND Statement.

On a threaded operating system such as HP_UX 11.0, the Adviser supports a THREAD LOOP. A thread loop can be nested inside a process loop in order to examine each thread for a particular process. If you do reference a PROC_ metric inside a thread loop, it could return unexpected results (thread information).

A thread loop can also exist outside a process loop. In this case, it will examine all threads active on the system. You should not nest a process loop within a thread loop.

Because LOOP statements are initiated at each interval, use them with discretion due to possible performance implications. This caution is especially appropriate with regards to using nested LOOP statements.

APPLICATION LOOP Example

Use the APPLICATION LOOP statement to cycle through all active applications.

You can use global (gbl), table(tbl), or application (app) metrics with the APPLICATION LOOP.

The following example uses an Application LOOP to find the application with the highest CPU for an interval.

```
big_app = ""
highest_cpu = 0
APPLICATION LOOP
  IF (app_cpu_total_util > highest_cpu) THEN
  {
    highest_cpu = app_cpu_total_util
    big_app = app_name
  }
  IF (highest_cpu > 20) THEN
    YELLOW ALERT "The application ", big_app,
      " is the highest CPU user at", highest_cpu, "%"
```

After finding the application, the Adviser writes a message to the Alarm History window with the app_name and CPU value, if the CPU value is greater than 20.

CPU LOOP Example

Use the CPU LOOP to cycle through data about CPU use on your system. You can use global (gbl), table(tbl), or by CPU metrics with the CPU LOOP.

The following example prints the CPU usage percentage for each CPU on your system.

```
Print "----------", glb_stattime, "----------"

CPU LOOP

  PRINT "CPU # ", bycpu_id, " used ", bycpu_cpu_total_util, " % CPU"
```

On a system with two CPUs, the resulting output printed for two intervals is:

```
----------10:52:01----------
CPU #              0 used  0.6 % CPU
CPU #              1 used  3.4 % CPU
----------10:52:11----------
CPU #              0 used  0.4 % CPU
CPU #              1 used  2.3 % CPU
```

DISK LOOP Example

Use the DISK LOOP to loop through your configured disk devices. When you use this LOOP, the Adviser checks for specific disk information that appears in the IO by Disk window. You can use global (gbl), table(tbl) or by disk metrics with the DISK LOOP.

The following example prints the physical write rate for each disk on your system.

```
PRINT "---------------------", gbl_stattime, "------------------------"

DISK LOOP

  PRINT bydsk_devname, " write rate: ", bydsk_phys_write_rate
```

On a system with three disks, the resulting output printed for two intervals is:

```
---------------------11:00:23------------------------
/dev/hdisk0          write rate:    2.4
/dev/hdisk1          write rate:    0.0
/dev/cd0             write rate:    0.0

---------------------11:00:33------------------------
/dev/hdisk0          write rate:    0.0
/dev/hdisk1          write rate:    0.0
/dev/cd0             write rate:    0.0
```

FILE SYSTEM LOOP Example

The FILE SYSTEM LOOP is designed to loop through configured file systems and allow the Adviser to report on information accessible in the IO By File System Window. You can use global (gbl) , table (tbl) , or IO by file system (fs) metrics with the FILE SYSTEM LOOP.

The following example reports the space utilized for each file system device on a system with three devices.

```
PRINT "---------------------", gbl_stattime, "------------------------"

FS LOOP

  PRINT fs_devname, " is ", fs_space_util, "% full at ",
    fs_max_size," megabytes"
```

The resulting output for two intervals on a system with three file systems is:

```
---------------------11:11:28------------------------
/dev/hd4             is   77.9% full at       32 megabytes
/dev/hd2             is   94.9% full at      928 megabytes
/dev/hd9var          is   93.9% full at       56 megabytes

---------------------11:11:38------------------------
/dev/hd4             is   77.9% full at       32 megabytes
/dev/hd2             is   94.9% full at      928 megabytes
/dev/hd9var          is   93.6% full at       56 megabytes
```

NFS BY OPERATION LOOP Example

Use the NF_OP_LOOP to loop through NFS operations performed. When you use this LOOP, the Adviser checks for specific NFS operations that appear in the NFS By Operation window. You can use either global (gbl) , table (tbl) , or by operation metrics with the NFS_OP LOOP.

The following example prints the server and client operations performed:

```
PRINT "---------------------", gbl_stattime, "----------------------"

NFS_OP LOOP

  PRINT byop_server_count," server and ",byop_client_count,
    " client ",byop_name," operations performed"
```

On a system performing no activity as an NFS server but with users doing directory listing on another NFS server, the resulting output is:

```
--------------------14:55:41-----------------------

     0 server and      0 client null           operations performed
     0 server and      2 client getattr        operations performed
     0 server and      0 client setattr        operations performed
     0 server and      0 client root           operations performed
     0 server and    886 client lookup         operations performed
     0 server and    884 client readlink       operations performed
     0 server and      0 client read           operations performed
     0 server and      0 client writecache     operations performed
     0 server and      0 client write          operations performed
     0 server and      0 client create         operations performed
     0 server and      0 client remove         operations performed
     0 server and      0 client rename         operations performed
     0 server and      0 client link           operations performed
     0 server and      0 client symlink        operations performed
     0 server and      0 client mkdir          operations performed
     0 server and      0 client rmdir          operations performed
     0 server and     28 client readdir        operations performed
     0 server and      1 client statfs         operations performed
```

NETWORK INTERFACE LOOP Example

Use the NETWORK INTERFACE LOOP to loop through configured LAN devices and to report on information from the Network by Interface window. You can use global (gbl) ,table (tbl) , or by network interface (bynetif) metrics with the LAN LOOP.

```
# This version will only work with hp-ux 11.x.  If you want it to work
# for 10.20 then you'd need to remove the "BYNETIF_QUEUE," string below
# as that metric is only available from 11.x glance.

# The following string variable should be changed to the interface of
# interest.  For example:
#   netif_to_examine = "lan0"
# If you want to see all interfaces, leave it an empty string (""):
#   netif_to_examine = ""

# initialize variables:

headers_printed = headers_printed

netif loop {

  # print information for the selected interface or if null then all:

  if (BYNETIF_NAME == netif_to_examine) or

    (netif_to_examine == "") then

  {

# print headers the first time through the loop:
```

```
    if headers_printed == 0 then
    {
      print "Time    Interface  InPkts OutPkts  OutQ  Colls  Errs"
      print "   "
      headers_printed = 1
    }
# print one line per interface reported:
    print GBL_STATTIME, " ", BYNETIF_NAME|8,
      BYNETIF_IN_PACKET, BYNETIF_OUT_PACKET,
      BYNETIF_QUEUE, BYNETIF_COLLISION, BYNETIF_ERROR
    # (note that some interface types do not report collisions or
    # errors)
}
}
print "  "
```

The resulting output:

```
Time    Interface  InPkts OutPkts  OutQ  Colls  Errs

22:43:42 lan3         49      3     0      0      0
22:43:42 lan0          0      0     0      0      0
22:43:42 lan1          0      0     0      0      0
22:43:42 lan2          0      0     0      0      0
22:43:42 lo0           0      0     0      0      0

22:43:47 lan3        329      2     0      0      0
22:43:47 lan0          0      0     0      0      0
22:43:47 lan1          0      0     0      0      0
22:43:47 lan2          0      0     0      0      0
22:43:47 lo0           0      0     0      0      0
```

LOGICAL VOLUME Example

Use LOGICAL VOLUME loops to loop through your configured logical volumes. You can use either global (gbl), table (tbl) , or logical volume metrics with the LOGICAL VOLUME LOOP.

```
PRINT "---------------------", gbl_stattime,
  "-----------------------"

LV LOOP

  PRINT "Volume ", lv_dirname, " was read at a rate of ",
    lv_read_rate, " per second"
```

The resulting output for two intervals on a system with logical volumes is:

```
------------------------------11:46:50-------------------------------
Volume /dev/vg00            was read at a rate of    0.0 per second
Volume /dev/vg00/group      was read at a rate of    0.0 per second
Volume /dev/vg00/1vol3      was read at a rate of  314.3 per second

------------------------------11:47:00-------------------------------
Volume /dev/vg00            was read at a rate of    0.0 per second
Volume /dev/vg00/group      was read at a rate of    0.0 per second
Volume /dev/vg00/1vol3      was read at a rate of    70.6 per second
```

PRM LOOP Example

Use the PRM LOOP to cycle through information found in the PRM Group List Window. You can use global (gbl), table (tbl), or application metrics with the PRM LOOP.

The following PRM LOOP example checks for high run queue and any PRM groups exceeding their CPU entitlements.

```
IF gbl_run_queue > 3 THEN {

  print " "

  print "--- High run queue = ", gbl_run_queue, " at ", gbl_stattime,
    " ---"

  prm loop {

    if app_prm_state > 2 then

      if app_cpu_total_util > app_prm_cpu_entitlement then

        print "    Note PRM group ", app_name_prm_groupname,
          " exceeds entitlement."

  }
}
```

The output printed at each interval is:

```
--- High run queue =   3.4 at 15:53:29 ---
    Note PRM group Testing exceeds entitlement.
```

PRM_BYVG LOOP Example

Use the PRM_BYVG loop to loop through PRM groups for a volume group. (Note that PRM information is only available for volume groups that are specified in the PRM configuration file.) The PRM_BYVG loop must be nested within a LV loop. The following example displays disk resource usage statistics by PRM group.

```
PRM loop {
  disk_state = app_prm_disk_state
}

if disk_state == 0 then {
  print "  Disk manager state: Not Installed"
}

else if disk_state == 1 then {
  print "  Disk manager state: Reset"
}

else if disk_state == 2 then {
  print "  Disk manager state: Disabled"
}

else if disk_state == 3 then {

  print "  Disk manager state: Enabled"

  lv loop {

    IF lv_type == "G" THEN {

      print "  Volume Group: ", lv_dirname
      print "                          %        %          KB"
      print "PRM Group     PRMID entitled achieved   transferred"
      print "-------------------------------------------------"

      prm_byvg loop {

        print prm_byvg_prm_groupname|13, prm_byvg_prm_groupid|5,
          prm_byvg_group_entitlement|8, prm_byvg_group_util|8,
          prm_byvg_transfer

      }
      print "   "
    }
  }
}
```

The output at each interval is:

```
Disk manager state: Enabled
Volume Group: /dev/vg00
                  %        %        KB
PRM Group     PRMID entitled achieved transferred
-------------------------------------------------
PRM_SYS           0       0     100         8
OTHERS            1      50       0         0
tools             2      50       0         0
```

PROCESS LOOP Example

Use the PROCESS LOOP statement to cycle through all active processes. You can use either global (gbl) , table (tbl) , or process (proc) metrics with the PROCESS LOOP. The following example uses a PROCESS LOOP to find the process with the highest CPU for an interval.

```
big_proc_id = 0
big_proc_name = ""
big_proc_cpu = 0
PROCESS LOOP
IF proc_cpu_total_util > big_proc_cpu THEN {
  big_proc_cpu = proc_cpu_total_util
  big_proc_name = proc_proc_name
  big_proc_id = proc_proc_id
}
IF big_proc_cpu > 10 THEN
  YELLOW ALERT "Possible loop, process ", big_proc_name,
    " pid ", big_proc_id|6|0, " using ", big_proc_cpu, " % CPU"
```

SWAP LOOP Example

Use the SWAP LOOP to LOOP through the configured swap areas and allow the Adviser to report on information from the Swap Space Window. You can use table (tbl) or global (gbl) or by swap (byswp) metrics with the SWAP LOOP.

The following example reports on the swap space available on a system with two swap devices.

```
PRINT "---------------------", gbl_stattime, "------------------------"

SWAP LOOP

  PRINT BYSWP_SWAP_SPACE_NAME, " has ", BYSWP_SWAP_SPACE_USED,
    " used out of", BYSWP_SWAP_SPACE_AVAIL, " megabytes"
```

On a system with one swap area, the output printed for two intervals is:

```
---------------------15:31:59------------------------
/dev/hd6              has     37 used out of   128 megabytes

---------------------15:32:09------------------------
/dev/hd6              has     37 used out of   128 megabytes
```

SYSTEM CALL LOOP Example

Use the SYSTEM CALL LOOP to cycle through calls on your system. When you use the SYSTEM CALL LOOP, the Adviser checks for information available in the System Call window. You can use global (gbl), table (tbl) , or system call (syscall) metrics with the SYSTEM CALL LOOP.

The following example checks for a high system call rate, then prints the most frequent call.

```
IF gbl_syscall_rate > 6000 THEN {
  print " "
  print "--- High syscall rate = ", gbl_syscall_rate, " at ",
    gbl_stattime, " ---"
  highestrate = 0

  systemcall loop {

    IF syscall_call_rate > highestrate THEN {
      highestrate = syscall_call_rate
      highestcall = syscall_call_name
    }
  }
  print "    Most frequent syscall was ", highestcall, " at",
    highestrate, " per second"
}
```

The output is:

```
--- High syscall rate =   6750.6 at 15:50:27 ---

    Most frequent syscall was gettimeofday at 6632.90 per second
```

TT LOOP Example

Use the TT LOOP to loop through transaction information that has been recorded during the last interval. When you use this LOOP, the Adviser checks for specific transaction information that appears in the Transaction Tracking window. You can use global (GBL), table (TBL), or transaction tracking (TT) metrics with TT LOOP.

The following example prints the number of completed transactions and the average response time for each registered transaction name on your system.

```
PRINT "-------------------------", gbl_stattime,
  "-------------------------"

TT LOOP

  PRINT tt_name, " had ", tt_count, " transactions; ",
    "response time ", tt_wall_time_per_tran, " secs"
```

On a system with four transactions, the resulting output for two intervals is:

```
-------------------------13:24:44-------------------------

First_Transaction  had   1 transactions; response time   1.000355 secs
Second_Transaction had   1 transactions; response time   2.000221 secs
Third_Transaction  had   1 transactions; response time   3.000231 secs
Fourth_Transaction had   0 transactions; response time   0.000000 secs

-------------------------13:24:54-------------------------

First_Transaction  had   3 transactions; response time   1.000383 secs
Second_Transaction had   1 transactions; response time   2.000216 secs
Third_Transaction  had   0 transactions; response time   0.000000 secs
Fourth_Transaction had   0 transactions; response time   0.000000 secs
```

TTBIN LOOP Example

Use the TTBIN LOOP, which must be nested within a TT loop, to loop through the response time bins of each active transaction on your system. When you use this LOOP, the Adviser checks for specific transaction information that appears in the Transaction Graph window. You can use global (gbl), table (tbl) , transaction tracking, or transaction tracking bin metrics with the TTBIN LOOP.

The following example prints the response time bins for each transaction name which had any completed transactions during the interval.

```
PRINT "-------------------------", gbl_stattime,
  "-------------------------"

TT LOOP

  IF (tt_count > 0) THEN

  {

    print "Transaction ", tt_name, " had ", tt_count, " transactions"

    lower_bin_limit = 0

    TTBIN LOOP

    {

      IF (ttbin_trans_count > 0) THEN {

        print "  ", ttbin_trans_count, " were between ",
          lower_bin_limit, " and ", ttbin_upper_range, " seconds"

        lower_bin_limit = ttbin_upper_range

      }

    }

  }
```

On a system with four transactions, the output printed for two intervals is:

```
-------------------------13:46:31-------------------------
Transaction First_Transaction    had      4 transactions
       2 were between     1.00 and   2.000000 seconds
Transaction Second_Transaction   had      1 transactions
       1 were between     2.00 and   3.000000 seconds
Transaction Third_Transaction    had      1 transactions
       1 were between     3.00 and   5.000000 seconds

-------------------------13:46:41-------------------------
Transaction First_Transaction    had      3 transactions
       1 were between     1.00 and   2.000000 seconds
Transaction Second_Transaction   had      1 transactions
       1 were between     2.00 and   3.000000 seconds
Transaction Fourth_Transaction   had      1 transactions
       1 were between     3.00 and   5.000000 seconds
```

TT LOOP ARM Example

With ARM 2.0, the TT_CLIENT, TT_INSTANCE and TT_UDM loops can be nested within a TT LOOP. The TT_CLIENT loop lists the correlated transactions, the TT_INSTANCE loop lists up to 2048 transaction instances, and the TT_UDM loop lists user measurements for a given transaction. You can use global (gbl), table (tbl) or transaction tracking metrics with the TT LOOP.

Within a TT_CLIENT loop a user can nest a TT_CLIENT_UDM loop to display user measurements on a per correlator basis. A TT_INSTANCE_UDM loop, or TT_INSTANCE_CLIENT loop may be nested within the TT_INSTANCE loop to see correlators or user measurements specific to a given instance.

The examples below show how multiple loops can be used to look at user measurements for any given transaction instance.

Example 1: Look for SLO Violations

```
# The following example loops thru all transactions looking for SLO
# violations, then prints the UDM information for all instances:

print "----------------------------", GBL_STATTIME,
  "----------------------------"

tt loop {

  IF tt_slo_count > 0 THEN {

    print "  "
    print "SLO violation count:", tt_slo_count,
      " for transaction:", tt_name, " user:", tt_uname,
      " app:", tt_app_name, " threshold: ", tt_slo_threshold

    tt_instance loop {

      starttime = gbl_stattime - gbl_interval
      IF tt_instance_stop_time > starttime THEN {

        # found a completed instance in the transaction, print info:
        print "instance pid:", tt_instance_proc_id,
          " wall time:", tt_instance_wall_time

        tt_instance_udm loop {
          print "   ", tt_instance_user_measurement_name|44,
            " value= ", tt_instance_user_measurement_value
        }
      }
    }
  }
}
```

The following is the output for one interval:

```
----------------------------17:19:03----------------------------


SLO violation count:     1 for transaction:Client_tra00
user:gracel          app:Client_Appl0         threshold:   5.000000

instance pid: 12137 wall time: 13.0407


SLO violation count:     1 for transaction:Server_transaction
user:joe             app:Server_Application   threshold:   5.000000

instance pid: 12137 wall time: 13.0358
   Metric #1 - Type 1 is a COUNTER32            value=           32
   Metric #2 - Type 4 is a GAUGE32              value=           37
   Metric #3 - Type 2 is a COUNTER64            value=     19088743
   Metric #4 - Type 9 is a STRING8              value=     String 8
   Metric #5 - Type 3 is a COUNTER32/DIVISOR32  value=        2.000
   Metric #6 - Type 8 is a NUMERICID64          value=     19088434
   The last field is always a STRING32          value=            0
instance pid: 12137 wall time:  3.0291
   Metric #1 - Type 1 is a COUNTER32            value=           32
   Metric #2 - Type 4 is a GAUGE32              value=           37
   Metric #3 - Type 2 is a COUNTER64            value=     19088743
   Metric #4 - Type 9 is a STRING8              value=     String 8
   Metric #5 - Type 3 is a COUNTER32/DIVISOR32  value=       21.333
   Metric #6 - Type 8 is a NUMERICID64          value=     19088434
   The last field is always a STRING32          value=            0
instance pid: 12137 wall time:  3.0256
   Metric #1 - Type 1 is a COUNTER32            value=           32
   Metric #2 - Type 4 is a GAUGE32              value=           37
   Metric #3 - Type 2 is a COUNTER64            value=     19088743
   Metric #4 - Type 9 is a STRING8              value=     String 8
   Metric #5 - Type 3 is a COUNTER32/DIVISOR32  value=       21.333
   Metric #6 - Type 8 is a NUMERICID64          value=     19088434
   The last field is always a STRING32          value=            0
instance pid: 12137 wall time:  2.0201
   Metric #1 - Type 1 is a COUNTER32            value=           32
```

```
   Metric #2 - Type 4 is a GAUGE32              value=           37
   Metric #3 - Type 2 is a COUNTER64            value=     19088743
   Metric #4 - Type 9 is a STRING8              value=     String 8
   Metric #5 - Type 3 is a COUNTER32/DIVISOR32  value=       21.333
   Metric #6 - Type 8 is a NUMERICID64          value=     19088434
   The last field is always a STRING32          value=            0
instance pid: 12137 wall time:  1.0101
   Metric #1 - Type 1 is a COUNTER32            value=           32
   Metric #2 - Type 4 is a GAUGE32              value=           37
   Metric #3 - Type 2 is a COUNTER64            value=     19088743
   Metric #4 - Type 9 is a STRING8              value=     String 8
   Metric #5 - Type 3 is a COUNTER32/DIVISOR32  value=       21.333
   Metric #6 - Type 8 is a NUMERICID64          value=     19088434
   The last field is always a STRING32          value=            0
```

Example 2: ARM 2.0 syntax

```
# The following example prints info for all completed transactions
# during the interval.
print "--------------------------------", GBL_STATTIME,
  "------------------------------"
header_printed = 0
tt loop {
  tt_instance loop {
    starttime = GBL_STATTIME - GBL_INTERVAL
    IF TT_INSTANCE_STOP_TIME > starttime THEN {
      IF header_printed == 0 THEN {
        print "  "
        print "TranID  StartTime                StopTime",
          "                "
        header_printed = 1
      }
      print TT_TRAN_ID|6, "  ", TT_INSTANCE_START_TIME, "  ",
        TT_INSTANCE_STOP_TIME
      print "          TranName: ",TT_NAME|40
    }
  }
}
```

The following is the output for one interval:

```
-------------------------------17:21:24-------------------------------
TranID  StartTime                 StopTime
     3  Wed Jun  3 17:21:07 1998  Wed Jun  3 17:21:20 1998
          TranName: Client_tra00
     7  Wed Jun  3 17:21:07 1998  Wed Jun  3 17:21:20 1998
          TranName: Server_transaction
     7  Wed Jun  3 17:21:17 1998  Wed Jun  3 17:21:20 1998
          TranName: Server_transaction
     7  Wed Jun  3 17:21:17 1998  Wed Jun  3 17:21:20 1998
          TranName: Server_transaction
     7  Wed Jun  3 17:21:18 1998  Wed Jun  3 17:21:20 1998
          TranName: Server_transaction
     7  Wed Jun  3 17:21:19 1998  Wed Jun  3 17:21:20 1998
          TranName: Server_transaction
```

PRINT Statement

Use the PRINT statement to print to stdout data you are collecting. You may want to use the PRINT Statement to log metrics or calculated variables.

Syntax:

```
PRINT printlist
```

PRINT Example

```
PRINT "The Application OTHER has a total CPU of ",
  other:app_cpu_total_util, "%"
```

When executed, this statement prints a message to the window that initiated GlancePlus like the following:

```
The Application OTHER has a total CPU of 89%
```

SYMPTOM Statement

Syntax:

```
SYMPTOM variable [TYPE = {CPU, DISK, MEMORY, NETWORK}]
RULE measurement {>, <, >=, <=, ==, !=} value PROB probability
[RULE measurement {>, <, >=, <=, ==, !=} value PROB probability]
    .
    .
    .
```

The keywords SYMPTOM and RULE are exclusive for the SYMPTOM statement and cannot be used in other syntax statements. The SYMPTOM statement must be a top-level statement and cannot be nested within any other statement.

variable is a variable name which will be the name of this symptom, as well as a graph title in the Symptom History window. Variable names defined in the SYMPTOM statement can be used in other syntax statements, but the variable value should not be changed in those statements.

TYPE defines the type of symptom and connects the SYMPTOM information to the CPU, Disk, Memory, or Network button on the Main GlancePlus window. The symptom type can only be CPU, Disk, Memory, or Network. However, you can define more than one CPU, Disk, Memory, or Network

symptom. For example, if you have two TYPE = CPU symptoms, each with their own set of RULEs, then the symptom with the highest probability determines the color of the CPU button label.

RULE is an option of the SYMPTOM statement and cannot be used independently. You can use as many RULE options within the SYMPTOM statement as you need.

The SYMPTOM variable is evaluated according to the RULEs at each interval.

- *measurement* is the name of a variable or metric that is evaluated as part of the RULE
- *value* is a constant, variable, or metric that is compared to the *measurement*
- *probability* is a numeric constant, variable, or metric

The probabilities for each true SYMPTOM RULE are added together to create a SYMPTOM value. The SYMPTOM value then appears in bar graph form in the Symptom History window. The SYMPTOM value also appears in the Symptom Status window and the Symptom Snapshot window alphanumerically, if the SYMPTOM evaluates to yellow or red.

The sum of all probabilities where the condition between measurement and value is true is the probability that the symptom is occurring.

SYMPTOM Example

Syntax:

```
SYMPTOM CPU_Bottleneck TYPE=CPU
RULE gbl_cpu_total_util > 50  PROB 25
RULE gbl_cpu_total_util > 85  PROB 25
RULE gbl_cpu_total_util > 90  PROB 25
RULE gbl_run_queue      >  3  PROB 50

SYMPTOM CPU_Level TYPE=CPU
RULE gbl_cpu_sys_mode_util > 40  PROB 25
RULE gbl_cpu_sys_mode_util > 50  PROB 25
RULE gbl_cpu_sys_mode_util > 60  PROB 25
RULE gbl_cpu_sys_mode_util > 70  PROB 50
```

Whichever CPU symptom defined above has the highest total probability (PROB), is the symptom that determines the label color of the CPU button on the GlancePlus Main window.

SYMPTOM Example: Global CPU Bottleneck

```
SYMPTOM Symp_Global_Cpu_Bottleneck TYPE=CPU
RULE gbl_cpu_total_util > 50  PROB 25
RULE gbl_cpu_total_util > 85  PROB 25
RULE gbl_cpu_total_util > 90  PROB 25
RULE gbl_run_queue      > 3   PROB 75
```

The SYMPTOM statement establishes a new variable called Symp_Global_Cpu_Bottleneck. TYPE=CPU links the SYMPTOM to the CPU button on the Main window.

The new variable receives a probability every update interval which is computed by summing a value according to the RULES below the SYMPTOM statement.

If the computed probability is between 51 and 90, the CPU button letters on the Main window are turned to yellow for that interval.

- If the probability is 91 or more, then the CPU button letters are turned red.
- If the probability is 50 or less, the CPU button letters are turned to their normal color.

For example, if the CPU utilization (gbl_cpu_total_util) for the interval was 93% and the run queue was 2, then the first three rules would all be true so that 25 would be added to the probability three times. Since the fourth rule would not be true, 75 would NOT be added. Thus global_cpu_bottleneck variable would have a value of 75 (percent) that interval and the Main screen CPU button letters would be turned yellow (probability between 51 and 90).

If there were several RULES which pertain to CPU in the Adviser Syntax and any of them were to achieve a sufficient probability, the CPU button letters turn the appropriate color. If a RULE would cause the letters to turn yellow and another RULE would cause them to turn red, the highest probability (turning red) is reflected on the CPU button.