PIMOS Manual
(V 3.0)

— Volume Operation —

ICOT PIMOS Developing Group
November 1991

Institute for New Generation Computer Technology

INTRODUCTION

This manual details the use of the Operating System PIMOS, Version 3.0, which is for parallel inference machines. Because the expansion of PIMOS's functions has increased the amount of accompanying description required, the PIMOS manual has been divided into two volumes, the "Operation Volume" and the "Programming Volume". The Operating Volume details the operational methods for using PIMOS. As for the programming interface, please refer to the Programming Volume. Please note that the pages are numbered consecutively through both volumes, and the attached table of contents and index cover both volumes. PIMOS is written in the concurrent logic language KL1, and it is assumed that the user programs running under PIMOS are also coded in KL1. KL1 is based on Flat GHC and has various extensions. The Programming Volume contains the outlined specification of the extensions of KL1 to Flat GHC. For programming techniques in KL1, please refer to the KL1 programming textbook "KL1 Programming Introduction/Beginning/Intermediate" (the fourth laboratory of ICOT).

Copyright 1991 Institute for New Generation Compute Technology, Japan All RIGHTS RESERVED

Chapter 1

LOGIN

This chapter outlines the login procedures to allow users of SIMPOS to login to PIMOS for use from a local terminal. For information on how to start PIMOS see Chapter ?? (page ??).

1.1 SIMPOS Preparation of a User Environment for SIMPOS

Add the following line: to the system menu of the login file (called `me:login.com`)

```
{"Login PIMOS", pimfep##pimos_remote_login_requester}
```

1.2 Login Procedure

There are two types of login procedures based on the setting of the task environment file. (See Section 6.3.1 How to Set the Task Environment (Page 206).)

- Normal Login
- Login Using the Simple Login Function

Each procedure is described below.

1.2.1 Normal Login Procedure

This login procedure is used when the normal login function is started. There are several parameters that the user must input during login such as the login partition and other information.

1. Confirm that the following message showing that the login condition is available is displayed on the CSP.

   ```
   DC> **** xxx LOGIN SERVICE STARTS
   DC> ('INTo) 00 0000000X
   ```

   【Caution】 xxx is the type of LAN available for login, and X is the FEP number that can accept the login service.

2. Select "Login PIMOS" from the SIMPOS system menu.

3. The login request window will be displayed. Input the required items according to the steps below.

(a) Input the I/O node name of the FEP appropriate for accepting the login service. (Refer to Section 11.1, Acquisition of Resources (Page 399) in the Programming Volume.) If you login from SIMPOS on the FEP, type "local."

(b) Input the user name of PIMOS.

(c) Input the corresponding password.

4. PIMOS confirms the user name and the password. If they can not be confirmed, this fact will be displayed on the screen, and the login attempt has failed.

5. PIMOS displays the login condition at each login partition as follows: (Example)

```
Part:     Mode      :Node No.:  User name (LID)
================================================
0   :  available    :0-31    :
1   : not available :32-47   : daigo (100)
2   : (multi user)  :48-63   : masahiro (102)
```

The meaning of each item is as follows:

Part: Login Partition Number

Mode: Login possibility

available Login possible

not available Login impossible

`(multi user)` Multi user (Login always available)

Node No: Node range used

User name: User name if currently used.

LID: `User Name` Login ID corresponding to the User name

【Caution】 As for partitions that cannot be logged in by the FEP specified at login, the Mode becomes unavailable regardless of whether a user is currently logged in.

6. (a) If an available partition for login exists:

Taking the above example, the following question is displayed on the screen.

```
select your partition (0 2 exit kill) ?
```

Then input the login partition number ('0' or '2') to log in, type 'e' to exit using PIMOS, or type 'k' to interrupt the program during login. If 'k' is selected, type the partition number and the login ID as shown, to select the specific program.

```
enter your partition ?1↩
enter your LID ?2↩
```

(b) If no partition is available for login:

The following display appears on the screen.

```
Confirm(<return> or kill) :
```

To stop while using PIMOS press the ¡return¿ key. Otherwise type 'k' as described above.

7. After completing the partition selection process, the user generally starts the shell, and the shell window is displayed. See Section 6.4.2 (Page 209) for information on how to set the initial program for start-up.

1.2.1.1 User Environment Initialization File

The user utilizes the window that is first displayed by PIMOS to interactively perform the selection of the above-mentioned partition to be used. It is possible for users to set the information to customize this window in the user environment initialization file. The file name is `pimos.init` under the home directory on the same I/O node that requested login.

The customization information at the moment consists of the following parameter only:

```
console_window(Pos, Size, Font).
```

Pos, Size Pos, Size Same as the first and the second arguments of the protocol reshape/3 (Refer to the Section 9.3 Controlling Input/Output (Page 372) in the Programming Volume.)

Font String to show the font file.

1.2.2 Login Method Using the Simple Login Function

When using the simple login function, it is possible for two or more users to use all of the nodes.

1. `CSP`

   ```
   DC> **** xxx LOGIN STARTS
   DC> ( INTo) 00 0000000X
   ```

   Confirm that the following is displayed on the screen.

   【Caution】 xxx is the type of LAN available for LOGIN, and X is the FEP number which can accept the login service.

2. Next, select "Login PIMOS" from the SIMPOS system menu.

3. Since the login request window will be displayed, the required items should be input according to the steps below:

   (a) Input the I/O node name of the FEP appropriate for accepting the login service. (Refer to Section 11.1 Acquisition of Resource (Page 399) in the Programming Volume.) If login is performed from SIMPOS on the FEP, type local.

   (b) Input the user name of the PIMOS.

   (c) Input the corresponding password.

4. The success or failure of the login will be displayed on the screen and the login process will be completed.

5. When login has been successful, the shell is activated and normally the shell window is displayed on the screen. See Chapter ?? (Page ??) for information on how to set the activation.

1.2.3 Setting Environmental Variables

Set the following environmental variables at the time of login:

task:package

Package Name Atom.

task:directory

Initial Directory Name String.

task:home

Home Directory Name String.

task:io_node

I/O Node Name Requesting Login.

task:fep

FEP Number Accepting Login Request.

task:user

Login User Name String.

【Caution】 The package name is immediately after login is the name of the user who has logged in.

1.2.4 Changing the Password

The procedure to change the password is described below.

【Caution】 Make sure to always change the password at the initial login to the PIMOS.

1. Execute the change_password command from the shell.
2. The window for changing password is displayed on the screen. Set the new password according to the directions displayed.
3. Terminate the login shell. Upon doing so, the file will be rewritten.

【Caution】 Login should be performed by the authorized user (usually, the system manager) of the SIMPOS who has the right to rewrite the user-defined file (pimos:pimos.logins).

1.3 Message Related to Login

This chapter provides a description of the messages that may be output at the time of login. See Appendix A for information on how to establish the A.

1.3.1 Errors Displayed on CSP

1. Someone has erased a login information

   Cause The privileged user has erased a login information.

   Solution Nothing in particular.

2. Cannot make a remote device

   Cause It is impossible to make the device on the I/O node used by the user who has been logged-in.

   - The network on the I/O node used by the user has an illegal status as against the FEP.
   - The package "mpsicsp" of SIMPOS has a different status from that at the time of release.

   Solution Change the network or the package "mpsicsp" of SIMPOS to normal status.

1.3.2 Login Request Window / Errors Displayed on Windows for Selecting Login Partitions

1. Unknown FEP mode name

   Cause The specified node name of the FEP is illegal or unknown.

   Solution
   - Input the correct FEP node name.
   - Correct the network information (See A).

2. Network Error

   Cause An error has occurred while accessing the login server (internal device for processing login) through the network.

   - The address of the login server has not been registered to the network system.
   - The login server has not been activated on the specified FEP.
   - The FEP has not been booted.
   - It is impossible to access to the login server on the specified FEP because the network has not been correctly set.
   - It is impossible to access to the login sever on the specified FEP because of a network disorder.

   Solution
   - Check the status and the settings of the network.
   - Set it to enable acceptance of the login request from the remote by using the task environment file `pimos:pimos.conf`.

3. Receive time-out

   Cause Time-out has occurred while waiting for the message receipt from the login server of the PIMOS.

Solution It is possible that because the multiple PSI or the FEP has a high load, PIMOS can not cope with the request for login.

4. Login Incorrect

   Cause There is an error with the user name or password typed in.

   Solution Type in the correct user name or password.

5. Input Error!!

   Cause It may have occurred, for example, at the time of selecting the partition of login.

   - The login partition number in the parentheses displayed by the input prompt or a command other than exit/kill has been input.

6. Your ID has been erased

   Cause A privileged user has erased the user during the login process.

   Solution Request a privileged user to re-register the user.

7. Someone has logged on PIMOS

   Cause Login has been attempted in the login partition in the single user mode when another user is already logged in during the login process.

   Solution Ask the user who has already logged in to logout.

8. Your initial program code doesn't exist.

   Cause No program code of the user's initial login program exists.

   Solution Ask the privileged user to register the required program module in the module database.

Chapter 2

Shell

2.1 Shell

2.1.1 Outline

The shell is a user interface used to start a goal specified by the user and to manage the execution of such goal.

The functions of shells are to create a task and execute the goal.

The shell starts the goal designated by the user interactively via the shell window. At this time, the shell creates a new task, inserts the goal into the task, and executes it.

Standard I/O stream

When starting a user task with the shell, assign the three task devices (streams): **standard input**, **standard output**, and **standard message output**. The user can specify these three with the goal. If the specification is omitted, the streams provided in the shell (usually streams to the shell window) will be assigned.

In addition to the above-mentioned streams, a stream called the 'standard interaction' stream is also assigned to the task. It is linked to the shell window of "the shell that has a shell window and is the nearest to the user task in the user task's ancestor" (usually the shell starting the user task). That is, it is guaranteed that the standard interaction stream is linked to the shell window of one of the shells.

The above mentioned streams (**standard input**, **standard output**, and **standard interaction**) are realized by the standard input/output functions (Please refer to Chapter 9.4 Standard Input/Output (Page 394) in the Programming Volume.) The standard message output is a function provided by the shell independently.

Inter-task communication facility (pipe)

It may be desirable to use output from a certain task as input to another task to implement task concatenation. In that case, please concatenate the standard input and output using the inter-task communication facility (pipe) provided by the shell. This enables communication between the tasks.

The pipe provides a buffering function. When the buffer in the pipe becomes full, the task in the front stage of the pipe (producer) is made to wait for an appropriate period until the buffer becomes empty.

In the case where, despite the fact that the buffer is full, the producer sends data that is not expected to be used, all tasks that are contacted in the pipe are forced to completion. This is called the 'pipe broken' state.

Job

The job consists of a series of tasks (user goal group) concatenated with the pipe. The shell uses this job as the unit for managing user tasks that it started. Please note that they are not controlled individually by the shell.

"Control" here covers the functions to monitor the job execution status, to change the job states such as activation, stop, restart, and end, and to query the job states.

Sharing the shell window

The shell usually has a shell window, which is used for interaction with the user. This window is also used as the standard I/O of the task that the shell started without assigning a window other than files or the shell window for the standard I/O of it. It may thus be used competitively by several tasks. For this reason, the shell maintain exclusive control over the shell window between the shell itself and the started job.

Foreground and background

The shell offers two job execution environments, foreground and background, which are used for exclusive control over the shell window.

Several jobs can be entered in the background, whereas only one job can be entered in the foreground.

The shell and the job entered in the foreground use the shell window exclusively. Access to the shell window by the jobs in the background is continuously suppressed. If such a job sends an I/O message to the shell window, it cannot reach the window until the job has been entered in the foreground.

The background job can however display the message on the shell window as for the standard message output.

Shell commands

Several job management commands are available to move a background job to the foreground or to stop or restart a job. The mechanism that moves the foreground job to the background is implemented using the window's attention mechanism.

2.1.2 Method for Starting the Shell

2.1.2.1 Method for starting the Normal Shell

There are three methods for starting the shell. When one of these goals is executed, the shell window is automatically produced and displayed on the screen. If this is followed by the prompt display, the shell is then in a state to accept various commands.

In usual mode shell:go

The mouse is used to specify the position and the size of the window. The default font file of the SIMPOS is used.

Specifying the position and the size of the window shell:go(at(X, Y), char(W, H),
The specified position and the size of the window are used. The default font file is used.

Specifying the window position, size and font shell:go (at(X, Y), char(W, H), Font)
The position, size, and font of the window are specified.

The meanings of the arguments in the goals are shown below:

X (X coordinate) at the top left corner of the window in dot increments

Y (Y coordinate) at the top left corner of the window in dot increments.
at(mouse) makes it is possible to specify the position by using a mouse instead of specifying X and Y individually.

W Number of characters per line in the window

H Number of lines in the window.
char(mouse) makes it is possible to specify the size by using a mouse instead of specifying X and Y individually.

Font Path name of the font file used for the display (character string)

【Caution1】 If a specified position includes some error (such as a negative number or a type mismatch), the position is to be specified using the mouse. The same can applied to specify the size of the window. If the window from the specified position and size is out of the display screen, it will be created using the default position and size. If the specified font file is undefined, the default font will be used.

【Caution2】 The window is created on the remote specified by the environmental variable `task:io_node`. Please refer to Section ?? (Page ??) I/O Node in the Programming Volume as for information on the environmental variable `task:io_node`.

2.1.2.2 Method for Staring the Shell for Login

There are three methods for starting the shell as mentioned in the previous Section 2.1.2.1. When one of these goals has been executed, the shell window is automatically produced and displayed on the screen, followed by the prompt display. Subsequently, the shell can accept various commands.

In usual mode shell:login
The mouse is used to specify the position and the size of the window. The default font file is used.

Specifying the position and the size of the window shell:login(at(X, Y), char(W, H)
The specified position and the size of the window are used. The default font file is used.

Specifying the window position, size and font shell:login (at(X, Y), char(W, H), Font)
All of the position, size, and font of the window are specified.

【Caution】 The shell used for logging in is normally used as the goal that was to first started at the time of login to PIMOS by the user.

2.1.3 Basic Operations of Shells

2.1.3.1 Commands General Regulations

The shell window commands are classified into two groups: commands to start the user goal and commands to start the shell or jobs. According to the syntax rules described in the next Section 2.1.3.2(Page 10), each of the commands to start the user goal may appear more than once in one line. This is not allowed for commands to start the shell or jobs. No variables may be included in a command line.

2.1.3.2 Starting User Goal

In command lines, several user goals can be specified according to the following syntax rules. Each specified goal is executed in an independent task.

<command line>	::=	<job> ["&" [<command line>]]
<job>	::=	<task group> ["=>" <std_out>]
<task group>	::=	<task> ["<=" <std_in>] { "|" <task> }
<task>	::=	<command> ["->" <std_mes>]

- Specify devices <std_in>, <¡std_out>, and <std_mes> as follows:

  File: `file("file-path-name")`

  File: `file(^"file^path-name")`

  Window: `window("window-name")`

  Window: `window("window-name", at(S, Y), char(W, H))`

  Window: `window("window-name", at(S, Y), char(W, H), Font)`

  【Caution1】 If a ^ is added to the file path name, that file will be opened in the (append) mode, and if not, it will be opened in the output (write) mode. For more information on these modes, please refer to (Section 9.3.6.1 (Page 382) File Request Devices in Volume Programming. The window name is the title attached to the window, and not the text buffer name of the PMACS corresponding to the window. See Section 2.1.2.1 (Page 8) describing the Method for Starting Shells for the meanings of the other arguments.

- Specify <command> in the format of package::module:predicate(Arg, ...), similarly to the inter-module call in a program.

  The package name and '`::`' can be omitted. If they are omitted, the value of the environmental variable "task:package" is used. If `task:package` is not specified or data other than an atom is stored, the package name is to be '`work`'.

  【Example1】 If the command line has no argument

  ```
  > test:start
  ```

  【Example2】 If the command line contains arguments

  ```
  > test:start(arg1, arg2)
  ```

  '`:`' can be omitted. If they are omitted, go is to be assumed as the predicate name.

- Only one standard message output device can be specified in <task>. In the case where nothing is assigned to it, the use of the shell's standard message output is to be shared.

  【Example1】 Using the shell's standard message output

  ```
  > test:start
  ```

  【Example2】 Specifying the file as the standard message output

  ```
  > test:start -> file("mes.doc")
  ```

- When <task group> consists of two or more <task>'s, the standard I/O devices between them are linked using the pipe.

  【Example1】 Linking two tasks using the pipe

  ```
  > test1:start | test2:start
  ```

  【Example2】 With two tasks linked through the pipe, specify that the standard messages are output from the individual tasks.

  ```
  > test1:start | test2:start
  ```

- Only one standard input device and only one standard output device can be specified in <job>. In the case where nothing is specified, the shell window is to be used.

  【Example1】 Specifying all of the standard input, standard output, and standard message.

  ```
  > test:start -> file("mes.doc") <= window("in") => window("out")
  ```

  【Example2】 Specifying one standard input and one standard output for each job.

  ```
  > test1:start <= window("in") | test2:start => file("out.doc")
  ```

- If the <command line> consists of one <job>, this <job> is to be started in the foreground or the background depending on the specification of &.

  【Example1】 Entering <job> in the foreground

  ```
  > test:start
  ```

  【Example2】 Entering ¡job¿ in the background

  ```
  > test:start &
  ```

- If the <command line> consists of two or more <job>'s, they are concurrently executed. In that case, the last specified <job> is started in the foreground or background according to the specification. The others are all started in the background.

  【Example1】 Only `test3:start` is entered in the foreground. The others are all entered in the background.

```
> test1:start & test2:start & test3:start
```

【Example2】 All the jobs enter in the background.

```
> test1:start & test2:start & test3:start &
```

2.1.4 Commands to Shells and Jobs

The shell provides the following commands to give instructions to started jobs of other shells.

2.1.4.1 Commands for Job Management

stop JobNo

Stop the job specified by "JobNo."

fore JobNo

Resume the execution of the job specified by "JobNo" in the foreground. Please note that the execution is to be resumed after the previously stated resource amount is added, if that job is stopped due to a shortage of the resource. For details, please see Section 2.1.7 Handling Errors in Resource Shortage (Page 23). For information on resources (computer resources) please refer to Section 7.5 Shoen Function in the Programming Volume (Page 302).

back JobNo

Resume the execution of the job specified by JobNo in the background. Note that the execution is to be resumed after the previously stated resource amount is added if that job was stopped due to a shortage of the resource. For details, please see Section 2.1.7 Handling Errors in Resource Shortage (Page 23). For resources (computer resources) please refer to section 7.5 Shoen Function in the Programming Volume (page 302).

kill JobNo

Force the completion of the job specified by "JobNo". If all are specified as "JobNo", all the jobs currently executed will be brought to completion.

status

Displays information about all jobs started by the shell. The content of the displayed information consists of the following three types. JobNo (integer), Execution State of job, and name of job (Name of User Goal) The following are the states of the executing jobs.

running Under execution

stopped Currently stopped. To restart the execution, it is necessary to execute the command fore or back as mentioned in this section.

rsc_low Currently stopped due to a shortage of the resource. To restart the execution, it is necessary to execute the command fore or back as mentioned in this section.

terminate The process of termination has been started. To be terminated soon.

zombi Under the process of termination now. To be terminated soon.

abnormal The job has fallen in to an abnormal state. This status cannot occur normally.

If "JobNo" is omitted from a format including the specification of "JobNo," the "JobNo" value of the last started job at that time will be regarded as the "JobNo."

2.1.4.2 Attention Keys

To move the active job from the foreground to the background press the Attention key (usually control-C). In such case, the job is to be stopped at the same time the shift to the background is completed.)

2.1.4.3 Commands to Shells

exit

Terminate the shell. In such cases, all the jobs currently started will be brought to completion.

suspend

In calling up a subsidiary shell (see the next Section 2.1.4.4 (Page 13) from within another shell (hereafter called the parent shell), temporarily stop such subsidiary shell temporarily and return control to the parent shell.

get

Display the currently set values of all the special variables called shell variables provided to store the parameters necessary for the operation of the shell.

help

Display the list of commands.

2.1.4.4 Commands for Starting Child Shells

shell

shell(at(X, Y), char(W, H))

shell(at(X, Y), char(W, H), Font)

Each of these creates a new shell window and then starts another shell. See the previous Section 2.1.2.1 Method for Starting a Normal Shell (Page 8) for the meaning of the arguments.

sh

Sh Enable the shell to call up another (subsidiary shell) in a recursively. In this case, a new shell window is not created.

2.1.4.5 Commands to Operate the Command History

history

history N

Display the list of the command lines stored in the current history stack. In the case where the numeric N is specified, the information for the specified number of lines (N) is displayed on the screen. If the specified number of lines is more than those stored, all the lines will be

displayed. Please note that all lines are also displayed when a negative number or data other than an integer is specified.

Event Number

The serial number (called event number) will invariably be assigned to the command stored in the history stack. It is possible to retrieve and execute the corresponding command line by specifying the event number. Note that the 0 is specified, the immediately prior command will be executed.

get hl

Display the length of the history stack.

set hl = Length

Set the length of the history stack. Length is a non-negative integer or an infinite. If the Length is 0, the command storage to the history stack will not be performed. The default value in starting the shell is 0.

!!

!! Character String

Retrieve the newest line in the commands stored in the history stack and display it to the command line as the already input character string. If the length of the history stack is 0, an error will result. It is also possible to describe the character string continuously after `!!`. In that case, concatenate the character string after expanding `!!`, and display it on the command line as the already input character string.

【Caution】 The character strings here are not enclosed in double quotation marks.

!Character String 1

!Character String 1 Character String 2

Search for the line whose starting part of the command coincides with character string 1 from among the command lines stored in the history stack, and display it on the command line as the already input character string. If two or more of such lines exist, the newest one is selected. And if there is no command line corresponding to that condition, an error will result. Also, it is possible to describe a blank line or a character string 2 that succeeds the character string 1 at one time. In those cases, concatenate character string 2 after extending character string 1, and display it on the command line as the already input character string.

【Caution】 Character string 1 and character string 2 mentioned here are not enclosed in double quotation marks.

!Event number

!Event numbercharacter string

Search for the line specified by the event number from among the command lines stored in the history stack, and display it on the command line as the already input character string. If two or more lines concerned exist, it will be a blank line. It is also possible to indicate a blank line or

a character string succeeding the event number. In such cases, concatenate the character string after expansion has occurred according to the event number, and display it on the command line as the already input character string. If the event number is 0, the immediately prior command will be regarded as specified.

【Caution】 The character strings mentioned here are not enclosed in double quotation marks.

2.1.4.6 Commands to Operate the Prompt

set prompt = Prompt

Change the prompt of the command input line. The prompt is specified by the character string or the atom. If `\!` is described in the character string, the event number is displayed at that location.

get prompt

Display the prompt currently set.

2.1.4.7 Commands to Operate the Current Directory

cd Directory

Set the value of the environmental variable "task:directory." To change it, confirm the existence of the specified directory. If the specified directory does not exist, the value of the environmental variable will not be changed.
The directory is specified by using > as a delimiter, and the combination of ~, .., and the sub-directory name atom. ~ means the value of the environmental variable "task:home" at that moment. Though ~ is used in the case where no value is set to "task:home," it is dealt with by assuming that the specified directory does not exist. And .. means the relative parent directory.
If > is used at the beginning, it is regarded as an absolute specification. In other cases, it is recognized as a relative specification from the directory specified by "task:directory."
It is possible to specify the directory using a character string. However, ~ is not expanded in this case. In the case where the directory is a character string and is started by >, it is regarded as the absolute specification. If the directory is a character string and is not started by >, it is regarded as an absolute specification of the current value of "task:directory."

【Caution1】 In specifying a directory name atom starting with a capital letter or one containing a period, it is necessary to enclose that atom in quotation marks (').

【Caution2】 If the value of "task:directory" is illegal at the time the relative specification is performed, that specification will naturally be invalid.

cd

This is the same as in the case where cd ~ is executed.

pushd Directory

pushd N

pushd

Set the value of the environmental variable "task:directory." To change it, confirm the existence of the specified directory. If the specified directory does not exist, the value of the environmental variable will not be changed.
The directory can be specified in the same manner used for the specification of cd command.
If the value of the environmental variable has been changed, the directory name is stored at the beginning of the directory stack.
If a numeric value (positive integer) is specified instead of the directory name, the stack is rotated to make the Nth element becomes the beginning, and the directory name become the beginning of "task:directory." Note that the stack begins from 0.
If the argument is omitted, the first element of the stack and the second element are exchanged.

popd

Extract the directory that was stored last from the directory stack, and store that value in the environmental variable "task:directory." To change it, confirm the existence of the specified directory. If the specified directory does not exist, the value of the environmental variable will not be changed.
If the directory stack is empty, an error results.

dirs

Display the list of the contents if the directory stack.

pwd

Output the environmental variable task:directory to the shell's standard output destination (normally a window).

2.1.4.8 Commands to Operate Print parameter

setset printing parameter name = value

Change the printing parameter used in displaying the message which the shell outputs. There are four kinds of printing parameters which can be specified: pd, pl, pal, and psl. They correspond to the printing parameter name of the input/output buffer, print_depth, print_length, print_atomlength, and print_string_length respectively. The value should be non-negative integers. The default values at the start of the shell are 20, 20, 80, and 80 respectively.

【Caution】 The default value is scheduled to be changed by later adjustment.

get printing parameter name

Inquire as to the printing parameter used for displaying the message to be output by the shell, and output the value to the shell's standard output destination (normally a window). There are four kinds of printing parameters which can be specified; pd, pl, pal, and psl. They are corresponding to the printing parameter name of the input/output buffer, print_depth, print_length, print_atomlength, and print_string_length respectively.

2.1.4.9 Commands to Operate Command Alias

alias atom = character string

As for the commands to be input later, if the token located in the starting part of the command is identical to the above- mentioned atom, replace that token with the above mentioned character string and execute the command. If two or more commands are contained in one command line, the replacement will occur for each of them. However, replacement across the multiple steps is not performed.

【Caution1】 An item containing any character other than alphanumerics should not be specified as an atom. If an atom is so specified, the above replacement cannot be guaranteed to be correctly performed.

【Caution2】 The character string may contain all the characters which can be described on the command line. Please note that a syntax error occurs because the character "!" is not processed even though the starting character of the character string is "!".

【Caution3】 In specifying the shell command name as an atom, it is necessary to enclose that atom with the parentheses. However, please note that alias and unalias should not be specified.

unalias atom

Delete the command alias corresponding to the specified atom.

alias atom

Display the command alias corresponding to the specified atom.

alias

Display the list of the command aliases currently defined.

2.1.4.10 Commands to Execute Command Line Sequentially

do [command line, command line, ...]

Execute, in order, a series of command lines described on the list of the command lines. Each command line should be a character string. If data other than the character string is described, that data is ignored.

【Caution1】 Attention is effective with respect to individual command lines specified by the command line list. However, it is impossible to cancel the do command itself at the moment. Therefore, if any command line remains at the time the attention occurs, the remaining command line(s) is(are) interpreted and executed.

【Caution2】 This command is very much likely to be revised in the future.

2.1.5 Messages Displayed by Shells

There are two kinds of messages displayed by the shell: information messages output to report various kinds of information to users, and the error messages output to report the occurrence of errors to users. Some of the error messages may be followed by the display of the incidental

messages to notify the detailed information of the error. Each message is output immediately upon the occurrence of the event causing the message display. These three kinds of messages are respectively described below.

2.1.5.1 Information Messages

`Resource-increment in job` *JobNo* `is` *Inc* `.`
Display the increased value of the resource at the resource shortage of the job of "*JobNo*."

`The value of` *Key* `is` *Value* `.`
Display the value of the environmental variable or shell variable.

`Non existent value` *Key* `.`
Although there has been an attempt to display the variable value, the environmental variable or the shell variable specified does not exist.

`No environment variable .`
Although the display of all environmental variables have been attempted to be displayed, no environmental variable exists.

`Resource-increment` *Inc* `added into job` *JobNo* `.`
The predefined unit value for the resources has been added because of the resource shortage.

`Alias :` *Atom* `=>` *String*
Display the command alias. The alias of the command "*Atom*" is "*String*."

`Un-alias :` *Atom*
The command alias of the command "*Atom*" has been deleted.

`Environment variable removed` *Name*
The variable "*Name*" has been deleted.

LineNo `:` *CommandLine*
Display the contents of the history stack. For the command line with the line number "*LineNo*", "*CommandLine*" is displayed.

JobNo `-->` *Status JobName*
Display the state of the job. The name of the job for "*JobNo*" is "*JobName*", and the current state of it is "Status."

Status: Execution state of job.

running Under execution.

stopped Currently stopped.

rsc_low Currently stopped by the shortage of resources.

terminate The process of termination has commenced.

zombi Proceeding with termination.

abnormal The job has fallen into an abnormal state.

2.1.5.2 Error Messages

`Syntax error !!`
There is a syntax error in the input command line.

【Caution】 The protocol "`getwt/1`" of the input/output buffer is used for reading the command of the shell. If a syntax error is found when reading this command, the error message is similarly displayed by using the protocol `print_error/1` of the input/output buffer. Therefore, the error message in this case is displayed in the format specified by the input/output buffer. (Refer to Chapter 9.2 Input/Output utility (Page 355) in the Programming Volume.)

`Syntax error !!`

`( Variable(s) found in Command-line . )`
There is a syntax error in the input command line. (The variable name is contained.)

`Job` *JobNo* `was aborted .`
The job with the identifier "*JobNo*" has terminated abnormally.

`Unable to open take-file` *FileName*
The specified command file does not exist.

`Can't create the file-device for` *FileName* `.`
It is impossible to make the file request device which is necessary to open the file specified by "*FileName*."

`Illegal Alias-data,` *Atom* `or` *String* `!!`
An error has been found in the command alias name.

`No alias existed for` *Atom* `!!`
The specified alias of the command does not exist.

String `can't found in history .`
The command line starting with the character string "*String*" does not exist in the history stack.

`Unable to change directory` *Directory*
The transfer of the current directory to "Directory" has failed.

`Empty directory-stack .`
The directory stack is empty.

`Unable to exchange directory .`
To operate the directory stack has failed.

`Non existent job-number` *JobNo*
The job with the job number *JobNo* does not exist.

`Illegal variable name` *Name*
The variable name of the shell variable is not correct.

`Illegal value` *Value* `of` *Name*
The value which was attempted to be stored in the shell variable *Name* was incorrect.

`Can't remove the environment variable` *Name*
It is impossible to delete the variable "*Name*."

`Can't connect to the node` *IoNode*
Connection with the remote node *IoNode* has failed.

`Resource-low occured in job` *JobNo* `.`
A shortage of resources has occurred.

`Deadlock detected !!`
A deadlock has occurred.

`Merger-deadlock detected !!`
A merger deadlock has occurred.

`Reduction failure occured !!`
User-defined predicate has failed in reduction.

`Unification failure occured !!`
Unification has failed.

`Illegal input occured !!`
An error has been found in the input argument of the built-in predicate.

`Range overflow occured !!`
The input argument exceeds the built-in predicate.

`Out of bounds occured !!`
The built-in predicate input arguments included a value exceeding the range.

`Integer overflow occured !!`
Overflow has occurred during integer operation.

`Integer zero division occured !!`
Zero division has occurred during integer operation.

`Invalid floating-point operation occured !!`
An error has occurred during floating-point operation.

`Arity mismatch occured !!`
An invalid number of the argument has been found during a call to a user-defined predicate.

`Firmware error occured (reserved_7) !!`
A firmware error has occurred.

`Undefined Module Detected !!`
There was an attempt to call up a undefined module among the activated code.

`Undefined Predicate Detected !!`
There was an attempt to call up an undefined predicate among the activated code.

`Illegal merger input occured !!`
Invalid data has been found in the merger input argument.

`Trace exception detected !!`
The shell has detected a trace exception.

`Spy exception detected !!`
The shell has detected a spy exception.

`Firmware error occured (reserved_14) !!`
A firmware error has occurred.

`Special exception detected !!`
A special exception has occurred.

【Caution】 This message is never displayed because the current firmware version does not support it.

`Illegal situation occured !!`
A failure has occurred during job execution.

`Unknown exception detected !!`
An exception that the shell does not know has occurred.

`Error reported !!`
An error has been reported by the activated job.

2.1.5.3 Incidental Messages

`Job-number` : *JobNo*
Display the job number.

`NODE-number` : *Node*
Display the node number.

`PE-number` : *PE*
Display the processor number inside the task.

`Predicate` : *Pred*
Display the predicate which has caused the exception.

`Undefined` : *UndefPred*
Display the undefined predicate.

`Builtin` : *Bltn*
Display the built-in predicate which has caused the exception.

`Position` : *Pos*
Display the position of the argument which has caused the exception.

`Called by` : *Caller*
Display the predicate that called up the built-in predicate which has caused the exception.

`Collected by` : *Collector*
Display the goal which collected the deadlock goal, if it is detected.

`Type` : *Type*
Display the type of the abnormal state such as a deadlock. See Section 4.1.5.3 for the detailed information on the types of deadlock.

- **[0]Copying GC** A deadlock has been detected during the execution of Copying CG.
- **[1]Copying GC (in Tracing)** A deadlock of the tracing goal has been detected during the execution of "Copying GC."
- **[2]Copying GC (in Spying)** A deadlock of the spy goal has been detected during the execution of "Copying GC."

- **[10]MRB-GC (Collect Value)** A deadlock has been detected in performing "Collect Value."
- **[11]MRB-GC (Suspend)** A deadlock has been detected in performing "Suspend." 検出した.
- **[12]MRB-GC (Active Unify)** A deadlock has been detected in performing "Active Unify."
- **[13]MRB-GC (Suspend)** A deadlock has been detected in performing "Suspend."
- **[14]MRB-GC (Active Unify)** A deadlock has been detected in performing Active Unify
- **[15]MRB-GC (Release)** A deadlock has been detected in performing "Release."
- **raised(Node, PE, Type)** An exception unknown to the shell has occurred.
- **illegal_filename(Direction, FileName)** An error has been found in the specification method of the file.
- **illegal_window_parameter(Direction, WindowParameter)** An error has been found in the specification method of window.
- **illegal_device_name(Direction, DeviceName)** There has been an attempt to specify a device which does not exist.
- **cannot_create_file_request(Type, File)** The file request device has not been created.
- **cannot_create_file(Type, File)** The file has not been opened.
- **cannot_create_window_request(Window)** The window request device has not been created.
- **cannot_create_window(Window)** The window has not been created.
- **illegal_goal_name** An error has been found in the goal name tried to be activated.
- **undefined_code** The code attempted to be activated does not exist.
- **pipeBroken** A pipe broken has been detected.

【Caution】 There is a very high possibility that the kinds of Type will be revised in the future.

`term1` : *Term1*

`term2` : *Term2*
Display the term which has failed in unification.

`Sub Code` : *SubCode*
Display the auxiliary code of the exception.

【Caution】 The detailed specifications are not yet determined.

`Error ID` : *ErrID*
Display the identifier of the error. In case an illegal message is sent to the PIMOS, for example, "illegal_message_to_PIMOS" is displayed on the screen. And if an "illegal message" is sent to the standard input/output stream, "illegal_message_to_Std_IO" is displayed.

`Error Info` : *ErrInfo*
Display the error information.

`Error Data` : *ErrData*
Display detailed information concerning the error.

2.1.6 Handling Errors

When the user goal fails, such as a failure in processing, or an occurrence of an exception, the shell forcefully terminates the entire job containing the task as well as the task itself.
In that case, some error message is to be displayed on the screen. The details of the displayed error message can be investigated by using the following commands.

last_error

Redisplay the error message last displayed on the screen. This is used to reconfirm the immediately prior error message, after changing the settings of printing parameter.

inspect

Start the "inspector" to investigate the error by using the data of the error message last displayed. This is used to investigate the more detailed content of the error message (see Chapter 4.2 Inspector (Page 164).

2.1.7 Handling Errors in Resource Shortage

When the resource (computer resource) runs short during the job execution, the message "resource_low" is output.

At this time, the shell stops the job, and asks the user whether to continue executing. Refer to Section 7.5, Shoen Function (Page 302) in the Programming Volume as for the resource (computer resource).

If the resource is added to continue the job, merely restart the job using the job management command (fore or back). In that case, add the predefined unit value regardless of whether fore or back is used, before restarting the job.

The amount of the resource to be added can be changed and inquired of using one of the following commands:

set rscinc = Inc

Set to Inc, the unit value of resource to be added when resource shortage occurs in the operation of subsequently started jobs. Inc is a non-negative integer equal to or less than 10^9, or a vector representation High, Low, or an atom infinite.

The ranges of the High and Low values are the same as Inc. If infinite is specified in Inc, the predefined value of resource (10^9) is added repeatedly and automatically. This means that an apparently limitless resource amount is assigned. The default is infinite.

【Caution】 The default value has been changed from that of Version 2.5.

get rscinc

Inquire about the predefined resource amount to be added. The information is output to the shell's standard output destination (usually the window).

set rscinc(JobNo) = Inc

As for the specified job, change to Inc, the unit value of resource to be added when resource shortage occurs.

Inc is a non-negative integer equal to or less than 10^9, or a vector representation High, Low, or an atom infinite. The ranges of the High and Low values are the same as Inc. If infinite is specified in Inc, the predefined value of resource (10^9) is added repeatedly and automatically. This means that an apparently limitless resource amount is assigned.

get rscinc(JobNo)

Inquire as to the incremental amount of resources that have been established for the job. The information is output to the shell's standard output destination (usually the window).

2.1.8 Specifying Nodes to Execute Jobs

set node = NODE

Specify the node that will execute subsequently activated jobs. The top level goal of every task under the job starts with the node that has a number specified with non-negative integer "NODE." The default is the node that started the shell.

get node

Once the current job starts, the top level goal of each task under the job will be executed by a node. The node number can be inquired about using this command. It is output to the shell's standard output destination (usually a window).

【Caution】 The mechanism of specifying a node is very much likely to be revised in the future.

2.1.9 Specifying Priority Range to Execute Jobs

set minprio = Min

set maxprio = Max

Specify the lower and upper bounds of the priority range for each task being under subsequently activated jobs. Relative specification applies to both Min and Max with respect to the priority range assigned to the shell itself. The following condition must be satisfied: $0 \leq \text{Min} \leq \text{Max} \leq 4000$. If an attempt is made to set a value which is not within the above stated range, it is impossible to be accepted, and a command error will occur. The default values are 0 and 4000, respectively.

get minprio

get maxprio

Inquire about the lower and upper bounds of the priority range for each task being carried out under subsequently started jobs. The values are output to the shell's standard output destination (usually window).

【Caution】 The mechanism of specifying the priority is very much likely to be revised in the future.

2.1.10 Operation of Environmental Variables

setenv Key = Value

Value in the environmental variable pool is something that the task of the shell itself has. "Key" is used as the key.

unsetenv Key

Delete the environmental variable(s) using "key" as the key, from the environmental variable pool that the task of the shell itself has.

getenv Key

Read the value(s) using "key" as the key from the environment variable pool that the task of the shell itself has. They are then displayed on the shell's standard output.

getenv

Read all values from the environmental variable pool that the task of the shell itself has. They are then displayed on the shell's standard output.

2.1.11 Environmental Variables Used by Shells

There are the following environmental variables used by the shell. (See Section 11.1.7 (Page 421).)

task:directory

Specify the default directory. Set the directory name (string).

task:home

Specify the home directory. Set the directory name (string).

task:package

Specify the default package. The value is a package name (atom).

task:io_node

Specify the name of the ?? performing input/output. For more information on the environmental variable "`task:il_node`", refer to Section ?? (Page ??) in the Programming Volume.

2.1.12 Command Files

The shell command is usually input from the shell window. The input process can be temporarily switched to a file to enable the command string written in the file to be executed.
When take command is executed, all the command lines from the file are to be read at first, and stored in the command buffer of the shell. Then the command line is retrieved one by one, and interpretation process is to be executed.

takecommand-file-name

Input the shell commands from the file specified in the command file name, then execute them. The command file name can be described in the same manner as used for specifying the directory for the cd command (see Section 2.1.4.7 (Page (15)).
If no file exists, the search is tried once again with the file name extension ".com" added.
It is possible to describe the take command in the command file.
Each command line in the command file is interpreted and executed in order.

【Caution1】 While the command file is read, attention keys can not be accepted.

【Caution2】 The attention is valid for each command line during the time the command is fetched from the command buffer and the interpretation process is executed. However, it is impossible to cancel the "take" command itself at the moment. Therefore, if any command line remains at the time the attention is started, the remaining command lines are interpreted.

set echo = MODE

Specify whether or not the command lines are to be echoed back during the execution of the command file. MODE should be Atom. If MODE is off, the echo back is not performed. In other cases, echo back is performed and the default value in starting shell is off.

get echo

Output to the standard output destination of the shell (normally window) whether or not the command lines are to be echoed back during the execution of the command file.

2.1.13 Initialization Files

The initialization file contains commands executed automatically when the shell starts. Its file name is fixed to "shell.com." When the shell starts, this file is searched according to the values of the environmental variables "task:node" and "task:directory." If the file is not found, searching the file is to be retried according to the values of the environmental variables "task:io_node" and "task:home". In the case where no file can be found even by retrial searches, no steps are to be taken. As for information on how to describe a file, see previous Section 2.1.12, "Command Files (Page 25)".

2.1.14 Login Files

The first command file automatically ex'cuted in starting the shell for login (pimos:shell:login), and its file name is fixed to "pimos.com." When the shell starts, this file is searched according to the values of the environmental variables "task:node" and "task:directory." If the file is not found, searching the file is to be retried according to the values of the environmental variable "task:io_node" and "task:home." In the case where no file can be found even by the retrial of searching, no steps are to be taken. As for the information on how to describe the file, see previous Section 2.1.12 Command Files (Page 25).

The shell for login reads the initialization file described above. The execution is performed in the following order: first "shell.com" is read, and pimos.com is read later. Note that the reading of "pimos.com" is performed regardless of whether "shell.com" exists or not.

2.1.15 Example of Login Files and Initialization Files

In the case where the login file and the initialization file as indicated in the above examples are prepared, and the login shell is activated under the environment described hereafter, the following series of initial settings can be automatically performed in addition to the environmental settings of the login shell itself.

1. Set the environment of the login shell.
2. Start the subsidiary shell from the login shell.
3. Set the environment of the subsidiary shell activated.
4. Start the compiler from the subsidiary shell activated.
5. Start the listener from the login shell.

Environment at login (In starting login shell)

- Environmental Variables

  task:directory = ">sys>user>User name"

  task:home = ">sys>user>User name"

- Directory Structure

  `>sys>user>User-name` Home directory

  `>sys>user>User-name>src` Directory the source file is stored

- Content of pimos.com under the home directory (specify the initial starting task.)

```
cd ~>src
shell(at(700, 3), char(50, 10), ">sys>font>test_11") &
cd ~
listener( at(300, 610), char(65, 12), ">sys>font>test_11") &
```

- Contents of "shell.com" under the home directory. (Describe the environmental settings common to all shells.)

```
set prompt = "Shell<<\!>> "
set minprio = 0
set maxprio = 4000
set rscinc = infinite
set node = 0
set hl = 50
set pd = 50
set pl = 50
set pal = 100
set psl = 100
alias h = "history"
alias printenv = "getenv"
alias fg = "fore"
alias bg = "back"
alias jobs = "status"
```

- Contents of "shell.com" under the directory src. (Describe the environmental settings of each shell.)

```
take ~>shell
compile
```

2.2 Shell Utility Programs

The shell utility program was designed to be started from the shell.

The respective command names correspond to the module names. As predicates, they are omitted from the formats below. They do ot need to be included in commands during input.

These modules have no relation to the main ppogram of the shell. They are exactly identical with the ordinary user-defined modules. For this reason, deleting or overwriting one or more of these modules has no influence on the main program of the shell.

2.2.1 Starting GC

Garbage Collection (GC) is a function to check the unused memory area (garbage) and collect it as a free space (collection) when the free area of the memory becomes insufficient because of the reduction of the goal. Two types of GC methods, **MRB-GC** and **Copying-GC** are currently provided by Multi-PSI. As all the processes required for operating **MRB-GC** are performed automatically, users do not necessarily to pay attention to the manipulation of it. **Copying-GC** is commenced if the free area of the memory is lacking, and may also be forcefully performed at user's option. The following commands start this Copying-GC regardless of the amount of free space.

【Hint】 Some types of deadlocks can not be found until `Copying-GC` is performed. If this type of deadlock occurs with the goal started by the user, it may seem as if the shell has hung up. In that case, press the attention key to move the job containing the deadlocked goal to the background and start the GC. Then all the deadlock goals which can be detected can be searched.

【Caution】 The CG method currently provided by the Multi-PSI system is local collection in the memory by node. As for now, the method to globally collect garbage to the Multi-PSI system in whole is not available.

gc

Instruct the nodes to initiate GC. Note that the instruction is presented last of all as to the node in which this program itself is activated.

gc(NODE)

Instruct the node to initiate GC. The instruction is presented for the nodes that have the node numbers specified in the argument NODE. An exception is the time when the atom self has been specified. In that case, check the node started by this program and instruct that node to initiate GC. If an error is found with the node specification such as the specified node does not exist, it is to be regarded as an error instead of performing GC.

seqgc

Instruct all nodes to initiate GC. The instruction is presented in the ascending order of the node numbers. Note that the instruction is presented last of all as to the node in which this program itself is activated.

seqgc(NODE)

Instruct the nodes to initiate GC. The instruction is presented for the nodes that have the node number assigned to the argument NODE. Exception is the time when the atom self has been specified. In that case, check the node started by this program and instruct that node to initiate GC. This command executes exactly the same function as gc/1 mentioned above. If an error is found with the node specification such as the specified node does not exist, it is to be regarded as an error instead of performing GC.

2.2.2 Display of Status

2.2.2.1 Displaying Resource Tree

The tasks and I/O devices offered by the PIMOS are managed under a tree structure called the resource tree. The following commands display its status.

One resource is always displayed in one line. Each line consists of three items: a resource identifier for uniquely identifying all the PIMOS resources, a resource type showing the type of the resource, and the name of the resource. And the indentation of the line indicates the hierarchy in the resource tree.

When the display of all the parts of the specified resource tree is completed, the inquiry sign "**OK?**" is displayed on the screen. The program is to be ended if the return key is pressed at this stage. Note that any key which is input before the return key is pressed is to be ignored except for the attention key.

rs

Examine the resource tree about the subsidiary resources of the shell that activated the program, then display its contents.

rs(Id)

Examine the resource tree about the subsidiary resources of the resource specified by the argument Id, then display its contents. Refer to general_request_stream protocol for specifying the argument Id.

rsall

Examine the resource tree about all the PIMOS-managed resources. then display their contents.

2.2.2.2 Displaying the Task Tree

The following commands display only the task information related to the resource tree.

One task should be always displayed on one line. Each line consists of five items: a task identifier for the purpose of uniquely identifying all of the PIMOS tasks, task name, execution state of task, upper limit value of resource consumption, and a resource consumption. And the indentation of the line shows its hierarchy in the resource tree.

The task name is the one set at the start of activating task. The five execution conditions of the task are described below:

running Under execution

resource_low Stopped due to a resource shortage

stop(Reason) Stopped by the stop request from the control stream or the parent task. If it has been stopped by the stop request from the control stream, "Reason" is the reason to stop specified in the message stopped/2 from the report stream, while in the case where the stopping state has been from the exclusive task, the parent is the reason for stopping. Please note that there is not always just one reason for stopping.

terminating In the process of terminating the task

shut_terminating Termination processing state by forceful termination of parent task

The upper limit value of resource consumption and the resource consumption are numeric data of 64 bits, and are shown by a vector with two elements. The first element of the vector corresponds to the upper 32 bits, and the second one to the lower 32 bits. Note that the numeric data is simply shown instead of treated as the vector in the case where the upper is 0, and "unknown" is shown when the value is not known. When the display of all the parts of the specified resource tree is completed, the inquiry sign "OK?" is displayed on the screen. The program will end if the return key is pressed at this stage. Note that any key which is input before the return key is pressed is to be ignored except for the attention key.

ts

Examine the resource tree with respect to the subsidiary task of the shell that started the program, then display its contents.

ts(Id)

Examine the resource tree about the subtasks of the task specified by the argument Id, then display its contents. Refer to general_request_stream protocol for specifying the argument Id.

tsall

Examine the resource tree concerning all the PIMOS-managed tasks, then display its contents.

2.2.3 Termination of PIMOS

The following command terminates the whole of the PIMOS:

shutdown

Terminate the PIMOS.

2.2.4 User Management

2.2.4.1 Displaying Login User

who

Display the user currently logging in. The information on the following items is displayed on the screen: user name, login partition currently used by that user, day and time logged in, node name of FEP or I/O node, user name on I/O node, and type of local area network.

See Chapter 1 Login (Page 1) in this Volume Operation as for the more details on login partitions.

2.2.4.2 Migration to Privileged Users

Become a privileged user. Similar to the UNIX command su.

Refer to Section 11.4.1 of the Programming Volume (Page 446) as for the details for privileged users.

su

Become a privileged user. Use this shell command to display the window for inputting the password. Input the password of the superuser (UID:0). If the password typed is wrong, "Sorry" is displayed on the window of the shell.

2.2.4.3 Changing Passwords

Change the password. Note the password change performed here is to be reflected in the file (pimos.logins) at the termination of the program started at the first(usually shell for login) after login(see Chapter ??.).

change_password

Change the password of the user name. The window to input the password is displayed on the screen by this command. Input the user name and the password.

Note that only the password of the user who is now logging in can be changed, and that those of the other users cannot be changed.

2.2.5 Copy

The following commands copy the data between standard I/O devices.

cat

Read the data line by line from the standard input, and output it to the standard output. If a null line is input, it is regarded as the end of reading and the program is to be terminated.

tee

Read the data line by line from the standard input, and output it to the standard output and standard message output. If a null line is input, it is regarded as the end of reading and the program is to be terminated.

2.2.6 Displaying Method for Using Modules

The following commands display the usage of various modules.

usage

Display the usage of usage command itself on the standard output.

usage(ModuleName)

usage(PackageName::ModuleName)

usage(PackageName, ModuleName)

Display the usage of the module specified in "PackageName" (atom) and "ModuleName" (atom). In the case where PackageName is omitted, the search is to be performed assuming that the target package is the one with the package name stored in task:package. Actually, the usage/1 predicate of the module that has the module name specified by the argument is executed. And the list of two-element vectors, which consist of the format character strings returned to the argument, and the argument list, is displayed by using the protocol putf/2 of the I/O buffer (refer to Section 9.2.1 (Page 355) in the Programming Volume). Therefore, the event is regarded as an error if the specified module does not contain the predicate usage/q or the value returned as the argument is invalid.

【Example】 In case of Shell

```
usage(Usage) :- true |
    Usage = [{string#"~s~n~s~n~s~n~s~n~s~n", [S1, S2, S3, S4, S5]}],
    S1 = string#"Usage : shell",
    S2 = string#"Usage : shell(at(X, Y), char(W, H))",
    S3 = string#"Usage : shell(X, Y, W, H)",
    S4 = string#"Usage : shell(at(X, Y), char(W, H), Font)",
    S5 = string#"Usage : shell(X, Y, W, H, Font)".
```

【Caution】 Scarcely any module provided by the PIMOS defines the predicate usage at the moment.

2.2.7 Commands to Character Stings and Text Files

2.2.7.1 Echo

The following command echoes back the value specified as the argument.

echo(X)

Output the contents of the argument X to the standard output.

2.2.7.2 Retrieval of Character Strings

The following commands retrieve a character string.

grep(Pattern)

Read the data line by line from the standard input. If a line contains the character string specified in the argument Pattern, the contents of a line is to be output to the standard output. When the input message "bf getl" returns "**end_of_file**" to the status, it is judged at the end of input. And it never reaches the **end_of_file** state if the window is specified to the standard input. Therefore, the program should be forcefully terminated by using the shell command **kill** after being interrupted by the attention key.

grep(PathnameWithWildcard, Pattern)

Read the data line by line from a file whose name matches the path name containing the wild card assigned to the argument PathnameWithWildcard (character string). If a line contains the character string assigned to the argument Pattern, the contents of the line are output to the standard output. The wild card may be contained in the directory.

【Caution】 Unlike the command name, this command currently does not support the retrieval processing of the regular expression. It may be revised in the future.

2.2.7.3 Counting Number of Lines

The following command counts the number of lines.

lc

Read the data line by line from the standard input, then output the number of lines to the standard output. If a null line is input, it is regarded as the end of the reading process and the program is terminated.

2.2.8 Displaying Directories

The following commands display the contents of the directory.

ls

List the files of the current directory.

ls(PathnameWithWildcard)

List the files whose names match the file path name containing the wild card assigned to the argument PathnameWithWildcard (character string). The wild card may be contained in the directory.

2.2.9 Checker of Variables

The module varchk is a shell program to check the specified KL1 source program and point out the symptom of bugs and the program which is not desirable from the viewpoint of efficiency.

Specifically, the following two facilities are equipped with the current version:

- Display of the variable which appears only once in a clause.
- Display of the variable which makes the MRB black.

The following kinds of state called "mode" selects the above mentioned functions.

one : Display the variable which appears only once in a clause.

mrb : Display the variable which makes the MRB black.

all : Display the variable indicated by both one and mrb modes.

Also, it is possible to select the following output formats.

short : Output the corresponding clause in one line.

long : Output the corresponding clause as line feed/indent.

varchk

Activate the checker of interactive variables. After it is activated, the checker investigates the program by indicating the target source program and option.

【Example】 Example of Starting varchk

```
*** KL1 Variable Checker ***
VARCHK[o,1] me:demo1, me:demo2
      :
VARCHK[o,1] !exit
```

Hereupon,

- `VARCHK[o,1]`
  Prompt. `[o,1]`displays the current mode and the output format (to be described later).
- `me:demo1, me:demo2`
  User input (file name). The following input is acceptable in this case.

  File name : Input the file name to be checked. Multiple files are permitted to be specified, and either a comma or a blank should be used as the delimiter of them. As shown in this example, the file name should not be enclosed in double quotation marks or quotation marks (‘"’, or ‘’’). The extension .kl1 may be omitted. And the file name can be specified by a wild card such as *kl1. In that case, the extension (usually ‘.kill’) should not be omitted. (That is because the name only with ‘*’ matches the file name without an extension.)

  !exit : Exit

  . : Switch to one mode.

  .. : Switch to mrb mode.

  ... : Switch to all mode.

  / : Switch to short output format.

  // : Switch to long output format.

  !! : Check the group of files checked by the last command again.

  ? : Help

  !help : Help

  !cd : Change the working directory. The directory name specified here should not be enclosed in double quotation marks or quotation marks (‘’’ or ‘"’).

  !pwd : Display the working directory.

varchk(FileNameList, Mode, Form)

Activate the batch version variable checker.

FileNameList: The file name string or its list.extension. .ko1 can be omitted.

Mode: Mode

one, o : Display the variable which appears only once in a clause.

mrb, m : Display the variable which makes the MRB black.

all, a : Display the variable indicated by both one and mrb modes.

Form: Output format

short, s : Output the corresponding clause in one line.

long, l : Output the corresponding clause as line feed/indent.

Chapter 3

Compiler

This chapter explains the KL1 compiler and how to operate the functions related to it.

【Hint】 The arguments described in list format can contain two or more elements, such as FileNameList and ModuleNameList, as mentioned in this chapter. When only one element is specified, the list format is not necessarily applied. For example, in specifying a single file name as FileNameList, there is no difference in meaning between the expressions "[FileName]" and "FileName".

3.1 Self-Compiler

The self-compiler is a shell program which mainly compiles, assembles, and links the KL1 program. Since it uses a standard I/O, self-complier must be started from the PIMOS shell or listener. The source program is read from the file specified. If all the processes for compiling are successfully completed, the self-complier automatically generates a module, which is registered in the module database. There are two ways to start the self-compiler, namely through batch compile and interactive compile.

compile

Starts the interactive compile method.

compile(FileNameList)

Starts the batch compile. The argument FileNameList specifies the file to be compiled by using a list containing the file name(s) (string(s)) as its element(s).

【Example1】 Example of starting batch compile (No.1)

```
SHELL> compile(["me:testprog1.kl1", "me:testprog2.kl1"])
```

【Example2】 Example of starting batch compile (No.2)
Assigning the destination for standard output to the file which enables the log data for compiling to be recorded in the file.

```
SHELL> compile(["me:test1", "me:test2"]) => file("me:comp.log")
```

【Example3】 Example of starting interactive compile

```
SHELL> compile
```

【Example4】

```
SHELL> compile(["me:demo1", "me:demo2"])
```

【Hint】 The execution results for the above example are equivalent to the results for executing the interactive compiling shown below.

```
SHELL> compile
COMPILE> me:demo1,me:demo2
              :
COMPILE> !exit
SHELL>
```

3.1.1 Description of Basic Items

- Initialization of Compiler

  The compiler has various parameters. The following two methods are available for initialization of these parameters. 1. By the environment variables, 2. By the parameter file of the compiler.

  When it is started, the searches the file "compile.param" from the current directory and from the home directory in order, reads the content of the appropriate file and sets the parameters if the file can be found. In the case where the corresponding file can not be found, the various parameters are set according to the values set for the environment variables.

  The rule for this parameter setting is common for batch compiling and interactive compiling.

  【Caution】 The content of the parameter file for PIMOS Version 3 has been a slightly changed from that of Version 2.5. Therefore, the alarm message has been designed to appear on the screen if the parameter file for the old version is read. Note that a conversion from Version 2.5 to 3.0 is not performed automatically.

  【Hint】 The parameter file can be generated by the `!keep` command for the interactive compiling (see page 49).

  【Hint】 When the parameter file is read, the full path name of such file is displayed on the screen.

- Estimation of causal relation between the file name extension and the file

  The name of the file specified to be compiled is automatically interpreted with a file name extension added. The extensions `mac` (macro definition file), `kl1` (ordinary kl1 source file), `pkl1` (kl1 source file without macro), and `sav` (unloaded file) are provided in this order as defaults and can be changed by the `!default_extension` command (see Page 41). The order for the file name extensions is regarded as a causal relation between files with the same name. The latter stated file is generated after the former one. If the final time update sequence of the files is reversed, the reversed file is compiled.

- Load distribution

  The `!distribute` command see Page 45) enables the load distribution of the compiler to be specified by a file. The compiler successively assigns the number specified by the !distribute `!parallel` command (see Page 46) as the maximum number of files for the node (processor). If the number of files to be compiled exceeds [the number of nodes x maximum number of

files], the files exceeding the limit will be assigned to the node which was compiled after the completion of compiling.

Therefore, in general, it is not guaranteed that compiling will be completed in the order of files specified by the user. The various display messages for compiling are in almost the same order as the completed compiling.

【Hint】 As for a simulated (pseudo) system, since, in general, process changes occur more frequently in performing the load distribution, there are delays in processing.

- Display message

  The format for the display messages output by the compiler of the PIMOS Version 3.0 is very much different from that of older versions. Each display message denotes the name and the location of the source file denoted at the top.

  【Example】 Display of Syntax Error

  ```
  !ERROR![test.kl1@1:pimos.h@3] *** Syntax Error ***
            :
  ```

  The above message indicates, through the phrase located in the second item (zero origin) of the test.kl1 file, that a syntax error has been detected in the fourth item of the file named pimos.h.

  The unit of the location depends on the parser which has read the source file. For example, the number of the terms is the unit for the standard parser of the KL1 (kl1_standard_preprocessor).

  【Hint】 The purpose of this mechanism is to easily relate the location which caused the message to the source file even if the preprocessor is used.

- Preprocessor

  The preprocessor function is equipped with a self-compiler in order to make the development of a compiler in a higher language based on KL1 easier. A higher language is finally converted into the KL1 through several language conversion filters, and is then compiled by the KL1 compiler. The developer of a higher language may only develop the following parts only.

  A preprocessor to convert the parser of a certain language and the language itself. An unparser to make a fair copy of the language, and a verifier to statically check the program (Refer to the Chapter D, "How to Produce Preprocessor" (Page 457) in Programming Volume for information on the definition method for the preprocessor.)

  In addition, a self-compiler functions such as a path, wild card, and a load distribution in specifying the files which can be utilized as they are, and the operation methods which can be unified. Further, the conversion results are not to be stored in the file, it can be expected that the total time required for compiling can be reduced.

  【Caution】 The verifier function has been prepared for this Version 3.0.

3.1.2 Batch Compiling

Through batch compiling, the file specified as the argument is compiled, assembled, and linked. The model generated is then registered.

If an undefined external reference is detected in linking, an error occurs and the compilation process is terminated without registering a module. Therefore, in the case where the compilation is likely to fail, such as under debugging steps, the interactive compiling described below is recommended. Batch compiling is convenient for the use of command files in a shell.

3.1.3 Interactive Compiling

In interactive compiling, the name of the file to be compiled and various commands are input from the standard input destination of the shell, and a message about the compiling error is output to the standard output destination or the destination of the standard message output.

It is possible to batch and store the input commands of the interactive compilation, and to specify them as the destination of the standard input at the time the compiler starts. The same operation can be done by using the `!take` command (See Page 50).

【Example】 A case of specifying a file to standard.

```
SHELL> compile <= file("compile.com")
```

3.1.3.1 Command Syntax

In the event that a line starts with an "!" mark, it is regarded as a compile command. The command name is the character string following "!" until the delimiter character (space, comma (,), or line feed). Capital letters are distinguished from small letters. The token after the command name is the argument of the command. The type of the argument is appropriately interpreted and converted to a character string, atom, or compound item by the location of the command and the argument. If an optional item is expected as an argument, it must be enclosed in double quotation marks if it is a character string to make it evident.

In the case where a delimiter is contained in an argument, it may be enclosed in "!"marks. If you wish to use an "!" mark between these "!" marks, you should repeat the "!" marks twice.

The other syntax rules are listed below.

- Continuation Line

  The line feed inside the "!" marks or the line with a comma (,) at its end.

- Specifying File Name

  It is possible to specify `"~>pimos", "..>.."`, and so on.

- Specifying Module

  In specifying the module of the default package, only the module name is to be indicated. If a package other than the default is specified, it should be explicitly indicated such as `Package::Module`.

【Caution】 The command name must not be enclosed in (double) quotation marks.

【Example】 The following commands are all equivalent.

```
COMPILE> !preprocess abc def package::module

COMPILE> !preprocess "abc", def !package :: module!

COMPILE> !preprocess abc,
         def  !package ::
         module!
```

3.1.3.2 Order of Link

In retrieving an external reference, the module(s) already compiled is (are) searched, and the module database is thereafter retrieved. If an undefined module is detected, the unresolved module(s) will be retained in the compiling process and regarded as an outstanding module(s). When this unresolved external reference is resolved by subsequent compilation, the external reference for that unresolved module is solved and registered as a module.

If the outstanding module becomes unnecessary and input a period (.) and return to discard it.

3.1.3.3 Termination of Interactive Compiling

The interactive compilation process is terminated by inputting "!exit". If any outstanding module exists, it is to be discarded.

【Example1】 Executing interactive compile

```
SHELL> compile
** KL1 Compiler **
COMPILE> me:demo1.kl1
              :       % Compiles information about the modules and
      % predicates is output.
              :
COMPILE> me:demo2.kl1
COMPILE> !exit
SHELL>
```

【Example2】 Executing interactive compile

```
SHELL> compile
** KL1 Compiler **
COMPILE> me:demo1,me:demo2
              :
COMPILE> !exit
SHELL>
```

【Caution】 The above two examples will result in the same state in the following cases: (a) there is no reference relationship between the module in the file demo1 and the one in the file demo2; or (b) the module in the demo2 references the one in the demo1. Otherwise, no compile results are guaranteed, if the specification stated in the above Example 1 is input.

3.1.3.4 Commands for Interactive Compiling

The command starting "!" is used for the interactive compiling. Refer to the following description.

- Commands regarding compilation

  !compile *FileName*, ..

  Compiles the file specified by FileName

  【Caution】 This is a new command additionally provided by PIMOS Version 3.0.

!klb *FileName, ..*

Compiles the file specified by FileName, and output the assemble command string to be produced as a result of compilation to the file with the same file name and the extension "klb".

【Caution1】 This command is prepared for tuning to realize high-speed performance and for debugging the compiler, but not for the use of general users.

【Caution2】 No module is to be generated as a results of the execution of this command.

【Caution3】 This is a new command additionally provided by PIMOS Version 3.0.

!make

Search and re-compile the file modified after the file change day/time in compiling the source file of the module in the package specified by working_package (command or environment variable) as for the module reference relation tree with the module specified by **top_modules** (command or environment variable) at the top of it. Specifying **auto_link** (command or environmental variable) is valid and re-linkage is performed automatically.

【Caution1】 This command is valid only for the module with the time stamp information. The module loaded by the IPL is not to be the object of a make command.

【Caution2】 The time stamp of the module is a new function included in PIMOS Version 3.0. Therefore, the make function is not valid for the module compiled by PIMOS Version 2.5 or an older version.

【Caution3】 The specification of path is invalid for searching the corresponding file. In the case in which the compiled source file is renamed or moved to a different directory after compilation, the file name needs to be explicitly specified for compilation.

【Caution4】 This is a new command additionally provided by PIMOS Version 3.0.

- Commands regarding specification of file

!!

Compiles the file specified immediately before.

!cd [*Directory*]

Regards the directory specified by Directory as the currentdirectory. It is possible to specify using "..>.." or "~>pimos". If the argument is omitted, the home directory is regarded as the current directory.

!default_extension [*Extension, ..*]

As for the file name which will be the object of compiling, specifies the file name extension to be defaulted by Extension. If the argument is omitted, the set file extension will be displayed on the screen. The order of the specified file name extensions is considered to

have a causal relation with the file of the same name. The latter one is regarded as the newer. If this time sequential order is reversed, the reversed file is compiled (or becomes the object of **!translate, !verify** command).

The extensions `mac` (macro definition file), `kl1` (kl1 source file), `pkl1` (kl1 source file without macro), and `sav` (unloaded file) are provided as defaults in this order and can be changed by the !default_extension command.

【Caution】 A period (.) to separate a file name from an extension is not necessary for specifying the file extension.

【Hint】 If the file with the same mac, kl1 pkl1, and sav exists and the mac file is older than the kl1 and the kl1 file is newer than the pkl1 file, the kl1 file will be compiled.

!ls [*Pattern*, ..]

Output each listing of the directories specified by path. The pattern indicates the specification of a wild card. If the argument is omitted, "*.*" is considered to be specified.

!path [*PathName*, ..]

Specifies the search path of the source file by PathName. If the file specified by the compilation does not exist in the current directory, or the wild card is specified, the directory specified by this path will be searched.

【Hint】 Through the above described function, the following usage can be applied for managing files. In managing files with different versions or managing the source file for a large file by using multiple directories, those directories can be specified as this variable.

!pwd

Displays the current directory.

- Commands regarding linkage

!auto_link (*on* | *off*)

Specifies whether an automatic linkage function is to be operated. The default is set to on (same as the former versions). In order to turn off the automatic linkage operation, the `!link` command should be explicitly input.

【Hint】 In the past, the steps for linkage were always taken at the end of executing the commands for the generating module (such as compile, relink, and load). Such a method left open the possibility that linking or relinking would be performed even when it was not necessary, and that memory might be wasted.

【Caution】 This command relink is valid only for the module loaded in the complete state after the linkage is finished. If the automatic linkage is turned off, relinking is invalid for the incomplete module which is maintained in the compiler as the unresolved module. Relinkage in the automatic linking off condition means to newly generate a module with an external module reference for the object module set as undefined.

!link

Explicitly specify to link in the automatic linkage function off condition (by auto_link command or environment variable). If automatic relinkage is specified (by a **top_modules** command or environmental variable), the linked module is relinked. This command is meaningless in the automatic linkage on condition.

!link_error (*warning* | *error*)

Provides the actions to be taken when a link error occurs (with the undefined module or undefined predicate. The warning message is output when an error occurs and linkage is not performed by error, and linkage is performed forcibly by warning.

【Caution】 If the undefined external module or the module containing the undefined external predicate is executed, an exception will occur in executing such a part.

【Hint】 As for the module containing the undefined reference, the undefined reference can be released by re-linkage or loaded if the undefined module has been generated.

!original_package (*on* | *off*)

The switch to specify whether the module generated by the re-linker or the loader is contained in the package. on: To the same module where the original module exists. off: To the default package. The default setting is off.

!package *Package*

Regards the package specified by Package as the default package.

【Caution】 When `working_package` is specified, the package to be searched by re-linkage and other means is different from the package of the module if the default package is changed by this command.

!top_modules [*ModuleName*, ..]

When compiling or loading is completed normally, the module specified by Module Name is regarded as the highest module, and re-linkage is automatically performed. If this setting is not made, re-linkage will not be performed. Omitting the argument makes this variable void.

- Commands regarding the preprocessor

!parser [*InLang* [*ParserModuleName*]]

Specifies the module for syntax analysis (ParserModuleName) corresponding to the input language specified by InLang (extension of the file). If the module name is omitted, the set module name is displayed. And if all the arguments are omitted, all the syntax analysis modules which have been set are to be displayed.

!preprocess [*InLang OutLang* [*PreprocModuleName*, ..]]

Specifies the preprocessor module (PreprocModuleName) to convert the input language (extension of file) specified by InLang to the output language (OutLang). By this specification, the stated preprocessor is automatically assigned at the moment of compiling rather than at the time of the extension of the file. If two or more preprocessors are specified for one language, they are assigned in order from the left (as a pipeline processing).

It is necessary that `pkl1` without containing the macro is specified as the output language in order to finally compile the language to the `KL1` language. If the output language can not be compiled to `pkl1` by a specification at one time, the specification for other preprocessor is chained and compilation is attempted until pkl1 is specified as the output language.

However, this chain of the preprocessor is not infinite, but is performed once. In the case where only the input language is specified, the setting of the preprocessor corresponding to it is released. And when no argument is stated, all the set preprocessors are to be displayed on the screen.

【Example】 Chain of Preprocessors

```
!preprocess abc def pp1 pp2
!preprocess ghi jkl pp3 pp4
!preprocess jkl pkl1 pp5
```

If the above lines are specified, the preprocessors are to be chained automatically from pp1 → pp2 → pp3 → pp4 → pp5.

As a default, the macro compiler is assigned to "mac", the macro expansion to "kl1", and the loader is "sav" specially. It is impossible to specify or change the preprocessor corresponding to "sav". Since the input language is normalized as a name written in small letters, classification by capital and small letters is not available.

【Caution1】 The command `!attach` is provided by PIMOS Version 2.5. It has been changed to this `!preprocess` command in Version 3.0. The `!attach` command is regarded as the command with the output language specified to `pkl1` in the `!process` command.

【Caution2】 This is a new command additionally provided by PIMOS Version 3.0. It is possible to use the `!attach` command for the time being.

!translate *FileName*

Translates the file specified by FileName to a file with a preprocessor output. The file to be generated will have the same name and an extension corresponding to the output language name of the preprocessor last specified.

【Caution】 No module is to be generated as the result of executing this command.

!unparser [*OutLang* [*UnparserModuleName*]]

Specifies the module for fair copy (UnparserModuleName) corresponding to the output language specified by OutLang. If the value is omitted, the module name which has been set is displayed on the screen.

- Commands regarding verifier

!verifier [*InLang* [*VerifierModuleName*]]

Assigns the verifier module specified by VerifierModuleName to the language specified by InLang. If the module specification is omitted, the verifier for the corresponding language is removed. If all the arguments are omitted, the verifier already set is displayed on the screen.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0.

!verify *FileName*, ..

Verifies the file specified by FileName. The verifier corresponding to the extension of the file (**!verifier** command reference) or the verifier for the later part of the language in the chain of the preprocessor is used. In the case where no verifier is specified in the chain of the preprocessor, an error occurs.

【Caution1】 No module is to be generated as the result of executing this command.

【Caution2】 This is a new command additionally provided by PIMOS Version 3.0

- Compile Option

!distribute ([(*PE#* | *PE#* : *N* | *PE#* − *PE#*), ..] | ∗)

Specifies the node (processor or cluster) for the load distribution for a compiler or a loader. The basic method of specification is the same as that for "load distribution utility". When ∗ is specified, all the nodes become objects. "Load Distribution" does not include the load distribution for the basic unit of the self-compiler (which is the top level).

An example of the specification method to assign the nodes 1 through 7 is shown below.

1. Explicit specification of node

   ```
   COMPILE> !distribute 1, 2, 3, 4, 5, 6, 7
   ```

2. Specification by the distribution start node and number of nodes

   ```
   COMPILE> !distribute 1:7
   ```

3. Specification by the distribution start node and end node

   ```
   COMPILE> !distribute 1-7
   ```

4. Mixture of the above

   ```
   COMPILE> !distribute 1, 2, 3:3, 6-7
   ```

The load distribution is performed by the unit of a file. Furthermore the specified nodes are used as circuitry.

If the actual processor configuration is less than the specified processor number, or number of nodes, the specification for the overflown node(s) will be ignored. If any meaningless load distribution or no load distribution is specified, only the current node becomes an object (and no load distribution is performed).

!define [*Key Value*]

Sets the parameter for compiling the condition(s). "Key" indicates the parameter name, and "Value" means the value (optional base term) of the parameter. If the argument is omitted, all the parameters which have been set are displayed on the screen.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!undefine *Key*

Releases the setting of the parameter for compiling with conditions specified by Key.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!describe [*PreprocModuleName*]

Displays the compile variable set to the preprocessor which is specified by PreprocmoduleName. If the name of a preprocessor is omitted, the variable set for the KL1 compiler is displayed on the screen. The **!set** command or the environment variable should be used for setting this variable.

【Caution1】 All the following items are covered by the preprocessor. How to display the variable using this command. Whether to display the variable or not

【Caution2】 This is a new command additionally provided by PIMOS Version 3.0

!get [*ParameterName*]

Displays the value of the compile variable specified by ParameterName. If the name of the variable is omitted, all the variables which have been set are displayed on the screen.

【Caution】 In general it is unknown whether the variable displayed by this command is valid for the preprocessor or the IKL1 compiler. To investigate the validity, use the **!describe** command.

!message (*silent* | *verbose*)

Sets the display label to show the locaiton of the error in the source file which is displayed on the screen when error message is output. Whether this setting is valid or not depends on the parser of the language to be compiled.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!parallel [*N*]

Sets the maximum number of modules compiled per node. The default value is 2.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!set [*ParemeterName* [=] [*Value*, ..]]

Sets the value specified by Value to the variable for the preprocessor which is specified by ParameterName. The name of the variable is an optional name depending on the preprocessor, and its value also depends on the preprocessor. If the value is omitted, the variable which was set is released. Note that "=" in the argument does not have any special meaning.

【Caution】 It is unknown whether the variable displayed by this command is valid for the preprocessor or the IKL1 compiler.

The reserved compile variables are listed below. These variables are related to the compilation of KL1 and do not depend on the language of the preprocessor. As for the variables which depend on the language, refer to the corresponding preprocessor.

- **compile:indexing** Specifies whether optimization by indexing is to be performed. The value "on" (Atom) sets the indexing. The indexing code will not be generated by specifying the other values. The default setting value is "on."
- **compile:profile** Specifies whether the information regarding the clause is to be written in the code generated by the compiler. When the value "on" (Atom) is set, the code with the clause information is (this mode has been used for all the old versions) generated. The file without the clause information will be generated by setting other values. The default is off. (If the code compiled in this mode on is traced, the clause number will be displayed on the screen. For details, see a later chapter describing the tracer (Page 128)).

- Relink, Load/Unload

!load *FileName, ..*

Loads the unloaded file which was specified by FileName. Same as input `load( [File-Name, ...] )` from the shell.

!Relink *ModuleName, ..*

Relinks the module specified by ModuleName. Same as input `relink( [ModuleName, ..] )` from the module.

!Relink *ModuleName, ..*

In the module specified by TopModuleName, search all the modules that have a reference relation with the module which is specified by ModuleName from the module. Then perform re-linkage of them. Same as input `relink(TopModule, [ModuleName, ..] )` from the shell.

【Caution】 If the default package is not contained in the **working_packages** specified by the command or the environmental variable, it is necessary to explicitly specify the package in specifying the module. Otherwise, relinking will not be performed, since no module to be relinked is considered to exist.

!Unload *FileName*

Unloads the (multiple) modules loaded immediately before (by compiling and other means) to the file specified by FileName.

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!unload *ModuleName, .. FileName*

Unloads the module specified by ModuleName to the file specified by FileName. Same as input `unload( [Mod1, Mod2, ..], Filename )` from the shell.

!unload *FileName*

Assuming that top_modules specified by the command or the environmental variable is the **top module** , unloads the module belonging to the package set by **working_packages** to the file which is specified by FileName.

【Caution】 If the default package is not contained in the **working_packages** specified by the command or the environmental variable, it is necessary to explicitly specify the package in specifying the module. Otherwise, unloading will not be performed, since no module to be unloaded is considered to exist.

!working_packages ([*Package*, ..] | ∗)

Specifies the package which the user is going to work with by Package. If this variable is specified, the only modules in the specified package are searched. If ∗ is specified, all the packages are searched. In the case where nothing in particular is specified, only the default package is set as the range to be searched.

- Others

!alias [*Alias* [*Command*]]

Specifies the alias of the command specified by Command by using Alias. If the argument is not specified, all the aliases which have been set are to be displayed on the screen. When one argument is specified, it is regarded as the release specification for setting alias. The specification for alias is saved by using **keep** command. It is not necessary to write an "!" mark at the top of the command name or the command alias.

【Caution】 It is impossible to assign an alias to the alias.

The default settings for the aliases are shown in the following Table 3.1.

Table 3.1: Default Aliases

Alias	Command in full
dstr	distribute
orgp	original_package
pkg	package
tm	top_modules
wp	working_packages

!exit

Exits compiling.

!help

Displays the list of commands.

!keep

Saves the set compile option in the file named compile.param in the current directory. The setting of the current directory should not be kept (to make this file valid even after it is copied to the another directory).

【Caution1】 The content kept in this file and the default value may be changed through the Version 3.0. Therefore, it is recommended to erase parameter files on Version 2.5 or older versions and produce new ones. In Version 3.0, for security, an alarm message is designed to be displayed on the screen when the parameter file of the old version is read.

【Caution2】 This file is to be basically created and modified by the self-compiler command. It is possible for a user to modify the self-complier by using the text editor or a similar tool. However, careful attention is necessary because there is a possibility that the compiler will not be started due to an incorrect specification or some other error. In such case, delete the file compile.param to avoid the problem.

Table 3.2: Compile Option Saved by the !keep Command

Command	Environmental Variable
—	compile:prompt
—	compile:more_prompt
alias	
auto_link	compile:auto_link
default_extension	compile:default_file_extension
distribute	compile:distribution
link_error	compile:link_error
message	compile:message
original_package	compile:original_package
package	task:package
parallel	compile:parallel
parser	compile:parsers
path	compile:path
preprocess	compile:preprocessors
set	compile:language_options
top_modules	compile:top_modules
unparser	compile:unparsers
verifier	compile:verifiers
working_packages	compile:working_packages

!revert

Reverts the option settings for the compiler to the condition that existed when it was saved in that directory or when the compiler was started.

The content of the file named compile.param containing in the current directory in starting compilation is to be read. If it does not exist in the current directory, a file with the same name in the home directory is to be read. In the case where it is not contained in either of them, the value of the environment variable will then be used.

!show[(*alias* | *preproc* | *param*)]

Displays the setting for the main compilers. If the argument alias[es] is (are) specified, (all) the alias(es) will be displayed on the screen. Specifying preproc[ess[or[s]]] displays all the preprocessor(s), the verifier(s), the parser(s), and the unparser(s) which have been specified. When param[eter[s]]is specified, the preprocessor option (parameters) and parameter for compiling with conditions are all displayed. If the argument is omitted, the other items regarding the main state of settings are displayed.

!take *FileName*

Reads the command string from the file specified by FileName.

【Hint】 It is convenient to describe the installation method for the preprocessor.

【Example】 Description Example for the Installing Preprocessor

```
!compile my_preproc                % Compile the preprocessor.
!Unload mypp                       % Unloads the compiled module.
!preprocess mylang kl1 my_preproc  % Specifies the preprocessor.
!keep                              % Saves the specification of preprocessor.
```

【Caution】 This is a new command additionally provided by PIMOS Version 3.0

!unalias *Alias*

Releases the setting of the command alias specified by Alias. An "!" at the head of the command alias is not necessary.

3.1.4 Environmental Variables

The following environmental variables are used by the self-compiler.

compile:auto_link

Specifies if the linkage is to be performed at the end of each compilation. Set the atom to either "on" or "off."

compile:default_file_extension

Specifies the file name extension in specifying the name of the file to be compiled. Sets the list containing the string of the extension as an element.

compile:distribution

Specifies the load distribution processor for compiling. Sets the specification of the processor (the processor number or terms such as PEi, PEj, PEi:n are the elements) by the list. (For the further details, see the **!distribute** command (Page 45).

Table 3.3: Relations between Commands and Environment Variables, and Default Setting Values

Environment Variable	Command	Default Value
compile:auto_link	auto_link	on
compile:default_file_extension	default_extension	[“mac”,”kl1”,”pkl1”,”sav”]
compile:distribution	distribute	[ ]
compile:language_options	set	[ ]
compile:link_error	link_error	error
compile:ls_column	—	4
compile:message	message	verbose
compile:more_prompt	—	Null Character String the same length as the length of the p
compile:original_package	original_package	on
compile:parallel	parallel	2
compile:parsers	parser	[{“kl1”,kl1_standard_preprocessor}, {“mac”,kl1_macro_preprocessor}]
compile:path	path	[ ]
compile:preprocessors	preprocessor	[{“kl1”,kl1_standard_preprocessor}, {“mac”,kl1_macro_preprocessor}, {“pkl1”,kl1_standard_preprocessor}]
compile:prompt	—	“COMPILE> “
compile:top_modules	top_modules	[ ]
compile:unparsers	unparser	[{“kl1”,kl1_standard_preprocessor}]
compile:verifiers	verifier	[{“pkl1”,kl1_standard_preprocessor}]
compile:working_packages	working_packages	[ ]
task:directory	cd	Value of task:directory
task:package	package	Value of task: package

compile:language_options

Sets the compile variable for the preprocessor. Sets the list consisting of the vectors in which the variable name (base term) and the optional value (base term) are paired.

compile:link_error

Sets whether the error by the undefined module or undefined external predicate is to be ignored. Sets the atom to either “error” or “warning.”

compile:ls_coulmn

Sets the number of columns in listing the directory.

compile:message

Sets the level of the message output for compiler. Sets the atom to either “silent” or “verbose.”

compile:more_prompt

Sets the input prompt character string in the continuation line.

compile:original_package

Specifies the package in relinking or loading. (See **!original_package** command (Page 43) for further details.) Sets the atom to either "on" or "off."

compile:parallel

Sets the maximum number of modules to be forked per node in the load distribution for compiling. A positive integer will be set.

compile:parsers

Sets the syntax analysis module for the user specified language. Sets the list consisting of the vectors in which the input language (string) and the module name (atom) or the term in the format of {**(::), Package, ModuleName**} are paired.

compile:path

Specifies the search path in opening the file. Sets the list containing the path name as an element.

compile:preprocessors

Specifies the user defined preprocessor. Sets the list consisting of the vectors in which the input language name (string) and the list of the preprocessor module are paired.

compile:prompt

Sets the input prompt (string) at the time of interactive-compilation.

compile:top_modules

Specifies the top module for automatic relinkage in compiling. Sets the list containing the top module name or a term in the format of {**(::), Package, ModuleName**} as its element.

compile:unparsers

Sets the fair copy module of the user specified language. Sets the list consisting of the vectors in which the output language name (string) and the module name (atom) or the term in the format of {**(::), Package, ModuleName**} are paired.

compile:verifiers

Sets the static analysis module for the user specified language. Sets the list consisting of the vectors in which the output language name (string) and the module name (atom) or the term in the format of {**(::), Package, ModuleName**} are paired.

compile:working_packages

Specifies the range of the package to be searched in relinking or unloading through the unit of the program. Sets the list containing the package (atom) as its element.

task:directory

Specifies the default directory. The value is specified by the package name (atom).

task:package

By this specification, the module to be generated as a result of compiling is registered in the package of the specified name. If no package is specified in the external reference, the default package is set to be the starting point of the search.

3.1.5 Preprocessor Provided by PIMOS

- **kl1_macro_expander** A preprocessor for macro expansion of the KL1. The input language is `kl1`, and the output language is `pkl1` (plain KL1).
- **kl1_macro_preprocessor**

  A preprocessor for compiling the user defined macro. The input language is `mac`, and the output language is `pkl1`. The syntax analysis function for the macro definition language is also provided by it.
- **kl1_standard_preprocessor**

  A preprocessor providing the default functions for analyzing syntax and fair copying. No language conversion processing is taken. **kl1** is used as the input and output language. As for the syntax analysis function, it defines the operator (:- op ..), processes, and detects the syntax errors. As for the fine copying function, it outputs each term in the appropriate format, assuming that it is a clause of the KL1. As for the verify function, it checks the variables which appear by the unit of clause (the same as the current varchk).

  The following options are available. (See `!set, !describe` commands (Page 46).)

  - **kl1:include_max**

    Sets the maximum number of nesting (by using a positive integer) the include the file in the parser.
  - **kl1:print_mode**

    Sets whether the guard is only to be displayed in the detailed output mode for the display message (set to verbose by the **!message** command (see Page 46)). By setting the atom "guard," only the guard is displayed. Through the other settings a whole clause is to be displayed on the screen.
  - **kl1:pretty_printer**

    Sets the code to perform the display output for the clause in the detailed output mode for the display message (set to verbose by **!message** command (see Page 46)).

3.2 Relinker

The relinker is a shell program that re-establishes the linkages between modules.

The compiler only generates the modules as specified. Therefore nothing affects the module referring to that module. If a certain module is recompiled, the module referring to the compiled module(s) refers to the old one (before the compilation). Since the compiler only knows which modules are to be linked, it is impossible to know the modules to be relinked that refer to the compiling modules.

One way to obtain the proper compilation is that all modules that consist the reference network should be compiled together. Developing a large program that consists of many modules is time consuming. This can be avoided by using this relinker. If an link error is neglected in compilation

Table 3.4: Environment Variables and Reference Programs

Environmental Variables	compile	relink	unload	load
compile:auto_link	○			
compile:default_file_extension	○			
compile:distribution	○			○
compile:language_options	○			
compile:link_error	○		○	○
compile:ls_column	○			
compile:message	○			
compile:more_prompt	○			
compile:original_package		○		○
compile:parallel	○			
compile:parsers	○			
compile:path	○			○
compile:preprocessors	○			
compile:prompt	○			
compile:top_modules	○			
compile:unparsers	○			
compile:verifiers	○			
compile:working_packages	○	○	○	
task:directory	○		○	○
task:package	○	○	○	○

and the undefined module reference remains, the undefined reference can be released by relinking. It is needless to mention that the referred module should be registered in advance.

As for searching the module from the top module, the package specified by the environment variable **compile:working_package** is the range to be searched.

【Caution】 In going backward in the linkage process while the search from the top module is performed, if the search comes to the package out of the range specified by the environment variable **compile:working_packages**, the search is stopped at that moment and the later part is not searched at all.

relink(ModuleNameList)

The command relink (ModuleAtomList) invokes a relink process. The argument ModuleAtomList is a list of module names (each module name is an atom). The references from the module specified in ModuleAtomList are updated.

【Hint】 When you want to save the original module in order not to include a bug in the module of the parent package to be updated, it is possible to create the copy of the module in the child package by using this relink command.

relink(TopModuleName, ModuleNameList)

Starts a relink process. The top level module (atom) specified by the argument TopModuleAtom is searched for all modules related to the ones specified in the argument ModuleAtomList (list

of module name atoms).

They are then relinked. The specification for the module and, for each argument, is either a three element vector (compound term) such as **Package::ModuleName** ({**(::), Package, ModuleName**}) in vector notation) or is described simply as ModuleName (atom). The Package here is the package name (atom), and ModuleName is the name of the module (atom).

relink:package

Relinks all the modules in the default package.

relink:package(Package)

Relinks all the modules in the specified package (Package).

3.2.1 How to Use Relinker

Starts the relinker through the following procedures.

【Example1】 Example of the use of relinker (No. 1)

```
SHELL> relink([module_1, module_2, module_3])
```

Relinks the modules module_1, module_2, and module_3.

【Example2】 Example of the use of relinker (No. 2)

```
SHELL> relink(top_module, [util_module_1, util_module_2])
```

Relinks all the related modules from the top level module top_module to the lower level modules util_module_1 and util_module_2. In this case, the two low-level modules would have been compiled by the self-compiler. Unless there is some reference relationship between these two, they will not be targets in this relink process.

【Hint】 It is possible to duplicate the module in the different package by utilizing the relinker. All the modules included in the source package can be duplicated by setting the default package to the target package and specifying the source package as `relink:package ( SourcePackage )`. Module Copy by Package

【Example3】 Duplicates all the packages in the package src_package to the package dst_package.

```
SHELL> setenv task:package = dst_package
SHELL> relink:package(src_package)
```

3.3 Unloader

After the modules are generated by the self-compiler, the unloader is used to save them in a file. The saved modules can be loaded into the memory as modules by using the loader (to be mentioned below).

An extension of the default for the unloaded file is "sav", and the extension "sav" is automatically added when the extension is omitted in the specification step.

【Caution】 As every file unloaded is in a binary form, they cannot be edited by the text-editor (such as the PMACS of the PIMPOS).

unload(ModuleNameList, FileName)

The argument ModuleAtomList is a list of the module names (atoms) specifying the target modules. The argument FileName is the file name to be saved.

The specification of the module for each argument of ModuleNameList is either a three-element vector (compound term) such as **Package::ModuleName** ({**(::),Package,ModuleName**}) in vector notation) or is described simply as ModuleName (atom). The Package here is the package name (atom), and ModuleName is the name of the module (atom).

It is possible to unload all the modules in the package specified by Package (atom) by specifying `unload(Package::*, FileName)`.

unload:package(Package, FileName)

Saves all the modules contained in the package specified by the argument Package (atom) into the file specified by FileName. The loading method for modules stored in this manner is the same as that used by specifying the module name.

【Caution】 Whenever a file stored in this method is loaded, all the modules are loaded with the same package as saved. Therefore, it is necessary that a package with the same name be ready at the time of such loading.

unload:program(TopModuleNameList, FileName)

Unloads all the modules called by the top module which are specified by the argument TopModuleNameList, and are existing in the package specified by the environment variable compile:working_packages to the specified file.

The specification of the module for each argument of ModuleNameList is either the three-element vector (compound term) such as **Package::TopModuleName** ({**(::),Package,ModuleName**}) in vector notation) or is described simply as ModuleName (atom). The Package here is the package name (atom), and ModuleName is the name of the module (atom).

【Example】 Example of Using Unloader

```
SHELL> unload([module_1, module_2, module_3], "module123")
```

Unloads the module module_1, module_2, and module_3 to the file "module123.sav"

【Hint】 The time taken for unloading and loading the module is usually shorter than the time required for recompiling by the self-compiler or reperforming the IPL by using the CAL.

3.4 Loader

A shell program to read the file saved by the unloader and to generate the original modules. The loaded module is generated into the default package.

The linkage (reference from the loaded module) of the module loaded by the loader is newly linked again at the time of loading. However, the linkage referring to the loaded module requires relinking because it is not affected by the load, in a manner similar to the self-compiler.

If any undefined module reference is contained in the module to be loaded, the undefined one is released by loading after the referenced module has been registered in advance.

In the case where the extension of the unloaded file is "sav", the extension at the time of specifying the file name can be omitted.

In a manner similar to the self-compiler, the unloaded files are searched in accordance with the value of the environmental variable **compile:path**.

It is possible to specify the load distribution by the unit of file based on the setting of the environmental variable **compile:distribution**.

【Caution1】 A file unloaded by the PIMOS Version 1.5 can not be loaded by the loader for the PIMOS Version 2.0 or a newer version.

【Caution2】

load(FileNameList)

Starts the loader. The argument FileName specifies the name of the file (or a list of the file names) generated by the unloader.

【Example】 Using load:go/1

```
SHELL> load("module123")
```

Loads the module which has been unloaded under the file name module123.sav.

3.5 Package

package:define(Package,SuperPackage)

Creates a package with the name specified by Package (atom) and the SuperPackage (atom) as its parent package.

If the package of the same name exists, or if no appropriate parent package exists, no package is created.

package:define(Package,SuperPackage,Protect)

Creates the package with the name specified by Package (atom) and the SuperPackage (atom) as its parent package. The argument Protect is the protection mode (integer) for the specified package. (Refer to the Chapter 11.1 Acquisition of Resources (Page 431) in the Programming Volume.)

If a package of the same name exists or no appropriate parent package exists, no package is created.

package:protect(Package,Protect)

Recognizes the protection mode for the package specified by Package (atom) as the value of the variable Protect (integer). (Refer to the Chapter 11.1 Acquisition of Resource (Page 431) in the Programming Volume.)

package:listing

Outputs the information related to the package registered at the moment to the standard output destination of the shell. The output items are the package name, the name of the parent package, the name of the owner (or the user ID), and the protection mode in order from the list. (Refer to Chapter 11.1 Acquisition of Resource (Page 431) in the Programming

Volime.) If the information on the package can not be obtained because it is protected, **** or ????????? is displayed on the screen.

【Example】 Display Example of the List of Information on Packages

```
SHELL> package:listing
  root       ----   0     r--r--r--
  user       pimos  0     rwxrwxr--
  privilege  root   ****  ?????????
  pimos      root   0     rwxrwxr-x
  work       pimos  0     rwxrwxrwx
SHELL>
```

package:listing(Package)

Displays the information related to all the modules registered in the package specified by Package (atom) at the moment.

package:listing(Package,Sort)

Displays the information related to all the modules registered in the package specified by Package (atom) at the moment. The atom (sort) indicates the key for sorting and for the module list. Sorting is to be performed in ascending order by using the name of the module as the key when a name is specified, by using the file name when a file is specified and by the time of file creation when the time is specified.

3.6 Module

A shell utility to display the information on the module and to delete the module. The argument ModuleName in the following explanation means either the three-element vector (compound term) such as ({**(::),Package,ModuleName**}) in vector notation) or is described simply as ModuleName (atom).

module(ModuleName)

Displays the list of predicates defined in the module specified by ModuleName (**Package::ModuleName**) after sorting the items by using the name of the predicate as the key.

【Caution】 Currently, non-disclosed predicates are displayed. Whether they are to be displayed by the PIM/XX is not known.

module:info(ModuleName)

Displays the time stamp information for the module specified by ModuleName.

module:where(ModuleNameAtom)

Displays the package defining the module which is specified by ModuleNameAtom (atom) while it is returning to the root package from the default package.

module:remove(ModuleName)

Deletes the registration of the module specified by ModuleName.

It is possible to delete all the modules contained in the default package or the specified package by indicating $*$ in place of the module name (atom).

Chapter 4

Debugging Tool

PIMOS provides the listener (debugger), the inspector, and the monitor as the debugging tools for the KL1 program. The users can utilize these tools to debug the KL1 programs interactively and symbolically.

The listener is the user interface system to extract and execute the program information on request from the users. Basic debugging functions such as the execution trace and the detection of spying and deadlock for the KL1 program areprovided. Also, the execution information is forwarded to the ParaGraph(see Section 5.2, Page 203), which is the evaluation tool for measuring the performance of the KL1 program, and the user interface to start the ParaGraphis prepared.

The inspector is the investigation system to examine the content of the optional KL1 data, and supports the debugging functions of the listener. It can be called from the listener or the shell (see the Section 2.1, Page 7). The execution results of the program and the KL1 data which appear during the trace procedure can be investigated immediately.

The monitor has the function of monitoring the specified KL1 data and waiting for concrete reports. It provides a user interface which is very much like that offered by the inspector. It is started from the tracer of the listener, and is used in cases where the specific state regarding the undefined variable(s) accompanies the execution of the program.

4.1 Listener

The listener is a user interface system for executing programs interactively. The difference from the shell, which has similar functions, is that, in contrast to the shell which focuses primarily on task processing during the execution of a completed program, the listener places emphasis on functions for debugging during the course of program development.

4.1.1 Outline of Functions

The listener has the functions outlined below.

◎ **Execution of Goal String**
The listener is able to execute goal sequences consisting of the user-defined predicates and the built-in predicates from the listener top level. And it manages this execution as a task. In addition, it stops the execution of a task by inputting the interrupt (attention key = Control-C) to allow inquires regarding the amount of the resources consumed and the state of the resources for the task.

◎ **Execution of Shell Utility Program**
Similar to the shell, it is possible to execute the shell utility program on the listener. Therefore, the tools inevitable for the programming environment such as the compiler, the linker, the loader, the unloader, and the variable checker which is prepared as the static analysis tool of the KL1 program can be started as commands.

◎ **Provision of Information Regarding Program Execution**
When the execution of a task is completed, the listener reports the amount of the resources that have been consumed and the execution time for such particular task execution. In addition, such values are to be reported by the listener in the case where the variables are contained in the goal sequence. The listener also measures the number of reduction processing steps and the number of suspension steps per predicate and the frequency of communicating the messages between nodes. The data measured can be output in a graph by utilizing ParaGraph(see the Section 5.2 on Page 203). Thus, the users can analyze the performance of the program visually.

◎ **Variable Pool**
The users are able to store the variables used for executing the goal(s) and the variables output during the trace processes in the variable pool provided by the listener. These functions make it possible to utilize the stored variable(s) as an argument(s) for other goal(s) and to realize the investigation of the details of those values by the inspector.

◎ **Standard Input/Output**
At the time a task is started, the listener assigns a standard input/output device (stream) for that task. Although these devices can be specified together with the goal sequence by the user, a stream for the listener window will be given if they have not specified. As a result of this function, debugging of the programs such as those started from the shell can be performed easily.

◎ **Tracing Function**
The tracing function provided by the listener allows the progression process with regard to the reduction of a specified goal to be viewed. In addition to the normal steps for the tracing function, a spying function, which only traces the calling of the specified predicate, and a function which causes temporarily pauses in the execution of the goal(s) are also available.

A multiwindow tracing function, which gives the output information on tracing by the unit of the optional process (goal) is also provided.

◎ **Starting Inspector and Monitor**

The listener is also equipped with an inspect function for analyzing the content of the data having complicated structures. By using this function, it is possible to analyze the values of the variables, and to analyze the state of the number of arguments for a goal during trace execution. Also, it is possible to proceed with the debugging processes while watching the variable(s) materialize. (See the Section 4.2 on Page 164.)

◎ **Actions to be Taken at Error Occurrence**

An error is reported by the listener in the following cases: The goal has failed; The treatment to the exception has been wrong; or any other abnormal state has arisen. In such cases, the user can analyze the content of the error in detail by using the inspector and then proceed with the execution process by specifying a new goal instead of the one which caused such error.

4.1.2 Operating Method

4.1.2.1 Starting

The listener starts from the shell. It waits for the input of the goal sequence at the "`[1]` prompt. The number of in the prompt is the **event number**, and is counted up whenever the goal string is executed by the user. An event number is used to re-display or to execute a goal string which has been already input. (See Section reflistener:event on Page 68.) This state is referred to as the listener top level. The goal sequences desired to be executed can be input from the top level, and users can execute the desired goal string(s) from **the top level of the listener**.

The following two goals are for starting the listener. One is to generate a window dedicated to the listener which is independent of the shell window to start the listener. It is called **the listener window**. The other is the method to use the window of the shell (standard input/output of the shell) which starts the listener.

◎ Starting with Generating a Listener Window

listener

listener(Position, Size)

listener(Position, Size, Font)

listener(Position, Size, Font, File) **[listen]**

Starts the listener, and create the listener window according to the specified Position, Size, and Font. Executes the listener commands described in the command file File at the moment of starting the listener.

Position: `mouse` or `char(characters, Lines)` `(Characters - number of characters, Lines - number of lines)`

If the specification is illegal or the argument is omitted, the `mouse` is set.

Size: 訳がありません。

Font: Path name of the font file (character string)

If the specification is illegal or the argument is omitted, the default font is set.

File: Path name of command file (character string)

If the command file is not specified, the initial execution file is retrieved and executed. If neither the specified command file or the initial execution file exists, nothing will be processed. Refer to the following sections for the details.

Command Files	→	Section 4.1.2.8 (Page 75)
Initial Execution File	→	Section 4.1.2.9 (Page 75)

【Example】 Starts up the listener from the shell by specifying location, size, font, and command file.

```
Shell>> listener(at(0,0),char(50,10),">sys>font>test_11.font","lis.com") &
```

【Caution】 The listener window is generated on the FEP specified by the environmental variable **task:io_node**.

◎ Starting by Using a Shell Window

listener:listener

listener:listener(File)

Starts the listener on the shell window. When File is specified, the specified file is to be executed as a command file. If the command file is not specified, the initial execution file is retrieved and executed. When the listener is started, the following items are displayed on the screen in order: Version of the PIMOS, Macro Module Name, Default Package, and Current Directory Refer to the following sections as for the details:

Command Files	→	Section 4.1.2.8 (Page 75)
Initial Execution File	→	Section 4.1.2.9 (Page 75)
Macro Modules	→	Seciton 4.1.2.4 (Page 64)
Default Package	→	Section 4.1.2.3 (Page 64)

【Example】 Start up the listener on the shell window

```
Shell>> listener:listener
*** KL1 LISTENER (V 3.0 17-Oct-91) ***
MACRO MODULES     : [pimos]
DEFAULT PACKAGE   : pimos
CURRENT DIRECTORY : >sys>user>icot

[1]
```

【Hint】 Call the top level of the shell from the listener top level. In other words, the movement of the listener to the background of the shell can be performed by executing the **suspend** command. For further details on the **suspend** command, see Section 4.1.6.1 (Page 93).

4.1.2.2 Environmental Variables

The following environmental variables are referred by the listener. For the details of the environmental variables refer to Section 11.1.7 in the Programming Volume (Page 421).

task:directory

Specify the current directory. The directory name is a string.

task:home

Specifies the home directory. The directory name is a string.

task:package

Specifies the default package. The package name is a string.

task:io_node

Specifies the I/O) node which performs Input/Output.

4.1.2.3 Handling Default Package

When the listener is started, it refers to the value of the environmental variable **task:package**. This value is treated as a default package at the start of the listener. It is possible to change the default package through the **default_package** command. See Section 4.1.6.3 (Page 96) for information on how to change the default package. The regulations for handling the default package for the listener are as follows.

1. The specification for the package name can be omitted as well as for the input the goal and the arguments of the listener commands.
2. The output of the package name is omitted at the time of tracing or the error report.
3. The **default trace mode** for the predicate(s) in the default package is set to ON when the tracing process is executed. See Section 4.1.7.4 (Page 127) for information on the trace mode.

4.1.2.4 Macro Expansion of Goal

When the goal is input from the listener top level, the listener expands the goal according to the expansion rules for macro modules (refer to Section 7.3.4 (Page 256) in the Programming Volume for information on the macro functions). The macro module at the start of the listener is **pimos**. The macro module for the listener can be changed by the **change_macro** command. Also, the specification of multiple macro modules is available. See Section 4.1.6.3 (Page 95) for information on how to change the macro module for the listener. Pay attention to the following items related to the macro expansion of the listener.

1. When the user updates the macro module specified by the listener, the updated expansion roles are automatically applied. This means that it is not necessary to execute the **change_macro** command every time the macro module is changed.

   【Hint】 The macro module is dynamically referred every time there is input from the listener top level.

2. If the macro module specified to the listener is updated to the illegal macro module by the user (for example, if the macro module is updated to the normal module) or if the macro module is deleted, the **macro expansion of the goal at the listener top level is suppressed (transferred to a non-conversion mode)**. In such case, an alarm is issued. The user

should load the correct macro module and set the macro module through the **change_macro** command.

【Caution】 When an alarm is issued, the execution of the goal already input is abandoned. Even if the goal which does not require macro expansion is input, this alarm is issued.

【Example1】 The macro module has been updated to the illegal

```
[3] layered#queen(8,Ans).
!WARNING! illegal macro module: my_macro
!CAUTION! switched to non-expansible mode.
```

【Example2】 The macro module has been deleted.

```
[2] module_table:cache_table([remove_module(my_macro,S)]).
 S = normal

[3] layered#queen(8,Ans).
!WARNING! non-existent macro module: my_macro
!CAUTION! switched to non-expansible mode.
```

4.1.2.5 Starting Goal

The user is able to start multiple goals such as user-defined predicates, built-in predicates and commands all at one time from the listener top level.

◎ Notation Arguments

The data notation of the arguments for the goal is the same as the KL1 data notation in the program. See Section 7.2 (Page 232) in the Programming Volume for information on the syntax of the KL1.

◎ End of Input

When input ends, enter a "." (period) and return.

◎ Calling Public Predicates

The goals which call user-defined predicates that are declared public predicates (public declaration `:- public`) in the program are specified by the following format in the same manner as calls between modules with in the program.

1. **In the case that no package is specified (Default Package)**

   ```
   ModuleName : PublicPredicateName(Arg, ...)
   ```

   【Example】 Calling Public Predicates in Default Package

   ```
   [1] layered:n_queens(8,Ans).
   ```

2. **In the case that the package is specified.**

   ```
   PackageName :: ModuleName : PublicPredicateName(Arg, ...)
   ```

【Example】 Calling Public Predicates with Specifying Package

```
[1] sample::layered:n_queens(8,Ans).
```

3. **In the case that a default module is declared.**

```
PublicPredicateName(Arg, ...)
```

【Example】 Calling Public Predicates in Default Package

```
[3] n_queens(8,Ans).
```

See Section 4.1.6.3 (Page 96) for information on the default module.

◎ Calling Local Predicates

The goals which call user-defined predicates that are not declared public predicates in the program are specified in the following format.

1. **In the case that no package is specified (Default Package)**

```
ModuleName ! LocalPredicateName(Arg, ...)
```

【Example】 Calling Local Predicates in Default Package

```
[1] layered!queens(1,8,begin,Ans).
```

2. **In the case that the package is specified.**

```
PackageName :: ModuleName ! LocalPredicateName(Arg, ...)
```

【Example】 Calling Local Predicates with Specifying Package

```
[1] sample::layered!queens(1,8,begin,Ans).
```

3. **In the case that a default module is declared.**

```
! LocalPredicateName(Arg, ...)
```

【Example】 Calling Local Predicates in Default Package

```
[3] !queens(1,8,begin,Ans).
```

See Section 4.1.6.3 (Page 96) for information on the default module.

【Caution1】 There is a possibility that the operator **!** to call the local predicate will be changed for the next version of PIMOS.

【Caution2】 There is a possibility that the local predicates will not be able to be called from the listener top level in the future.

◎ Calling Built-in Predicates

The Built-in predicates are described in the same manner as in the body of the program clause.

【Example1】 Calling a Built-in Predicate

```
[1] predicate_to_code(module#layered,n_queens,2,Code).
```

【Example2】 Operation of Integer

```
[1] Priority := (Pri-Min)/(Max-Min)*4096.
```

【Example3】 Operation of a Floating Point Number

```
[1] Priority $:= (float(Pri)-float(Min))/(float(Max)-float(Min))*4096.0.
```

◎ Execution of Multiple Goals

1. **Parallel Execution**
   In executing multiple goals at one time, a "," (comma) is used as the delimiter to separate each of the goals.

   ```
   Goal1, Goal2
   ```

   【Example1】 Executes the multiple goals in parallel.

   ```
   [1] test!jump(Jump,Step), sample::test:hop(Hop), test:step(Hop,Step).
   ```

   【Example2】 Mixture of Built-in Predicates

   ```
   [1] current_node(Node,Nodes), X := Nodes-Node, layered:n_queens(X,Ans).
   ```

2. **Execution in Order**
   In executing multiple goals one after another, a ";" (semicolon) is used as the delimiter to separate each of the goals.

   ```
   Goal1; Goal2
   ```

   Executes the multiple goals one after another. The Goal2 is started only when the execution of the Goal1 is completed.

   【Example】 Executes multiple goals one in order.

   ```
   [1] test!jump(Jump,Step); sample::test:hop(Hop); test:step(Hop,Step).
   ```

◎ Specification of Execution Node

The goal input from the listener top level is started in the node where the listener has been activated. It is possible to optionally change the node by the goal at the time when the goal string is input. To specify the node where the goal is to be executed, input the following line in a manner similar to the specification of the execution node in the program.

```
Goal @ node( NodeNumber )
```

【Example】Specification of Execution Node

```
[1] sample:generator(S)@node(7), sample:consumer(S,0)@node(3),
    utility:display(0).
```

Refers to Section ?? (Page ??) in the Programming Volume for information on the concept related to the specification of node for execution.

◎ Specification of Execution Priority

In general, the goal input from the listener top level is executed with the highest priority in the task to which the goal to be executed belongs. This priority order can be changed optionally by the goal unit at the time the goal string is input. To specify the priority, input the line shown below in a manner similar to that used for the specification of execution priority in the program.

```
Goal @ priority( Rate )
%Specifying the proportion in the Shoen to which it belongs.

Goal @ relative_priority( Rate )
%Specifying the self-relative in the Shoen to which it belongs.
```

【Example】Specification of Execution Priority

```
[1] sample:generator(S), sample:consumer(S,0)@priority(4000),
    utility:display(0)@relative_priority(-2000).
```

Refer to Section 7.2.6 (Page 238) in the Programming Volume for information on the concept related to the specification of execution priority.

◎ Standard Input/Output

The standard input/output (I/O) is normally set in the listener window (or the shell window). It is possible to change this setting as desired during the time the goal sequence is input. As for the details, see Section 4.1.4 (Page 78).

4.1.2.6 Re-Displaying and Re-Executing Goal Strings

The listener has a history queue to manage the input goal string for realizing the re-display and re-execution of the goal string already input (executed). The user can refer to the input history. Also, the goal string already input can be re-displayed or re-executed by using the event number displayed as the prompt for the listener.

◎ Re-displaying Goal Strings

The following two methods are available to re-display a goal string already input.

! Event number

Displays the goal string through the event number. See Section 4.1.6.1 (Page 87) for further details.

!!

Displays the goal string input immediately before. See Section 4.1.6.1 (Page 87) for further details.

【Caution】 `!"character string"` is not supported at this moment.

◎ Re-execution of Goal Strings

The following three methods are available to re-execute a goal string already input.

Event number

Re-executes the goal string through the event number. See Section 4.1.6.1 (Page 88) for further details.

!

Re-executes a goal string input immediately before. See Section 4.1.6.1 (Page 88) for further details.

Event number list

Executes the goal strings specified by the list of the event numbers one after another. See Section 4.1.6.1 (Page 88).

◎ Referencing to Input History

The user can refer to the content of the history queue in the listener. The length of the queue can be changed.

history

Lists the input history of the goal strings stored in the history queue. See Section 4.1.6.1 (Page 89) for further details.

history_length

Changes the length of the history queue in the listener. See Section 4.1.6.1 (Page 89) for further details.

4.1.2.7 Interruption of Execution

The user can interrupt the execution of a task by pressing[1] the attention key ("**Control-C**"). If the listener detects the attention, it interrupts the execution of the task and waits for the instruction provided by the user. In addition, the actual execution time (from the start to the stop of the task execution) is also reported. The unit of the execution time to be reported is a millisecond.

【Example】 Interruption of Execution

```
[37] bestpath(a,z,Path).
 ?? Interruption (1167807 msec): ?
```

The following key operations can be performed for the execution of a task that has been interrupted.

[1] Press C while while the control key remains pressed.

< *cr* >: continue
`g`: gc & continue
`r`: consumed resource
`s`: status
`@`: status all
`l`: list variables
`v`: display variables
`t`: list traced goals
`k`: list kept goals
`m`: list monitored terms
`a`: abort
`b`: break
`&`: background
`h`: help

The meaning of each operation is described below.

Continue (return)

Continue (resume) execution.

GC & Continue (g)

Starts the garbage collection for all processors and continues (resume) execution.

【Hint】 In the case where the execution of the user program has become deadlocked, it is useful to perform this operation. For detailed information on deadlock, see Section 4.1.5 (Page 79).

【Example】 Detection of Deadlock

```
[1] add(_,_,_).
?? Interruption (1326 msec): g
*** Deadlock detected during GC.
Builtin> add(A,B,C)
 Caller> pimos::listener_cmd_builtin: do_command/ 1
   Node> 0
!! Exception (7254 msec):
```

Consumed resource (r)

Inquires about the amount of resources that were consumed (number of reductions) up until the time the execution of the task was interrupted. "**Unknown**" is displayed on the screen when the amount of resources consumed is unknown.

【Hint】 The amount of resources consumed can be inquired by providing the following message in the control stream for the generated task.

```
status([], task, detail, ^Result)
```

When the management control pert of PIMOS receives `state/4` message, inquires about the amount of resources consumed by sending a `statistics/2` message to the report string for the Shoen. The time-out management function is equipped to handle the case in which the execution of the inquired Shoen was already terminated[2] (for example, if the child Shoen was

[2] Otherwise PIMOS is deadlocked.

aborted through the execution of the parent Shoen, and other cases). If there is no response within a certain time after the Shoen was inquired about the amount of resources consumed, "**unknown**" is displayed on the screen.

Status (s)

Inquires as to the status of the task. The following information is displayed on the screen.

```
%%% STATUS %%%
id      : 203
name    : "Task_35"
status  : normal([stop(none)])
limit   : 75000000 reductions
consume : 10129 reductions
[children] (id name status limit consume)
33 "Task_35_child" normal([]) 75000000 unknown
```

The meanings for each field are indicated below.

id: id: Relative resource identifier for task that is executing the goal. Refer to Section 11.1.6 (Page 421) for information on the relative resource identifier.

name: task name. Normally, the event number is appended to "Task_".

status: Status of task. A vector in the format of {Cond,Stop} is displayed on the screen. If the status is not evident (such as the case when the parent Shoen is stopped) "**unknown**" is displayed. The Cond indicates the execution states for the task, and the types of such states are as follows.

normal: Currently in normal execution (not in the state of resource shortage)

resource_low: The amount of resources is becoming low.

terminating: Currently terminating processing.

shut_terminating: Current terminating processing through the forced termination of the parent task.

Stops is a list of reasons for stopping the task, and the format for each element is stop(Reason). Reason is an atom to show the reason for stop. The listener normally determines it.

limit: Upper limit of the amount of consumed resources.

consume: Amount of consumed resources (number of reductions) until the time that the execution of the task was interrupted.
"**unknown**" is displayed if it is not evident.

children:

Status all (@)

Amount of consumed resources (number of reductions) until the time that the execution of the task was interrupted.
"**unknown**" is displayed if it is not evident. Inquires as to the resource status of the task. An example of the information that is displayed is indicated below.

```
%%% STATUS ALL %%%
id   : 203
type : task
name : "Task_35"
[children] (id type name)
5 atom "atom"
8 atom "atom"
12 module "module"
15 file_request "file request"
16 window "window"
18 window "window"
19 file ">sys>csp>PIMOS.DIR>pimos.mac"
20 fep_timer fep
23 atom "atom"
```

The meanings of each field are listed below.

id: Relative resource identifier for the resources. Refer to Section 11.1.6 (Page ??) in the Programming Volume for the relative resource identifier.

type: Type of OS-defined resource.

name: Resource name (comment provided to the resource)

children: When information pertaining to the child tasks has been obtained, that same information is provided for all subject child tasks.

List variables (l)

Displays the list of the variables stored in the variable poll for the listener. See Section 4.1.6.5 **list** command (Page 105) for further details.

【Hint】 The variable input as the argument of the goal at the listener top level (except for the one with the name starting underline_) is restrained as that value until the end of the execution of the program. It is possible to view the list regardless of the mode for memorizing the value of the variable.

Display variables (v)

Displays the list of the variables and their values in a variable pool for the listener.

【Hint】 The variable input as the argument of the goal at the listener top level (except for the one with the name staring with an underline_) is to be restrained as that value until the end of the execution of the program. It is possible to view the list regardless to the mode for memorizing the value of the variable.

List traced goals (t)

Lists those goals that have not been reduced at the time this command is executed from among those goals that have been executed following the specification of trace. For further details see Section 4.1.7.9 (Page 143).

List kept goals (k)

Displays the list of the goals for which execution has been paused during the trace execution. For further details, see Section 4.1.7.9 (Page 140).

List monitored terms (m)

Displays the list of items which are specified to be the monitoring target but for which no report has been specifically provided by the time this command is executed. For further details, see Section 4.2.5.2 (Page 192).

Abort (a)

Abort execution.

Break (b)

Returns the listener to the top level state while the execution remains interrupted. At that time, the interruption level is displayed at the top level. Enter "**exit**" in order to go down to the interruption level.

【Example】 Calling and Returning at Top Level During Execution

```
 ?? Interruption (112794 msec): b

{1}
[7] inspect(X).
{0, code$sample::layered:queens/4$662, {1,f,begin,K}}> me
   ...

{1}
[8] exit.
 ?? ------------ (112794 msec):
```

Background (&)

Returns the listener to the top level state while the execution continues. At such time, the level is displayed at the top level. To make the level lower, type **exit**.

【Hint1】 The current specification is a preliminary version. A real multitask management function which is similar to the shell is scheduled to be incorporated in the future.

【Hint2】 Assume the case in which the user utilizes the device resources provided by PIMOS. If the following line is input from the top level of the listener in order to obtain the general request, the goal is interrupted.

```
[1] shoen:raise(pimos_tag#task, general_request, GR).
```

Many users might have experienced this phenomenon already, since PIMOS is waiting for the message to generate and manipulates the resources sent from the user.[3] The execution will neither be resumed nor stopped unless an appropriate message is sent from the user. If the background function is used at such time, the results indicated in the following example will occur.

【Example】 Obtains the resource hierarchically and creates a window.

[3] Precisely speaking, a filter for protection is inserted in the user task together with the user goal in order to prevent a mistake made by the user from affecting PIMOS in cases in which an illegal message is given to PIMOS (the general request in the above example) or in which a stream is discarded because of a user mistake. In the above example, the filter is waiting for the message protocol to be provided by the user (the stream is closed), and that causes the execution of goal to be interrupted.

```
% Sets the mode to recall the value of the variable to ON, and
% selects the reporting mode for the value all.
[1] remember, display_variables(all).

% Obtains the window request device, and move to the background.
[2] shoen:raise(pimos_tag#task, general_request, GR),
    GR=[window(normal(WR,_,_))].
 ?? interruption (1007 msec): &

% Obtaina the window device.
{1}
[3] WR=[create(normal(W,_,_))].
 GR = [window(normal([ & ],D,E))]
 WR = [create(normal(W,F,G))]
 W = W

% Creates a window.
{1}
[4] W=[reshape(mouse,mouse,_),activate(_)|W1].
 GR = [window(normal([ & ],E,[]))]
 W = [reshape(mouse,mouse,F),activate(G)|W1]
 WR = [create(normal([ & , & |W1],H,I))]
 W1 = W1

% Inserts the buffer, and moves to the background.
{1}
[5] buffer:interaction_filter(BUF,W1).
 ?? interruption (1102 msec): &

% Outputs to the window.
{2}
[7] BUF=[putb("ABCDEFG"),nl,flush(_)|B1].
 BUF = [putb("ABCDEFG"),nl,flush(G)|H]
 GR = [window(normal([ & ],I,[]))]
 W = [reshape(mouse,mouse,normal),activate(normal)|W1]
 W1 = W1
 WR = [create(normal([ & , & |W1],J,K))]
 B1 = H

% Closes the stream and terminates the task.
{2}
[8] B1=[putt(end),nl,getl(L)].
 B1 = [putt(end),nl,getl(L)]
 BUF = [putb("ABCDEFG"),nl,flush(normal),putt(end),nl,getl(L)]
 GR = [window(normal([ & ],H,[]))]
 W = [reshape(mouse,mouse,normal),activate(normal),putb("ABCDEFG
",normal)|I]
 W1 = [putb("ABCDEFG
",normal)|I]
 WR = [create(normal([ & , & , & |I],J,K))]
 L = L
```

Help (h)

Display the help menu.

4.1.2.8 Command Files

The command is usually input to the listener from the listener top level. It is possible to switch it to the file temporally and execute the command string written in the file. The command file is executed by the **take** command. For further details, see Section 4.1.6.1 (Page 90).

4.1.2.9 Initial Execution File

The command file is automatically executed at the time the listener is started. The file name is fixed as "**listener.com**". This file retrieves the current directory when the listener starts. If no file exists, nothing will be processed.

4.1.2.10 Obtaining Information Related to Execution of Programs

The user can get the information on some execution steps for the program executed from the listener top level. The information which can be provided by the listener is as follows.

1. **Amount of Consumed Resources (total number of reductions) and Execution Time (in milliseconds)**
   Sets the mode reported in the information related to the execution of the program to ON. The switching of modes is performed by the **statistics** command and the **nostatistics** command. See Section 4.1.6.3 for further details. (See Page 94)

2. **Information on Execution State**
   Measures the execution state through ParaGraph-related commands. It is possible to output the information on the measured results in graphs by starting ParaGraphby using the **paragraph** command.

   (a) **Number of reductions and number of suspensions by predicate Measured by profile command.**

   (b) **Information on the transmission/reception of messages between nodes and the operation state of each node Measured by profile_msg command.**

   (c) **Calculation time, idle time, time for sending/receiving messages, and GC execution time as for each node Measured by profile_node command.**

   As for the commands, see Section 4.1.6.9 (Page 116), and for ParaGraphSection 5.2 (Page 203).

3. State of Program in the middle of execution
   It is possible to know the execution state and the resource consumption amount of the program, as well as the usage state of the OS definition resources, by inputting **Control-C** during the execution interruption. For further details, see Section 4.1.2.7 (Page 69).

4. **The variable whose name starts with a capital letter and**
   is used for the argument of the goal is reported when the execution of program has ended normally. The mode of the report on the value of the variable can be changed by the **display_variables** command. For further details see Section 4.1.6.5 (Page 103).

4.1.2.11 Termination

Type exit from the listener top level. If the inspector, the monitor, or ParaGraphhas been activated, it is terminated when the listener **exits**.

4.1.3 Variable Pool

The listener has a pool called "**variable pool**" to store the variables as its operation environment. The user can store the variables used as the arguments of the goals or the variables that appeared during the trace output in the variable pool of the listener. To "store variables in the variable pool," means "to maintain the bound condition of the value of the variable". The values of variables stored in the variable pool are bound, and can be used as many times as desired as arguments of the goals in the listener top level.

4.1.3.1 Notation of Variables

To store the variable used as an argument for the goal in the variable pool or to have the value reported at the end of the execution of the goal, an appropriate mode assigned to activate those functions should be used. However, the following variables, which do not recognize such modes are also available for use. Note the naming method for such variables.

1. **Variables whose name starts with a capital letter**

   ```
   X, Priority, CODE, X1, Ans_1, ...
   ```

   These variables are handled in accordance with the mode for storing the value of the variable, the mode for reporting the value at the end of execution, and the mode of inquiry.

2. **Variables whose name starts with an underline.**

   ```
   _, _X, _priority, _x1, _Ans_1, __XX, ...
   ```

   The variables whose names start with an underline are not stored in the variable pool, and their values are not reported at the end of the execution, regardless of the modes.

   【Example】 Variables with an underline at the beginning of their name

   ```
   [1] _X=2, XX := _X*_X.
    XX = 4
   ```

   The variable `XX` is handled according to the storage mode, the report mode, and the inquiry mode. However, the variable `_X` which has nothing to do with each such mode is not stored in the variable pool, nor it is reported.

4.1.3.2 Storage Method for Variables

1. **Sets the mode to store the value of the variable to ON.**
   The mode is prepared for storing the variable used as the argument of the goal in the variable pool. When this mode is set to ON, the variable used as the argument of the goal is stored in the variable pool automatically when the execution of that goal is successively completed. The default value at the start of the listener is OFF. To switch the mode, use the **remember** command and the **forget** command. For further details see Section 4.1.6.5 (Page 101).

2. **Attaches "^" (hat) immediately before the variable name.**
   Even when the mode for storing variables in the variable pool is set to OFF, the variable name can be stored in the variable pool by attaching "^" immediately before the variable name which is input.

```
^Variable Name
```

【Example】 Stores the variable with "^" attached (assuming the current mode is "forget").

```
[1] samp:length([a],M), samp:length([a,b,c],^Len).  % Stores only Len.
 M = 1
^Len = 3

[2] value.  % Display the list of the stored variables.
    % (To be described later).
^Len = 3

[3] .       % Releases the bound condition of the value for the stored
    % variable.  (To be described later.)

[4] ^X=1, Y=2, ^Z := X+Y.
^X = 1
 Y = 2
^Z = 3

[5] value.  % The variable Y has not been stored.
^X = 1
^Z = 3

[6] X=Y.
 Y = 1
```

【Caution1】 When "^" is attached to a variable starting with an underline, such variable is not to be stored in the variable pool ("^" is ignored).

```
[1] ^X=1, ^_Y=2.
^X = 1

[2] value.
^X = 1
```

【Caution2】 When the mode for storing variables in the variable pool is set to ON, "^" is meaningless.

3. **Using the command under tracing**
   It is possible to store the variables which appear during the trace output process by using the **rem** command of the tracer. For further details see Section 4.1.7.9 (Page 147).

4. **Using the inspector command**
   It is possible to store the data with an optional variable name by using the **name** command of the inspector. For further details, see Section 4.2.4.5 (Page 187).

4.1.3.3 List of Variables Stored

The **value** command and the list command are used to display the name and the value of the variables which are stored in the variable pool. For further details, see Section 4.1.6.5 (Page 105).

4.1.3.4 Releasing Variables

The following are two methods for releasing the variables stored in the variable pool.

1. **Releases all the variables.**
   All the variables stored in the variable pool are released by inputting ".", "`cr`" (a period return). For further details, see Section 4.1.6.5 (Page 103).

2. **It is possible to release the variables individually by using the forget** (variable name) command. For further details, see Section 4.1.6.5 (Page 103).

4.1.3.5 Displaying the Values of Variables for Terminating Execution

The mode is prepared for displaying the variable used as the argument of the goal on the screen at the end of executing the task. To switch the mode, use the **display_variable** command. For further details, see Section 4.1.6.5 (Page 103).

4.1.3.6 Inquiry for Variables Used for Execution

At the end of executing a task, it is possible to send several inquiries to the variable used as the argument of the goal. The inquiries available include referring and inspecting the variable values, and storing the variables in the variable pool. To set the inquiry mode, use the **query** command. For further details, see Section 4.1.6.5 (Page 104).

4.1.4 Standard Input/Output

The listener has five devices (streams) available, which consist of standard input, standard output, standard message output, standard interaction and standard input/output for tasks. With respect to standard input, standard output, standard message output and standard input/output, the user can specify the desired device.

【Hint】 The initial values of the output limit value (depth and length of structure, length of string, and length of atom name) of the holophrasis for standard output, standard message output, standard interaction, of the standard input/output stream, which are provided by the listener, are the values set in the listener at the time the goal is executed. The standard operator of the filter for input/output is defined as the operator used to parse/unparse. Refer to Table 9.1 (Page 365) for a list of standard operators.

Normally, the destination of the standard input/output is set to the listener window. The user can specify this setting by writing the following operators after the goal string at the listener top level.

Standard input: <= (device)

Standard output: => (device)

Standard message output: -> (device)

In addition, the device is designated in the following manner.

File: `file( FilePathnameString )`
Opens the file in either the **read** or **write** mode. FilePathnameString is a character string.

File: `file( ^ FilePathnameString )`
Opens the file in the **append** mode.

【Caution】 The methods for specifying the mode used to open the file, either in write mode or append mode, have been interchanged. (whether or not "^" exists had been interchanged.) by Version 3.0.

Window: `window( WindowNameString )`

Window: `window( WindowNameString, Position, Size )`

Window: `window( WindowNameString, Position, Size, Font )`

Window: `window( WindowNameString, X, Y, W, H )`

Window: 英語の訳が無いです。
`window( WindowNameString, X, Y, W, H, Font )`
WindowNameString はウインドウのタイトル (文字列). X, Y はウインドウの位置 (ドット), W, H はウインドウのサイズ (文字数と行数), Font は表示に使用するフォントファイルのパス名 (文字列) である. また, Position には `mouse` または `at(X,Y)` を, Size には `mouse` または `char(W,H)` を指定する.

標準入力, 標準出力, 標準メッセージ出力は, それぞれ一度に入力するゴール列 (タスク) に対して高々一つずつ指定することができる.

【Example1】 Using the standard input/output of the listener.

```
[1] test1:start, test2!start.
```

【Example2】 Designating a file as a standard message output.

```
[1] test1:start, test2!start -> file( ^ "mes.doc").
```

【Example3】 Designating all standard input, standard output and standard message output.

```
[1] test:start -> file("mes.doc") <= window("in") => window("out").
```

As for information on how to acquire the standard input/output stream from the task and about the message protocol, refer to Section 9.4 (Page 394) in Programming Volume.

4.1.5 Deadlock Processing

4.1.5.1 Outline

Deadlock refers to the execution of a certain goal being interrupted and the occurrence of a deadlocked state in which restarting is impossible within the range of normal processing in IL1. More specifically, those goals that contain the materialization of a variable for which there is no reference anywhere (no reference path), become deadlocked.

When one goal becomes deadlocked, there is a strong possibility that all goals in a group of goals which have a dependent relationship with the data to be generated by the deadlocked goal will also become deadlocked. Therefore, the number of deadlocked goals generally tends to be

large. It is necessary when actually debugging programs in which deadlock may occur to detect the goal group that is the "cause" in the cause-and-effect relationship involved in deadlock. Such a goal that is the cause of deadlock is referred to here as the "**maximum goal**". The deadlocked goals that are reported in the PIMOS are all maximum goals. In addition, in the case where the group of goals in the cause-and-effect relationship involving a deadlock form a cyclic structure, since all the goals have the same value in the dependent relationship, only one of those goals is reported.

In looking at the program execution behavior, although the detection of a deadlock in the source program is an extremely difficult task, the efficiency of program debugging has improved considerably as a result of this function.

【Hint】 It is recommended to use the standard utilities such as the **verifier** and the **variable checker** provided by the compiler to prevent deadlock from occurring. These are tools which statically analyze and detect the variables which appear only once in one program clause (so-called "void variables"). The performance of this check alone has considerable efficacy. As for the verifier, see Chapter 3 (Page 36), and for the variable checking utility see Section 2.2 (Page 28).

4.1.5.2 Detecting Function of Deadlock Goal

There are two deadlock detection functions provided by the PIMOS.; One is to discover the deadlock during the batch GC (Garbage Collector). The other is to discover the deadlock during the execution of the incremental GC.[4]

◎ Almost all the deadlocked program executions in one node can be detected by the batch GC

Almost all the deadlocked program executions in one node can be detected by the batch GC, while only the deadlock with a certain condition is found by executing the incremental GC. To use the batch GC for detecting the deadlocked goals, it has to be activated while the user program is in a deadlocked condition. Although the GC is started automatically when there is a shortage of the free space in the memory, in order to manually perform deadlock detection, it is also necessary to manually start the GC. The following method is used to forcibly start the GC.

1. Execution of the user program is interrupted (by inputting attention using Ctr-C), and the GC is started by using the **g** commands available during the interruption of execution. For further details, see Section 4.1.2.7 (Page 69).

2. The GC is started from the shell.

【Hint】 The following methods are used for confirming that the execution of the program has stopped.

[4] It is found by the incremental GC, which uses the information of the Multiple Reference Bit (MRB). The incremental GC counts the number of references to the data object, and determines if the reference pointer to that data object no longer exists. It then collects the memory area used by the data object. The MRB is a 1 bit counter, and determines if the reference to the data object is either "1" or "many". For example, in the case where a certain data object was generated by the reference number "1", and the number of references has not been changed, the process which has consumed that data can collect the data object (so-called MRB white). When the number of references has been changed to "many", the collection of unnecessary data is terminated because nothing can be detected after the change of the reference number.

1. As for the Pseudo Multi-PSI, the "multi PSI-CSP status monitoring panel" becomes black when the CSP silent mode is OFF.
2. As for the Pseudo PIM/m, "**idle**" is displayed on the "multi PSI-CSP status monitoring panel" when the CSP silent mode is OFF.
3. Repeatedly interrupts the execution of the user program (by inputting attention using Control-C) and confirms that the amount of the resource consumption during the execution has not been increasing by using the **r** command during interruptions in execution. (See Section 4.1.2.7, Page 69)

【Caution】 The Deadlock is not detected in the following cases.

【Example】 Effect of variable pool

```
[1] add(X,1,_).
```

Although the execution of the built-in predicate add/3 is interrupted when the variable X is stored in the variable pool (in other words, a reference path to variable X from the variable pool exists), it is not treated as a deadlock. In the case where the user performs debugging of deadlocks, it is important to make effective use of variables whose names begin with an underline.

4.1.5.3 Types of Deadlock Goals and Output Examples of such Goals

◎ Deadlock goals detected by batch GC

- **The deadlock of the user defined predicate was found by the batch GC.**

```
[1] shoen:raise(pimos_tag#task, general_request, _GR).
 *** Deadlock detected during GC.
 Goal> pimos:: genreq_filter:genreq_filter(A,B,C,[])
 Node> 0
```

- **The deadlock of the built-in predicate by the batch GC.**

```
[1] add(X,_,Y), divide(Y,_,Z), modulo(Z,_,_).
 *** Deadlock detected during GC.
 Builtin> add(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
```

The built-in predicate add/3 is detected as the maximum goal.

- **The merger deadlock was found by the batch GC.**

```
[1] merge({_,[koichi],_,[nakao],_},OUT).
 *** Merger deadlock detected during GC.
 *** 3 merger inputs were abandoned.
 Abandoned> {B,C,D}
    Caller> pimos:: listener_cmd_builtin: do_command/ 1
      Node> 0
 !! Exception (6307 msec): e            % Execute the alternative goal.
% (To be described later.)
 >Please execute substitutive goals.
 |?- B=[27], C=[10], D=[1961].          % Close the input argument of the merger.
 OUT = [koichi,nakao,27,10,1961]        %End the execution of the merger normally.
```

In detecting the merger deadlock, the variables "B", "C", and "D" indicated by "Abandoned>" are the input argument of the merger which has remained unmaterialized and been abandoned. As for "execution of the alternative goal" shown in the above example, see Section 4.1.8.1 (Page 153).

【Hint】**How to use the input argument of a merger** which remains unmaterialized and abandoned As shown in the above example, the unmaterialized and abandoned input argument of the merger, which is reported when the deadlock occurs, is the input variable to send the correct data to the merger with the exception. Therefore, the user can resume the execution of the merger by materializing the reported input variable to the optional data.

◎ Deadlock goal detected by incremental GC

- **No goal remains, although there exists the goal which is waiting the variable to be materialized.**[5]

  (collect_value)

```
% collect_hook :- true | p(X), q(X).
% p(a) :- true | true.
% q(_) :- true | true.
%
% p/1 が q/1 よりも先に実行された場合.

[1] test:collect_hook.
 *** Deadlock occurred. [collect_value(HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
```

- **The case where p/1 was executed before q/1.** The goal to be realized by the input of the merger has shortened.[6]

```
% collect_mghook :- true | merge({X},_), q(X).
% q(_) :- true | true.
%
% The case where merge/2 was executed before q/1.

[1] test:collect_mghook.
 *** Merger deadlock occurred. [collect_value(MGHOKo)]
 Abandoned> {A}
    Caller> test: collect_mghook/ 0
      With> test: q/1
      Node> 0
```

[5] The white path indicating HOOKo has been collected.

[6] The white path indicating MGHOK has been collected.

```
% collect_mghook(Node) :- true | merge({X},_), q(X)@node(Node).
% q(_) :- true | true.
%
% The case where merge/2 was executed before q/1 and
% q/1 was executed by a different processor from that which
% executed merge/2.

[1] current_node(Node,_), test:collect_mghook(~(Node+1)).
 *** Merger deadlock occurred. [collect_value(MGHOKo)]
 Abandoned> {B}
    Caller> test: collect_mghook/ 1
      Node> 0
```

【Caution】 If an eternal interruption of the merger has been detected by the collect_value command as a result of the mounting of the processing system, and if the reference to the input variable of the merger is extinguished in a node different from the one where the merger is executed, the goal which abandoned that input value can not be detected.

- **An attempt was made to wait for the variable not to be materialized by another goal.[7](suspend)**

```
% hook_hook_1 :- true | p(X), q(X).
% p(0) :- true | true.
% q(1) :- true | true.

[1] test:hook_hook_1.
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
 !! Exception (115 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: q(A)
 With> test: p/ 1
 Node> 0
```

```
% hook_hook_2 :- true | add(X,1,_), q(X).
% q(1) :- true | true.

[1] test:hook_hook_2.
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> add(A,1,B)
  Caller> test: hook_hook_2/ 0
    With> test: q/ 1
    Node> 0
 !! Exception (116 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: q(A)
 With> add called from test: hook_hook_2/ 0
 Node> 0
```

[7] The goal has been hooked to the white path indicating HOOKo.

```
% hook_hook_3 :- true | add(X,1,_), multiply(X,2,_).

[1] test:hook_hook_3.
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> add(A,1,B)
  Caller> test: hook_hook_3/ 0
    With> multiply called from test: hook_hook_3/ 0
    Node> 0
 !! Exception (117 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> multiply(A,2,C)
  Caller> test: hook_hook_3/ 0
    With> add called from test: hook_hook_3/ 0
    Node> 0
```

【Caution】 When the deadlock goals are detected by the suspend command, they are reported separately to the Shoen to which each goal causing the deadlock belongs. (Two deadlocks are always reported.)

- **An attempt was made to wait for the input variable of the merger to refer to itself only.[8](suspend)**

```
% mghook_hook_1 :- true | merge({X},_), q(X).
% q(1) :- true | true.

[1] test:mghook_hook_1.
 *** Merger Deadlock occurred. [suspend(HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_1/ 0
      With> test: q/ 1
      Node> 0
 !! Exception (115 msec):
 *** Deadlock occurred. [suspend(MGHOKo)]
 Goal> test: q(A)
 With> merge called from test: mghook_hook_1/ 0
 Node> 0
```

```
% mghook_hook_2 :- true | merge({X},_), add(X,1,_).

[1] test:mghook_hook_2.
 *** Merger Deadlock occurred. [suspend(HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_2/ 0
      With> add called from test: mghook_hook_2/ 0
      Node> 0
 !! Exception (116 msec):
 *** Deadlock occurred. [suspend(MGHOKo)]
 Builtin> add(A,1,B)
  Caller> test: mghook_hook_2/ 0
    With> merge called from test: mghook_hook_2/ 0
    Node> 0
```

- **The variables to be materialized were unified.[9](unify)**

[8] The goal has been hooked in the white path indicating MGHOKo.

[9] The white paths both indicate that HOOKo was active-unified.

```
% hook_hook_unify :- true | p(X), q(Y), r(X,Y).
% p(0) :- true | true.
% q(1) :- true | true.
% r(X,Y) :- true | X=Y.
%
% The variables to be materialized were unified.

[1] test:hook_hook_unify.
 *** Deadlock occurred. [unify(HOOKo,HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
 !! Exception (117 msec):
 *** Deadlock occurred. [unify(HOOKo,HOOKo)]
 Goal> test: q(A)
 With> test: p/ 1
 Node> 0
```

【Caution】 When deadlock goals are detected by the unify command, they are to be reported separately to the Shoen to which each goal causing the deadlock belongs. (Two deadlocks are always reported.)

- **The variable to be materialized and the input variable of the merger were unified.[10] (unify)**

```
% mghook_hook_unify :- true | merge({X},_), q(Y), r(X,Y).
% q(1) :- true | true.
% r(X,Y) :- true | X=Y.
%
% The case where merge/2 and q/1 were executed before r/2.

[1] test:mghook_hook_unify.
 *** Merger Deadlock occurred. [unify(MGHOKo,HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_unify/ 0
      With> test: q/ 1
      Node> 0
 !! Exception (118 msec):
 *** Deadlock occurred. [unify(HOOKo,MGHOKo)]
 Goal> test: q(A)
 With> merge called from test: mghook_hook_unify/ 0
 Node> 0
```

- **The input variables of the merger were unified.[11] (unify)**

[10] The white path indicating that HOOKo was active-unified to the white path indicating MGHOKo.

[11] The white paths both indicate MGHOKo was active-unified.

```
% mghook_mghook_unify :- true | merge({X},_), merge({Y},_), r(X,Y).
% r(X,Y) :- true | X=Y.
%
% In the case where two merge/2's were executed before r/2.

[1] test:mghook_mghook_unify.
 *** Merger Deadlock occurred. [unify(MGHOKo,MGHOKo)]
 Abandoned> {A}
    Caller> test: mghook_mghook_unify/ 0
      With> merge called from test: mghook_mghook_unify/ 0
      Node> 0
 !! Exception (119 msec):
 *** Merger Deadlock occurred. [unify(MGHOKo,MGHOKo)]
 Abandoned> {B}
    Caller> test: mghook_mghook_unify/ 0
      With> merge called from test: mghook_mghook_unify/ 0
      Node> 0
```

4.1.6 Commands

There are several commands available for performing program debugging and interactive program execution. As the arguments of those commands, the values listed below are provided unless specially described. For information on the marks "^" and "?", refer to Section 7.1 "How to read mode display" (Page 231) in the Programming Volume.

Package	: Package name atom
ModuleName	: Module name atom
ModuleSpec	: Package::ModuleName or ModuleName
	: When Package is omitted ,the default package is considered to be specified.
Predicate	: Predicate name atom
Arity	: Number of predicates of arguments (Arity>= 0)
Operator	: Operator name atom
Precedence	: Priority order of operator (0< Precedence =<1200)
OpType	: Operator type. fx, fy, xfx, yfx, xfy, xf, yf
Term	: Optional KL1 data
ONorOFF	: Either on (atom) or off (atom)
Depth	: Output limit value of structure depth (Depth ≥ 0)
Length	: Output limit value for the length of the structure and other values (Length ≥ 0)

4.1.6.1 Basic Commands

ModuleSpec : Public

Executes Public, which is the public predicate of the module specified by ModuleSpec. For further details, see "Calling public predicates" (Page 65).

ModuleSpec ! Local

英訳無し

Goal @ node(NodeNumber)

Executes Goal, under the node of the node number NodeNumber. For further details, see Section 4.1.2.5 "Specification of Execution Node" (Page 67).

Goal @ priority(Rate)

Goal @ relative_priority(Rate)

Executes Goal in the execution priority Rate. When "priority" is specified, the proportion in the Shoen is indicated. And if "relative_priority" is specified, the self-relative within the Shoen is indicated. For further details, see Section 4.1.2.5 "Specification of Execution Priority" (Page 68). The concept of the execution priority specification is described in Section 7.2.6 (Page 238) in the Programming Volume.

Goal1, Goal2

Executes Goal1 and Goal2 in parallel. For further details, see Section 4.1.2.5 "Parallel Execution" (Page 67).

Goal1; Goal2

Executes Goal1 and Goal2 one after another. The execution of Goal2 is not to be started until the execution of Goal1 has ended. For further details, see Section 4.1.2.5 "Execution in Order" (Page 67).

!Event Number

Displays the goal string specified by the event number. The displayed goal string can be treated as the input for a new goal string to the listener after it was optionally modified by the user. If the specified event does not exist in the string queue, it is to be regarded as an error. Note that this command is not stored in the history queue.

【Example】 Re-display the goal string.

```
[10] !7.
[10] inter([prompt(" TERM>> "),getwt(^ WT)]) .
```

【Hint】 If a negative number is specified as the event number, it is processed as the relative specification from the current event (number).

【Caution】 "`!character string''` is not currently supported.

!!

Displays the goal string input immediately before. The displayed goal string can be treated as an input for a new goal string to the listener after beingoptionally modified by the user. If the length of the history queue is 0, it is to be regarded as an error. Note that this command is not stored in the history queue.

【Example】 Re-displays the goal string input immediately before.

```
[6] inter([prompt(" TERM>> "), getwt(WT)]).
 TERM>> abc(X).
 WT = normal(vector({atom(abc),variable("X")}))

[7] !!.
[7] inter([prompt(" TERM>> "),getwt(^WT)]) .     % Perform modification.
```

【Caution】 ‘‘character string’’ is not currently supported.

Event number

Displays the goal string specified by the event number. If the specified event does not exist in the string queue, it is to be regarded as an error. Note that this command is not stored in the history queue.

【Example】 Re-display the goal string.

```
[10] 7.
[10] inter([prompt(" TERM>> "),getwt(^ WT)]) .
 TERM>>
```

【Hint】 If a negative number is specified as the event number, it is processed as a relative specification from the current event (number).

!

Displays the goal string input immediately before. If the length of the history queue is 0, it is to be regarded as an error. Note that this command is not stored in the history queue.

【Example】 Re-display the goal string input immediately before.

```
[6] inter([prompt(" TERM>> "), getwt(WT)]).
 TERM>> abc(X).
 WT = normal(vector({atom(abc),variable("X")}))

[7] !.
[7] inter([prompt(" TERM>> "),getwt(WT)]) .
 TERM>>
```

Event number list

Executes the group of goal strings specified by the list of the event numbers in order of their specification. If the specified event does not exist in the history queue, it becomes an error. When the execution of the goal string specified by the event number fails, the execution of the remaining event(s) will be discarded. Note that the command is not stored in the history queue.

【Example】 Re-executes the group of goal strings already input.

```
[10] [3,5,6,7].

[10] setenv compile: link_error= warning .

[11] cd"~>TMP".

[12] compile"*.kl1".
```

【Hint】 If a negative number is specified as the event number, it is processed as a relative specification from the current event (number).

history

history(N) [his]

Displays the list of the history regarding the input goals which are stored in the history queue. If the positive integer N is specified, the input history from the N times before including this command input will be displayed on the screen.

【Example】 Displays the information on the last five input occasions.

```
[37] history(5).
 33  test: check( [ 100 ], start , A , B , C < D )
 34  test: check([100],start,A,B,C,D)
 35  test: check([],start,A,B,C,D)
 36  .
 37  history(5)

[38] !34.
[38] test: check([100],start,A,B,C,D) .
```

【Caution】 A period is displayed when the input is invalid.

history_length

history_length(Length)

history_length(^Old, Length) [hl]

Sets the history queue length of the listener to the value indicated by Length. If the length shorter than the current one specified as Length, the most recently input history for the specified number of queues (for its length) is maintained. The initial value is 50. If the argument is omitted, the current value is displayed, and the input from the user is awaited. Moreover, if Old is provided, the current value is unified.

prompt(FormatString)

prompt(^Old, FormatString)

Changes the prompt in the command input line according to the format FormatString. The format FormatString is provided by the string, and the characters other than the format items which start with \ are output as is. In the case where Old is given, the format string for the current prompt is output. The general form of the format item is `\C`, and the format type is specified to `C`. The initial value is `"[\!]"`. When Old is given, the format string for the current prompt is unified. The type of format is specified as follows.

- **\!** : Output the event number.
- **\d** : Output the current directory.
- **\p** : Output the default package.
- **\m** : Output the default module.
- **\M** : Output the macro module.
- **\n** : Output the line feed.
- \\ : Output the character "\" itself.

【Example】 Example of Changing Prompt

```
[5] prompt(^Old,"?- ").
 Old = "[\!] "

?- prompt(Old).

[7] sample:test(abc,Ans).
 Ans = [a,b,c]
```

take(FilePathname)

Executes the command file specified by FilePathname. If the top element is ">", it is regarded as an absolute specification. If it is "～", ("～" means the home directory.) it is to be regarded as the relative specification from the home directory, and for all the other specifications, it is to be regarded as the relative specification from the current directory. To specify the parent directory, ".." (which means to go one step upward in the directory) can be used. If the file name extension is omitted, the file that was that file name without the extension is searched. When no file with that file name is found, then ".com" is automatically added as an extension to find the command file.

- The input format of the commands is the same as that of the listener top level. It is possible all of the commands which can be executed by the listener in the command file. It is also possible to state the **take** command in the command file.
- % and /*... */ can be used for the comment.

【Caution】 When the execution of the goal specified in the command file has failed, the execution of the command file is abondanded at that moment.

【Hint】 Since the execution of the command file is processed in order, the following items can be executed.

【Example】 Example of Describing Command File

```
pushd(">sys>user>me>SAMPLE").
load("test.exe").
cd("..>PATCH").
compile( ["test1", "test2", "test3"] ).
relink( test, [ test1, test2, test3 ] ).
popd.
profile(test:go(1,Ans1)).
profile(test:go(2,Ans2)).
```

alias(Alias, Goals)

Registers Alias as the alias of the Goals. If the goal input from the listener top level identifies with Alias, it is replaced by Goals to be executed. It is no problem that the argument of the goal to be registered as Alias contains the variable. If expansion result (goal) of Alias has already been registered by the alias, it will be expanded recursively. The default setting for the times for expansion is five. Changes in the number of times for expansion are made by using the **alias_expansion** command (Page 91).

【Example1】 Registration of Alias

```
[1] alias( q(N,Ans), layered:n_queens(N,Ans) ).
  alias: q(N,Ans) => layered: n_queens(N,Ans)

[2] q(4,X).
 X = [1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]
```

【Example2】 Expansion to Multiple Goals

```
[1] alias( q(N), (q(N,Ans); display(Ans)) ).
 alias: q(N) => q(N,Ans); display(Ans)

[2] q(4)
 [1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]
```

【Caution1】 In the case where an alias which has already been registered is registered again, the older alias is deleted.

【Caution2】 Whether the input goal is identical with the alias is determined by using the Functor (name of predicate) and Arity (number of arguments). These are scheduled to be improved in the future.

【Hint】 The abbreviated listener command is realized by using this alias function.

unalias(Alias)

Deletes Alias which is registered as the alias.

【Example】 Deletion of Alias

```
[1] unalias(q(_)).
 unalias: q(N) => q(N,Ans); display(Ans)
```

alias(Alias)

Displays the expansion format of the alias.

【Example】 Display of Expansion Format of Alias

```
[1] alias(q(_,_)).
 alias: q(N,Ans) => layered: n_queens(N,Ans)
```

alias

Displays the aliases registered in the listener.

alias_expansion

alias_expansion(N)

Sets the upper limit value of the expansion times to N in the case where the expansion of the alias will be performed recursively. If 0 is set to N, expansion will not be performed at all. The initial value is five times. If the argument is omitted, the value currently set is displayed and input is awaited from the user.

errors

Lists the errors stored in the error queue of the listener. The initial value of the error queue length is 5. The following contents are listed on the screen.

```
"Error Number"  "[Event Number]"  "Exception Type"  "Exception Information"
```

【Example】Displaying List of Errors Occurred

```
[17] errors.
  1  [4]   unification_failure {{0,0},b, ...}
  2  [7]   illegal_input {{0,0},add( & ), ...}
  3  [10]  reduction_failure {{0,0}, code$nakao::...}
  4  [14]  merger_deadlock {{0,0},merge( & ), ...} ? 3
                                  % screen by typing in
 Failure>> layered: queens/4      % "error number"
    Goal > queens(1,f,begin,C)
    Node > 0
 !! Exception[3.10]: ins
```

The detailed content of the error will be displayed on the screen by typing in "error number" after "?" prompt. Pressing only the return key causes the content of the latest error to be displayed on the screen.

【Hint】This function is very useful in the following case. The error display routine has been removed by some mistake (because of the low level holophrasis) without sufficiently examining the error content after the error occurrence.

last_error

Re-displays the latest error.

error_queue_length

error_queue_length(Length)

Changes the length of the error queue in order to store the error which occurred. The default value is 5. if the argument is omitted, the current value is displayed and the input from the user is awaited.

display(?Term)

display(?Term, Depth, Length) [disp]

Displays Term on the listener window. When the terms Depth and Length which define the structure are given, the output depth limit value Depth and output length limit value Length are displayed.

date

Displays the current date and time.

suspend

Moves the listener operating on the shell window to the background of the shell. If the command is executed, the top level (prompt) of the shell appears on the screen. Note that nothing will be processed when the listener is operating on the window independent of the shell.

【Example】 Moves the listener on the shell window to the background

```
MyShell>> listener:listener
*** KL1 LISTENER (V 3.0 17-Oct-91) ***
MACRO MODULES     : [pimos]
DEFAULT PACKAGE   : pimos
CURRENT DIRECTORY : >sys>user>icot

[1] suspend.
MyShell>>                 % The prompt of the shell appears.
MyShell>> status
  2 --> stopped listener: listener
MyShell>> fore            % Returns the listener to the foreground.

[2]                       % The listener top level appears.
```

help(Type) [he]

Displays the list of commands of the type specified by Type (atom). The following items can be specified as Type.

builtin	:	Built-in predicates
basic	:	Basic commands
window	:	Listener window operation commands
varpool	:	Variable pool operation commands
env	:	Environmental commands
dir	:	Directory commands
library	:	Library commands
debug	:	Debugging commands
pg	:	ParaGraphcommands
util	:	Utility commands
all	:	All commands

help [he]

Dispalys the list of the command types which can be specified by help/1.

commands(Type) [com]

Displays the command name of the type which is specified by Type. Type is the same as that provided by **help/1**.

commands [com]

Displays the list of the command types which can be specified by commands/1.

exit [ex]

Exits to the execution one level above in the nested aprocess. When the level is zero (top level, when the level is not displayed), the listener is terminated.

4.1.6.2 Listener Window Operation Commands

【Caution】 The following commands supported by PIMOS Version 2.5 have been abolished and are not available for PIMOS Version 3.0.

title(FormatString), title(^Old, FormatString)
position, position(Position)
size, size(Size)
font, font(Pathname)
hide, close, open

In starting the listener, items such as title, position, size, and font can be set. (See Section 4.1.2.1 (Page 62).)

operator [op]

Displays the definition of all operators referred by the listener.

【Caution】 The operator of the buffer equipped with the standard input/output provided in the user program by the listener is not applied. However, the operator is referred by the listener to parse/unparse the term (goal) which is input from the listener top level.

operator(Operator) [op]

Displays the definition of the Operator which is the operator of the listener.

operator(Operator, ^Definition) [op]

Unifies the definition of Operator, which is the operator of the listener, to Definition. The format is the list of {Proceedence, Op-Type}.

add_op(Precedence, OpType, Operator)

Adds the operator of the listener.

remove_op(Precedence, OpType, Operator)

Removes the operator of the listener.

4.1.6.3 Environmental Commands

statistics [st]

Turns ON the mode which reports informaiton related to the execution of the task. The information that is reported consists of the amount of the resources consumed by the task (reductions) and the time required for execution (in the unit of milliseconds).

【Example】 When statistics is ON

```
[7] layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ... ]
 152 reductions
 432 msec
```

【Hint1】 The time to be measured is from the start to the termination (until the time terminated is reported) of the execution of the task.

【Hint2】 The built-in predicate system_timer/2 is used for the measurement of execution time.

【Caution】 A slight amount of listener overhead is contained in the number of reductions that is displayed on the screen.

nostatistics [nst]

Turns OFF the mode which reports informaiton related to the execution of the task.

【Example】 When statistics is OFF

```
[1] layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ... ]
```

statistics(ONorOFF)

statistics(^Old, ONorOFF) [st]

Changes the mode for reporting the information related to the execution of the task to ONorOFF. When Old is given, the current mode is unified.

change_macro

change_macro(MacroModuleName)

Changes the macro module for the listener to MacroModuleName. To set the multiple macro modules, specify the list of the macro module name (atom). For example, the multiple macro modules `[a,b,c]` are equivalent to :-with_macro a,b,c.— For the rules regarding the application of the multiple macros, refer to Section 7.3.4.4 (Page 266) in the Programming Volume. The macro module at the start of the listener is **pimos**. If the argument is omitted, the current value is displayed and the input from the user is awaited. If the input value does not exist or if any module which is not a macro module is specified, an error will occur. If no expansion is desired to be performed, designate 0 (intenger) to MacroModuleName. For the notes on the macro expansion of the listener, see Section 4.1.2.4 (Page 64).

【Example1】 Changes the macro module of the listener.

```
[1] change_macro.
 Current: pimos
 New    : my_macro
 pimos => my_macro
```

【Example2】 Controls the macro expansion of the listener.

```
[2] change_macro
 Current: my_macro
 New    : 0
 my_macro => Non-expansible mode
```

【Example3】 The specified module does not exist.

```
[3] change_macro
 Current: pimos
 New    : another_macro
 Nonexistent module>> another_macro
```

【Example4】 An illegal module was specified.

```
[4] change_macro
 Current: pimos
 New    : my_module
 Illegal macro module>> my_module
```

change_nodes

Changes the node available for the task which is started by the listener. The node to be utilized is specified in the format, "number of nodes from the node of Start number". The following message is displayed on the screen by executing the command.

【Example】 8 nodes from the node No. 8 are to be used.

```
[1] change_nodes.
 Start node     : 0
 Number of nodes: 16
 >Input format is as follows.
   StartNode,NumberOfNodes
  ( 0 =< StartNode =< 15,
    1 =< NumberOfNodes,
    StartNode+NumberOfNodes =< 16 )
 => 8,8                              % Use 8 nodes from the node No. 8.
```

default_package

default_package(Package)

default_package(^Old, Package)　[dfp]

Changes the default package to Package. The default package at the start of the listener is the value of **task:package**. If the argument is omitted, the current value is displayed on the screen and input from the user is awaited. In the case where Old is provided, the current value is unified. For more information on how to handle the default package of the listener, see Section 4.1.2.3 (Page 64).

default_module

default_module(ModuleSpec)

default_module(^Old, ModuleSpec)　[dfm]

Sets the default module to the module specified by ModuleSpec. The predicate which the default module is declared can be executed at the listener top level with input of module name omitted. The initial value is not set (there is no default module). If the argument is omitted, the current value is displayed on the screen and input from the user is awaited. In the case where Old is provided, the current value is unified. For more information on the format used for calling predicate in case the module name is omitted, see Section 4.1.2.5 "Calling public predicates" (Page 65) and "Calling local predicates" (Page 66).

print_depth

print_depth(Depth)

print_depth(^Old, Depth) [pd]

Sets the limit of the output depth of the structure to the value indicated by Depth. The initial value is 4. If the argument is omitted, the current value is displayed and input from the user is awaited. In the case where Old is provided, the current value is to be unified. Furthermore, when the structure to be output exceeds the output restriction value of the depth, "`&`" is displayed.

print_length

print_length(Length)

print_length(^Old, Length) [pl]

Sets the limit of the output length of the structure (number of elements) to the value indicated by Length. The initial value is 10. If the argument is omitted, the current value is displayed and input from the user is awaited. In the case where Old is provided, the current value is unified. Furthermore, when the structure to be output exceeds the output restriction value of the length, and "`...`" is displayed.

print_string_length

print_string_length(Length)

print_string_length(^Old, Length) [psl]

Sets the limit of the output number of elements in string to the value indicated by Length. The initial value is 70. If the argument is omitted, the current value is displayed and the input from the user is awaited. In the case where Old is provided, the current value is unified. Furthermore, when the number of the elements in the string to be output exceeds the output restriction value of the length, "`??`" is displayed.

print_atom_length

print_atom_length(Length)

print_atom_length(^Old, Length) [pal]

Sets the limit of the output number of print letters in the atom print name to the value indicated by Length. The initial value is 40. If the argument is omitted, the current value is displayed and input from the user is awaited. In the case where Old is provided, the current value is unified. Furthermore, when the number of letters assigned for the atom print name to be output exceeds the output restriction value of the length, "`??`" is displayed.

print_variable_mode

print_variable_mode(Mode)

print_variable_mode(^Old, Mode) [pv]

Changes the output format of the term which shows the variable to the format indicated by Mode. Mode is an atom and either name or address is given. The initial value is **name**. If the argument is omitted, it is changed by toggle (if it is currently set to name, it will be changed

to address, and if currently set to address, to be changed to name). In the case where Old is provided, the current value is to be unified.

name	:	Name mode	::	A,B,C, ...
address	:	Address mode	::	Address1$Address2

【Hint】As for the multi PSI and the PIM/m, Address1 shows the processor number where that variable is contained, and Address2 is the address in that processor.

rewrite_depth

rewrite_depth(Depth)

rewrite_depth(^Old, Depth)　[rwd]

Sets the output limit of the structure depth in performing re-display temporarily with the output limit changed during the tracing process or error processing. The initial value is set to 30. If the argument is omitted, the current value is displayed and input from the user is awaited. In the case where Old is provided, the current value is unified.

rewrite_length

rewrite_length(Length)

rewrite_length(^Old, Length)　[rwl]

Sets the output limit of the structure length in performing re-display temporarily with the output limit changed during the tracing process or error processing. The initial value is set to 30. If the argument is omitted, the current value is displayed and input from the user is to be waited for. In the case where Old is provided, the current value is to be unified.

environment　[env]

Inquires the parameter values of the environment for executing the listener. The initial values of each of the parameters are listed below.

【Example】Displaying execution environment parameters when the listener is started

```
[1] environment.
 Current directory   : "icpsi255::>sys>user>me"
 Default package     : me
 Default module      : 0
 Macro module        : [pimos]
 Alias expansion     : 5
 Statistics          : off
 Remembering mode    : off
 Display variables   : used
 Query mode          : no
 Display ancestor    : abbr
 Scope of variables  : tracer
 Print depth         : 4
 Print length        : 10
 Print string length : 70
 Print atom   length : 40
 Print variable mode : name
 Re-write depth      : 30
 Re-write length     : 30
 Error queue length  : 5
 History length      : 50
 Event number        : 1
```

Each of the files show the following meanings.

Current directory : Current directory

Default package : Default package

Default module : Default module
The default module set to 0 shows that the initial value is not set.

Macro module : Macro expansion module

Alias expansion : Time of alias expansions

Statistics : Mode whether the execution information (number of reductions and time for execution) is to be reported

Remembering mode : Mode whether the value of the variable used for execution is to be stored

Display variables : Mode to output the value of the variable when the execution is terminated

Query mode : Mode to inquire about the variable when the execution is terminated

Display ancestor : Output format for ancestor goal at spy execution

Scope of variables : Range of identity of the variable at trace

Print depth : Output depth limit of structure

Print length : Output length limit of structure

Print string length : Output limit of number of elements in string

Print atom length : Output limit of number of letters in atom print name

Print variable mode : Output format of term showing the variable

Rewrite depth : Output depth limit of structure for re-displaying with the output limit temporarily changed.

Rewrite length : Output length limit of structure for re-displaying with the output limit changed

Error queue length : Length of error queue

History length : Length of history queue

Event number : Number of commands executed since the listener started including the execution of this command.

4.1.6.4 Operation of Environmental Variables

environment(Env) [env]

Extracts and displays the value whose name is in the form of "env:×××" from the pool of environmental variables which is possessed by the task of the listener.

```
[7] env(task).
 task: io_node = 0
 task: directory = "icpsi255::>sys>user>nakao>LISTENER"
 task: user = nakao
 task: home = "icpsi255::>sys>user>nakao"
 task: package = nakao
```

printenv

Extracts and displays all the values in the pool of environmental variables which is possessed by the task of the listener.

printenv(Name)

Extracts and displays the value which uses name as a key from the pool of environmental variables which is possessed by the task of the listener.

getenv(^Status)

Extracts all the values from the pool of environmental variables which the task of the listener has, and unifies the processing results to Status. Status is one of either of the following.

normal(NameValueList) : Normal termination. If no environmental variable exists, a blank list is returned to NormalValueList. Moreover, if environmental variables exist, the list with the vector in the format of {**Name, Value**} is to be returned according to each variable. The name of the environmental variable is set as Name, and its value is stated as Value.

abnormal : Abnormal termination.

getenv(Name, ^Status)

Extracts the value whose key is Name from the pool of environmental variables which the task of the listener has, and unifies the processing results to Status. If Name is a structure, values (to the element level) should be completely materialized. Status is one of either of the following.

normal(Value) : Normal termination. Returns the value to the argument.

abnormal : The specified environmental variable does not exist.

setenv Name = Value

setenv(Name, Value, ^Status)

Sets the value specified by Value by using Name as the key to the pool of environmental variables which is possessed by the task of the listener. The processing result is unified to Status. If Name is a structure, values (to the element level) should be completely materialized. Status is one of either of the following.

normal : Normal termination.

abnormal : Abnormal termination

【Caution】It is not possible to set the value of the environmental variable which currently has the name listener: ×××.

unsetenv Name

unsetenv(Name, ^Status)

Deletes the value specified by Value by using Name as the key to the pool of environmental variables which is possessed by the task of the listener. The processing result is unified to Status. If Name is a structure, values (to the element level) should be completely materialized. Status is one of the followings.

normal : Normal termination. This result is to be returned in the case that no environmental variable exists.

abnormal : Abnormal termination.

【CautionNote】It is not possible to set the value of the environmental variable which currently has the name listener: ×××.

4.1.6.5 Variable Pool operation Commands

remember [rem]

Turns ON the mode which remembers the values of the variable that have newly appeared at the listener top level. The values of variables stored in the variable pool are bound, and can be used as many times as desired as arguments for a goal at the listener top level unless the user releases it. Even if this mode is on, those variable whose names start with an underline are not stored in the variable pool. The default value when the listener starts is set to OFF.

【Example】Mode to store the variable in variable pool

```
[1] remember.          % Sets the mode for storing the variable to ON.

[2] layered:n_queens(4,Ans).  % The variable Ans is stored.
^Ans = [1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']

[3] inspect Ans.       % The value of the variable Ans remains bound.
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

【Hint1】The variables stored in the variable pool are displayed with "^" mark immediately before their name.

【Hint2】It is also possible to store the variables in the variable pool by attaching "^" mark immediately before a variable name. For further details, see Section 4.1.3.2 (Page 77).

【Hint3】All the variables stored in the variable pool are released by inputting **a period return** to be described later in this document or by using the **forget** command.

forget [fgt]

Turns OFF the mode which remembers the values of the variable that have newly appeared at the listener top level. After this command, the values newly appearing at the listener top level will be forgotten at the time when the task is terminated. Note that the values already stored in the variable pool will remain even the mode is set to OFF.

【Example】Mode not to store the variable in variable pool

```
[1] remember.          % Sets the mode for storing the variable to ON.

[2] X=1.               % The variable X is stored.
^X = 1

[3] forget.            % Sets the mode for storing the variable to OFF.

[4] Y=2.
 Y = 2

[5] value.             % Displays the list of the stored variables.
^X = 1                 % The value of the variable X remains bound.
                       % The variable Y is not stored.
[6] .                  % Release by a period return.

[7] value.             % Displays the list of the stored variables.
                       % No variable is stored.
[8]
```

remember(ONorOFF) [rem]

Changes the mode to store the values of the variable that have newly appeared at the listener top level ONorOFF.

remember(VariableName) [rem]

Stores the variable with the name VariableName to the variable pool of the listener. If the variable whose name starts with an underline is specified, the specification is invalid.

【Example】Only the particular variables are stored in the variable pool.

```
[1] layered:n_queens(4,Ans), remember(Ans).
                          % 変数 Ans は格納される.(trans nothing)
^Ans = [1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']

[2] inspect Ans.          % 変数 Ans は値が束縛されたままである.(trans nothing
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

【Hint1】The variables stored in the variable pool are displayed with "^" mark immediately before its name attached.

【Hint2】It is also possible to store the variable in the variable pool by attaching "^" mark immediately before the variable name. For further details see Section 4.1.3.2 (Page 77).

forget(VariableName) [fgt]

Deletes the variable with the name VariableName from the variable pool of the listener.

【Example】Releases the variables stored in the variable pool individually.

```
[11] value.          % Variables X and Y are stored.
^X = 1
^Y = 2

[12] forget(X).      % Deletes the variable X from the variable pool.

[13] value.
^Y = 2               % The variable X has been deleted.

[14] X=Y.
 X = 2
^Y = 2
```

【Hint】The **forget** command can be executed from **list** command or the **query** command.

Period return

Releases all the variables stored in the variable pool. A period return is to input a null line consisting of only "." and "cr".

【Example】Releases all the variables stored in the variable pool.

```
[11] value.          % The variables X and Y are stored.
^X = 1
^Y = 2

[12] .               % Releases the stored variables.

[13] value.          % No variable has been stored.

[14] X=a, Y=b.
 X = a
 Y = b
```

display_variables

display_variables(Mode)

display_variables(^Old, Mode) [dv]

Sets the mode to display the value of the variable which has been used as for the argument of the goal at the time when the execution of the task is terminated. One the following atoms is provided as Mode; all, used, new, or no. The initial value is **used**. If the argument is omitted, the current value is displayed on the screen, and, after that, the input from the user is awaited. In the case where Old is provided, the current value is unified. The meanings indicated by each command are as follows.

all : All the variables stored in the variable pool and the ones used by that goal.
used : The variables used by that goal.
new : The variables newly used by that goal.
no : The value of the variable is displayed.

【Caution】 Any variable whose variable name starts with "_" (an underline) is never displayed .

query

query(Mode)

query(^Old, Mode)

Sets the mode to send the inquiries to the variable used as the argument for the goal at the end of executing the task. One of the following atoms is provided as Mode; all, used, new, or **no**. The initial value is no. If the argument is omitted, the current value is displayed on the screen and, after that, input from the user is awaited. In the case where Old is provided, the current value is to be unified. The meanings indicated by each command are as follows.

all : All the variables stored in the variable pool and the ones used by that goal.
used : The variables used by that goal.
new : The variables newly used by that goal.
no : Nothing is inquired.

【Caution】 Any variable whose variable name starts with "_" (an underline) is never be the object of the inquiry.

【Example】 Inquires to the variables given to the execution

```
[1] display_variables(no), query(used).

[2] current_node(_,Total), layered:n_queens(Total,Ans).
( Ans, Total )? rem Ans              % Store Ans in the variable pool.
(^Ans, Total )? ins Ans              % Inspects Ans.
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

The following operation can be used as an inquiry. The argument included in < and > can be omitted. If the specification of the variable name is omitted for an operation for which the specification of the variables is available, all the variables are to be the objects of that operation.

v		:	Displays the values of all the variables.
	<Variable name,...>	:	Displays the value of the specified
w	**<Variable name,...>**	:	Displays the value of the variable with the output limit value temporarily increased.
ins	**<Variable name,...>**	:	Inspects the specified variable.
rem	**<Variable name,...>**	:	Stores the specified variable in the variable pool.
fgt	**<Variable name,...>**	:	Deletes the specified variable from the variable pool.
pd	**<Depth>**	:	Changes the output limit value regarding the depth of the structure.
pl	**<Length>**	:	Changes the output limit value regarding the length of the structure.
pal	**<Length>**	:	Changes the output limit value regarding the length of atom print name.
psl	**<Length>**	:	Changes the output limit value regarding the strength of the string.
pv		:	Changes the output mode (name/address) of the variable.
q,	**Return**	:	Quit

value [v]

Displays the list of the variables with those values stored in the variable pool.

【Example】 Displays the list of the stored variables (assuming that the current mode is forget mode).

```
[1] ^X=1, Y=2, ^Z := X+Y.       % Only the variable with ^ is stored.
^X = 1
 Y = 2
^Z = 3

[2] value.
^X = 1
^Z = 3
```

list [l]

Lists the list of the variables stored in the variable pool.

【Example】 Displays the list of the stored variables (assuming that the current mode is the forget mode).

```
[1] ^X=1, Y=2, ^Z := X+Y.       % Only the variables with ^ is stored.
^X = 1
 Y = 2
^Z = 3

[2] list.
(^X,^Z )? fgt Z
(^X, Z )?
```

The commands which can be input after the "?" prompt are the same as the ones for the **query** command (Page 104).

4.1.6.6 Directory Commands

cd

Returns to the home directory.

cd(Directory)

cd(^Old, Directory)

Changes the current directory to Directory. In specifying the directory, if the directory starting with ">" is regarded as an absolute specification, and if it starts with a "~" mark, it is regarded as the relative specification from the home directory ("~" means the home directory). Moreover, in all the other cases, it is regarded as a relative specification from the current directory. ".." (to go one step upward directory) can be used for specifying the parent directory. In the case where Old is provided, the current value is unified.

【Example1】 The directory name which starts with a capital letter is included in quotation marks. (Note that a quotation mark is not necessary when the directory name is specified by a character string.)

```
[1] cd  >sys>user>me>'LISTENER'>'NEW'.

[1] cd ">sys>user>me>LISTENER>NEW".
```

【Example2】 When the special atoms ("~", "..") are used, insert a blank space. (The space is not necessary when the directory name is specified by a character string.)

```
[1] cd  ~ > .. >me>'LISTENER'>'NEW'.  % Insert a blank spaces.

[1] cd  "~>..>me>LISTENER>NEW".       % Space is not necessary.

[1] cd  ~ .                     % Inserts one space before the period.

[1] cd  .. > .. > me.                 % Inserts sblank spaces.

[1] cd  "..>..>me".                   % Space is not necessary.
```

pushd(Directory)

pushd(N)

pushd

Changes the current directory, and store the name of the current directory at the top of the directory stack. The specification for Directory is the same as that for the cd command. If any numeric value (positive integer) is specified, rotation processing is performed to make the Nth element become the top of the stack. However the actual starting of the stack is the 0(zero)th element. In the case where argument is omitted, the elements of 0(zero)th stack and the elements of the 1st stack are interchanged.

popd

Picks up the 0(zero)th element of the directory stack, and changes the current directory to contain the element specified by the first element. If the directory stack is empty, an error will occur.

dirs

Displays the list of the contents in the directory stack. The directory displayed at first (the 0th element) is the current directory.

pwd

Displays the current directory.

pwd(^Directory)

Unifies the current directory to Directory.

def(Atom, PathnameString)

Defines the correspondence between "logical directory name" Atom and the"directory name" PathnameString which are used by the command **con/1**. If the already defined logical directory name is redefined, the directory name is replaced.

```
[1] def(samp,">sys>user>me>SAMPLE").
 Define: samp => ">sys>user>me>SAMPLE"
```

【Hint】 The logical directory name defined on the shell which has activated the listener is succeeded at the time when the listener is activated.

【Caution】 PIMOS Version 3.0 does not support the logical directory name. The commands def and con are provided to somewhat make up the inconvenience caused by the logical directory name not being able to be used for changing the current directory. When a file system or the world management mechanism is introduced in the future, these commands are scheduled to be abolished.

def

Displays the list of the logical directory names currently set.

【Caution】 This command is scheduled to be abolished in the future.

con(Atom)

Picks up "directory name" corresponding to the"logical directory name" Atom defined by def/1, and changes the current directory to that directory.

```
[1] con(samp).
>sys>user>me>SAMPLE
```

【Caution】 This command is scheduled to be abolished in the future.

listing(Wildcard, ^Files) [ls]

Unifies the file with the name which conforms to the wild card Wildcard (character string) to Files.

delete(Wildcard)

Deletes the file with the name which conforms to the wild card Wildcard (character string) in the current directory.

deleted

Displays the file names of the deleted files in the current directory.

deleted(Wildcard)

Displays the file name which conforms with the wild card Wildcard (character string) of the deleted file in the current directory.

undelete(Wildcard)

Recovers the deleted file which conforms with the wild card at the Wildcard (character string) in the current directory.

purge

Leaves te file which has the largest version number and deletes all the other files in the current directory.

expunge

Displays the attributes of the file specified by Pathname. The specification of Pathname is the same as that for cd command.

attributes(Pathname)

Displays the attributes of the file specified by Pathname. The specification of Pathname is the same as that for cd command. The attributes to be displayed are the owner (Owner), the person who has updated the file (Update user), the day and time file generated (Created), the day and time file updated (Modified), the number of links (Link counter), the file size (File size), and the record size (Record size).

【Example】 Displaying the attributes of the file

```
[1] attributes(">sys>csp>PIMOS.DIR>pimos.conf").
 icpsi255::>sys>csp>PIMOS.DIR>pimos.conf.7
 Owner        : icpsi255!s-nakao
 Update user  : icpsi255!s-nakao
 Created      : 27-Oct-90 10:38:42
 Modified     : 27-Oct-90 10:38:42
 Link counter : 1
 File size    : 438 byte(s)
 Record size  : 0 byte(s)
```

4.1.6.7 Library Commands

predicates(ModuleSpec, Predicate, Arity)

Displays the list of the predicates with the predicate name Predicate and the number of arguments Arity in the module specified by ModuleSpec. The wild card ("*") can be used to specify the package name, the module name, the predicate name, and the number of arguments. The package has not succeeded.

info(ModuleSpec)

Displays the module information on the module specified by ModuleSpecs. The contents to be displayed cover the following items: day and time module generated, day and time linked, name of source file and day and time it was produced, name of dependent file for compilation and day and time that the file was created (such as the files which include the process was taken to), and other functions.

【Example】 Displaying module information

```
[3] info(listener_cmd_builtin).
 %%% nakao:: listener_cmd_builtin %%%
 Created: 13-Oct-1991 16:37:22
 Linked : 13-Oct-1991 16:37:22
 Source : icpsi255::>sys>user>nakao>LISTENER>NEW>com_built.kl1.1
        : 03-Oct-1991 17:08:49
 [Dependent Files]
 icpsi255::>sys>user>nakao>LISTENER>NEW>pimos.h.1 24-Sep-1991 17:28:05
```

【Caution】 The module information on the module loaded by the IPL can not be seen.

xref(ModuleSpecs)

xref(ModuleSpecs, Packages)

Displays the list of the modules in Packages referring the module (externally) by ModuleSpecs. Either a module name or a list of module names can be given to ModuleSpecs, and a package name or a list of package names can be provided to Packages. If a package name is not specified to the module name, the default package is regarded as being specified. In the care where Packages is omitted, the module in the default package is the object.

【Example】 Displaying Module Reference Relations

```
[3] xref(listener).
[nakao:: listener]
  nakao::
    listen
    listener
    listener_trace_command
    listener_exception_handler
    listener_interrupter
```

A list of the module names in the default package referring to the module listener in the default package (in this example nakao) is displayed.

4.1.6.8 Debug Commands

See Section 4.1.7 (Page 121) for various concepts related to the tracing system presented below.

trace [tr]

Enters into the trace mode. The following prompt is output in the trace mode.

【Example】Tracing mode prompt trace(Goals)

```
{Trace}
[3]
```

trace(Goals) [tr]

English nothing

【Example】English nothing

```
[1] trace( layered:n_queens(4,Ans) ).
@0@4096   1#1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)?
```

spy_reduction [sr]

Enters into the execution mode of the reduction spy. The following prompt is output in the reduction spy mode.

【Example】Reduction spy mode prompt

```
{SpyReduction}
[3]
```

spy_reduction(Goals) [sr]

Executes Goals in the reduction spy mode.

【Example】Executing reduction spy

```
{Trace}
[3] sr( layered:n_queens(4,Ans) ).
<1> layered: n_queens/2 -->
@0@4096    #1 + filter(begin,1,1,D)
                 D= begin?
```

spy_fork [sf]

Enters into the execution mode of the fork spy. The following prompt is output in the fork spy mode.

【Example】Fork spy mode prompt

```
{SpyFork}
[3]
```

spy_fork(Goals) [sf]

Executes Goals in the fork spy mode.

【Example】Executing fork spy notrace

```
[3] sf( layered:n_queens(4,Ans) ).
<1> layered: n_queens/2 -->
@0@4096    #1 queen/4
   2   + * (1)  filter(begin,1,1,D)?
```

notrace [ntr]

Allows the user to leave the trace and spy execution modes and return to the normal execution mode.

notrace(Goals) [ntr]

Suppresses the trace and spy modes to execute Goals in the normal mode.

【Example】Suppressing trace mode

```
{Trace}
[3] notrace( layered:n_queens(4,Ans) ).
 Ans = [1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']
```

【Hint】When this command is an alias, the user-defined predicate can be registered as the command of the listener. (It is possible to suppress the trace even in the trace mode.)

```
[1] alias( q4, (notrace(layered:n_queens(4,Ans)); display(Ans)) ).
 alias: q4 => notrace(layered: n_queens(4,Ans)); display(Ans)

[2] trace.

{Trace}
[3] q4.
[1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']
```

spy(ModuleSpec, Predicate, Arity)

spy(Package, ModuleName, Predicate, Arity)

Set the spy to the predicate Predicate/Arity in the module indicated by Package, ModuleName (ModuleSpec). It is possible to set the public predicate and also the local predicate. Also, a wild card "*" can be used for ModuleName, Predicate, and Arity.

【Example1】Designating a spypoint predicate by using wild cards

```
[1] spy(listener, *, *).
 Spypoint set on listener:*/*
```

【Caution】 For using the wild card in the spy setting by the current Modulename:predicate/Arity notation, the wild card should be enclosed in "(", ")". The command "/*...*/" is set to comment. If, however, "/*" is entered by mistake, close the comment with "*.".

【Example2】 Designating spypoint predicates using wild cards

```
[1] spy listener:(*)/(*).
```

An analysis of the inclusive relationship between wild cards and spypoint predicates that have already been set is performed in order to designate spypoint predicates using wild cards.

【Example3】 Analysis of inclusive relationship with wild cards spy(Predicate, Arity)

```
[1] spy(listener, *, *).
 Spypoint set on listener:*/*

[2] spy(listener, listener, *).
 There already is a spypoint on listener:listener/*
```

spy(Predicate, Arity)

Allows user to designate a spy when the default module is omitted. The wild cards can be used for Predicate and Arity.

nospy(ModuleSpec, Predicate, Arity)

nospy(Package, ModuleName, Predicate, Arity)

Cancels the spy of the predicate indicated as Predicate/Arity in the module specified by package, ModuleName (ModuleSpec). The wild card "*" can be used for Package, ModuleName, Predicate, and Arity.

【Caution】 The predicate in which the spy is set by the wild card cannot be canceled separately. To cancel it, the wild card should be used, with either the same format as when the wild card was designated, or the format which includes the wild card.

【Example】 Spying cannot be canceled.

```
[1] spy(listener, listener, *).
 Spypoint set on listener:listener/*.

[2] nospy(listener, listener, 10).
 Spypoint can't reset on listener:listener/10

[3] nospy(listener, *, *).
 %%% Reset spied pattern (listener:*/*) %%%
listener:listener/*
```

nospy(Predicate, Arity)

Cancels spy when the default module is omitted. The wild card can be used for Predicate and Arity.

nospy

Cancels all spies. Executed in the same manner as `nospy(*,*,*,*)` is executed.

spying [si]

Analyzes the setting of the spy format, and displays the list of the predicates to which the spy is set.

【Example1】Displaying the List of spying object predicates.

```
[1] spy(listener,*,*).
 Spypoint set on listener: */ *

[2] spying.
 %%% SPYING PATTERNS (1 pattern) %%%
nakao:: listener: */ *
 %%% SPYING PREDICATES (29 predicates) %%%
nakao::
  listener:
    go/ 0 (public) - nakao:: listener: */ *
    go/ 2 (public) - nakao:: listener: */ *
    go/ 3 (public) - nakao:: listener: */ *
    go/ 4 (public) - nakao:: listener: */ *
    terminate/ 8 - nakao:: listener: */ *
      ...
      ...
    prompt/ 4 - nakao:: listener: */ *
... (10/29) ...?
```

The list of the spy object predicates is displayed with 10 predicates on the screen at a time. The (public) mark is attached to the public predicate(s). Also, the setting format of the corresponding spy is displayed. The following commands can be input in the list display.

$< cr >$	:	list 10 predicates
N(>0)	:	list N predicates
q	:	quit
h	:	help

The meaning of each operation is as follows.

List 10 predicates :: Return
Keep the list display.

List predicates :: N (N>0)
Displays as many predicates as specified by N. The spy object predicates are then displayed on the screen by N.

Quit :: q
Quits the list display.

Help :: h
Help

spying(ModuleSpec)

spying(Package, ModuleName) [si]

Lists the predicates which are the objects of the spy in the module which has been specified by Package, and indicated as Predicate/Arity in the module specified by package, ModuleName (ModuleSpec). The wild card "*" can be used for Package and ModuleName. The commands which can be input to display the list are the same as those for spying/0.

【Example】 Displaying a list of spy object predicates in a specified module

```
[1] spy(*, go, *), spy(tee, *, *).
 Spypoint set on *:go/*
 Spypoint set on tee:*/*

[2] spying(tee).
 %%% SPIED PATTERNS (2 patterns) %%%
nakao:: *: go/ *
nakao:: tee: */ *
 %%% SPYING PREDICATES (2 predicates) %%%
nakao::
  tee:
    go/ 0 (public) - nakao:: tee: */ *, nakao:: *: go/ *
    loop/ 3 - nakao:: tee: */ *
```

trace(ModuleSpec, Predicate, Arity)

trace(Package, ModuleName, Predicate, Arity) [tr]

The wild card "*" can be used for Package, ModuleName, Predicate, and Arity. Sets the default trace mode (so that the tracing mode is performed by simply entering the return without any need for a particular specification) for the predicates specified by Package, ModuleName, Predicate, and Arity. The wild card "*" can be used for Package, ModuleName, Predicate, and Arity. As for the predicates to which the default trace mode is set, the goal is output with s "*" mark attached during the execution of the trace process. When the listener is activated, the default trace mode is set to all the predicates in the module belonging to the default package. When the default package is changed, the same function is available. For the further details, see Section 4.1.7.4 (Page 127).

【Caution】 The default trace mode does not influence the execution of the spy at all.

notrace(ModuleSpec, Predicate, Arity)

notrace(Package, ModuleName, Predicate, Arity) [ntr]

Cancels the default trace mode of the predicates, Package, ModuleName, Predicate, and Arity. The wild card ("*") can be used for Package, ModuleName, Predicate, and Arity.

【Caution】 Similar to a predicate to which the default trace mode is set by using a wild card, this predicate cannot be canceled individually.

notrace_all [ntr_all]

Cancels all the default trace modes. Executes in the same manner as `notrace(*,*,*,*)`.

tracing [tri]

Displays the list of the setting formats of the default trace modes.

display_ancestor

display_ancestor(Mode) [da]

Changes the display mode of the ancestor goal output in executing the spy to Mode. The ancestor goal is the goal which has specified the execution in the reduction (fork) spy mode. Mode can be selected from full, abbr, and no. The default value is **abbr**. When the argument is omitted, the current value is displayed on the screen and input from the user is to awaited.

full: Displays the argument of the ancestor goal.

```
{SpyReduction}
[3] layered:n_queens(4,Ans).
<1> layered: n_queens(4,[1*C, ...]) -->     % Ancestor goal
@0@4096    #1 + filter(begin,1,1,D)         % Spy objective goal
                 D= begin?
```

abbr: Abbr: Displays the ancestor goal in the format of ModuleSpec::Predicate/Arity.

```
{SpyReduction}
[3] layered:n_queens(4,Ans).
<1> layered: n_queens/2  -->                % Ancestor goal
@0@4096    #1 + filter(begin,1,1,D)         % Spy objective goal
                 D= begin?
```

no: Ancestor goal is not displayed, (only the goal number is displayed).

```
[3] layered:n_queens(4,Ans).
<1> -->                                 % Goal number of ancestor goal
@0@4096    #1 + filter(begin,1,1,D)     % Spy objective goal
                 D= begin?
```

inspect(?Term) [ins]

Calls the inspector from the listener top level. Inspects the arbitrary KL1 data specified by Term. This function enables the user to inspect the content of data with a complicated structure. For more information on the inspector, see Section 4.2 (Page 164).

monitor(?Term) [m]

Calls the inspector from the listener top level. Monitors and displays the materialization state of the arbitrary KL1 data specified by Term. When the variable is specified to Term, that variable print name becomes the name of Term. For more information on the monitor, see Section 4.2.5 (Page 188).

monitor(?Term, Name) [m]

Calls the inspector from the listener top level. Monitors and displays by Name the materialization state of the arbitrary KL1 data specified by Term. An optional atom, string, or the variable can be specified according to Name. For more information on the monitor, see Section 4.2.5 (Page 188).

【Caution】 When the variable is specified according to Name, the print name of that variable (string) is regarded as specified. Therefore, this variable does not refer to the variable pool of the listener, and it will not be registered.

monitor_and_fork(?Term) [mf]

Calls the inspector from the listener top level, and forks that process. The input/output of the activated monitor is switched to the new window which will be generated. When the variable is specified according to Term, that variable print name is the name of Term. For more information on the monitor, see Section 4.2.5 (Page 188).

monitor_and_fork(?Term, Name) [mf]

Calls the inspector from the listener top level, and forks that process. The input/output of the activated monitor is switched to the new window which will be generated. The arbitrary atom, string, or variable can be specified according to Name. For more information on the monitor, see Section 4.2.5 (Page 188).

【Caution】 When the variable is specified to Name, the print name for that variable (string) is regarded as specified. Therefore, this variable does not refer to the variable pool of the listener, and it will not be registered.

4.1.6.9 ParaGraphCommands Related to ParaGraph

ParaGraph is a performance evaluation supporting tool to graphically output the execution state of the parallel program. Displays the function of the in the various graphs based on the execution state information collected by using the **profile, profile_msg, and profile_node** commands of the listener. ParaGraph is activated by using the **paragraph** command from the listener top level. For more information on ParaGraph, see Section 5.2 (Page 203). Currently, the following items can be measured as execution state information.

1. The number of reductions and suspensions of each predicate which has been executed in each node.
2. Transmission/receipt information of messages between nodes and operation state in each node.
3. Calculation time, idle time, message transmission/receipt time, and GC execution time for each node.

profile(Goals)

Measures the execution state information of the program which is specified by Goals. The execution state information is collected at certain regular time intervals and saved in the listener **pool for data storage** when the execution of the program is terminated. At that moment, the data identification number, input goal string, number of total reductions required for execution and execution time are displayed on the listener window. The information to be measured is **the number of reductions and suspensions of each predicate** which has been executed in each node. The data stored can be output in graphs by using the **paragraph** command. The time interval for collecting the execution state information (cycle time) can be changed by the **interval** command. The default value is 2,000 milliseconds.

Methods of Specifying Goals

The specification method for goals is normally the same as the input method at the listener top level. The macro notation, handling of variables in the goal, specification of the execution node, default module, and the execution of the local predicate and the shell utility program can all described in the same manner. It is possible to specify multiple goals. **However, it is impossible to specify the built-in predicate or the command of the listener.**

【Example1】 Measurement of execution state information

```
[7] current_node(_,N), profile( layered:n_queens(N,Ans) ).
 1: layered: n_queens(N,Ans)    % "1" is a data identification number.
  19421 reductions
  989 msec
 Ans = [1*'[' & ']',2*'[' & ']', ... ]
```

【Example2】 Measurement of multiple goals (Measuring two or more goals at a time.)

```
[8] profile( (test:generator(100,Data), test:consumer(Data,Ans)) ).
 2: test: generator(100,Data),test: consumer(Data,Ans)
  204 reductions
  401 msec
 Ans = [0,2,4,6, ... ]
 Data = [0,1,2,3,4, ... ]
```

【Example3】 Measurement of multiple goals (Measuring two or more goals separately.)

```
[9] profile(test:generator(100,Data)), profile(test:consumer(Data,Ans)).
 3: test: generator(100,Data)
  103 reductions
  231 msec
 4: test: consumer(Data,Ans)
  103 reductions
  331 msec
 Data = [0,1,2,3,4, ... ]
 Ans = [0,2,4,6, ... ]
```

【Example4】 Specification of execution node, omission of default module, and execution of local predicate

```
[10] profile((generator(_D),consumer(_D,_)@node(3))),
     profile(layered!queens(1,8,begin,_)).
 5: genarator(_D),consumer(_D,_)@ node(3)
  204 reductions
  398 msec
 6: layered! queens(1,8,begin,_)
  19421 reductions
  1024 msec
```

Data identification number

When the execution of the goal is terminated, the number to identify the data, input goal string, number of reductions, and execution time are output. The data identification number is unique for one listener. If two or more **profile** commands are executed at one time, these commands will be allocated in the order of input. The commands will also be allocated when the execution state information is loaded from the **History manipulator** of the ParaGraph.

Occurrence of Errors

When any errors occur with the goal under measurement for execution state information, such errors will be reported. If the execution is aborted by a command resulting from an error, the execution state information which existed up until the erroneous command will be stored in the pool for data storage.

【Example5】 Occurrence of Errors

```
[11] profile( layered:n_queens(f,Ans) ).
 Failure>> layered: queens/ 4
    Goal > queens(1,f,begin,Ans)
    Node > 0
 !! Exception (13452 msec): a
 7: layered: n_queens(f,Ans)
  44 reductions
  1351 msec
```

Interruption and abortion of execution

Interrupting execution by inputting Control-C during the measurement of the execution state information and aborting the execution by using **abort** command allows the user to acquire the execution state information up to the point of such interruption.

【Example6】 Interrupting execution during measurement

```
[11] profile( test:loop ).
 ?? Interruption (37564 msec): a
 8: test: loop
  848 reductions
  1228 msec
 Execution aborted !!
```

【Hint】 When the execution of the user program causes an infinite loop, aborting the execution of the profile command during execution and outputting the execution results allows the user to detect the looped predicate.

【Hint】 The execution state information stored in the listener can be deleted and saved in the file by using the commands which are supported by ParaGraph. Also, it is possible to load the data saved in the file.

【Caution1】 Whenever the **profile** command is executed, the execution state information is saved in the pool for storage. Therefore, it is desirable to judiciously delete the unnecessary data.

【Caution2】 When the listener is terminated, the execution state information stored in the listener disappears. Important data should be saved in the file before the listener is terminated.

profile_msg(Goals)

Measures the transmission and receipt information **of messages between nodes and the operation state of each node in executing the program specified by Goals**. The method of specifying the goal, and the operating method for interrupting or aborting an execution are the same as those for the **profile** command.

profile_node(Goals)

Measures the calculation time, idle time, message transmission/receipt time, and GC execution time for each node in executing the program specified by Goals. The method of specifying the goal, and the operation method of interrupting or aborting an execution are the same as those for the **profile** command.

paragraph [pg]

Start ParaGraph. Performs the graph output for the execution state information which has been measured by the **profile, profile_mag, and profile_node** commands. For information on how to operate ParaGraph, see Section 5.2 (Page 203).

【Hint】 If ParaGraph is executed in the background, data measured after the activation of ParaGraph can be obtained by using the **renew** command of the **History manipulator** when necessary. Thus, this function is convenient since it allows users to avoid the necessity of starting Paragraph at the time of each execution.

interval

interval(Interval)

Changes the time interval for correcting the execution state information (cycle time) to the value specified by Interval. The collection time interval is specified in milliseconds. If the collection is not to be performed until the execution is terminated, specify the atom infinite. The default value when the listener is started is 2,000 milliseconds.

4.1.6.10 Utility Commands

number(Integer)

Displays the integers specified by Integer in binary, octal digit, decimal, and hexadecimal numbers.

【Example】

```
[1] number(~(16'FFFFFFFF xor exception_tag#trace)).
 Binary  : 11011111111111111111111111111111
 Octa    : 33777777777
 Decimal : 3758096383 (-536870913)
 Hexa    : DFFFFFFF
```

calculator [calc]

Activates the calculator. When the calculator is started, the prompt "!" is displayed on the screen. An optional computational expression (or arbitrary integer) can be input. The calculator is terminated by merely entering the return.

【Example】

```
[1] calc.
>Hexa Calculator(? for help)
! (10+2'100)+16'FF/3>>8'7
 Binary  : 1110
 Octa    : 16
 Decimal : 14
 Hexa    : E
! 16'FFFFFFFF xor exception_tag#trace
 Binary  : 11011111111111111111111111111111
 Octa    : 33777777777
 Decimal : 3758096383 (-536870913)
 Hexa    : DFFFFFFF
!
```

in(Stream)

Acquires the standard input stream of the listener.

【Hint】This function is convenient for the case in which the protocol of the input filter (buffer:input_filter) is attempted.

out(Stream)

Acquires the standard output stream of the listener.

【Hint】This function is convenient in the case in which the protocol of the output filter (buffer:output_filter) is attempted.

inter(Stream)

Acquires the standard interaction stream of the listener.

【Hint】This function is convenient for the case in which the protocol of both the input and output filters (buffer:interaction_filter) is attempted.

4.1.6.11 Shell Utility Programs

It is possible to execute the shell utility program as a command from the listener top level, in a manner similar to the shell execution. This function enables the following items to be activated as the command; compiler, linker, loader, unloader, all of which are essential for the programming environment, and the variable checker as tools for the static analysis of the KL1 program.

- The predicate with the predicate name "**go**" can be executed by merely specifying the module name and its argument, and omitting the predicate name.

  ```
  ModuleName( Arg, ... ) or Package::ModuleName( Arg, ... )
  ```

  【Example】Execution of a shell utility program

  ```
  [1] compile("*.kl1").
  ```

- **The goals which have been executed with the predicate name "go" omitted are not the objectives of trace and spy.** In the case where tracing or spying are performed, the predicate name "**go**" should be included in the specification.

【Example1】 When the predicate name "go" is omitted, trace is not available.

```
{Trace}
[3] varchk.
*** KL1 Variable Checker ***
VARCHK[o,l,v]
```

【Example2】 When the predicate name "go" is input, trace is available.

```
{Trace}
[3] varchk:go.
@0@4096   1   varchk: go
   2   + * (1)  pimos:: shoen: raise(256,get_std(in), ... )
   3   + * (2)  pimos:: shoen: raise(256,get_std(out), ... )
   4     * (3)  variable_check(ready,B, ... )?
```

- If the goal specified with the predicate name "**go**" omitted conflicts with the commands of the listener or the specification of the goal with the default module omitted, it will be recognized in the following priority order.

  1. Command of the listener or built-in predicate
  2. Shell utility program
  3. Goal input with the module name omitted by the default module

4.1.7 Tracer

4.1.7.1 Outline

The important feature of a function which traces execution in order to debug a program is that it suppresses the tracing of those portions that are not necessary, thereby allowing the concentration of attention on the target for debugging.

In ordinary sequential languages, the execution of one portion of the program can not begin until the execution of another portion of the program that was started from a location within the program has been completed. Because of this, if the entire executed portion is traced at suitable time intervals during execution, it is possible to perform a trace that is limited to programs that were started from a certain location.

In the programs of kl1, a parallel language, all goals are conceptually executed in parallel. In actuality, execution is also performed from the goal for which all the necessary data is present. As a result, it is not possible to realize the desired suppression of tracing in tracing systems in which the time interval is limited.

Therefore, in the tracer used here, a tracing system is realized in which execution is traced by focusing attention on only the parent-child relationship of the goal. In this system, execution proceeds while suppressing those traces for branches of the goal for which tracing is unnecessary in the tree structure of the calling relationship.

In this tracer, "**reduction**" of the goal (replacing with a child goal specified in the body as a result of application of a program clause for which execution of the goal by the guard part is successful) is the target of tracing. Therefore, execution of tracing proceeds for every completion of one reduction in the form of reporting the child goal (sub-goal) which resulted from that reduction. In addition to user-defined predicates, body built-in predicates and unification are also included in the child goal that is reported. Consider the program indicated below. いま,

```
a :- b, c.
b :- d, e.
c :- f, g, h.
d.
e :- g, h.
f :- g, h.
g :- h.
h.
```

The execution process for this program is in the form of thefollowing tree structure:

$$a \rightarrow \left\{ \begin{array}{l} b \rightarrow \left\{ \begin{array}{l} d \\ e \rightarrow \left\{ \begin{array}{l} g \rightarrow h \\ h \end{array} \right. \end{array} \right. \\ c \rightarrow \left\{ \begin{array}{l} f \rightarrow \left\{ \begin{array}{l} g \rightarrow h \\ h \end{array} \right. \\ g \rightarrow h \\ h \end{array} \right. \end{array} \right.$$

Here, suppression of tracing means to specify not to perform tracing for arbitrary portions of the tree within this tree structure.

Trace

Trace When the goal has been reduced, reports its contents (child goal).

【Example】 Execution example of tracing

```
{Trace}
[7] sample:qsort([5,1,4,2,3],List).
@0@4096   1#1   sample: qsort([5,1,4,2,3],List)
   2     * (1)  append(B,[5|C],List)
   3     * (2)  qsort(D,B)
   4     * (3)  qsort(E,C)
   5     * (4)  part([1,4,2,3],5,D,E)?
```

The reduction of network:go/1 that was executed in the trace mode is displayed.

Spy

In this tracer, spy refers to reporting the reduction or a particular predicate only. In this tracer, two types of spies are realized.

Reduction spy

This is a spy which reports a reduction when the goal that is the target of the spy is reduced. Therefore, in this spy, when the goal that is the target of the spy is suspended, that reduction is not to be reported so long as that suspension is not canceled.

【Example】 Execution of a reduction spy

```
[11] spy(layered,colors,*).

[12] sr.

{SpyReduction}
[13] layered:map(Ans).
<1> layered: map/1 -->
@0@4096    #1 + colors(B)
                 [red,yellow,blue,white]=B ?
```

When the goal corresponding to layered:colors/1 for which the spy is set has been reduced, the content of that reduction is reported.

Fork spy

This spy reports reduction when the goal, that is the target of the spy and the goal of the body, is reduced. In other words, it reports the goal of the parent goal of the spy objective goal.

【Example】 Execution of a fork spy

```
[11] spy(layered,colors,*).

[2] sf.

{SpyFork}
[3] layered:map(Ans).
<1> layered:map(A) -->
@0@4096       area/3
   2   + * (1)  colors(B)?
```

When the goal of layered:area/3, which has the goal corresponding to layered:colors/1 as the child goal, for which the spy is set, is reduced, the content of that reduction is reported.

【Caution】 In the case in which the fork spy is executed, it is not possible to know the status of the argument of the goal (parent goal) which is directly calling the detected spy goal or brother goals of the goal due to reasons related to implementation.

4.1.7.2 Start and End of Tracing

The following two methods are available to activate the tracer.

1. Changes the execution mode of the listener top level. The execution level is changed by one of the following commands: **trace** (trace mode), **spy reduction** (spy reduction mode), or **spy_fork** (spy fork mode).

2. Executes a goal by one of the following commands: **trace(Goals), spy_reduction(Goals), or spy_fork(Goals)**.

For detailed information on the above-mentioned commands, see section 4.1.6.8 (Page 110). To terminate tracing, either the trace command **a**(aborting execution) or **ntr**(trace termination) is executed.

4.1.7.3 Output of Trace

Note regarding trace output

The trace output will not always agree with what was described by the user with respect to the following items.

1. There are cases in which the execution sequence of the built-in predicates and unification may be changed by the compiler.

2. There are cases in which the built-in predicates that have been described by the user may be expanded into the multiple built-in predicates or a combination of unifications by the compiler.

   【Example】 Expansion of a built-in predicate

   ```
   X := Y+Z*R
   --> A := Z*R, B := Y+A, X=B.
   ```

3. Arithmetic operations are output in macro notation.

   (a) Integer operations

   ```
   add(X,Y,Z)
   --> Z := X*Y
   ```

   (b) Floating point number operations

   ```
   floating_point_add(X,Y,Z)
   --> Z $:= X+Y
   ```

4. Variable names are generated within the tracer. Variables are noted with a capital letter as the first letter. Although the identity of variables having the same name is guaranteed within a tracer, in the event that the garbage collector is started during the tracing process, the same variables will be displayed with the different names.

   【Hint】 It is possible to change the range for the identity check of the variables to the unit of trace output at one time (unit of 1 clause) by using the **scope** command of the tracer. This function can realize the faster trace output and easy-to-read change numbers by avoiding the possibility of becoming a large number (such as 49 of "A49"). Such are the merits in using it.

Trace output

The following types of outputs can be performed with the trace function.

```
@0@4096  20#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)          (Example 1)
                  X=[2*H1|I1]
                  J1:=2+1                                      (Example 2)
   30  + * (1)  filter([1*L,2*T,3*E1|F1],2,1,H1)
1  31    * (2)  queen(J1,4,[1*L,2*T,3*E1|F1],I1)?               (Example 3)

<2> layered: queen/4 -->                                       (Example 4)
@0@4096    #1 + filter(begin,1,1,D)                            (Example 5)
                  D=begin?

<3> layered: queens/4 -->
@0@4096       queen/4                                          (Example 6)
   4   +   (1)  filter(C,1,1,D)?

@0@4096   6#2   job:create(createJob( & ,Q, ... ))
   11    * (1)  @node(0) <= create1(createJob( ... ),Q, ... ) ? (Example 7)

@0@4096   1#3   queenw:queen(4,[1,2,3,4],A,4,0,1)
                  calculate_priority_from_current(-3000,B)
   1       (1)  @pri(B) => queen_n([1,2,3,4],[],[],A,4,0,0,4,1)?(Example 8)

@0@4096   1     listener:go(mouse,mouse, ... )
                  calculate_priority_from_minimum(4092,A)
                ....
   10    * (9)  @pri(A) => get_command(prompt,K, ... )?         (Example 9)

[37-62]@0@4096  79#3   inspector_io_manager:manager(N6,M4, ...) (Example 10)
   87    * (1)  manager(N6,M4, ... )?
```

【Example1】 Trace goal

```
@0@4096  20#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)
(a) (b) (c)(d)            (e)
```

【Example2】 Body built-in predicate

```
        J1:=2+1
          (f)
```

【Example3】 Child goal

```
1  31    * (2)  queen(J1,4,[1*L,2*T,3*E1|F1],I1)?
(g)(c)  (h)(i)              (j)               (k)
```

【Example4】 Ancestor goal

```
<2> layered: queen/4 -->
(l)           (m)
```

【Example5】 Goal detected by reduction spy

```
@0@4096    #1 + filter(begin,1,1,D)
             (n)       (o)
```

【Example6】 Parent goal detected by fork spy

```
@0@4096        queen/4
                 (p)
```

【Example7】 Child goal with node specification

```
11     * (1)  @node(0) <= create1(createJob( ... ),Q, ... ) ?
                (q)
```

【Example8】 Child goal with specification of execution priority (Relative self - specification in the Shoen to which it belongs)

```
                  calculate_priority_from_current(-3000,B)
1        (1)  @pri(B) => queen_n([1,2,3,4],[],[],A,4,0,0,4,1)?
                (r)
```

【Example9】 Child goal with specification of execution priority (Rate specification in the relevant Shoen)

```
                  calculate_priority_from_minimum(4092,A)
                ....
10     * (9)  @pri(A) => get_command(prompt,K, ... )?
                (r)
```

【Example10】 Goal in user defined task

```
[37-62]@0@4096  79#3   inspector_io_manager:manager(M6,M4, ... )
  (s)
```

The meanings for each field on the trace displays stated in the above examples are as listed below.

(a) Node number: The number of the node that reduced the goal.

(b) Priority: The priority of the goal in the relevant Shoen (proportioned within the affiliated domain).

(c) Goal number: The goal identification number. The number that is given exclusively to the goal by the tracer during the execution of the goal in the trace mode. Management of these numbers allows the user to know which goal has been reduced.

(d) Clause number: The number of the clause (1 origin) committed by the goal. This information is added when compiling is performed in the profile mode ON state.

(e) Trace goal: The reduced goal.

(f) Body built-in predicate

(g) Tracer number: The number of the clause (1 origin) committed by the goal. This information is added when compiling is performed in the profile mode ON state.

(h) Trace mark: The mark to indicate that a trace is specified for a child goal.

(i) Child goal serial number: A serial number that is given to child goal of the trace goal. This is used for specifying child goal by means of the trace command.

(j) Child goal: Child goals are displayed with the module name omitted when the trace goal and the module name are the same.

(k) Prompt: Indicates that the tracer is waiting for the input of the trace command.

(l) Ancestor goal number: The goal number of the ancestor goal. It is used in the same manner as explained in (c).

(m) Ancestor goal: The goal corresponding to the ancestor of the goal that was detected by the spy (this may be the parent in some cases). In other words, it is regarded that the user executed this goal in the spy mode.

(n) Spy mark: This mark is attached when the goal corresponds to the spypoint predicate.

(o) Spy goal: The spypoint goal that was detected as a result of executing the reduction spy.

(p) Parent goal: The goal corresponding to the parent of the goal that was detected by the fork spy (the goal that is calling the spypoint goal in the body). The display represents the predicate name and the number of arguments. The parent goal is displayed with the module name omitted when the ancestor goal and the module name are the same.

(q) Destination node: This is displayed when the child goal is a goal with a node specified.

(r) Execution priority specification: This is displayed when the child goal is a goal with the execution priority specified. The tracer displays the predicates indicated below together with the execution priority specification corresponding to the specification method for the execution priority.

- Case of relative - self specification belonging Shoen domain:

```
calculate_priority_from_current(Rate,X)
...
@pri(X) => Goal
```

- Case of relative - self specification belonging Shoen domain:

```
calculate_priority_from_minimum(Rate,X)
...
@pri(X) => Goal
```

【Caution】 In the case where a current goal with the execution priority specified is detected by the fork spy, the predicate which expresses the specification method of the execution priority is not displayed. This is scheduled to be modified in the future.

(s) Task (Shoen) identifier: The identifier to distinguish the task to which the goal reported by the trace (spy) belongs. The number in the identifier is the goal number of the goal (normally shoen:raise/3) which has generated that task. The line formed by the goal numbers shows the depth of the nest.

4.1.7.4 Default Trace Mode

Whether or not to re-trace the execution of the child goal of the goal reported by trace or spy can be specified individually at the time of executing the trace process. To select the default, it is possible to set the mode to each package, each module and each predicate by using a **trace** command.

The child goal to which the default trace mode is set can be traced by entering the return only. For such a child goal, the trace mark ("*") is attached when it is displayed at the trace output.

When the listener is activated, the predicates in the module which belong to the default package are all set to be traced in default. Also, in the case where the default package is changed, the default trace mode is automatically set to the predicate in the module belonging to the changed package.

【Example】 Cases where the default trace mode is set to the module tee.

```
[7] tee:go.
@0@4096   1   tee: go
   2       (1)  pimos:: shoen: raise(256,get_std(in), ... )
   3       (2)  pimos:: shoen: raise(256,get_std(out), ... )
   4       (3)  pimos:: shoen: raise(256,get_std(mes), ... )
   5     * (4)  loop(A,B, ... )?       % Inputs return key
@0@4096   5   tee: loop("abcdefg",C, ... )
                C=[prompt("TEE>"),getl(D)|E]
                  ...
```

Since the default trace mode is set to the predicate `loop(A,B,...)` in the module tee, it is displayed with the trace mark ("*") attached. If the return is entered at this moment, only the goal `loop` of (4) is traced and the predicate of the package pimos is not to be traced.

【Caution】 The default trace mode does not influence the execution of the spy at all.

4.1.7.5 Displaying Close Number

If the compiling process is executed with the profile mode ON, the number of the clause (clause) committed (selected) by the goal is displayed on the screen when the predicate in the generated module is traced. The clause number is 1 origin and is allocated to the upper clause and the lower clause in their description order in the program. The clause number is displayed in the format of "**#clause number**" after "goal number" under the trace output.

```
@0@4096  20#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)
                  X=[2*H1|I1]
                  J1:=2+1
   30  + * (1)  filter([1*L,2*T,3*E1|F1],2,1,H1)
```

"**#2** in the first line is the number of the clause committed. To switch the profile mode to ON, execute the **!set** command on the interactive compiler as follows.

```
[7] compile.
** KL1 Compiler **
COMPILE> !set compile:profile on
compile: profile := on
COMPILE>
```

Similarly, **compile:profile := off** allows the user to reset the profile mode. For information on the profile mode of the compiler, see Section 3.1.3.4 (Page 47).

【Hint】 The **!keep** command enables the environment which has been set by the interactive compiler to be saved. The file name is "**compile.param**". Thereafter, the initial setting of the compilers (both interactive and patch) is to be performed according to the content of this file when the compiler is activated.

【Caution1】 The clause number is not displayed to the parent goal which is reported by the fork spy because of the usefulness of mounting.

【Caution2】 The execution speed of the program is influenced little by compilation in the profile mode.

4.1.7.6 Automatic Trace and Automatic Spy

When debugging by using the tracer, it often happens that the return has been entered many times continuously, or debugging has been performed in the spy mode repeatedly to confirm the execution state of the specific predicate. In such cases, the automatic trace and automatic spy functions can be utilized. The steps to input commands in executing trace can be omitted by using these functions. Also, the execution log can be easily recorded.

Automatic trace
: Repeatedly executes the **return or c** command (to continuously perform the execution according to the trace mode) as many times as specified. Consequently, only the goal(s) with the default trace mode ("*") attached will be traced. See Section 4.1.7.9 "**at** (Auto Trace) command" (Page 132) for an execution example.

Automatic spy
: Repeatedly executes **the s** command or **sr** command (to execution all the child goals by the reduction spy) repeatedly as many times as specified. This function is very useful to trace only the specific reduction(s). See Section 4.1.7.9 "**as** (Auto reduction-Spy) command" (Page 136) for an execution example.

4.1.7.7 Trace Using Multiwindow

As already described in the outline of this chapter, all execution of the programs written in kl1 is conceptually in parallel. Therefore, the debugging of such programs in such state would be extremely difficult, since the execution information on the multiple processes are reported randomly and mixed. One of the methods of avoiding this problem is to output the trace information after it is classified and readjusted by the appropriate unit of the processes. The trace method utilizing the multi-window provides this function. The user can optionally activate the tracer which is dedicated to each process (goal). If these processes are executed individually in parallel, the output for each tracer will be performed in parallel. Control of the execution (scheduling) can be easily realized by merely changing the input order of the commands to the individual tracer.

Specification of tracer
: Specifies the tracer number through the **f** (Fork) command to set the tracer to perform the trace output of the child goal. The tracer number is the arbitrary positive integer to identify the individual tracer. The tracer number of the tracer which is activated in the basic listener unit is 0 (zero). It is also possible to specify the tracer which already exists. In that case, the trace output is to be merged. For an example of executing the command, see Section 4.1.7.9 (Page 137).

Tracer Window
: If the child goal for which a tracer other than the trace specified to the basic unit of the listener is reduced, a tracer window is generated when such tracer has not been created. The tracer window is set to the appropriate size in advance, and the user can specify the desired position by using the mouse. The tracer number is displayed on the label of the generated tracer window as follows.

```
1: CHILD LISTENER
```

"1" of "1:" is the tracer number. "CHILD LISTENER" is the title of the window.

Termination of tracer

When the execution of the task is terminated, the following characters are output to the tracer window.

```
!!! Return to exit. !!!
```

The tracer then disappears if return is entered.

Items to be noted

1. The default trace mode and the spy setting predicate in the environment of the basic listener unit are succeeded.
2. Also, the environment of the basic listener unit is succeeded for the environmental variables (including the current directory and the default package). Thereafter, these settings for the environment can be changed for each tracer individually.

4.1.7.8 Trace Within a Task (Shoen)

When the user generates the task (Shoen) in the program, the functions of trace and spy can be executed in that task as well. The specification for the trace (spy) in the task is performed to the following predicate which generates the task (Shoen).

pimos::shoen:raise(pimos_tag#task,general_request,GR)

Tracing is applied when the **execute** message reaches the stream GR for the general request device.

pimos::shoen:execute/7

pimos::shoen:profile/7

【Example】 Execution of trace in task

```
@0@4096   3   test:create_task(C,E, ... )
                GR=[execute(C,Arg, ...)],
   7       (1)  pimos::shoen:raise(pimos_tag#task,general_request,GR)
   8     * (2)  monitor(Rpt,Ctr, ... )? 1          % Trace (19 and (2).
    ...

    % It is possible to trace in the task by specifying the trace
    % to the goal which will acquire the general request.

[7]@0@4096   7   test:create_child_task(A1,B1, ... )
                GR1=[execute(C1,D1, ...)],
   23      (1)  pimos::shoen:raise(pimos_tag#task,general_request,GR1)
   24    * (2)  new_monitor(J1,K1, ... )? sr 1     % Execute (1) by spy.
    ...                                           % Trace (2)

    % It is possible to spy in the task by specifying the spy
    % to the goal which will acquire the general request.

<23> pimos::shoen:raise/3 -->
[7-23]@0@4096    + layered:filter([],X1, ... )
   35    * (1)  filter([],Y1, ... )
    ...
```

The identifier to indicate the task to which that goal belongs is displayed in the trace output of the goal under execution in the task. In the above example "[7]" and "[7-23]" are the identifiers of the task. The number in the identifier is the goal number of the goal which has generated such task. The line formed by the goal numbers indicates the depth of the task nest.

"[7-23]" indicates that the child task has been generated by the execution of the goal with the goal number 23 (shoen:raise/3) in the task which was generated by the execution of the goal with the goal number 7 (shoen:raise/3). The goal which reported the tracing by the reduction spy (layered:filter) is executed in the task with the goal number 23.

【Caution】 The trace bit of the mask pattern for exception processing of the task which is generated by the user is forcefully set to **OFF** while the trace function is performed.

4.1.7.9 Trace Commands

The following portion describes the commands that can be input after the display during the trace processing. The meaning of the heading for each command will be indicated below.

```
Command name :: Input format
```

The argument enclosed in "<", ">" indicates that it can be omitted.

Continue :: Return or c

Continues execution. Execution is performed in accordance with the trace mode that is set for the child goal. Only the goal with the trace mark ("*") attached is to be traced.

【Example1】 Tracing all the child goals.

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? <cr>
```

The child goals **(1) and (2)** will be traced.

【Example2】 All the child goals will not be traced.

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)? <cr>
```

Neither the child goal **(1) nor (2)** will be traced.

【Example3】 Only the goal with the trace mark ("*") will to be traced.

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? <cr>
```

The child goal **(2)** is to be traced, though the tracing of the child goal **(1)** will be suppressed.

Auto trace :: at COUNT

Continuously executes tracing as many times as specified by COUNT (positive integer). This means that the **return** is entered as many times as specified by COUNT. Therefore, only a goal with the trace mode ("*") set is to be traced.

【Example】 Automatic trace

```
@0@4096   1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)? at 3      % Performs automatic
@0@4096   2   layered: queens(1,4,begin,Ans)     % trace three times.
                 B:= 1+ 1
   3     * (1)  queen(1,4,begin,C)
   4     * (2)  queens(B,4,C,Ans)? c % 2           % Two more time
@0@4096   3   layered: queen(1,4,begin,C)
                 C=[1* D |E]
                 F:= 1+ 1
   5     * (1)  filter(begin,1,1,D)
   6     * (2)  queen(F,4,begin,E)? c % 1          % One more time
@0@4096   5   layered: filter(begin,1,1,D)
                 D= begin?                % Waiting for command input
```

No trace all and Continue :: n

Tracing of all the child goals is suppressed regardless of whether the trace mode ("*") set to the child goals.

【Example】 Execution of tracing of all the child goals suppressed.

```
@0@4096  25   test:go(IN,Status)
   26      (1)  pimos::buffer:output_filter(IN,C)
   27    * (2)  display(C,D)
   28    * (3)  check_status(D,Status)? n
```

The child goals **(1), (2), or (3)** will not be traced regardless of whether the trace mode already set. The trace mode set to the child goals are executed with the previous settings reversed (if the trace mode is On, it is executed as if set to OFF, and if OFF, it is performed as if set to ON) by the **x** command (Toggle trace mode all and Continue) to be described later in this document.

【Hint】 In such the case stated in the above example in which the predicate provided by the PIMOS and the user defined predicate are both called in the body part, the predicate provided by the PIMOS is traced. It is convenient to use **n** command to suppress the trace execution of all the displayed child goals.

Toggle trace mode all and Continue :: x

Continues the trace execution with the trace mode set to the child goals reversed (if the trace mode is On, it is executed as if set to OFF, and OFF it is performed as if set to ON). Only the goal without the trace mark ("*") is to be traced.

【Example1】 Tracing of all the child goals is suppressed.

```
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? x
```

Neither the child goal **(1) or (2)** is to be traced.

【Example2】 Tracing of all the goals.

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)? x
```

The child goals **(1) and (2)** are both to be traced.

【Example3】 Only the goal without the trace mark ("*") is to be traced.

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? x
```

The child goal **(1)** is to be traced though tracing of the child goal **(2)** is to be suppressed.

Toggle trace mode selectively and Continue :: N1,N2,...

Toggle trace mode selectively and Continue :: N1,N2,... Reverses the trace mode individually and continue the execution. Execution is performed with the trace modes of the child goals specified with the child goal serial numbers n1, n2,... reversed. The child goals that are not specified are executed in accordance with individually set trace modes.

【Example1】 Canceling the trace mode individually and continuing execution.

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? 1,2,3
```

The child goals `(4), (5) and (6)` are to be traced though tracing of the child goals `(1), (2) and (3)` are to be suppressed.

【Example2】 Specifying the trace mode individually and continuing the execution.

```
@0@4096   1   layered:map(A)
   2       (1)  area(ok,begin,B)
   3       (2)  area(on(ok),B,C)
   4       (3)  area(on(on(ok)),C,D)
   5       (4)  area(on(off(on(ok))),D,E)
   6       (5)  area(off(on(on(ok))),E,F)
   7       (6)  area(on(on(on(ok))),F,A)? 1,2,3
```

The child goals `(1), (2) and (3)` are to be traced though tracing of the child goals `(4), (5) and (6)` are to be suppressed.

Toggle trace mode all :: t

Reverses the setting mode of tracing for all the child goals. This does not perform the execution but only reverses the trace mode.

【Example】 Reversing the setting mode of tracing

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? t
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)?
```

The trace modes of all the child goals have been reversed.

Toggle trace mode selectively :: t N1,N2,...

Toggle trace mode selectively :: t N1,N2,... Reverses the trace mode for the child goals specified for the child goal serial numbers N1, N2,.... It does not perform the execution but only reverses the trace mode.

【Example】 Reversing the trace mode individually

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? t 1
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)?
```

The specification of tracing set to the child goal (1) has been canceled.

Set reduction-spy mode all and Continue :: s or sr

All the child goals are executed by the reduction spy. Those descendent goals that are targets of the spy are not to be traced until they are reduced.

【Example】 Executing all the child goals by the reduction spy.

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? s
```

All the child goals are executed by the reduction spy.

【Caution】 When the child goal of the spy target (the goal with the spy mark "+" attached) is executed by the reduction spy, the reduction of that own goal is reported. This has the same function as the execution of the trace mode.

```
@0@4096  37   listener_utility: integer_to_string(123,X)
   38   + * (1)  i_to_s(123,"",X)? s     % Since the spy is set to goal
@0@4096  38 + listener_utility: i_to_s(123,"",X)
                                  % number 38 listener_utility:i_to_s/3,
                  B:= 123 mod 10
                  C:= 123 / 10
   39    * (1)  integer_to_string(B,D)
   40      (2)  pimos:: builtin_utility: append_string(D,"",E)
   41  + * (3)  i_to_s(C,E,X)?
```

Since the spy is set to goal number 38 listener_utility:i_to_s/3, the reduction of this goal is to be reported if this goal is executed by the spy reduction.

Set reduction-spy mode selectively and Continue :: s N1,N2,... or sr N1,N2,...

Those child goals specified by the child goal serial numbers N1, N2,... are executed by the reduction spy. Those child goals that were not specified are executed in accordance with the status of the trace mode.

【Example1】 Specifying the reduction spy individually and performing execution

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? s 1,2,3
```

The child goals `(1), (2) and (3)` are to be executed by the reduction spy, and the child goals `(4), (5) and (6)` are to be traced.

【Example2】 Specifying the reduction spy individually and performing execution

```
@0@4096   1   layered:map(A)
   2       (1)  area(ok,begin,B)
   3       (2)  area(on(ok),B,C)
   4       (3)  area(on(on(ok)),C,D)
   5       (4)  area(on(off(on(ok))),D,E)
   6       (5)  area(off(on(on(ok))),E,F)
   7       (6)  area(on(on(on(ok))),F,A)? s 1,2,3
```

The child goals `(1), (2) and (3)` are to be executed by the reduction spy, and the child goals `(4), (5) and (6)` are not to be traced.

Set fork-spy mode all and Continue :: sf

All the child goals are executed by he fork spy. The trace is to be reported when the descendant goals to be traced has been generated.

【Example】 Executing all the child goals by the foal spy

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? sf
```

All the child goals are executed by the fork spy.

Set fork-spy mode selectively and Continue :: sf N1,N2,...

Those child goals specified by the child goal serial numbers N1, N2,... are executed by the fork spy. Those child goals that were not specified are executed in accordance with the status of the trace mode.

【Example1】 Specifying the fork spy individually and performing execution

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? sf 1,2,3
```

The child goals `(1), (2) and (3)` are to be executed by the fork spy, and the child goal `(4), (5) and (6)` are to be traced.

【Example2】 Specifying the fork spy individually and performing execution

```
@0@4096   1   layered:map(A)
   2       (1)  area(ok,begin,B)
   3       (2)  area(on(ok),B,C)
   4       (3)  area(on(on(ok)),C,D)
   5       (4)  area(on(off(on(ok))),D,E)
   6       (5)  area(off(on(on(ok))),E,F)
   7       (6)  area(on(on(on(ok))),F,A)? sf 1,2,3
```

The child goals `(1), (2) and (3)` are to be executed by the fork spy, and the child goal `(4), (5) and (6)` are not to be traced.

Auto reduction-spy :: as COUNT

Executes the reduction spy as many times as specified by COUNT (positive integer). This has the same function as continuously entering the **s or sr** commands as many times as specified by COUNT.

【Example】 Automatic spy

```
[7] spy(layered,queen,*).
 Spypoint set on layered:queen/ * .

[8] trace( layered:n_queens(4,Ans) ).
@0@4096   1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)? as 3
                          % Perform automatic trace three times.
<2> layered: queens/4 -->
@0@4096     + queen(1,4,begin,B)
                 B=[1* C |D]
                 E:= 1+ 1
   3     * (1)  filter(begin,1,1,C)
   4   + * (2)  queen(E,4,begin,D)? s % 2          % Two more times
@0@4096   4 + layered: queen(2,4,begin,D)
                 D=[2* F |G]
                 H:= 2+ 1
   5     * (1)  filter(begin,2,1,F)
   6   + * (2)  queen(H,4,begin,G)? s % 1          % One more time
@0@4096   6 + layered: queen(3,4,begin,G)
                 G=[3* I |J]
                 K:= 3+ 1
   7     * (1)  filter(begin,3,1,I)
   8   + * (2)  queen(K,4,begin,J)?       % Waiting for command input
```

No trace anymore :: ntr

No trace is performed any more.

Abort :: a

Abort execution.

Break :: b

Interrupt tracing. Makes the listener in the state of the top level which is nested one step deeper. At that time, the level interrupted is displayed for the top level.

Fork all (set the tracer to trace) :: f

Specifies the tracer to perform the trace output of the child goal (which utilizes the multi-window). Thereafter, the trace output for all the child goals is to be performed for the window of the tracer which has been specified by the user. When this command is input, the following prompt is output on the screen.

```
Tracer number:
```

Types in the tracer number (arbitrary integer 0, or positive). The tracer number of the listener basic unit is 0 (zero). If the tracer which already exists is specified, the trace output is merged. With respect to the trace method which uses the multi-window, see Section 4.1.7.7 (Page 129).

【Example】 Specification of tracing by using multi-window.

```
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? f
 Tracer number: 1                          % Input of tracer number
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
1  7     * (1)  queen(1,4,C,H)     % "1" at the top is a tracer number.
1  8     * (2)  queens(G,4,H,A)? s 2       % (1) is trace.
                                           % (2) is reduction spy.
```

Thereafter, the trace output of the child goals `(1) and (2)` is performed for the tracer with the tracer number 1.

Fork selectively (set the tracer to trace) :: f N1,N2,...

Specifies the tracer to perform the trace output of the child goal (which utilizes the multi-window). Thereafter, trace output of the child goals specified by the child goals serial numbers N1, N2,... are performed for the window of the tracer which has been specified by the user. When this command is input, the following prompt is output on the screen.

```
Tracer number:
```

Types in the tracer number (arbitrary integer 0, or positive). The tracer number of the listener basic unit is 0 (zero). If the tracer which already exists is specified, the trace output is merged. In the case where the output destination tracer is set to the child goal, the tracer number is displayed at the top of the line. As for the trace method which uses the multi-window, see Section 4.1.7.7 (Page 129).

【Example1】 Specification of tracing using multi-window.

```
@0@4096   4   layered: queens(2,4,C,A)
                 G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? f 1  % Specify the destination tracer individually.
 Tracer number: 1                     % Input of tracer number
@0@4096   4   layered: queens(2,4,C,A)
                 G:=2+1
1  7     * (1)  queen(1,4,C,H)   % "1" at the top is a tracer number.
   8     * (2)  queens(G,4,H,A)? % The output destination of (1) is the tracer number 1.
                                 % The output destination of (2) is not  changed.
```

Thereafter, the trace output of the child goal `(1)` is performed for the tracer with the tracer number 1. The trace output of the child goal `(2)` remains in the current state and is not changed.

【Example2】 Specification of tracing by using multi-window

```
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off & )),D,E)
   6     * (5)  area(off(on & )),E,F)
   7     * (6)  area(on(on & )),F,Ans)? f 3,4
                               % Specify the destination tracer individually.
 Tracer number: 1              % Specify the tracer number 1.
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
1  4     * (3)  area(on(on(ok)),C,D)  % "1" at the top is a tracer number.
1  5     * (4)  area(on(off & )),D,E) % "1" at the top is a tracer number.
   6     * (5)  area(off(on & )),E,F)
   7     * (6)  area(on(on & )),F,Ans)? f 5,6
                             % Specifies the destination tracer individually.
 Tracer number: 2            % Specifies the tracer number 2.
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
1  4     * (3)  area(on(on(ok)),C,D)
1  5     * (4)  area(on(off & )),D,E)
2  6     * (5)  area(off(on & )),E,F)     % "2" at the top is a tracer number.
2  7     * (6)  area(on(on & )),F,Ans)?   % "2" at the top is a tracer number.
```

Thereafter, the trace output of child goals `(3) and (4)` is performed for the tracer with the tracer number 1, and that of the child goals `(5) and (6)` is performed for the tracer with the tracer number 2. Note that trace output of the child goals `(1) and (2)` remains in the current state and is not changed.

Toggle keep mode all :: k

Reverses the execution keep mode for all the child goals. When the execution keep is not specified to the child goal, the execution keep is to be set. If the execution keep is already specified, that specification is to be canceled. The scheduling of goal execution can be dynamically changed by using this function.

【Example】 Reversing the execution keep mode

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? k
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     k (1)  queen(1,4,C,H)
   8     k (2)  queens(G,4,H,A)? k          % Keep specification
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)?            % Cancel specification
```

The letter "k" attached before the sub-goal serial number indicates that the keeping of goal execution has been specified.

【Hint1】 The following three goal states exist during the trace processing. No single goal can be in a multiple number ofstates.

1. Executed within Shoen.
2. Waiting for reporting or reporting within the tracer.
3. State in which execution is not being kept.

Toggle keep mode selectively :: k N1,N2,...

Reverses the execution keep mode of the child goals specified by the child goal serial numbers N1, N2,... When the execution keep is not specified to the child goal, the execution keep is to be set. And if the execution keep is already specified, that specification is to be canceled.

【Example】 Reversing the execution keep mode individually.

```
{Trace}
[7] layered:map(Ans).
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? k 1,2,3
                                    % Specification of keeping execution
@0@4096   1   layered:map(A)
   2     k (1)  area(ok,begin,B)
   3     k (2)  area(on(ok),B,C)
   4     k (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? k 3 % Cancellation of keep specification
@0@4096   1   layered:map(A)
   2     k (1)  area(ok,begin,B)
   3     k (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? 3,4     % Suppression of tracing

(1), (2) - Execution is kept.
(3), (4) - Tracing is suppressed.
(5), (6) - Tracing is performed.
```

List kept goals :: kg <TRACERS>

Displays a list of those goals for which the execution is being kept. Specifies the trace number or atom all to TRACERS. When TRACERS is not specified, the list of the goals kept by the tracer which executed this command is displayed on the screen. If the tracer number is specified, the list of the goals kept by the appropriate tracer is displayed. In the case where all is specified to a tracer number, a list of all the goals kept in all the tracers is displayed.

【Example1】 Displaying the list of kept goals

```
@0@4096   5   layered:area(on(off( & )),D,E)
   8     * (1)  colors(G)
   9     * (2)  area1(on(off(on(ok))),D,G,E)? kg
 %%% KEPT GOALS (3/5 goals) %%%
k:2 layered:area(ok,begin,B)
k:3 layered:area(on(ok),B,C)
k:4 layered:area(on(on(ok)),C,D)? tr 1         % Execution in trace mode
```

"3/5" in the title "`%%% KEPT GOALS (3/5 goals) %%%`" shows the proportion of the number of goals kept by the tracer which has executed this command (3 goals) and that number kept by all the tracers in total (5 goals). The "number" in k:"number" is the **goal number**.

【Example2】 Displaying a list of all the goals for which the execution is suppressed.

```
@0@4096   7   layered:area(on(on( & )),F,O)
   18    * (1)  colors(L)
   19    * (2)  area1(on(on( & )),F,L,O)? kg all
 %%% KEPT GOALS (2/6 goals) %%%
1:8  layered:colors(H)
1:9  layered:area1(ok,begin,H,B)
2:10 layered:colors(I)
2:11 layered:area1(on(on(ok)),C,I,D)
k:12 layered:colors(G)
k:13 layered:area(off(on( & )),E,G,F)? tr 12   % Execution in trace mode
```

"1" and "2" stated at the top of each line of "1:82 and "2:10" are the tracer numbers. "k" is marked for the goal kept in the tracer which has executed this command.

The commands which can be input during the display of the list are shown below.

Goal numbers	:	Executes the kept goal with tracing suppressed.
tr goal numbers	:	Traces the kept goal.
s goal numbers	:	Executes the kept goal by the reduction spy.
sr goal numbers	:	Executes the kept goal by the reduction spy.
sf goal numbers	:	Executes the kept goal by the fork spy.
spy module name, predicate name, number of arguments	:	Sets the spy.
nospy module name, predicate name, number of arguments	:	Cancels the spy.
si <module name>	:	Display the list of spies.
f <goal numbers>	:	Specifies the tracer to perform the trace output.
tg <tracer number>	:	Lists of the goals not reduced.
kg <tracer numer>	:	Lists of the kept goals
ins goal number	:	Inspects the kept goal.
m variable name <,name>	:	Activates the monitor.
mf variable <,name>	:	Activates the monitor in the other window.
mm	:	Changes the stream automatic monitor mode of the monitor.
mt	:	Lists the monitors activated.
rem variable name <,name>	:	Stores the variable in the variable pool.
l	:	Lists the variables stored in the variable pool.
v	:	Displays the value of the variable stored in the variable pool.
w <depth,length>	:	Redisplays with the output limit value temporarily changed.
pd <depth>	:	Changes the output limit value of the structure depth.
pl <length>	:	Changes the output limit value of the structure length.
psl <length>	:	Changes the output limit value of the number of string elements.
pal <length>	:	Changes the output limit value of the number of atom print name characters.
pv	:	Changes the output formad of the variable.
Return or q	:	Quits the list.

The arguments enclosed in <, > can be omitted. The goal numbers are the two or more

goal numbers punctuated by a comma (","). For each term and concept, see the following sections.

Setting Spy	→	Section 4.1.6.8 Page 111
Canceling Spy	→	Section 4.1.6.8 Page 112
List of Commands	→	Section 4.1.6.8 Page 113
f command	→	Section 4.1.7.9 Page 137
tg command	→	Section 4.1.7.9 Page 143
Inspector	→	Section 4.2 Page 164
Monitor	→	Section 4.2.5 Page 188
Stream automatic monitor mode	→	Section 4.2.5.2 Page 190
List of monitors	→	Section 4.2.5.2 Page 192
l command	→	Section 4.1.6.5 Page 105

【Hint】 The list of the kept goals can also be viewed when the execution of the program has been interrupted (Ctr-C).

List traced goals :: tg <TRACERS>

Displays a list of those goals for whHch reduction has not yet been completed at the moment of executing this command, among the goals executed by specifying the trace. Specifies the trace number or atom all to TRACERS. When TRACERS is omitted, the list of the goals which have not yet reduced in the goals to be reported to the tracer which executed this command is displayed on the screen. If the tracer number is specified, the list of the goals in the appropriate tracer is displayed. In the case where all are specified to the tracer number, a list of all the goals not yet reduced in all the tracers is displayed.

【Hint】 Those goals whiCh may b deadlocked are included in this group of goals whenever they are executed with tracing specified.

【Example1】 List of goals not yet reduced "5/7" in the

```
@0@4096   7   layered: colors(G)
                  [red,yellow,blue,white]= G ? tg
 %%% TRACED GOALS (5/7 goals) %%%
t:8  layered:area1(ok,begin,G,B)
t:10 layered:area1(on(ok),B,H,C)
t:12 layered:area1(on(on(ok)),C,I,D)
t:14 layered:area1(on(off( & )),D,J,E)
t:16 layered:area1(off(on( & )),E,K,F)?
```

[Example 1] title "`%%% TRACED GOALS (5/7 goals) %%%`" indicates the proportion of the number of goals not yet reduced among the ones to be reported to the tracer (5 goals) and the number of goals not yet reduced by the all tracers in total (7 goals). The "number" of t:"number" is the goal number.

【Example2】 Displaying a list of all the goals not yet reduced in all the tracers

```
@0@4096   7   layered: colors(G)
                  [red,yellow,blue,white]= G ? tg all
 %%% TRACED GOALS (2/5 goals) %%%
1:8  layered:area1(ok,begin,G,B)
2:10 layered:area1(on(ok),B,H,C)
2:12 layered:area1(on(on(ok)),C,I,D)
t:14 layered:area1(on(off( & )),D,J,E)
t:16 layered:area1(off(on( & )),E,K,F)?
```

"1" and "2" stated at the top of each line of "1:82 and "2:10" are the tracer numbers. "t" is attached to the goal kept in the tracer which executed this command.

The commands which can be input during the display of the list are shown below.

tg <tracer number>	:	Lists of goals not reduced.
kg <tracer number>	:	Lists of kept goals
ins goal number	:	Inspects the kept goal.
m varialbe name <,name>	:	Activates the monitor.
mf variable name <,name>	:	Activates the monitor in the other window.
mm	:	Changes the stream automatic monitor mode of the monitor.
mt	:	lists the monitors activated.
rem variable name <,name>	:	stores the variable in the variable pool.
l	:	Lists the variables stored in the variable pool.
v	:	Displays the value of the variables stored in the variable pool.
w <depth,length>	:	Redisplays the output limit value tmporArily changed.
pd <depth>	:	Changes the output limit value of the structure depth.
pl <length>	:	Changes the output limit value of the structure length.
psl <length>	:	Changes the output limit value of the number of string elements.
pal <length>	:	Changes the output limit value of the number of atom print name characters.
pv	:	Changes the output format of the variable.
Return or q	:	Quits the list.

The arguments enclosed in <, > can be omitted. The goal numbers are two or more goal numbers punctuated by a comma (","). For each term and concept, see the following sections.

kg command	→	Section 4.1.7.9 Page 140
Inspector	→	Section 4.2 Page 164
Monitor	→	Section 4.2.5 Page 188
Stream automatic monitor mode	→	Section 4.2.5.2 Page 190
List of monitors	→	Section 4.2.5.2 Page 192
l command	→	Section 4.1.6.5 Page 105

【Hint】 The list of the goals which have not yet reduced can also be viewed when the execution of the program has been interrupted (Ctr-C).

Set predicate spy on parent(trace) goal :: spy

When the predicate of the parent (trace) goal is not registered as a spypoint predicate, this function sets such predicate as the spypoint predicate. The goal for which the spy has been set is displayed with the ("+") mark attached.

【Example】 Setting the predicate in the parent goal as the spypoint predicate.

```
@0@4096   3   layered:queens(2,4,C,A)
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy
@0@4096   3 + layered:queens(2,4,C,A)   % The spy mark (+) is attached.
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)?
```

Reset predicate spy on parent(trace) goal :: spy

The spy has been set to layered:queens/4. Reset predicate spy on parent(trace) goal :: spy When the predicate of the parent (trace) goal has been set as the spypoint predicate, this function resets the spy of that predicate.

【Example】 Resetting the spy set to the predicate of the parent goal.

```
@0@4096   3 + layered:queens(2,4,C,A)
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy
@0@4096   3   layered:queens(2,4,C,A)   % The spy mark (+) is disappears.
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)?
```

The spy of layered:queens/4 has been reset.

【Caution】 It is impossible to delete the registration of the predicates which have been registered as the spy objectives by the wild card.

Set predicate spy on subgoal :: spy N

Sets the spy to the predicate of the goal specified by the child goal serial number N.

【Example】 Setting the spy to the predicate of the child goal.

```
@0@4096   3   layered:queens(2,4,C,A)
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy 1
@0@4096   3   layered:queens(2,4,C,A)
                  G:=2+1
   6   + * (1)  queen(1,4,C,H)          %The spy mark (+) is attached.
   7     * (2)  queens(G,4,H,A)?
```

Reset predicate spy on subgoal :: spy N

Resets the spy having been set to the predicate of the goal specified by the child goal serial number N.

【Example】 Resetting the spy having been set to the predicate of the child goal.

```
@0@4096   3   layered:queens(2,4,C,A)
                 G:=2+1
   6   + * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy 1
@0@4096   3   layered:queens(2,4,C,A)
                 G:=2+1
   6     * (1)  queen(1,4,C,H)          % The spy mark (+) disappears.
   7     * (2)  queens(G,4,H,A)?
```

The spy of layered:queen/4 has been reset.

Set predicate spy :: spy ModuleSpec, Predicate, Arity

Sets the spy for the predicate of ModuleSpec:Predicate/Arity. The wild card ("*") can be used for ModuleSpec, Predicate, and Arity.

Reset predicate spy :: nospy ModuleSpec, Predicate, Arity

Resets the spy set to the predicate of ModuleSpec:Predicate/Arity. The wild card ("*") can be used for ModuleSpec, Predicate, and Arity.

List spying predicates :: si

Displays the list of the predicates that are the targets of the spy. As for the commands which can be input during the display of the list, see Section 4.1.6.8.

List spying predicates in the specified module :: si ModuleSpec

List spying predicates in the specified module :: si ModuleSpec Displays the list of the predicates that are the targets of the spy in the module specified by ModuleName. As for the commands which can be input during the display of the list, see Section 4.1.6.8.

Set default trace mode :: tr ModuleSpec, Predicate, Arity

ModuleSpec:Predicate/Arity to ON., The wild card ("*") can be used for ModuleSpec, Predicate, and Arity. The trace mark ("*") is attached to the goal which the trace mode has been turned ON.

Reset default trace mode :: ntr ModuleSpec, Predicate, Arity

Sets the default trace mode if the predicate specified by Reset default trace mode :: ntr ModuleSpec, Predicate, Arity Resets the trace mode of the predicate specified by ModuleSpec:Predicate/Arity which the default trace mode has set to on. The wild card ("*") can be used for ModuleSpec, Predicate, and Arity.

Reset default trace mode all :: ntr_all

Resets the trace mode to OFF for all the predicates which the default trace mode has set to ON.

List tracing predicates :: tri

List the predicates which the default trace mode is set to ON.

Inspect parent(trace) goal :: ins

Inspect the parent (trace) goal. For more detailed information on the inspector, see Section 4.2 (Page 164).

Inspect subgoal :: ins N

Inspects the goal specified by the child goal serial number (N). For more detailed information on the inspector, see Section 4.2 (Page 164).

Monitor variable :: m Variable <,?NewName>

When the variable specified by Variable is materialized, its value is monitored. It is possible to name the variable to be monitored appropriately, and the value is displayed by using that name. Newname is the atom or arbitrary variable. If NewName is not specified, it is to be monitored by using Variable as its variable name. For more details on the monitor, see Section 4.2.5 (Page 188).

【Example1】 Monitoring the materialization of the variable.

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? m G    % Monitor of variable G
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? 1      % Suppress tracing.
aq:1 G == [red,yellow,blue,white] ? % Reports on materialization status
```

【Example2】 Monitoring the materialization of the variable with naming.

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? m G,Colors
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? 1
aq:1 Colors == [red,yellow,blue,white] ?
```

Monitor variable and Fork :: mf Variable <,?NewName>

When the variable specified by Variable is materialized, its value is monitored. Input/output is performed to the new window to be generated. It is possible to name the variable to be monitored appropriately, and the value is displayed by using that name. Newname is the atom or arbitrary variable. If NewName is not specified, it is monitored by using Variable as its variable name. For more details on the monitor, see 4.2.5 (Page 188).

Change monitor mode :: mm

Changes the stream automatic monitoring mode of the monitor. Thereafter, the monitor to be activated will be executed in the changed mode. For further details, see Section 4.2.5.2 (Page 4.2.5.2).

List monitored terms :: mt

Lists the terms not yet reported on at the time of executing this command among the ones to which monitoring has been specified. For further details, see Section 4.2.5.2 (Page 192).

Remember the binding :: rem Variable <,?NewVariable>

Registers the variable specified by Variable in the variable pool of the listener. If NewVariable is specified, it is to be registered as a new variable. Those names which start with an underline or the ones which are already stored in the variable pool of the listener cannot be used.

【Example1】 Registering the variable under tracing.

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? rem G    % Store the variable G.
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(H)
   8     * (2)  area1(ok,begin,H,B)? 1        % Suppresses tracing
@0@4096   2   layered: area(on(ok),B,C)
   9     * (1)  colors(I)
   10    * (2)  area1(on(ok),B,I,C)? v        % Lists variables
 Ans = Ans
 G = [red,yellow,blue,white]
```

【Example2】 Registering the variable under tracing with optional name.

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? rem G,Colors
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(H)
   8     * (2)  area1(ok,begin,H,B)? 1
@0@4096   2   layered: area(on(ok),B,C)
   9     * (1)  colors(I)
   10    * (2)  area1(on(ok),B,I,C)? v
 Ans = Ans
 Colors = [red,yellow,blue,white]
```

List variables :: l

Lists the variables stored in the variable pool of the listener. For further details see Section 4.1.6.5 List command (Page 105).

【Hint】 The values of the variables input as the argument of the goal at the listener top level (except the ones with a name starting with an underline) are bound until the end of the program execution. Therefore, it is possible to view the list of the values of those variables regardless of the mode used for storing them.

Display variables :: v

Displays the variables stored in the variable pool of the listener with their values.

【Hint】 The values of the variables input as the argument of the goal at the listener top level (except for the ones with the name starting an underline) are bound until the end of the program execution. Therefore, it is possible to view the list of the values of those variables regardless of the mode used for storing them.

Change holophrasting parameters :: ho

Changes the environmental variable values related to holophrasting. It changes the output restriction values of the depth and length of the structure, the number of elements of a string and the number of characters of an atom. The tracer displays the current values and then waits for input of values by the user. Entering return results in no change in the values.

Change print depth :: pd Depth

Changes the output restriction value of the depth of the structure. The specification of Depth may be omitted.

Change print length :: pl Length

Changes the output restriction value of the length (number of elements) of the structure. The specification of Length may be omitted.

Change print string length :: psl Length

Changes the output restriction value of the number of elements of the string. The specification of Length may be omitted.

Change print atom length :: pal Length

Changes the output restriction value of the number of characters of the atom. The specification of Length may be omitted.

Toggle print variable mode :: pv

Switches the variable display mode (between name and address).

Print environment variables :: env

Displays the values of the environmental variables related to the tracer.

Rewrite reduction :: w

Changes the values of Print-Depth, Print-length temporarily, and redisplays the reduction. The values for Print-Depth, Print-Length can be changed by the rewrite_depth, rewrite_length commands. The initial values when the listener is started are 30, 30. For further details see Section 4.1.6.3 (Page 98).

【Example】 Re-displaying the reduction with the output limit value changed temporarily

```
@0@4096   7   layered: queens(3, ... )
                  0:= 3+ 1
   12    * (1)  queen(1, ... )
   13    * (2)  queens(0, ... )? w
@0@4096   7   layered: queens(3,4,[1* L |M],Ans)
                  0:= 3+ 1
   12    * (1)  queen(1,4,[1* L |M],P)
   13    * (2)  queens(0,4,P,Ans)? 1
@0@4096   8   layered: filter(begin, ... )
                  H= begin?
```

Rewrite reduction :: w Depth,Length

Changes the values of Print-Depth, Print-Length temporarily, and redisplays the reduction.

Change scope of variables :: scope <MODE>

Changes the range of the identity check for the variables under tracing. Either **tracer** or **clause** can be set to MODE. When MODE is tracer, the identity of the variables with the same name in the tracer (until the time when the execution of the task is terminated) is guaranteed. If MODE is a clause, it is guaranteed by the unit of trace output at one time (by clause).

【Hint】 The merits of the clause mode are that a faster trace output and a easy-to-read change number can be realized by avoiding the possibility of becoming a large number (such as 49 of "A49"). However, it is impossible to check the identity of the variables in different goals.

Change output format of ancestor goal :: da <MODE>

Changes the display format of the ancestor goal output at the spy execution to Mode. full, abbr, or no can be specified to MODE. For further details, see Section 4.1.6.8 (Page 115).

Help :: h

Displays the help menu.

4.1.7.10 List of Commands for Tracer

The Table 4.1 on Page 151 lists the commands of the tracer.

4.1.8 Treatment of Errors

When the goal specified by the user results in some type of error, such as goal failure or an error in handling an exception, the listener will display information pertaining to the error. The user can analyze the details of the error by using the inspector. The execution can be continued by specifying the substitutive goal instead of the goal with the error.

4.1.8.1 Commands Available for Input in Reporting Errors

When the listener detects an error during the execution of a user program, it reports the time that the error occurs (the execution time from the start of executing program to the moment the error is detected) and the content of the error.

```
[7] layered:n_queens(f,Ans).
 Failure>> layered: queens/ 4
    Goal > queens(1,f,begin,Ans)
    Node > 0
 !! Exception (115 msec):
```

The following list indicates the commands which can be input at the time of error occurrence.

$< cr >$	:	abort & look into other exceptions
`q`	:	abort & look over other exceptions
`a`	:	abort & ignore other exceptions
`b`	:	break
`c`	:	continue
`e`	:	execute substitutive goals
`ins`	:	inspect exception
`info`	:	print exception informations
`l`	:	list variables
`v`	:	display variables
`w <D,L>`	:	rewrite exception
`pd <D>`	:	change print depth
`pl <L>`	:	change print length
`psl <L>`	:	change print string length
`pal <L>`	:	change print atom length
`pv`	:	toggle print variable mode
`h, ?`	:	help

Abort & Look into other exceptions :: Return

Aborts the execution. If an error other than this has been detected, the error is reported again and the user can issue a command at the time of error occurrence. (Note that it is not possible to continue execution or to execute a substitutive goal thereafter.)

【Example】 Return is entered at the error occurrence

Command name	Input format	Meaning	Reference Page
Continue	**<cr>, c**	Continues execution	131
Auto trace	**at COUNT**	Traces automatically	132
No trace & Continue	**n**	Suppresses trace	132
Toggle trace mode all & Continue	**x**	Reverses execution of trace mode for all goals	132
Toggle trace mode & Continue	**N1,N2, ...**	Reverses execution of trace mode	133
Toggle trace mode	**t <N1,N2, ...>**	Reverses of trace mode	133
Apply reduction-spy	**s <N1,N2, ...>**	Execution by reduction spy	134
Apply reduction-spy	**sr <N1,N2, ...>**	Execution by reduction spy	134
Apply fork-spy	**sf <N1,N2, ...>**	Execution by fork spy	135
Auto reduction-spy	**as COUNT**	Automatic reduction spy	136
No trace anymore	**ntr**	Will not trace any more	136
Abort	**a**	Aborts execution	137
Break	**b**	Interrupts tracing	137
Fork (set the tracer)	**f <N1,N2, ...>**	Specification of output destination tracer	137
Toggle keep mode	**k <N1,N2, ...>**	Specification of execution keep	138
List kept goals	**kg <Tracers>**	Lists execution kept goals	140
List traced goals	**tg <Tracers>**	Lists unexecuted trace goals	143
Set(reset) spy on parent goal	**spy**	Sets and resets spy of parent goal	144
Set(reset) spy on subgoal	**spy Number**	Sets and resets spy of subgoal.	145
Set spy	**spy M,P,A**	Sets spy to the predicate M:P/A	146
Reset spy	**nospy M,P,A**	Resets spy on the predicate M:P/A	146
List spying predicates	**si <ModSpec>**	Lists of predicates spy is set	146
Set default trace mode	**tr M,P,A**	Sets the default trace mode	146
Reset default trace mode	**ntr M,P,A**	Resets the default trace mode	146
Reset default trace mode all	**ntr_all**	Resets all the default trace modes	146
List tracing predicates	**tri**	Sets list predicates the default trace mode	146
Inspect parent goal	**ins**	Inspects the parent goal	146
Inspect subgoal	**ins Number**	Inspects the sub-goal	146
Monitor variable	**m Var <,Name>**	Activates the monitor	147
Monitor variable & Fork	**mf Var <,Name>**	Activates the monitor in the other window	147
Change monitor mode	**mm**	Changes the stream automatic mode of the monitor	147
List monitored terms	**mt**	Lists of monitors	147
Remember the binding	**rem Var <,Name>**	Stores the variables in the variable pool	147
List variables	**l**	xxxxx	148
Display variables	**v**	xxxxx	148
Change holophrasting parameters	**ho**	Changes the holophrasting.	148
Change print depth	**pd <Dep>**	Changes the output limit value of the structure depth.	148
Change print length	**pl <Len>**	Changes the output limit value of the structure length.	149
Change print string length	**psl <Len>**	Changes the output limit value of the number of string elements.	149
Change print atom length	**pal <Len>**	Changes the output limit value of the number of atom print names characters.	149
Toggle print variable mode	**pv**	Changes the output format of the variable	149
Print environment variables	**env**	Displays the environmental variable value related to the tracer	149
Rewrite reduction	**w <Dep,Len>**	Redisplays the output limit value temporarily changed	149
Change scope of variables	**scope <Mode>**	Changes the scope of the variable un-	149

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % Activate batch GC
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec):               % Enter return
 *** Deadlock detected during GC.
 Builtin> subtract(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5123 msec): h             % Possible to enter commands
```

Abort & Look over other exceptions :: q

Aborts the execution. If an error other than this has been detected, only the error output is performed. The user can not issue a command at time of the error occurrence.

【Example】 **q** command is entered at time of the error occurrence

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % Activate batch GC
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec): q       % Enter q command
 *** Deadlock detected during GC.  % Only an error report is performed.
 Builtin> subtract(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0

[2]           % The command at the error occurrence can not be accepted.
```

Abort & Ignore other exceptions :: a

Aborts the execution. Even if an error other than this has been detected, the error is not reported but is returned to the listener top level.

【Example】 **a** command is entered at the time an error occurs

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % Activate batch GC
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec): a             % Enter a command

[2]   % Even when the other error has been detected, it is not reported.
```

【Hint】 There are cases in which two or more errors are reported by the execution of one task. For example, multiple deadlocks are detected by a batch GC at one time. Generally, the number of errors which occurred is unknown. However, it is possible to avoid unnecessary error reports by using this command in the case in which many errors are expected to occur because of nature of the program.

Break :: b

Places the listener in the top level with the error processing interrupted. At that moment, the interrupt level is displayed at the top level. To make it go to a lower level from the interrupt, type **exit**.

Continue :: c

Executes the true goal (the goal to perform nothing) as a substitutive goal instead of the goal with an error and continues execution.

【Caution】 If the error occurred is by the merger deadlock or the illegal input of a merger, [ ] is to be unified to the merger input variable (which has been abolished).

【Hint】 This command is useful to utilize at the time when an undefined module exception or an undefined predicate exception occurs.

【Example】 Continuing execution.

```
[1] test:go(A)
 Undefined module>> rsc_manager: create/1
             Goal > create(B)
           Caller > test: go/ 0
             Node > 0
 !! Exception (32452 msec): c            % Continue execution
 A = A

[2]
```

Execute substitutive goals :: e

Specifies the substitutive goal instead of the goal with the error and continues the execution. The specification method for the substitutive goal is the same as that of the listener top level. Multiple goals may be specified. Note that the listener commands other than the built-in predicates can not be specified.

【Hint】 This command is useful to utilize at the time when an undefined module exception or an undefined predicate exception occurs.

【Example】 Executing the substitutive goal.

```
[1] sieve:go(10,Ans).
 Undefined predicate>> generator: go/ 2
               Goal > go(10,B)
             Caller > sieve: go/ 2
               Node > 0
 !! Exception (43242 msec): e
 >Please execute substitutive goals.
 |?- B=[0,1,2,3,4,5,6,7,8,9].
 Ans = [0,2,4,6,8]
```

Inspect exception :: ins

Inspects the exception information. The information to be forwarded to the inspector is the information for the following exception events.

```
exception( Kind, Info, NewCode, NewArgv )
```

As for the details of the Info, refer to Section 7.5.4 (Page 305). For information on the inspector, see Section 4.2 (Page 164).

Print exception informations :: info

Displays the module information of the goal for which the exception occurred. The content to be displayed is as follows: day and time that the module was generated, the day and time of link, the name, day and time of the source file, add the name, day and time of the dependent file (such as the files included to be performed).

【Example】 Displaying module information

```
[7] layered:n_queens(f,Ans).
v Failure>> layered: queens/4
    Goal > queens(1,f,begin,K)
    Node > 0
 !! Exception (135 msec): info
 %%% PREDICATE INFORMATIONS %%%
 Module  : nakao:: layered
 Created : 10-Aug-1991 20:03:14
 Linked  : 10-Aug-1991 20:03:14
 Source  : icpsi255::>sys>user>s-nakao>SAMPLE>...
         : 12-Jul-1990 19:55:53
```

【Caution】 It is impossible to view the module information of the module loaded by the IPL.

List variables :: l

Lists the variables stored in the variable pool of the listener together with their values. (see 4.1.6.5 Page 105)

【Hint】 The values of the variables input as arguments of the goal at the listener top level (except for the ones with a name starting with an underline) are bound until the end of the program execution. Therefore, it is possible to view the list of the values of those variables regardless of the mode used for storing them.

Display variables :: v

Displays the variables stored in the variable pool of the listener together with their values.

【Hint】 The values of the variables input as arguments of the goal at the listener top level (except for the ones with a name starting with an underline) are bound until the end of the program execution. Therefore, it is possible to view the list of the values of those variables regardless of the mode used for storing them.

Rewrite exception :: w

Changes the values of output restriction for the depth and length of the structure and redisplay the exception information. The values for Print-Depth, Print-Length can be changed by the rewrite_depth, rewrite_length commands. For further details, see Section 4.1.6.3 (Page 98).

【Example】 Re-displaying exception information

```
[7] {0,1,2,3,4,5,6,7,8,9} = {0,1,2,3,4,5,6,7,8}.
 Unification failure>> {0,1,2, ... }
                With > {0,1,2, ... }
                Node > 0
 !! Exception (114 msec): w
 Unification failure>> {0,1,2,3,4,5,6,7,8,9}
                With > {0,1,2,3,4,5,6,7,8}
                Node > 0
 !! --------- (114 msec):
```

Rewrite exception :: w Depth,Length
Temporarily changes the limit values of the depth and length of the structure to Depth, Length and redisplay the exception information.

Change print depth :: pd Depth
Changes the output restriction value of the depth of the structure. The specification of Depth may be omitted.

Change print length :: pl Length
Changes the output restriction value of the length of the structure. The specification of Length may be omitted.

Change print string length :: psl Length
Changes the output restriction value of the number of elements of the string. The specification of Length may be omitted.

Change print atom length :: pal Length
Changes the output restriction value of the number or characters of an atom. The specification of Length may be omitted.

Toggle print variable mode :: pv
Switches the variable display mode (between name and address).

Help :: h
Displays the help menu.

4.1.8.2 Re-displaying Errors

At the error occurrence, there is a possibility that the error display routine will be quit before examining the content of the error by mistake (for example if the level of holophrasting is small). Also, the user may want to reconfirm the content of the error occurred in the past.

The error reported by the listener is normally stored in the error queue of the listener. The error information stored in the error queue can be referred to by the command **error** and **last_error**. The length of the queue when the listener is started is 5. The length of the error queue can be changed by the **error_queue_length** command. If the number of errors occurred exceeds the length of the error queue, the information is deleted beginning with the oldest one in the chronological order. For the information on these commands see Section 4.1.6.1 (Page 92).

4.1.8.3 List of Error Messages

The following Table 4.2 lists the error messages displayed by the listener. Examples of each error which occurred are provided below. For an example of the deadlock exception occurrence, see Section 4.1.5.3 (Page 81).

【Example1】 Event does not exist.

`Event not found`	: Event does not exist.	(Example 1)
`!WARNING! illegal macro module`	: Macro-module of the listener has been updated to the illegal module	(Example 2)
`!WARNING! non-existent macro module`	: The macro-module of the listener does not exist. (it has been deleted)	(Example 3)
`Nonexistent package`	: Package does not exist.	(Example 4)
`Nonexistent module`	: Module does not exist.	(Example 5)
`Protection violation`	: Protected package is accessed	(Example 6)
`Nonexistent public predicate`	: Public predicate does not exist.	(Example 7)
`Nonexistent local predicate`	: Local predicate does not exist.	(Example 8)
`Local predicate`	: Local predicate is executed as public predicate.	(Example 9)
`Public predicate`	: Public predicate is executed as local predicate.	(Example 10)
`Default moduel is not set`	: Default module is not set.	(Example 11)
`Nonexistent command`	: Command (built-in predicate) does not exist.	(Example 12)
`Illegal argument`	: Illegal input of argument of command	(Example 13)
`Illegal input`	: Illegal specification of goal input	(Example 14)
`Illegal goal`	: Illegal specification of goal input	(Example 15)
`Syntax error`	: Syntax error in input	(Example 16)
`Illegal specification`	: Error in specification of standard input/output device	(Example 17)
`Cannot open file`	: Standard input/output device cannot be created.	(Example 18)
`Failure`	: All the guards of candidate clauses have failed.	(Example 19)
`Unification failure`	: Body unification has failed.	(Example 20 - 21)
`Illegal merger input`	: Illegal input of merger	(Example 22)
`Undefined module`	: Predicate in the undefined module was attempted to be called.	(Example 23)
`Undefined predicate`	: The undefined predicate was attempted to be called.	(Example 24)
`Raised`	: An exception unknown to the listener has occurred.	(Example 25)
`Trace exception occurred`	: A trace exception unknown to the listener has occurred.	(Example 26)
`Spy exception occurred`	: A spy exception unknown to the listener has occurred.	(Example 27)
`Priority rate X is illegal input`	: Illegal specification of the goal execution priority	(Example 28)
`Priority rate X is range overflow`	: Specification of goal execution priority is not within the legal range.	(Example 29)
`Node number X is illegal input`	: Illegal specification of goal execution node	(Example 30)
`Cannot throw the goal at node X`	: Specification of goal execution node is not within the legal range.	(Example 31)
`Exception`	: An exception has occurred in execution of a built-in predicate.	(Example 32 - 38)
`Error`	: An illegal message has been sent to the PIMOS.	(Example 39)
`Deadlock detected in during GC`	: A deadlock has been detected during batch GC.	
`Merger deadlock detected in during GC`	: A deadlock of the merger has been detected during batch GC.	
`Deadlock occurred. [×××]`	: A deadlock has been detected during MRB-GC.	
`Merger deadlock occurred. [×××]`	: A deadlock of the merger has been detected during MRB-GC.	
`Firmware error`	: Occurrence indicates a bug in firmware.	

```
[77] !3.
 3: Event not found.
```

The event number 3 does not exist.

【Example2】 The macro-module of the listener has been updated to the illegal module.

```
[7] layered#queen(8,Ans).
!WARNING! illegal macro module: my_macro
!CAUTION! switched to non-expansible mode
```

Since the micromodule of the listener has been updated to the illegal module as the micromodule (such as the normal module), the expansion is not possible for the macro of the input goal. Therefore, the listener moves to a non-expansible mode, and the !CAUTION! sign appears on the screen.

【Example3】 The macro-module of the listener does not exist(it has been deleted).

```
[6] module_table:cache_table([remove_module(my_macro,S)]).
 S = normal

[7] layered#queen(8,Ans).
!WARNING! non-existent macro module: my_macro
!CAUTION! switched to non-expansible mode
```

Since the micromodule of the listener has been deleted, the macro expansion of the input goal cannot be performed. Therefore, the listener moves to a non-expansible mode, and the !CAUTION! sign appears on the screen.

【Example4】 Package does not exist.

```
[1] default_package(abc).
 Nonexistent package>> abc
```

The package abc does not exist.

【Example5】 Module does not exist.

```
[1] hash:hash({1,2,3},Hash,Data).
 Nonexistent module>> hash
```

The module hash does not exist.

【Example6】 The protected package is accessed.

```
[1] hanako::sample:n_queens(4,Ans).
 Protection violation>> hanako:: sample
```

Since the package "hanako" is protected, it is impossible to access it.

【Example7】 Public predicate does not exist.

```
[1] pool:keyed_sort_bag([put(2,b), put(1,a), get_all(Content)]).
 Nonexistent public predicate>> pool: keyed_sort_bag/ 1
```

keyed_sort_bag/1 does not exist in the module pool.

【Example8】 Local predicate does not exist. (Example 8)

```
[1] layered!area(Ans).
 Nonexistent local predicate>> layered: area/ 1
```

`area/1` does on exist in the module `layered`.

【Example9】 Local predicate executed as a public predicate.

```
[1] layered:queens(1,4,begin,List).
 Local predicate>> layered: queens/ 4
```

`queens/4` of the module `layered` is a local predicate.

【Example10】 Public predicate executed as a local predicate.

```
[1] layered!n_queens(4,Ans).
 Public predicate>> layered: n_queens/ 2
```

`n_queens/2` of the module `layered` is a public predicate.

【Example11】 Default module is not set.

```
[1] !queens(1,4,begin,List).
 >> Default module is not set.
```

A local predicate was attempted to be executed with the syntax with the name of the module omitted despite the default module not being set.

【Example12】 Command (built-in predicate) does not exist.

```
[1] string("listener",Size,Elm).
 Nonexistent command>> string("listener",Size,Elm)
```

The command `string("listener",Size,Elm)` does not exist (because it is a built-in predicate for guards only).

【Example13】 Illegal input of argument of command

```
[1] print_length(infinite).
 Illegal argument>> print_length infinite
```

The argument of `infinite` of the command `print_length` is an illegal input.

【Example14】 Illegal specification of goal input

```
[1] X:test(100).
 Illegal input>> X
```

The specification of the module is a variable.

【Example15】 Illegal specification of goal input

```
[1] X.
 Illegal goal>> X
```

It is illegal to input the variable as a goal.

【Example16】 Syntax error in input

```
[1] comparator:sort("applied",technology<Small,Large,Swapped).
*** Syntax Error ***
comparator:sort("applied",technology
*** Here ***
 <Small,Large,Swapped)
```

There is a syntax error at the location indicated with `*** Here ***`.

【Example17】 Error in specification of standard input/output device

```
[1] tee:go -> file("mes.doc") => windoe("OUT").
 Illegal specification>>  =>(windoe("OUT"))
```

`window("OUT")` can not be specified to the standard input/output device.

【Example18】Standard input/output device can not be created.

```
[1] tee:go <= file("in.doc") => window("OUT").
 Cannot open file :  in.doc
```

The standard input/output device can not be generated because in the file "`in.doc`", the standard input device, does not exist.

【Example19】All the guards of candidate clauses have failed

```
[1] layered:n_queens(*,Ans).
 Failure>> layered: queens/ 4
    Goal > queens(1,*,begin,Ans)
    Node > 0
```

All the candidate clauses of the predicate keyed_set/4 of the module keyed_bag in node 0 have failed.

【Example20】Body unification has failed.

```
[1] {1,2,3}={1,2,3,4}.
 Unification failure>> {1,2,3}
                With > {1,2,3,4}
              Caller > pimos:: listener_cmd_builtin: do_command/1
                Node > 0
```

The unification of {`1, 2, 3`} and {`1, 2, 3, 4`} executed by pimos::listener_cmd_builtin:do_command/1 has failed in node 0.

【Example21】Body unification has failed.

```
[1] {a,b,c}={a,d,c}.
  Unification failure>> d
                 With > b
                 Node > 0
```

Unification of `d` and `b` failed in processor 0.

【Caution】When the unification of two structures fails at the element level, that element is displayed on the screen.

【Example22】Illegal input of merger

```
[1] merge(abc,OUT).
 Illegal merger input>> abc
     New merger input > B
                Caller > pimos:: listener_cmd_builtin: do_command/1
                  Node > 0
 !! Execption (116 msec): e
 >Please execute substitutive goals.
 |?- B=[a,b,c].
 OUT = [a,b,c]
```

An illegal input has occurred with the merger which was executed by pimos::listener_cmd_builtin:do_command/1 in node 0.

【Hint】 As shown in the above example, in the case where an illegal input occurs with the merger, the merger input with the illegal input (atom `abc`, in the above example) and the one to be used in returning from the exception (the input variable to send the correct data to the merger with the exception occurred, variable `B` in the above example) are to be reported. Materializing that input variable for the optional stream (or atom `[]`) allows the restart (termination) of the execution of the merger.

【Example23】 The predicate in the undefined module was attempted to be called.

```
[1] test:go.
 Undefined module>> abc: create/2
             Goal > create(100,A)
           Caller > test: go/ 0
             Node > 0
```

The predicate `abc:create/2` which is called in `test:go/0` executed in node 0 can not be executed, since the module `abc` does not exist.

【Example24】 The undefined predicate was attempted to be called.

```
[1] test:go.
 Undefined predicate>> test: create/2
                Goal > create(100,A)
              Caller > test: go/ 0
                Node > 0
```

The predicate `test:create/2` which is called in `test:go/0` executed in node 0 cannot be executed , for it is not to be defined in the module `test`.

【Example25】 An exception unknown to the listener has occurred.

```
[1] shoen:raise(user_tag#test,get_stream,Stream), Stream=[put(abc)].
 Raised>> get_stream
   Data > [put(abc)]
   Node > 0
```

The exception `Info=get_stream_il`, `Data=[put(abc)]` of `shoen:raise(Tag,Info,Data)` has occurred in node 0.

【Example26】 A trace exception unknown to the listener has occurred.

```
[1] module:get_code(layered,n_queens,2,Cde), apply_tracing(Cde,{4,Ans},1).
 >> Trace exception occurred.
 Trace ID> 1
 SubGoals>[internal( code$nakao::layered:queens/4$668 ,{1,4,begin,Ans})]
     Node> 0
```

A trace exception of the trace identifier 1 unknown to the listener has occurred in node 0.

【Example27】 A spy exception unknown to the listener has occurred.

```
[1] apply_spying(Cde,{4,Ans},{layered,filter,4},1).
 >> Spy exception occurred.
 Spy ID> 1
 Parent> layered: queen/ 4
   Goal> internal( code$nakao::layered:filter/4$668 ,begin(1,1,C))
   Node> 0
```

A spy exception of spy identifier 1 unknown to the listener has occurred in node 0.

【Example28】 Illegal specification of the goal execution priority

```
[1] pragma_test:go(1,rate).
 >> Priority rate "rate" is illegal input.
 Pragma> @ priority(*,rate)
 Caller> pragma_test: go/ 2
   Node> 0
```

The exception priority `@ priority(*,rate)` specified for the goal being called in the body of `pragma_test:go/2` in node 0 is an illegal input.

【Example29】 Specification of goal execution priority is not within the legal range

```
[1] pragma_test:go(1,-100).
 >> Priority rate "-100" is range overflow.
 Pragma> @ priority(*,-100)
 Caller> pragma_test: go/ 2
   Node> 0
```

The execution priority `@ priority(*,-100)` specified for the goal which is being called in the body of `pragma_test:go/2` in node 0 is not within the allowed range (specification using a negative number is not possible for the rate specification for Shoen).

【Example30】 Illegal specification of the goal execution node

```
[1] pragma_test:go(3,node7).
 >> Node number "node7" is illegal input.
 Pragma> a/ 0 @ node(node7)
 Caller> pragma_test: go/ 2
   Node> 0
```

Execution node number `node7` specified for goal 1/0 being called in the body of `pragma_test:go/2` in node 0 is an illegal input.

【Example31】 Specification of the goal execution node

```
[1] pragma_test:go(3,8).
 >> Cannot throw the goal at node "8".
 Pragma> a/ 0 @ node(8)
 Caller> pragma_test: go/ 2
   Node> 0
```

which is not within the legal Execution, is not possible since the execution node 8 specified for the goal a/0 which is being called in the body of `pragma_test:go/2` in node 0 does not exist (for example, in the case of 8 nodes configuration).

【Example32】 An exception has occurred in execution of a built-in predicate.

```
[1] predicate_to_code(layered,n_queens,2,Code).
 Exception>> illegal_input_at_arg1
   Builtin > predicate_to_code(layered,n_queens,2,B)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

The first argument of the built-in predicate `predicate_to_code/4` which is executed by `pimos::listener_cmd_builtin:do_command/1` in node 0 is illegal.

【Example33】 The argument of the built-in predicate is not withinthe allowed range.

```
[1] new_string(String,10,64).
 Exception>> range_overflow
   Builtin > new_string(B,10,64)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

The argument of the built-in predicate `new_string/3` which is executed by `pimos::listener_cmd_builtin:do_command/1` in node 0 is not within the correct range (element sizes larger than 23 bits can not be specified for the third argument).

【Example34】 Index of structure out of range

```
[1] new_vector(Vector,3),
   set_vector_element(Vector,3,Old,new,NewVector).
Exception>> out_of_bounds
  Builtin > set_vector_element({0,0,0},3,D,new,E)
   Caller > pimos:: listener_cmd_builtin: do_command/ 1
     Node > 0
```

The specification of the element location of the built-in predicate `set_vector_element/5` which is executed by `pimos::listener_cmd_builtin:do_command/1` in node 0 is outside of the structure.

【Example35】 Overflow of integer operation

```
[1] current_priority(Pri,Min,Max), Rate := ((Pri-Min)*4096)/(Max-Min).
Exception>> integer_overflow
  Builtin > multiply(-289851296,4096,E)
   Caller > pimos:: listener_cmd_builtin: do_command/ 1
     Node > 0
```

An overflow has occurred in the operation of the built-in predicate `multiply/3` which is executed by `pimos::listener_cmd_builtin:do_command/1` in node 0.

【Example36】 Division by zero

```
[1] X := 512/0.
Exception>> integer_zero_division
  Builtin > divide(512,0,B)
   Caller > pimos:: listener_cmd_builtin: do_command/ 1
     Node > 0
```

Division by zero was performed in the operation on the built-in predicate `divide/3` which was executed by `pimos::listener_cmd_builtin:do_command/1` in node 0.

【Example37】 Invalid operation on floating point number

```
[1] X $:= 0.0/0.0.
Exception>> invalid_floating_point_operation
  Builtin > floating_point_divide( +0.0 , +0.0 ,B)
   Caller > pimos:: listener_cmd_builtin: do_command/ 1
     Node > 0
```

The built-in predicate `floating_point_divide/3` which was executed by `pimos::listener_cmd_builtin:do_command/1` in node 0 is an invalid operation.

【Example38】 Mismatch in arity of high-order predicate

```
[1] predicate_to_code(module#comparator,sort,5,Code),
   apply(Code,{"abc",abc, Small, Large}).
Exception>> arity_mismatch
  Builtin > apply( code$comparator:sort/5$435 ,{"abc",abc,D,E})
   Caller > pimos:: listener_cmd_builtin: do_command/ 1
     Node > 0
```

The built-in predicate `apply/2` executed by `pimos::listener_cmd_builtin:do_command/1` in node 0 does not match the arity given to `comparator:sort/5`.

【Example39】 Error message detection

```
[1] module_table:cache_table([get_atom_name(abc,Status)]).
 Error>> module_table_error
  Info > Illegal Message
  Data > get_atom_name(abc,Status)
  Node > 0
```

An illegal message to the PIMOS was detected in node 0.

【HintReference】 If the user provides the utility, such as the devices and pool managed by the PIMOS, with the illegal message (unknown message for a device, a message whose argument type is illegal, or other messages), the PIMOS reports the error to the user. For further details, refer to Section 11.5 (Page 450) in the Programming Volume.

4.2 Inspector

The inspector is a tool to debug the KL1 program, which inspects the contents of any KL1 data item at the user's instructions and displays them. The data item to be inspected can be specified and called at the top level of the listener or during the trace processing. In addition, it can be activated from the shell, as well. In such case, the data specified by the shell is the data to be inspected. (Refer to Section 2.1.6 "Error Processing" of shell (Page 23).)

4.2.1 Outline of Functions

The inspector has the following functions.

Data inspection and display: The data to be inspected is analyzed according to the instruction from users, and is then displayed. Any KL1 data item can be analyzed and inspected. To specify the data analysis/display position, the element number, the car, cdr, or tail parameter can be used depending on the data type. Also it is possible to specify two or more parameters with punctuation by using a '>' mark.

Data monitor and display: The data is monitored by instructions from the user, materialized, and then displayed on the screen. To specify the data monitoring position, the element number, or the car, cdr, or tail parameter can be used depending on the data type. It is also possible to specify two or more parameters with punctuation by using '>' mark.

Variable name and variable pool: A variable name can be given to the data which is found during the inspection process. The variable name is registered in the variable pool and used to refer to the data transfer.

The inspector can refer to its own local variable pool and the variable pool of the listener which has invoked the inspector, and can register the data in them.

Output interruption: The user can interrupt the output of the inspector during the processing. With an instruction received from the user, thereafter, the output process may be resumed or canceled.

Nested call of the inspector: The inspector can be nested for calling. The nesting depth is indicated at the time of activation and termination.

Input/output destination switching: When the inspector is activated, it is given the input/output authority from the activation source process. Normally, the input/output authority is returned to the activation source process when the inspector is terminated. The input/output authority is also returned to the activation source process when the input/output destination of the inspector is switched to a new window. Thereafter, the switched inspector and the activation source processes become independent of each other and run in parallel.

4.2.2 Operation Method

4.2.2.1 Starting

The inspector is activated with a command input from the top level of the listener or a trace command during the trace processing.; When the inspector is activated during the tracing process, goals without a node number or a module name is passed for the inspection at the time of activation. When the inspector is activated at the top level of the listener, any KL1 data item specified by the user is passed.

In addition, it can be activated from the shell, too. In that case, the shell specifies the data to be inspected. For example, it is possible to investigate the content of the error message output from the shell by activating the inspector from the shell when some error occurs during the execution of the user goal on the shell. (Refer to Section2.1.6 "Error Processing" of shell (Page 23).)

The data to be inspected, which is passed at the time of activation, is put into the local variable pool of the inspector under the variable name of `ROOT`. This is used as the root of the retrieval.

【Example1】 Activating inspector at the top level of the listener

```
[1] inspect a.
a>
```

【Example2】 Activating inspector during trace processing

```
{Trace}
[1] layered:n_queens(8,Out).
@0@4096 0     layered: n_queens(8,Out)
1     * (1)     queens(1,8,begin,Out)? ins
n_queens(8,Out)>
```

4.2.2.2 Input of Commands

When the inspector is activated, a prompt is displayed, waiting for the user to input a command. On a prompt, the contents of the data kept at that stage are briefly displayed. Enter the command after the prompt.

【Example】 Displaying the prompt when the kept data is `{0,1,2,3,4,5,6,7,8,9}`

```
{0,1,2,3, ... }>
```

4.2.2.3 Variable Pool

The data item found during the inspection can have a variable name. The variable names are registered in the variable pool, and can be used to refer to the data thereafter. The inspector can refer to its own local variable pool and the variable pools of the activation source, listener, and the registered data in them. Normally, the local variable pool of the inspector is used. When user refers to the listener variable pool or the register data in it, he/she must pad the variable name with the leading "`_`" when it is registered in the listener variable pool or referred to. By this step, the variable name with "`_`" deleted is registered in the variable pool or referred to.

In giving a variable name to the data, a variable name which has already been registered cannot be used. This is difference from unification. If the activation source, the listener, is executing a goal, the variable name used to call the goal is unusable.

The variables registered in the local variable pool of the inspector except `ROOT` can be released with "`.`" and "`cr`" (period and return).

4.2.2.4 Parameters Concerned with Displaying to Window

The inspector displays the data according to the following six environmental parameters:

- Output limit value related to the structure depth
- Output limit value related to the structure length
- Output limit value related to the character string length

- Output limit value related to the atom print name length
- Parameter related to the character string output format
- Parameter related to the general data output format

The first four of the above are the parameters related to the holophrasting output, and are common to the listener. They can be referred to or set by the following commands. **print_depth**(See Page 185), for the Output limit value related to the structure depth **print_length**(See Page 186), for Output the limi tvalue related to the structure length **print_string_length**(See Page 186), for the Output limit value related to the character string length **print_atom_length**(See Page 186) for the Output limit value related to the atom print name length
The value of the listener window where the inspector has been activated is regarded as the initial value when the inspector is started.

The parameter related to the character string output format specifies the character print format, or the code format, as the character sting output format, and is valid only inside the inspector. The **print_string_mode** command (see Page 186) is used to refer to or modify the parameter value. The character print format is set as the initial value when the inspector is activated.

The parameter related to the output format of the data in general specifies the general format for display according to KL1 data type or one which displays the tag part and the value part of the data as the display format, and is valid only inside the inspector. The **print_mode** command (see Page 187) is used to refer to or modify the parameter value. The initial value when the inspector is activated is the general format.

【Caution1】 When the values of these parameters are changed through the process which has been called by nesting, the changed values will never return to the original values at the end of that process.

【Caution2】 The MRB of the data may become black by through the use of the tracer or the inspector during the execution of the goal from the listener, even when the MRB on the program written by the user is originally white. Therefore, it is impossible to precisely judge the white/black of the MRB of the data even the data in the goal is displayed by setting the parameters related to the output form of the data in general is to the format of displaying the tag part and the value part.

4.2.2.5 Switching the Destination of Input/Output

When the **fork** command (see Page 175) is executed, a new window is created, and the input/output destination of the inspector is switched to the window. The initial value of the parameter related to the display for the window is the same as that of the original window. By switching the input/output destination, the original window waits for the activation source process to input a command. Thereafter, the switched inspector and the activation source process are independent of each other and run in parallel.

When the inspector operating on the inspector window is terminated, the input/output of the monitoring process which remains active is properly performed thereafter. If it is in a state in which the input/output can not be performed, the following functions available by entering `^C`. Changes the stream automatic monitoring mode for the monitoring process activated on the window, Changes the default value of the query mode, Displays the list of the monitoring processes activated on the window, Outputs the help message, Terminating the inspector window.

【Hint1】 For more information on the monitoring process, see Section 4.2.5.1 regarding the monitor and display of data (Page 188).

【Hint2】 It is planned that the inspector will be able to be newly called with nesting in the future.

【Example1】 Closing the window after terminating the inspector on the **inspector window**

```
a> exit

 ?? Interruption (h for help): ex
```

【Example2】 Operation example for the inspector window where no input/output is performed (After activating the monitoring process from the inspector, terminate the inspector. Then output the help message for the operation to the inspector window, and display a list of the existing monitor processes.)

```
get_device(A)> m 1 result
>monitor:1 result started
get_device(A)> ex

 ?? Interruption (h for help): h
     <cr> : continue
      sam : change stream auto mode
       qm : change query mode
       mt : show monitored terms list
       ex : kill all monitoring processes and exit
        h : help
(h for help) ? mt

 %%% MONITORED TERMS %%%
  aq: 1 result B == A ?
```

4.2.2.6 Interruption of Execution

A user can interrupt an output operation of the inspector by entering `^C` from the keyboard. Thereafter, the output operation may be restarted or canceled according to instructions from the user. To resume the output operation, a linefeed code or `c` is entered from the keyboard. When `t` is entered, the output operation is canceled and the system returns to the command input level of the inspector.

【Example】 Though 100 elements are output from the 20th element of a vector consisting of 1,000 elements, the output operation is canceled intermediately.

```
{0,1,2,3, ... }> me 20,100
20 : 20
21 : 21
22 : 22
23 : 23
24 : 24
25 : 25
26 : 26

 ?? Interruption (h for help): h

   <cr>: continue
      t: return to inspector top level
      c: continue
      h: help

 ?? --------- (h for help): t
{0,1,2,3, ... }>
```

4.2.2.7 Termination

Enter the '**exit**' command (See Page 175). Note that the inspector is terminated at the same time as the activation source listener.

4.2.2.8 Environmental Variables for Inspector

If the any of the environmental variables is set when the inspector is called, that value is to be applied.

inspector:print_string_mode

Specify the output form of the character string. Set the output format of the character strings to the code format if the value is a code(atom). In other cases, character print format is specified.

【Example】Specification of the output form of a character string by an environmental variable

```
[1] env(inspector).        % From the environmental variable pool of the
   % listener, pick up and display the environmental
   % variables related to the inspector.
[2] inspect(String).
"^@^@^@^@?? "> me
string : "^@^@^@^@^@^@^@^@^@^@"  % If no environmental variable is set,
"^@^@^@^@?? "> exit            % the character print format is used
 % for displaying.
 % If the environmental variable is set,...
[3] setenv(inspector:print_string_mode,code,S).
 S = normal

[4] env(inspector).
 inspector: print_string_mode = code

[72] inspect(String).
"^@^@^@^@?? "> me
string : 16#{0,0,0,0,0,0,0,0,0,0}
                        % The code format is to be used for displaying.
"^@^@^@^@?? ">
```

inspector:initial_window_size

Specifies the size of the inspector window when it is generated. When the value is char(Characters, lines), and both the Characters and Lines specified are positive integers, the size of the window is enlarged by the number of characters specified by Character, and by the number of lines specified by the line of Lines. In the case where the value is mouse, the size and the position of the window are set by using the mouse. In other cases, char(40,10) is the value to be set.

【Caution】 The size specified should be larger than the minimum size of the window which can be generated by the PIMOS.

inspector:window_size_increment

Specifies the degree of the size change selected to the inspector window when the bigger command is executed. When the value is char(Characters, lines), and both the Characters and Lines specified are positive integers, the size of the window is enlarged by the number of characters specified by Character and by the number of lines specified by Lines. In the case where the value is mouse, the size and the position of the window are set by using the mouse. In other cases, char(10,5) is the value to be set.

inspector:window_size_increment_l

Specifies the degree of the size change selected to the inspector window when the a_little_bigger command is executed. When the value is char(Characters, lines) and both the Characters and Lines specified are positive integers, the size of the window is enlarged by the number of characters specified by Character, and the number of lines specified by Lines. In the case where the value is mouse, the size and the position of the window are set by using the mouse. In other cases, char(5,2) is the value to be set.

4.2.3 Displaying Format of Data, and the Specification Method of the Analysis/Display/Location for Monitor

4.2.3.1 Displaying Format of Data

The data format displayed by the inspector is shown below.

integer:

`integer` : integer

【Example】 Display of integer

```
1> me              % Command to display the data (to be described later)
integer : 1
1>
```

Floating point number:

`floating_point` : Floating point number

【Example】 Display of floating point number

```
1.000000E+0> me    % Command to display the data (to be described later)
floating_point : 1.000000E+0
1.000000E+0>
```

Atom:

`atom` : Atom print name [Example] Display of atom

【Example】 Display of character string

```
atom> me           % Command to display the data (to be described later)
atom : atom
atom>
```

Character:

`string` : "Character string", or,
`string` : `N#{`code, ... `}` (`N` indicates the element size.)

【Example】 Command to display the data

```
"abc"> me            % Command to change the display format of
string : "abc"       % the character string (to be described later).
"abc"> print_string_mode
 Print mode for string: character => code
"abc"> me
string : 16#{9057,9058,9059}
"abc">
```

List:

Element number : Value of the element, or
`car` : Value of car on the list
`cdr` : Value of cdr on the list
`tail` : Value of tail on the list

【Caution】 Tail indicates the end of the case for tracing the list structure. The `tail` of the list `[a,b,c]` is `[]`, and the tail of `[a|x]` is `X` at this moment.

【Example】 Display of list

```
[a,b]> me          % Command to display the data (to be described later)
0 : a
1 : b
tail : []
[a,b]> me car  % Command to display car of the list (to be described later)
car : a
[a,b]> me cdr  % Command to display cdr of the list (to be described later)
cdr : [b]
[a,b]> x
```

Vector:

Element number : Value of the element

【Example】 Display of vector

```
getb(256,L)> me % Command to display cdr of the data (to be described later)
0 : getb
1 : 256
2 : L
getb(256,L)>
```

Module:

`module$`package::module name`$`module size, or,
Element number : Value of the element

【Example】 Display of module

```
 module$pimos::pool$231 > me ,4
                    % Command to display the 4 elements
                    % from the 0(zero)th one of the structure
0 : 49
1 : pool
2 : pimos
3 : []
 module$pimos::pool$231 >
```

Code:

`code` : `code$`package name::module name : code name/number of arguments`$`module size

【Example】 Display of code

```
 code$pimos::pool:stack/1$231 > me
                % Command to display the data (to be described later)
code :  code$pimos::pool:stack/1$231
 code$pimos::pool:stack/1$231 >
```

Undefined variable:

>Unbound

【Example】 Display of undefined variable

```
X> me           % Command to display the data (to be described later)
>Unbound
X>
```

Others:

$Unknown/Tag number of data/value/, or
(tag number of data) value

【Example】 Display of data with the data type unknown to the inspector

```
 module$pimos::pool$231 > me 209
                      % Command to display the 209th element
                      % of the structure (to be described later)
209 :  $Unknown/36/16777426
 module$pimos::pool$231 > down 209
                      % Command to go down to the 209th element
                      % of the structure (to be described later)
unknown_data_type : (PREDH) 16777426
 $Unknown/36/16777426 >
```

In cases where the output form parameter of the data is in general set to the format to display the tag part and the value part, all the data is displayed in the below stated format.

$Unknown/tag number of data/value/, or
(Tag name of data) value

Display of list

【Example】 Display of data in format to display tag part and value part

```
 module$pimos::pool$231 > print_mode
                      % Command to change the display format
                      % of data (to be described later)
 Print mode: general => tag_and_value
 module$pimos::pool$231 > me ,8
                      % Command to display the 8 elements from
                      % the 0(zero)th one of the structure
0 : (INTo) 49
1 : (ATOMo) 8388749
2 : (ATOMo) 8388608
3 : (ATOMo) 0
4 : (VECTx) -1073735427
5 : (INTo) 0
6 : (MOD) -1073739797
7 : (ATOMo) 25165966
 module$pimos::pool$231 >
```

4.2.3.2 Specification Method of Analysis/Display Location for Monitor

Indicates the specification method of analysis/display location to analyze/display only the part needed by the user in the complicated structure data. The same method is used for specifying the monitoring position for monitoring the materialization of the data. To specify a part of the data one level deeper, the following parameter is specified according to each data type.

Character string: Specifies the element number of the character string.

【Example】 displaying the 5th element of the character string and going down to it.

```
[1] inspect "abcdefghijklmnopqrstuvwxyz".
"abcde?? "> me 5
string : " ... f ... "
"abcde?? "> down 5
integer : 9062
9062>
```

List:

Specifies the element number from the top of the list, or specifies the parameter **car**, **cdr**, or **tail**.

【Caution】 The list consists of the car and cdr from the structural point of view. However, each of the elements in the list is normally noted and the element numbers from the top of the list to be used for specification. Especially, it is possible to analyze/display the content of the **car** and **cdr** by specifying the atoms car and cdr as parameters. An analysis and display of the end of the list are available by specifying the parameter **tail**.

【Example】 Displaying the fifth element of the list, car, cdr, and tail, then going down to the fifth element.

```
[1] inspect [a,b,c,d,e,f,g].
[a,b,c,d, ... ]> me 5              % Display the 5th element of the list.
5 : f
[a,b,c,d, ... ]> me car            % Display car of the list.
car : a
[a,b,c,d, ... ]> me cdr            % Display cdr of the list.
cdr : [b,c,d,e,f,g]
[a,b,c,d, ... ]> me tail           % Display tail of the list.
tail : []
[a,b,c,d, ... ]> down 5
atom : f
f>
```

Vector: Specifies the element number of the vector.

【Example】 Displaying the fifth element of the vector and going down to it.

```
[1] inspect {a,b,c,d,e,f,g}.
a(b,c,d,e, ... )> me 5
5 : f
a(b,c,d,e, ... )> down 5
atom : f
f>
```

Module: Specifies the element number of the module.

【Example】 Displaying the 209th element of the module and going down to it.

```
[1] inspect module#pool.
 module$pimos::pool$231 > me 209
209 :  $Unknown/36/16777426
 module$pimos::pool$231 > down 209
unknown_data_type : (PREDH) 16777426
 $Unknown/36/16777426 >
```

Undefined variable: It is impossible to analyze any deeper at such moment.

【Example】

```
X> me
>Unbound
X> down 0
>Unbound
X>
```

Integer, floating point number, atom, code, and others: It is impossible to analyze any deeper.

【Example】

```
atom> me 1
>Illegal parameter: 1
atom> down 1
>Cannot go down: atom
atom>
```

To specify a part of the data one level shallower, the parameter '**..**' is specified. Note that it is impossible to specify the shallower part when the current data to be inspected is located at the root of the retrieval.

To specify the N level(s) deeper/shallower part of the data, the parameters described above are specified by punctuated with a '>' mark.

4.2.4 Commands

In commands entered by the user, the command name is specified in the first argument, and the needed parameters, in the second and the subsequent arguments. A comma and a null are used to punctuate the parameter arguments. The commands have abbreviations. The entry of an abbreviation also enables the command to run normally.

Some commands may not contain the second and subsequent argument parameters because the default values are supported.

The following subsections explain the commands, giving examples for some of them. In the examples, unless otherwise mentioned, 5 is specified as the output limit value related to the structure depth. 4 is specified as the output limit value related to the structure length and the character output format is specified as the character print format.

4.2.4.1 Basic Commands

help Type [h, he]

Displays the command list of the type specified in Type(atom). The type includes the following:

basic: Basic command
insp: Inspection command
monitor: Monitor command
env: Environment command
varpool: Variable pool manipulation command
all: All commands

help [h, he]

Displays the list of the command types that can be specified by the help command (help/1) which performs a type of specification.

commands Type [com]

Displays the command list of the type specified in Type(atom). The type include the following:

basic: Command name of the basic command
insp: Command name of the inspection command
monitor: Command name of the monitor command
env: Command name of the environment command
varpool: Command name of the variable pool manipulation
all: Command name of all commands

【Hint】 The types can be displayed by the help command without any parameters, help/0.

【Hint】 A brief description of each command can be displayed by using help command.

commands [com]

Displays the list of the command types which can be specified by the commands command (commands/1) which performs type specification.

exit [ex]

Terminates the inspector and returns the window input/output authority to the activation source process.

fork [f]

Creates a new window and switches the input/output destination of the inspector to the window. The input/output authority for the original window is returned to the activation source process.

inspect ?Term [i]

Suspend the execution of the inspector currently invocation and start up a new inspector. Term is the data to be inspected by the inspector.

【Example】 Nested inspector calling

```
[put(a,A),put(b,B),put(a,C),put(c,D), ... ]> inspect _Table
>Inspector break level: 1
{0,c(A,b(B,0)),a(C,a(D, & )),0}>
                ......
{0,c(A,b(B,0)),a(C,a(D, & )),0}> exit
>Inspector break level: 0
[put(a,A),put(b,B),put(a,C),put(c,D), ... ]>
```

set_title String

Sets the title of the inspector window to the character string `INSPECTOR:` with String (character string) appended.

【Caution】 If this is executed to an inspector which is operating on a window other than the inspector window, nothing is performed.

【Example】 When the following operation is executed in the inspection window, the title of the window is regarded as `INSPECTOR: generator process`.

```
process(A,B)> set_title "generator process"
process(A,B)>
```

bigger [big]

Specifies the size of the window to be 10 characters and 5 lines larger. It is also possible to specify the size by the environmental variable **inspector:window_size_increment**. (See Section 4.2.2.8 "Environmental Variables for Inspector".) In cases where it is impossible to make a window larger with a specified degree because of the position limit of the window, it may be changed within the allowed range.

【Caution】 If this is executed for an inspector which is operating on a window other than the inspector window, nothing is performed.

a_little_bigger [lbig]

Specifies the size of the window to be 5 characters and 2 lines larger. It is also possible to specify the size by the environmental variable **inspector:window_size_increment_l**. (See Section 4.2.2.8 (Page 169) "Environmental Variables for Inspector".) In the cases where it is impossible to make the window larger by the specified degree because of the position limit of the window, it may be changed within the allowed range.

【Caution】 If this is executed for an inspector which is operating on a window other than the inspector window, nothing is performed.

4.2.4.2 Commands for Investigation

me M, N

Displays N elements from the Mth element of the data currently kept.

If the M value exceeds the number of the elements in the data, an error occurs. If the number of the elements to be displayed does not reach N, only the elements available are displayed.

With respect to the specification method for the display position M see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

A negative integer can also be specified as N. In such case, N elements refers to the concrete position M0 of E inside the structure one level higher than the one containing the element E which is indicated by M, which is retrieved from the M0+N+1 (however this is 0 when M0+N+1 < 0)th element in the minus direct and displayed.

The parameters M and N can be omitted. When they are omitted, 0 and the output limit value regarding the length of the structure are used as the default values respectively.

【Hint】 **me**, is the same as **me** (to be described later) which is not accompanied with parameter(s).

me M

Displays the Mth element of the data currently kept. If the specification of the display position M is illegal, an error occurs. As for information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

me

Displays the elements starting from the 0th element of the data currently kept by the output limit value related to the structure length.

If the number of elements that can be displayed do not reach the specified number, only the elements available are displayed.

【Example1】 Displaying the contents of the factor with 100 elements.

```
{0,1,2,3, ... }> me
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
{0,1,2,3, ... }> me 50
50 : 50
{0,1,2,3, ... }> me 50,2
50 : 50
51 : 51
{0,1,2,3, ... }> me 9,-3
7 : 7
8 : 8
9 : 9
{0,1,2,3, ... }> me 80,
80 : 80
81 : 81
82 : 82
83 : 83
84 : 84
{0,1,2,3, ... }> me ,2
0 : 0
1 : 1
{0,1,2,3, ... }>
```

【Example2】 Displaying the contents of the list `[1,2,3,4,5,6,7,8,9|A]` (where `A` is an undefined variable).

```
[0,1,2,3, ... ]> me
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
[0,1,2,3, ... ]> me 5
5 : 5
[0,1,2,3, ... ]> me 8,10
8 : 8
9 : 9
tail : A
[0,1,2,3, ... ]> me 9,-3
7 : 7
8 : 8
9 : 9
[0,1,2,3, ... ]> me 10,-3
8 : 8
9 : 9
tail : A
[0,1,2,3, ... ]> me car
car : 0
[0,1,2,3, ... ]> me cdr
cdr : [1,2,3,4,5, ... ]
[0,1,2,3, ... ]> me tail
tail : A
[0,1,2,3, ... ]>
```

【Example3】Displaying the contents of the string "0123456789".

```
"01234?? "> me
string : "01234 ... "
"01234?? "> me 5
string : " ... 5 ... "
"01234?? "> me 8,10
string : " ... 89"
"01234?? "> me 9,-3
string : " ... 789"
"01234?? ">
```

【Example4】Displaying the contents of the vector {x(y(z(1,2,3))), s(t(u(4,5,6)))}.

```
{x(y( & )),s(t( & ))}> me 0>1>1
1 : z(1,2,3)
{x(y( & )),s(t( & ))}> me 0>1>1>0,10
0 : z
1 : 1
2 : 2
3 : 3
{x(y( & )),s(t( & ))}>
```

down M [null]

Goes down deeper to the position specified by M. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】It is possible to omit the command and only input parameter M.

【Example】Operation of down to the data {{{{{{0},1},2},3},4},{5,{6,{7,{8,{9}}}}}}.

```
{{{ & ,3},4},{5,{6, &}}}> me
0 : {{{{ & ,1},2},3},4}
1 : {5,{6,{7,{8, & }}}}
{{{ & ,3},4},{5,{6, &}}}> 0
0 : {{{ & ,1},2},3}
1 : 4
{{{ & ,2},3},4}> 0>0>0>0>0
integer : 0
0> ..>..>..>..>..>..>1>0
integer : 5
5>
```

up N [u]

Traces data upward by N levels.

up [u]

Traces data upward by one level.

【Hint】The same as executing down ...

【Example】Operation of down and up to the data {{{{{{0},1},2},3},4},{5,{6,{7,{8,{9}}}}}}.

```
{{{ & ,3},4},{5,{6, &}}}> me
0 : {{{{ & ,1},2},3},4}
1 : {5,{6,{7,{8, & }}}}
{{{ & ,3},4},{5,{6, &}}}> 0
0 : {{{ & ,1},2},3}
1 : 4
{{{ & ,2},3},4}> 0>0>0>0>0
integer : 0
0> up 99
>Root
{{{ & ,3},4},{5,{6, & }}}> 1>0
integer : 5
5>
```

length [len]

Displays the number of elements in the data.

If the data is a character string, the element size of that length is displayed. In the case where the cdr part has not yet materialized in a list, the portion already known portion is displayed (see [Example 3]).

【Example1】 Investigates the number of elements in a vector.

```
{0,1,2,3, ... }> length
 100
{0,1,2,3, ... }>
```

【Example2】 Investigates the size of the character string.

```
"01234?? "> length
 Length of the string is 82,
 element size of the string is 16
"01234?? ">
```

【Example3】 Investigates the number of the elements in the list `[0,1,2,3,4|A]`. (where `A` is a variable)

```
[0,1,2,3, ... ]> length
>The tail of the list is variable: A,
 length of the list is greater or equal to 5
[0,1,2,3, ... ]>
```

show ?Term [sh]

Shows the value of the specified data Term briefly.

history N

Displays the history of data tracing for the past N times of the operations. The position of the data traced with the 'down7 command and the data list at that time are displayed on the screen.

history

Displays the history of the data tracing for the past several times of the operations. The position of the data traced with the 'down' command and the data list at that time are displayed on the screen. The output limit value related to the length of the structure is used as the number of times for tracing operations.

【Example】 Displaying the history of the data tracing when the data `{0,{1,{2}}}` is traced several times.

```
2> history
 0 root  :  {0,{1,{2}}}
 1 down 1 :  {1,{2}}
 2 down 1 :  {2}
 3 down 0 :  2
2>
```

push_root ?Term [pu]

The specified data, Term, is inspected. The contents of the variable `ROOT` registered in the local variable pool of the inspector is replaced with new inspection data.

【Example】

```
[0,1,2,3, ... ]> pu a
atom : a
a>
```

4.2.4.3 Commands for Monitor

monitor M Name [m]

Activates the monitoring process to the Mth element of the data currently stored. Any character string which does not include a null can be specified to Name as a memorandum of the data. As for information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details on monitoring data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

【Example】 Starts monitoring process from the inspector monitor M

```
process(A,B)> m 1 input_stream
>monitor:1 input_stream started
process(A,B)>
```

monitor M [m]

Activates the monitoring process to the Mth element of the data currently stored. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details on monitoring the data see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

monitor_and_fork M Name [mf]

Activates the monitoring process to the Mth element of the data currently stored. Creates a new inspector window to switch the destination of the input/output for the monitoring process to the new window at the same time. Any character string which does not include a null can be specified to Name as a memorandum of the data. The identification number for this monitoring process and Name are displayed in the title of the new window. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details of monitoring the data see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

【Example】 Example of activation from the inspector

```
process(A,B)> mf 1 input_stream
process(A,B)>
```

The message of monitoring process is displayed on the new window.

```
>monitor:1 input_stream started
```

monitor_and_fork M [mf]

Activates the monitoring process to the Mth element of the data currently stored. Creates a new inspector window to switch the destination of the input/output for monitoring process to the new window at the same time. The identification number is displayed in the title of the new window. For information on how to specify the display position M, see 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details regarding monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

conditional_monitor M Term Name [cm]

Activates the monitoring process to the Mth element of the data currently stored. However, the process of monitoring is reported only when the data to be monitored has materialized for the data which is collated with the term specified to Term. If any variable is contained in the Term, a comparison is not performed for the variable part. Any character string which does not include a null can be specified to Name as a memorandum of the data. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details on monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

【Example】 Example of activation with condition(s) from the inspector

```
process(A,B)> cm 1 [get(_)|_] input_stream
>monitor:1 input_stream started
process(A,B)>
```

conditional_monitor M Term [cm]

Activates the monitoring process to the Mth element of the data currently stored. However, the process of monitoring is reported only when the data to be monitored has materialized for the data which is collated with the term specified to Term. If any variable is contained in the Term, a comparison is not performed for the variable part. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details on monitoring the data, see Section 4.2.5.1 "Monitoring and Displaying of Data".

conditional_monitor_and_fork M Term Name [cmf]

Activates the monitoring process to the Mth element of the data currently stored. Creates a new inspector window to switch the destination of the input/output for monitoring process to the new window at the same time. However, the process of monitoring is reported only when the data to be monitored has materialized for the data which is collated with the term specified to Term. If any variable is contained in the Term, a comparison is not performed for the variable part. Any character string which does not include a null can be specified to Name as a memorandum of the data. The identification number is displayed in the title of the new window. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details of monitoring the data see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

【Example】 Example of activation from the inspector

```
process(A,B)> cmf 1 [get(_)|_] input_stream
process(A,B)>
```

The message for the monitoring process is displayed on the new window.

```
>monitor:1 input_stream started
```

conditional_monitor_and_fork M Term [cmf]

Activates the monitoring process to the Mth element of the data currently stored. Creates the new inspector window to switch the destination of the input/output for monitoring process to the new window at the same time. However, the process of monitoring is reported only when the data to be monitored has materialized for the data which is collated with the term specified to Term. If any variable is contained in the Term, a comparison is not performed for the variable part. The identification number is displayed in the title of the new window. For information on how to specify the display position M, see Section 4.2.3.2 (Page 173) "Specification Method of Analysis/Display Location for Monitor".

【Hint】 For the details on monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

stream_auto_mode [sam]

Changes the value of the stream automatic monitoring mode which is set as the default when the monitoring process is activated by using the toggle function.

【Hint】 For the details on monitoring the data see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data". (see page 190)

【Example】 Example of the operation of the inspector

```
process(A,B)> sam
 Monitor: stream_auto_mode = off
process(A,B)>
```

query_mode [qm]

Changes the value of the stream query mode which is set as a default when the monitoring process is activated by using a toggle function.

【Hint】 For the details on monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data". (see page 190)

【Example】 Example of operation from the inspector

```
process(A,B)> qm
 Monitor: query_mode = off
process(A,B)>
```

silent_mode Param [sm]

Changes the mode to issue a beep sound when the monitoring process is displayed. Param parameters are as follows:

all_on: Never sounds a beep for any monitoring process.

all_off: Sounds a beep for all monitoring processes.

window_on: Never sounds a beep for any monitoring process which performs input/output on the window executed.

window_off: Sounds a beep for all monitoring processes which perform input/output on the window executed.

【Caution1】 Monitoring process as a whole which is activated to 1 goal under execution from the listener is regarded as the entire monitoring process. Whatever remains active for the goal which has been terminated the execution is not included in the process to be controlled.

【Caution2】 It is planned that the mode for each monitoring process will be changed in the future. The current specification method for Param may be modified according to such change.

【HintReference】 For the details on monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data".

monitored_terms [mt]

Displays the list of monitoring processes currently activated.

【Hint】For the details on monitoring the data, see Section 4.2.5.1 (Page 188) "Monitoring and Displaying of Data". Regarding the operation of the group of monitoring processes, see "Operation to Group of Generated Monitoring Process" (Page 192).

【Example】Example of operation from the inspector

```
process(A,B)> m 1 input_stream
>monitor:1 input_stream monitored
process(A,B)> cm 1 [get(_)|_] input_stream2
>monitor:2 input_stream2 monitored
process(A,B)> mt

 %%% MONITORED TERMS %%%
  aq:1 input_stream C == A
  aq:2 input_stream2 [get(D)|E] == A ?
```

4.2.4.4 Commands for Environment

print_depth D [pd]

Sets the output limit value related to the depth of the structure to D.

【Example】

```
p({{ & ,3},4})> me
0 : p
1 : {{ & ,3},4}
p({{ & ,3},4})> pd 10
p({{ & ,3},4})> me
0 : p
1 : {{{{{0},1},2},3},4}
p({{ & ,3},4})>
```

print_depth [pd]

Refers to and sets the output limit value related to the depth of the structure.

After the current value is displayed, it is replaced with a new value entered by the user. If only the linefeed code is entered, the value is not changed.

【Example】

```
p({{ & ,3},4})> me
0 : p
1 : {{ & ,3},4}
p({{ & ,3},4})> pd
 Print depth: 2 => 10
p({{ & ,3},4})> me
0 : p
1 : {{{{{0},1},2},3},4}
p({{ & ,3},4})>
```

print_length L [pl]

Sets the output limit value related to the length of the structure to L.

print_length [pl]

Refers to and sets the output limit value related to the length of the structure.

After the current value is displayed, it is replaced with a new value entered by the user. If only the linefeed code is entered, the value is not changed.

print_string_length L [psl]

Sets the output limit value related to the length of the character string to L.

print_string_length [psl]

Refers to and sets the output limit value related to the length of the character stream.

After the current value is displayed, it is replaced with a new value entered by the user. If only the linefeed code is entered, the value is not changed.

print_atom_length L [pal]

Sets the output limit value related to the length of the atom print name to L.

print_atom_length [pal]

Refers to and sets the output limit value related to the length of the atom print name.

After the current value is displayed, it is replaced with a new value entered by the user. If only the linefeed code is entered, the value is not changed.

print_string_mode [psm]

Changes the output form of the character string. The following two formats are prepared for the output form: character print format characters and code format (code). They are switched by toggle operation. The initial value when the inspector is activated is the character print format.

【Example】Example of output in changing the output form of the character string

```
"ab"> me
string : "ab"
"ab"> psm
 Print mode for string: character => code
"ab"> me
string : 16#{9057,9058}
"ab"> psm
 Print mode for string: code => character
"ab">
```

print_mode [pm]

Changes the data output form. The following two types of output formats are prepared as output forms general format general and a format displaying the tag part and the value part (tag_andvalue). They are switched by toggle operation. The initial value when the inspector is activated is the general format.

【Caution】 The MRB of the data may become black through the use of the tracer or the inspector during the execution of the goal from the listener, even for MRB on programs written by the user which are originally white. Therefore, it is impossible to correctly judge the black/white of the MRB for data in the goal by using this mode.

environment [env]

Displays the value of the parameter for the display of the inspector.

【Example】 Example of output

```
a> environment
 Print depth           : 4
 Print length          : 10
 Print string length   : 20
 Print atom length     : 30
 Print mode for string : character
 Print mode            : general
a>
```

4.2.4.5 Commands to Operate Variable Pool

name ^VarName [n]

Substitutes the data currently kept for the specified variable. When a name with the leading "_" is specified as a variable name, the name is registered in the listener variable pool. The other names are registered in the local variable pool of the inspector.

【Caution】 A variable name which has already been registered can not be used. This is a difference from the unification process.

list Type [ls]

Lists the variable names and data for the variables registered in the inspector local variable pool and the listener variable pool. The variable names of the variables registered in the listener variable pool are displayed with the leading "_". Type parameters are as follows: The variable names for variables registered in the listener variable pool are displayed with a leading -s.

local: Displays a list regarding only the local variable pool of the inspector. The followings can be specified as Type.

listener: Displays a list regarding only the listener variable pool.

all: Displays a list regarding both the local variable pool of the inspector and the listener variable pool.

list [ls]

Lists the variable names and data for the variables registered in the inspector local variable loop and the listener variable pools.

The variable names of the variables registered in the listener variable pool are displayed with a leading "_".

Period return

When "." and "cr" are entered, all the variables except ROOT registered in the inspector local variable pool are released.

【Caution】 To release the variables from the listener variable pool, "period return" must be executed at the top level of the listener.

【Example】 Executing a variable pool manipulation command.

1. Substitutes current data for a variable A by **name** command.
2. Uses **list** command to display a list of the variables. The variables A and T are the user-defined local variables of the inspector. ROOT is a variable indicating the data passed when the inspector is activated, and _L and _Table are the variables registered in the listener variable pool.
3. When "." command is executed, all the user-defined inspector local variables are released.

```
a> name A                                                    % (1)
a> list                                                      % (2)
 A = a
 ROOT = a(b,c,A)
 T = {{"b",A,0},{"a",B,{"e",C,0}},{"d",D,0},{"c",E,0}}
 _L = [put(a,A),put(b,B),put(a,C),put(c,D),put(a,E), ... ]
 _Table = {0,c(A,b(B,0)),a(C,a(D,a(E, & ))),0}
a> .                                                         % (3)
 ROOT = a(b,c,A)
 _L = [put(a,A),put(b,B),put(a,C),put(c,D),put(a,E), ... ]
 _Table = {0,c(A,b(B,0)),a(C,a(D,a(E, & ))),0}
a>
```

4.2.5 Monitoring and Displaying Data

4.2.5.1 Outline

The variable may be undefined at the moment, though the data has been investigated. The data monitoring/displaying function monitors and displays the materialization of those undefined variables. The user can take the same steps to investigate the part with the data to specify the position to be monitored.

When the monitoring process for the data is specified, the process to be monitored for the materialization of that data is generated. Thereafter, this process is called **the monitoring process**. If the data under monitoring is materialized at depth level 1, it is determined whether such data matches with the term specified as a condition for reporting during the monitoring process for the data. When they are matched, the content of the data is reported to the window where the inspection process was activated. A prompt is displayed at the time of reporting, and

the commands for the inspector can be executed without certain exceptions. In the case where they are not matched, or matching can not be determined, no report can be issued.

Basically, only the materialization of data at depth level 1 is investigated by the monitoring process. A mode to continue monitoring the cdr part as a default when the data is materialized to the list is also provided. This mode is called **the stream automatic monitoring mode**.

The monitoring process as a default accepts a command from the user after displaying the materialization. It is possible to perform the display function only. For such purposes, **the query mode** is used.

It is possible to perform the following functions regarding the monitoring process generated: display of a list, forceful termination, and change of query mode.

Furthermore, the message is output at the start and the end of monitoring.

4.2.5.2 Operation Method

Start

The operation is started by a command from the inspector or the listener. The commands listed below are provided for this function. **monitor** (See Page 181): Starts from the inspector. **monitor_and_fork** (See Page 182): Reports to the other window. **conditional_monitor** (See Page 182): Reports only when the materialization has resulted for the value complying with the specified condition. **conditional_monitor_and_fork** (See Page 183): Reports to the other window only when the materialization has resulted for the value complying with the specified condition.

For information on how to start the monitoring process, see Section 4.1.6.8 (Page 110), "Debug commands". For information on how to start it during the trace processing, see Section 4.1.7.9 (Page 131) "Trace Commands".

The following message is output to the output destination window when the monitoring process is started.

```
>monitor:1 stream started
```

`1`, `stream` The above mentioned stream indicates the character string for memorandum and the identifier number of the monitoring process which has been started.

【Example1】 Starting the monitoring process from the inspector

```
get_device(A)> m 1 result
>monitor:1 result started
get_device(A)> ex
aq:1 result E == normal(B,C,D) ?
```

【Example2】 Starting the monitor from the listener.

```
{Trace}
?- del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? m A, result
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? 1
aq:1 result  E == normal(B,C,D) ?
```

Command input

If data under monitoring is materialized at depth level 1, it is determined whether such data matches with the term specified as a condition for reporting during the monitoring process of the data. When they are matched, the content of the data is reported to the window where the inspection process was activated.

【Example】 Display of materialization

```
aq:1 stream [N|O] == [putl(L)|M] ?
```

In the above example, `a` indicates that the stream automatic monitoring mode is on, and `q` indicates that the query mode is set to on. When those modes are off, the characters `a` and `q` are not displayed. `1` indicates the identification number given to the monitoring process. The identifier number is uniquely assigned for the execution of each goal one time from the listener. The `stream` is the memorandum (character string) specified by the user when the monitoring process is generated. If the memorandum is omitted, nothing is displayed. `[N|O]` is the condition for reporting. The report is issued only when the data under monitoring is a list, as in the above example. `[putl(L)|M]` is the content of materialization. The prompt "?" is simultaneously displayed on the screen, and an input of command from the user is awaited. Enters the command after the prompt. The commands the same as those for the inspector are valid for this function with certain exceptions. (See Section 4.2.5.3 (Page 193) "Commands".)

Stream automatic monitoring mode

The data is only generally investigated at depth level 1. The monitoring process is terminated after the end of displaying accompanied by the materialization and executing the above command(s). However, it is indicated that the monitoring of the cdr part of the list is to be continued in those cases where the stream spanned the multiple processes.

Therefore, the mode called stream automatic monitoring mode is equipped with an inspector. The stream automatic monitoring mode is set to either on or off. If the data specified to be monitored is materialized to the list with this mode on, such cdr part is to be monitored continuously by the default. In the case where it is materialized to a data type other than the list, or where the stream automatic monitoring mode is set to off, the subsequent monitoring process is not performed. The user needs to generate a monitoring process newly by using the command in such cases. The command **stream_auto_mode** is used for referring to and setting the stream automatic monitoring mode (See Pages 184, 194).

Query mode

As a default, the investigation of the data in the conversational mode and a new process to be monitored is generated after the materialization of the data is displayed during the monitoring process. It is also possible to display the materialization of the mode only, which is called the query mode, when command execution in the conversational mode is evidently recognized as unnecessary in advance.

The query mode is set to either on or off. When the query mode is set to off, and the stream automatic monitoring mode is set to on, the display of the materialization is only performed by the monitoring process. In other cases, the command input by the user is awaited. The command **query_mode** is used for referring to and setting the query mode (see Pages 184, 194).

Validation term for monitoring process

The monitoring process generated at the execution of a certain goal from the listener (at the start of the execution or during the trace process) remains active until the termination of the execution on the goal which is waiting for the materialization of the data. Thereafter, it is terminated at the same time as the goal is terminated. If the monitoring process is desired to be continued even after the termination of the execution of the goal, the destination of input/output must be changed in the following method. The monitoring process which is switched according to the above-mentioned method is not forcefully terminated, even when the execution of the goal is terminated, and operates independently until the termination of the listener.

Change of destination for input/output of monitoring process

The following methods are available for changing the destination of the monitoring process.

- Specifies the other window at the starting form of the listener or the inspector by using the **monitor_and_fork** command or the **conditional_monitor_and_fork** command.
- Switches the input/output destination of the monitoring process to the other window by using the **fork** command.

Initial value and environmental variables of each mode

The initial value of each mode is set each time that the goal is executed from the listener. When the following environmental variables are set when the execution of the goal from the listener is started, each mode follows the setting of those parameters.

monitor:stream_auto_mode

Sets the default value of the stream automatic monitoring mode when the monitoring process of the inspector is generated. If the value is off, the value of the stream automatic monitoring mode is to be set to off. In other cases, it is to be set to on.

monitor:query_mode

Sets the default value of the query mode when the monitoring process of the inspector is generated. If the value is off, the value of the query mode is to be set to off. In other cases, it is to be set to on.

monitor:silent_mode

Specifies the default value of the mode to sound a beep when the monitoring process is displayed. When this value is on, the mode which does not sound a beep is set. In other cases, the mode to sound a beep at the time of displaying the monitoring process is set.

Termination

To terminate the monitoring process which is waiting for the command input, execute the **exit** command. When only the linefeed code is entered to the prompt, the operation of the monitoring process develops as follows.

- If the monitoring process is trying to continue monitoring automatically, go on monitoring.
- In other cases, the monitoring process is to be terminated.

The following message is output on the output destination window when the monitoring process is terminated.

```
>monitor:1 stream terminated
```

`1`, `stream` The stream in the above message indicates the identification number of the monitoring process which has been activated and the character string for the memorandum.

Operation to the set the monitoring processes generated

The following operations for the monitoring process generated during the execution of the goal can be performed by the user through the starting of the goal and its termination: display of a list, forceful termination, change of report condition, switch of stream automatic monitoring mode and query mode. The **monitored_terms** command (see Page 185) is used for such cases.

When the **monitored_terms** command is executed, a list of the monitoring processes which are specified to the window executed whose input/output destination is command among the ones currently activated is displayed. After this display, the following operations are available by entering a command in the conversational form. The monitoring processes to be operated are specified by the identification numbers displayed in the list.

kill N, ... :	Forcefully terminates the monitoring process which has the identification number N, ...
sam N, ... :	Changes the stream automatic monitoring mode of the monitoring process which has the identification number N, ...
qm N, ... :	Changes the query mode of the monitoring process which has the identification number N, ...
cc N Term:	Changes the reporting condition of the monitoring process which has the identification number N.
return key or q:	quits the list display.
h:	Outputs the help message.

【Example1】 Displaying a list of the monitoring process (Activating the monitoring process during the trace operation and displaying the list.)

```
{Trace}
?- del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? m A, result
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? mt

 %%% MONITORED TERMS %%%
aq:1 result B == A ? h

     <cr> : quit
  k N,... : kill monitor
sam N,... : change stream_auto_mode
 qm N,... : change query_mode
cc N Term : change condition
        q : quit
        h : help

(h for help) ?
```

【Example2】 Displaying a list of the monitoring process and their operations (After displaying the list, the stream automatic monitoring mode of the monitoring process with the identification number 1 is changed.)

```
 %%% MONITORED TERMS %%%
aq:1 file G == F
aq:2 pathname I == H
aq:4 message K == J ? sam 1

 %%% MONITORED TERMS %%%
 q:1 file G == F
aq:2 pathname I == H
aq:4 message K == J ?
```

4.2.5.3 Commands

The commands of the inspector which can be operated for the monitoring process are as follows. The operation content is also indicated if the execution from the inspector differs from the operation content.

- All the basic commands except **inspect** (See Section 4.2.4.1 (Page 175) "Basic commands)"
- All the inspection commands except **push_root** (See Section 4.2.4.2 (Page 177) "Command for Investigation").
- All the monitoring commands (See Section 4.2.4.3 (Page 181) "Commands for Monitor").
- All the environmental commands (See Section 4.2.4.4 (Page 185) "Commands for Environment".)
- All the variable pool operation commands (See Section 4.2.4.5 (Page 187) "Commands to Operate Variable Pool".)

【Hint】 **Inspect** is scheduled to be available for execution in the future.

change_condition Term [cc]

Changes the conditions for reporting the process under monitoring process.

stream_auto_mode [sam]

Changes the value of the stream automatic monitoring mode of the monitoring process, when it is issued to the monitoring process.

query_mode [qm]

Changes the value of the query mode of the monitoring process, when it is issued to the monitoring process.

monitored_terms [mt]

Displays a list of the monitoring processes other than itself, when it is issued to the monitor process.

【Example】Monitoring the input stream to buffer and window when input/output is performed to the window where the buffer is inserted Monitors the input stream to the window and the buffer during trace processing.

Listener Window: Monitor the input stream to window and buffer during trace processing.

```
{Trace}
[1] del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? mf A, device
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? 1
@0@4096 2     del: wait_result(normal(B,C,D))
3       (1)     pimos:: buffer: interaction_filter(E,B)
4     * (2)     p(E)? mf E, buffer
@0@4096 2     del: wait_result(normal(B,C,D))
3       (1)     pimos:: buffer: interaction_filter(E,B)
4     * (2)     p(E)? 2
```

Inspector window 1: Monitoring the input stream of the window. Reports the arrival of the get b message, and further monitors the input contents.

```
>monitor:1 device started
aq:1 device D == normal(A,B,C) ? cm 1 [getb(_,_)|_] wait_getb_msg
>monitor:2 wait_getb_msg started
aq:1 device E == normal(A,B,C) ? ex
>monitor:1 device terminated

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:2 wait_getb_msg [getb(F,G)|H] == A ?
aq:2 wait_getb_msg [getb(K,L)|M] == [getb(490,I)|J] ? m car>2 getb_result
>monitor:4 getb_result started
aq:2 wait_getb_msg [getb(N,O)|P] == [getb(490,I)|J] ?

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:2 wait_getb_msg [getb(Q,R)|S] == J
  aq:4 getb_result T == I ?
aq:4 getb_result V == normal(U) ? m 1 input_line
>monitor:5 input_line started
aq:4 getb_result W == normal("pd.
") ?
>monitor:4 getb_result terminated
aq:5 input_line X == "pd.
" ?
>monitor:5 input_line terminated
```

Inspector window 2: Monitoring the input stream of the buffer

```
>monitor:3 buffer started

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:3 buffer B == A ? qm 3

 %%% MONITORED TERMS %%%
  a :3 buffer C == A ?
a :3 buffer H == [set_size(char(20,8),D),activate(E),prompt("input>"),gett(F)|G]
a :3 buffer J == [activate(E),prompt("input>"),gett(F)|I]
a :3 buffer K == [prompt("input>"),gett(F)|I]
a :3 buffer L == [gett(F)|I]
a :3 buffer O == [print_depth(M,M),putb("pd= "),putt(M),nl|N]
a :3 buffer P == [putb("pd= "),putt(4),nl|N]
a :3 buffer Q == [putt(4),nl|N]
a :3 buffer R == [nl|N]
```

4.2.6 List of Commands for Inspector and Monitor

Table 4.3: List of commands for inspector and monitor

Command	Abbreviation	Meaning	Referential
help Type	he,h	Displaying the list of the commands specified by Type	175
help	he,h	Displaying the list of the type which can be displayed by help/1	175
commands Type	com	Displaying the list of the commands specified by Type	175
commands	com	Displaying the list of the type which can be displayed by command/1	175
exit	ex	Termination of the inspector	175
fork	f	Switching the input/output destination window	175
inspect ?Term	i	Activating the child inspector	176
set_title String		Setting the title of the window	176
bigger	big	Changing the size of the window	176
a_little_bigger	lbig	Changing the size of the window	176
me M,N		Displaying N of the data from the Mth element.	177
me M		Displaying M element of the data	177
me		Displaying the data of the specified number from the 0th element of the data	177
down M	[Null]	Go down the data	179
up N	u	Go to the data N levels above.	179
up	u	Go to the data 1 level above.	179
length	len	Displaying the number of elements the data has.	180
show ?Term	sh	Displaying the data in simple form.	180
history N		Displaying the history gone back for N times	180
history		Displaying the history gone back for the specified times	181
push_root ?Term	pu	Setting the data to be investigated	181
monitor M Name	m	Monitoring the Mth parameter of the data	181
monitor M	m	Monitoring the Mth parameter of the data	181
monitor_and_fork M Name	mf	Monitoring on the other window	182
monitor_and_fork M	mf	Monitoring on the other window	182
conditional_monitor M Term Name	cm	Monitoring the Mth element of the data with condition(s)	182
conditional_monitor M Term	cm	Monitoring the Mth element of the data with condition(s)	183
conditional_monitor_and_fork M Term Name	cmf	Monitoring on the other window with condition(s)	183
conditional_monitor_and_fork M Term	cmf	Monitoring on the other window with condition(s)	183
change_condition Term	cc	Setting the condition(s) of the monitor	194
stream_auto_mode	sam	Setting the stream automatic monitoring mode	184,194
query_mode	qm	Setting the query mode	184,194
silent_mode Param	sm	Setting the existence of the beep sound	184
monitored_terms	mt	Displaying a list of monitors	185,194
print_depth <D>	pd	Setting the output limit value of the structure depth	185
print_length <L>	pl	Setting the output limit value of the	186

Chapter 5

Tuning Tools

As tuning tools, the following functions have been prepared: a performance meter (parameter, rmonitor, amonitor) which displays the operation state of each node when the program is executed, and ParaGraph which displays the operation state of each program after the execution state of the program is measured by the command for profile provided by the listener, the occurrence state of communication between modes and the execution state of the program in various graph formats.

5.1 Performance Meter

The performance meter is a tool to observe the operation state of each node. **pmeter** displays the state of each node on the mode configuration diagram **rmonitor**[1] displays the operation state of each node in the gradation graph, and amonitor,[2] displays the average operation rate of all the node in a bar graph.

5.1.1 Outline of Functions

The main functions of the performance meter are described below.

Displaying Operation State

The operation state of each node is measured and displayed at certain regular time intervals by the performance meter. The following items are measured as the operation state: calculation time for each time interval, [3] receiving time for the message for communication, transmission time, GC time, and idle time. The ratio of the non-idle time (sum of calculation time, receiving time, and transmission time) for each time interval is displayed as the operating ratio of each node in **pmeter** and **rmonitor**. Setting the environmental variables enables the operation state to be displayed. It is also possible to display the environmental variables. In **amonitor**, the average value of the non-idle time ratio for each time interval is displayed as the average operating ratio.

【Caution1】 If the same node is measured/displayed by multiple performance meters on the multi PSI or PIM, the operation has not been correctly performed. Similarly, the operation is not correct if the command for the profile of the listener is executed while the performance meter is activated.

【Caution2】 An appropriate operation is not available on Pseudo multi PSI or Pseudo PIM/m.

[1] rmonitor is the abbreviation of runtime monitor.

[2] amonitor is the abbreviation of runtime average monitor.

[3] calculation time: non-idle time - (receiving time + transmission time + GC time).

Figure 5.1: Display example of rmonitor

5.1.2 Operation Method

Calls from the shell as a command.

pmeter

Activates pmeter. Allocates each node on the matrix, and displays the operating ratio of each node by the gradation of the color.

【Hint1】 It is possible to display each node by the operating state (calculation, transmission, receipt) by setting the environmental parameter **pmeter:item**.

rmonitor

Activates **rmonitor**. Displays the operation state of each node in the gradation graph. The axis of abscissas is the cycle number [4], and the axis of the ordinates is the node number. The operating ratio each node is indicated by the gradation of the color. If GC occurs, a triangle mark (△) is displayed on the graph corresponding to the node with GC. An example of the **rmonitor** output screen is shown in Figure5.1.

【Hint1】 It is possible to display each operation state (calculation, transmission, receipt) by setting the environmental parameter **pmeter:item**.

【Hint2】 The operating ratio of each node is displayed from the left side in the gradation graph. Moreover, the screen is automatically scrolled when the full display is on the screen.

[4] Cycle number: The number sequentially assigned to each time interval from 1.

Figure 5.2: Output screen example of amonitor

amonitor

Activates **amonitor**. The average operating ratio of all nodes is displayed in the bar graph by using colors to classify the operation state. The axis of abscissas is the cycle number, and the axis of the ordinates is the average operating ratio. An example of an **amonitor** output screen is shown in Figure5.2.

【Caution1】 The display does not consider the set value of the environmental variable **pmeter:item**.

5.1.3 Environmental Variables

The following environmental variables are used for setting the attributes of the performance meter.

pmeter:interval

Sets the time interval of the measurement by integers indicating millisecond(s). When no environmental variable is set, the measurement is performed at intervals of 2,000 milliseconds.

pmeter:display

Sets the atom color displayed in the color display, and sets the atom mono displayed in the monochrome mode. When a color display is used, and no environmental variable is set, the display will be in the color mode. For monochrome displays, the display will be in monochrome mode regardless of the value of the environmental variable.

pmeter:timer_node

Specifies the system timer built into the environmental variable or the timer for measurement on the FEP. To specify the built-in system timer, set the atom system_timer, and set the FEP number for specifying the timer for measurement on the FEP. As for systems with two or more FEPs, it is appropriate to assign FEPs which are different from each other to set the window for displaying ones to set the timer for measurement (in order to make the measurement error small). The FEP number when the master FEP is used is 0(zero). The FEP number is set by integers from 0 to 3. It is necessary to specify the environmental variables **pmeter:timer_username and pmeter:timer_password** without fail in order to specify values other than

the FEP number or the I/O node name used for activating the performance meter. In the case in which no environmental variable is set, the built-in system timer is to be used.

【Caution】 When the environmental variable is undefined, the system timer which is built in the PIM or the PSI is used as a timer for measurement. It is needless to define this environmental variable normally except for the PIM which does not incorporate the system timer.

pmeter:timer_username

pmeter:timer_password

When a value other than the FEP number or the I/O node name is used for activating the performance meter by the environmental variable **pmeter:timer_node**, it is necessary to set the user name and the password for the SIMPOS which are registered in the appropriate FEP or the I/O node by using the atoms.

pmeter:display_node

Specifies the FEP number or the I/O node name to set the window for display. As for systems with a slave FEP, the integers 0 to 3 are used for specification. When only the master FEP is used, the FEP number is 0 (zero). It is necessary to specify the environmental variables **pmeter:display_username and pmeter:display_password** without fail in order to specify values other than the FEP number or the I/O node name used for activating the performance meter.

pmeter:display_username

pmeter:display_password

When a value other than the FEP number or the I/O node name used for activating the performance meter by the environmental variable **pmeter:display_node**, it is necessary to set the user name and the password for the SIMPOS which are registered in the appropriate FEP or the I/O node by using the atoms.

pmeter:gateway_node

Specifies the FEP number for the gateway to display in the remote mode. As for systems with two or more FEPs, it is appropriate to assign FEPs which are different from each other to set the window to display to the FEP which sets the timer for measurement (to make the measurement error small). The FEP number when the master FEP is used is 0(zero). The FEP number is set by the integers from 0 to 3. In the case where no environmental variable is set, the FEP number specified to the environmental variable task:fep is used.

pmeter:item

Specifies the contents to be displayed by the state of the operation. The state of the operation is specified by a bit string with 3 bits (which is expressed by the binary integer string from 001 to 111). The bit string is set in accordance with the order of the calculation, transmission , and receipt. When 1 is specified, those items are displayed, and they are not to be displayed if 0 is specified. For example, 100 is specified to display only the proportion of the calculation which is noted. Similarly, 011 displays the communication (transmission and receipt) portion. When no environmental variable is set, all the operation states (111) are measured and displayed.

【Example1】Setting Environmental Variables

```
SHELL> setenv pmeter:display = mono         % Displays in monochrome
SHELL> setenv pmeter:interval = 3000
SHELL> setenv pmeter:display_node = icpsi523
SHELL> setenv pmeter:display_username = akana
SHELL> setenv pmeter:display_password = rika
SHELL> setenv pmeter:timer_node = 0
SHELL> setenv pmeter:gateway_node = 0
SHELL> setenv pmeter:item = 100             % Displays calculation only
SHELL> rmonitor &
```

5.2 ParaGraph

The ParaGraph[5] is a performance evaluation supporting tool to carry out a graphic display of the execution process regarding the parallel programs by graphics. It displays the operation status of each node, the generation status of the communication between nodes, and the execution status of the programs in the graph based on the execution status information gathered by using the commands profile_node, profile_msg, and the profile of the listener. This tool is activated from the top level of the listener.

5.2.1 Outline of Functions

The main function of the Paragraph is outlined below.

(1) Graphics Display of Execution Status Information

What, When, Where and how much the process has been performed is regarded as the unit of the basic display, and the various execution status information is output in a combined format by the ParaGraph. Each display unit depends on the type of the execution status information measured. When the execution status information is the operation status of the node. What corresponds to the operation status (calculation, transmission, receipt GC)[6], and is indicated by each operation status name. If it is the generation status of the communication between nodes, What corresponds to the message for communication and is expressed by the message name for communication. In the case where it is the execution status of the program, What corresponds to the goal and shown by the name of the predicate. When corresponds the cycle[7] which has collected the execution status information and is indicated by the cycle number[8]. Where corresponds to the node which has executed the goal and expressed by the node number. The extent to which a process has been performed is indicated as follows: Operation status of the node: ratio (%) of each operation state to the operation time of the node Generation status of the communication between nodes: times of transmission or times of receipt Execution status of the program: number of reductions[9] or number of suspensions. The outline of the display format is as follows. (Refer to Section 5.2.3.2 for an example of a display.)

- **When**
  Individually displays What, When, and How much the process has been performed (for example how many times something has been reduced) in the line chart.
- **Overall What $\times$ When**
  Displays What, When, and how much the process has been performed in the format of piling the line charts in one figure.
- **What $\times$ Where**
  Individually displays What, Where, and how much the process has been performed in the line chart.

[5] ParaGraph $\fallingdotseq$ Parallel + Graphic

[6] The measurement is performed based on the following assumption. Operation Time of Node = Calculation Time + Transmission Time + Receipt Time + GC time + Idle Time.

[7] The execution status information at a certain regular time interval is collected by the profile command of the listener. This time interval is called a cycle.

[8] The number sequentially assigned to each time interval from 1.

[9] The number of reductions and suspensions as for each goal are combined into a total by the predicate name.

- **Overall What × Where**
  Displays What, Where, and how much the process has been performed in the format of piling the line charts in one figure.

- **When × Where**
  Displays When, Where, and how much the process has been performed through a gradation graph. The processed amount (such as the number of reductions) is shown by the depth of the color.

- **Result**
  Displays the list of the measurement results. Combines the measurement results by item (What), and displays the amount of the process (such as total receipt times and total number of reductions) sorted according to the amount. Note that the average value for each operation status is displayed for the operation status of a node and the order of display is fixed.

(2) Individual Specification of Display Item (What)

Sets all the measurement items (What) in the execution status information collected to be displayed/not to be displayed individually by using What manipulator. Only the item to be paid attention to is displayed in the various formats.

(3) Operation of Execution Status Information

Save, load, or switch the execution status information to be displayed for the execution status information collected by the history manipulator.

5.2.2 Collecting Information on Execution Status

5.2.2.1 Measuring Execution Operating Status

When the **profile_node** command is executed from the top level of the listener, the calculation time, transmission time, receipt time, idle time, and GC time of each node are measured at regular time intervals. The ratio (%) for each operation state at certain regular time intervals is displayed by ParaGraph. The measured execution status information is kept in the pool of the listener which is prepared for storing data. (See Section 4.1.6.9 for further details.)

【Example】 Use of profile_node command

```
[7] profile_node(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
```

5.2.2.2 Measuring Occurrence Status of Communication Between Nodes

When the **profile_msg** command is executed from the top level of the listener, the times of the receipt and transmission of the message for the communication between nodes are measured at regular time intervals. The measured execution status information is kept in the pool of the listener which is prepared for storing data. (See Section 4.1.6.9 for further details.)

【Example】 Use of profile_msg command

```
[7] profile_msg(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
```

5.2.2.3 Measuring Execution Status of Programs

When the **profile** command is executed from the top level of the listener, the number of reductions and suspensions of the goal executed in each node are measured at regular time intervals. The measured execution status information is kept in the pool of the listener which is prepared for storing data. (See Section 4.1.6.9 for further details.)

【Example】 Use of profile command

```
[7] profile(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
```

5.2.2.4 Changing Time Interval to Collect Information

Sets the time **interval** to collect the execution status information by the interval command. Specifies the argument of the interval command for each integer by indicating the millisecond units. The default time interval is set to 2,000 milliseconds. (See Section 4.1.6.9 for further details.)

【Example】 Changing the timer interval

```
[6] interval(3000).    % Changes the time interval to 3,000 milliseconds.
```

5.2.2.5 Pools for Storing Data

The measured execution status information is kept in the pool of the listener which is prepared for storing data. Since all the execution status information collected is saved in a pool for data storage, it is possible to display each execution status information switched by using the history manipulator after gathering the various execution status information by the listener in advance. Also, the processes to save, load, and delete the execution status information registered in the pool for data storage are carried out through the history manipulator.

5.2.3 Operation Method

5.2.3.1 Starting ParaGraph

Entering the **paragraph** (or **pg**) command from the top level of the listener allows the ParaGraph to start. When the activation of the ParaGraph is completed, the View selector is displayed on the screen.

【Example】 Activation of ParaGraph

```
[7] profile(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
```

Figure 5.3: Example of View selector

```
 4228msec
Ans=[1,2,3,4,...]
[8] paragraph.        % View selector is displayed at the
                      % right bottom of the screen.
```

5.2.3.2 Operation Method of View selector

Selection of Display Format

A screen example of the View selector is shown in the Figure 5.3. The following items are on the View selector screen: What × When, Overall What × When, What x Where, Overall What x Where, When × Where, Result, and Overall What x Where. When one of the above items is selected by clicking with the mouse, the item selected is reverse displayed. Clicking the item co_it then displays the execution status information in the format corresponding to the selected item name. The display appears on the screen based on the type on the View selector r, s, & and the attribute values Num, To, and Window in each display format. (See Section 5.2.4.1 for further details.) The selection method and the display example of each display format are described below.

What × When

Clicking the item do_it by the mouse in the state where the item What × When is selected displays the execution status information on the screen in the format of What × When. Figure 5.4 shows a display example of What × When in the execution status of the program. In this figure, five goals are displayed in a line chart in the order of number of total reductions. (The axis of abscissas is the cycle number and the axis of ordinates is the number of reductions.)

Overall What × When

Clicking the item do_it by the mouse in the state where the item Overall What × When is selected displays the execution status information on the screen in the format of Overall What × When. Figure 5.5 shows the display example of Overall What × When in the execution status of the program. In this figure, five goals are displayed in the line chart in the order of the number of total reductions, and the other goals are displayed by the line at the lowest on the chart as others. (The axis of abscissas is the cycle number and the axis of ordinates is the number of reductions.)

What × Where

Clicking the item do_it by the mouse in the state where the item What × Where is selected displays the execution status information on the screen in the format of What x Where. Figure 5.6 shows a display example of What × Where in the execution status of the program. In this figure, five goals are displayed in the bar graph in order of the number of total

Figure 5.4: Display Example of What × When

Figure 5.5: Display Example of Overall What × When

Figure 5.6: Display Example of What × Where

reductions. The number of reductions is indicated in a black pattern (blue in color mode), the number of suspensions is shown in a dotted pattern (green in the color mode), and the part where the number of reductions and the number of suspensions are overlapped is displayed in a hatched pattern (light blue in the color mode). (The axis of abscissas is the cycle number and the axis of the ordinates is the number of reductions or suspensions.)

Overall What × Where

Clicking the item do_it by the mouse in the state where the item Overall What × Where is selected displays the execution status information on the screen in the format of What x Where. Figure 5.7 shows a display example of Overall What × Where in the execution status of the program. In this figure, five goals are displayed in the bar graph in the order of number of total reductions, and the other goals are displayed in the lowest part of the graph as others. (The axis of abscissas is the cycle number and the axis of the ordinates is the number of reductions.)

When × Where

Clicking the item do_it by the mouse in the state where the item When × Where is selected displays the execution status information on the screen in the format of When × Where. Figure 5.8 shows a display example of When × Where in the execution status of the program. In this figure, the number of reductions executed is displayed in the gradation of colors (patterns by the depth of the color) (The axis of abscissas is the cycle number and the axis of the ordinates is the number of reductions.)

Result

Clicking the item do_it by the mouse in the state where the item Result is selected displays the execution status information on the screen in the format of Result. Figure 5.9 shows a display example of Result in the execution status of the program. In this figure, the number

Figure 5.7: Display Example of Overall What × Where

Figure 5.8: Display Example of When × Where

Figure 5.9: Display Example of Result

of total reductions and suspensions in each goal are listed at the left side and the correlation between the total number of reductions and that of suspensions is indicated in bar graphs at the right. (The axis of abscissas is the number of reductions or suspensions and the axis of the ordinates is the ranking when the goals are placed in the order of number of the total reductions.)

Activation of What manipulator

Clicking the item What in the View selector displays the What manipulator on the screen.

【Caution】 When the execution status information does not exist (empty), the What manipulator can not be opened.

Activation of History manipulator

Clicking the item History on View selector screen by using the mouse displays the History manipulator on the screen.

【Hint】 The History manipulator can be opened even when no execution status information exists (empty).

Termination of Paragraph

Clicking the button (rectangle) at the right corner on the View selector screen by the mouse deletes all the windows and View selector currently displayed on the screen to terminate ParaGraph. Since the execution status information is kept in the pool for data storage which is equipped with the listener, it can be referred to again even the ParaGraph is once terminated.

5.2.3.3 Operation Method of What manipulator

The user interface to individually specify the item (What) displayed by ParaGraph is the What manipulator. The item(s) with the display status set is(are) regarded as the objective to perform the statistic processing and the graphic display. It is possible to display only the item(s) already set in the various display formats by setting to be displayed or not to be displayed for items noted in the What manipulator.

Selection of item (What)

A list of the item names in order of their processing amount is displayed on the menu of the What manipulator screen. Each item is selected by a mouse click, and the selected item is reverse displayed. Multiple items may be selected at one time. When the ParaGraph is started and the execution status information is switched, all the items are initially set to be in the display status. The name of the item is displayed in the following format according to its type.

Operation state:	compute (calculation) — receive (receipt) — send (transmission) — GC (Garbage Collection)
Message for Communication:	Message name for communication
User-defined Predicate:	Package name: :Module name: Predicate name/Number of arguments
Built-in Predicate:	Name of built-in predicate

【Example】 Display Example of item

```
* me::test:go/2
* unify_vector
  pimos::variable:wrap/4
* unify
  unify_list
    .. .. ..
```

【Caution】 When the type of the execution status information is the operation status of the node, the display order is fixed regardless of the processing amount.

Setting of Display Item

The item set to the display objectives is displayed with a * mark attached at the head of the item name. To set each item to be the display objectives, click the item name in the What manipulator by using the mouse. The selected item is reverse displayed on the screen. Then click the item set by the mouse to set the currently selected item to the display objectives. Other than this setting method, set* to set all the items to be the display objectives and the shell for the What manipulator to set by entering from the keyboard are available to set the display item. (See Section 5.2.4.2)

Reset of Display Items

Selects the item name in the What manipulator by using the mouse to reset the display status of the item. The selected item name is reverse displayed. Clicking the item reset at that time resets

the display status of the currently selected item(s). The *mark at the head of the item name is deleted when the display status is reset. Other than this setting method, set* to set all the items to be the display objectives and the shell for What manipulator to set by entering from the keyboard are available to set the display item. (See Section 5.2.4.2.)

Close of What manipulator exit

Clicking the item exit by using the mouse closes What manipulator. The item which is set at this moment is regarded as the display objective. However, the What manipulator will not be able to be closed until the time when the item is set in the case where no item has been set.

5.2.3.4 Operation Method of History manipulator

The user interface to select the execution status information to be displayed by ParaGraph. Since two or more data concerning the execution status information have been kept in a pool for data storage equipped with the listener, the execution status information selected by the History manipulator will be displayed in various display formats.

Selection of item (What)

On the menu of the History manipulator screen, the data identification number, the type of the execution status information, and the pair of the goals specified by the argument of the command for the profile of the listener are displayed as the items. {n} is shown when type of the execution status information is set to the operation status of the node, {m} is shown when the generation status of the communication between processors, and {p} is shown for cases concerning the execution status of the program. Clicking by using the mouse selects an item, and the selected item is reverse displayed. Only one item can be selected at one time. By using the initial setting when the ParaGraph is activated, the latest execution status information which has been collected is selected.

【Example】 Display of Item

```
5:{n} test:go([5,4,3],Ans)     % In case of operation status
4:{m} bmtp:go(5,Ans)           % In case of communication between nodes
3:{p} bmtp:go(5,Ans)           % In case of execution status of program
   .. .. ..
```

Loading and Saving of Execution Status Information

Clicking the item to be loaded by using the mouse displays the window for input the file name. Loading is started by pressing the return key after entering the filename. When the processing of load is terminated, the menu is updated to the latest status and the execution status information is registered in the pool for data storage equipped with the listener. Similarly, the execution status information is saved in the specified file by clicking the item save by using the mouse and entering the file name. The absolute path specification and the relative path name specification are available for specifying the file name. (Refer to Section 5.2.4.3.)

【Example】 Specifying absolute path name

```
FileName: >sys>user>me>bmtp32.log
```

【Example】 Specifying relative path name

```
FileName: bmtp32.log
```

Close of History manipulator exit

Clicking the item exit by using the mouse closes the History manipulator. The execution status information selected at this moment is to be the display objective. When the execution status information is not selected, the History manipulator will not be closed until the time when the execution status information will be selected. However, the closing process is to be taken as it is when the execution status information does not exist (empty).

5.2.3.5 Window Interface

Label of Window

On the label of the window, the window number, the input goal name of the execution status information, the name of the display format, the proportion of calculation, the proportion of communication, and the proportion of GC and so on are displayed according to the type of the execution status information and the display format.

Scroll of Graph

The button for scroll is displayed at either the right center or lower center on the frame of the window in order to easily refer to the graph which is not displayed inside the window in the display format to display the graph individually. Clicking the button by the mouse scrolls the screen for 1 graph, for 1 page, or to the top or the end.

【Caution】 Currently, the scale is readjusted in accordance with the maximum processing amount of the graph which is located at the top in the window when the screen is scrolled. Therefore, the scale is changed every time of scrolling. You should pay attention to the unit of the scale.

Scale of graph

When the graph can not be displayed on one screen (for example when the number of nodes is too large or the case where the measurement time is too long), the whole display is to be performed in order to display the entire view on one screen. At the whole display, the cycle number is displayed with the multiple cycles regarded as one scale, and one scale of the node number is regarded as 8 processors unit. Clicking the button for scale at the right bottom of the screen by the mouse switches the display mode from the whole display to the standard display. "$\cdots$" is displayed at the character string which displays the name of each axis if the display can not appear on one screen in the standard display mode.

【Caution】 The whole display and the standard display are switched each time of clicking the button for scale by using the mouse.

Adjustment of Number of Graphs

The button for adjusting the number of graphs is displayed at the right center in the window to adjust the number of graphs to be displayed within the window, in the format to display the graph individually. The number of goals to be displayed inside the window is increased by one by clicking the button + with using the mouse. To decrease by one goal, click the button - by the mouse. In the display formats What $\times$ When and What x Where, the number of graphs to

be displayed inside the window increases/decreases each time the mouse is clicked, and as for the display formats Overall What × When, Overall What x Where, the number of goals displayed in the same graph is to be increased/decreased.

Switch of Display Type

The bottom to switch the display according to the type of execution status display is displayed at the right bottom of the window. The following table shows the correspondence between the type of the execution status information and each button.

Type of Execution Status Information	r	s &	
Operation status of node	Ratio of Each Operation States (%)	NONE	NONE
Generation Status of Communication Between Modes	Times of Receipt	Times of Transmission	Correlation Display
Execution Status of Program	Number of Reductions	Number of Suspensions	Correlation Display

Reshape of Window

The window can be re-shaped by utilizing the function provided by the window manipulator of PIMOS. By reshaping, the graphics in the window are clipped. Displaying the graphics in accordance with the window size requires the indication from the View selector once again.

【Caution】 It is impossible to make the size smaller than 350 dots (height) x 350 dots (width).

【Caution】 When the window size is smaller than the one required for displaying the execution status information, nothing is to be displayed on the screen. In such cases, reshape the window again to make the size of it larger enough for the requirement.

5.2.4 Commands

ParaGraph displays the execution status information according to the content set by the above items on View selector, What manipulator, and History manipulator. Each item is described below.

5.2.4.1 Operation Commands for View selector

What × When

Displays What, When, and How much the process has been performed in the line chart individually. The axis of abscissas differs based on the cycle number and the axis of the ordinates changes according to the execution status information and the type.

- **r :** Displays the following items according to the type of the execution status information. Operation status of the node: ratio (%) of each operation status to the operation time of the node Generation status of the communication between nodes: times of receiving each message for communication Execution status of the program: number of reductions of each goal. The display is shown in order of the processed amount. The order of display is fixed for the display of the operation status of the mode.

- **s :** Displays the following items according to the type of the execution status information. Generation status of the communication between nodes: times of transmitting each message for communication Execution status of the program: number of suspensions of each goal The display is shown in order of the processed amount.
- **& :** Displays the following items according to the type of the execution status information. Operation status of the communication between nodes: correlation between times of receipt and times of transmission regarding each message for communication Execution status of the program: correlation between number of reductions and number of suspensions regarding each goal The times of receipt or the number of reductions is displayed by a continuous line (blue in color mode), and the times of transmission or the number of suspensions is displayed by a dotted line (green in color mode) in one chart. The display is shown in the order of the processed amount.

The number of graphs to be displayed is the value which is set by the attribute value Num. The window for the output destination is the one corresponding to the number which is displayed as the attribute value window.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select s and &.

Overall What × When

Displays What, When, and How much the process has been performed in the format of piling the line charts on one figure. The axis of abscissas differs based on the cycle number and the axis of the ordinates changes according to the execution status information and the type.

- **r :** Displays the following items according to the type of the execution status information in the line chart. Operation status of the node: ratio (%) of each operation status to the operation time of the node Generation status of the communication between nodes: times of receiving each message for communication Execution status of the program: number of reductions of each goal. The order of display is fixed for the display of the operation status of the mode.
- **s :** Displays the following items according to the type of the execution status information in the individual line chart. Generation status of the communication between nodes: times of transmitting each message for communication Execution status of the program: number of suspensions of each goal
- **& :** Displays the following items according to the type of the execution status information. Operation status of the communication between nodes: correlation between times of receipt and times of transmission regarding the message for communication Execution status of the program: correlation between number of reductions and number of suspensions regarding each goal The times of receipt or the number of reductions is displayed by the continuous line (blue in color mode), and the times of transmission or the number of suspensions is displayed by the dotted line (green in color mode) in one chart.

The number of items to be displayed is the value which is set by the attribute value To. When the total number of items is larger than the value set by To, all the other items are displayed by one line located at the lowest in the chart. The window of the output destination is the one corresponding to the number which is displayed as the attribute value window.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select s and &.

What × Where

Displays What, Where, and How much the process has been performed in the line chart individually. The axis of abscissas differs based on the node number and the axis of the ordinates changes according to the execution status information and the type.

- **r :** Displays the following items according to the type of the execution status information. Operation status of the node: ratio (%) of each operation status to the operation time of the node Generation status of the communication between nodes: times of receiving each message for communication Execution status of the program: number of reductions of each goal. The display is shown in order of the processed amount. The order of display is fixed as for the display of the operation status of the mode.
- **s :** Displays the following items according to the type of the execution status information. Generation status of the communication between nodes: times of transmitting each message for communication Execution status of the program: number of suspensions of each goal. The display is shown in order of the amount processed.
- **& :** Displays the following items according to the type of the execution status information. Operation status of the communication between nodes: correlation between times of receipt and times of transmission regarding each message for communication Execution status of the program: correlation between number of reductions and number of suspensions regarding each goal. The times of receipt or the number of reductions is displayed by the black pattern light_paint (blue in color mode), and the times of transmission or the number of suspensions is displayed by the dotted pattern dotted (green in color mode) in one chart. The display is shown in order of the amount processed. Moreover, the hatched pattern right_hatch (light blue in color mode) shows the part where both of them are overlapped.

The number of the graph to be displayed is the value which is set by the attribute value Num. The window of the output destination is the one corresponding to the number which is displayed as the attribute value window.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select s and &.

Overall What × Where

Displays What, Where, and How much the process has been performed in the format of piling the line charts on one figure. The axis of abscissas differs based on the node number and the axis of the ordinates changes according to the execution status information and the type.

- **r :** Displays the following items according to the type of the execution status information. Operation status of the node: ratio (%) of each operation status to the operation time of the node Generation status of the communication between nodes: times of receiving each message for communication. Execution status of the program: number of reductions of each goal. The display is shown in order of the processed amount. The order of display is fixed as for the display of the operation status of the mode.
- **s :** Displays the following items according to the type of the execution status information by the individual pattern. Generation status of the communication between nodes: times of transmitting each message for communication. Execution status of the program: number of suspensions or each goal. The display is shown in order of the processed amount.

- **& :** Displays the following items according to the type of the execution status information. Operation status of the communication between nodes: correlation between times of receipt and times of transmission regarding each message for communication. Execution status of the program: correlation between number of reductions and number of suspensions regarding each goal. The times of receipt or the number of reductions is displayed by the black pattern light_paint (blue in color mode), and the times of transmission or the number of suspensions is displayed by the dotted pattern dotted (green in color mode) in one chart. The display is shown in order of the amount processed. Moreover, the hatched pattern right_hatch (light blue in color mode) shows the part where both of them are overlapped.

The number of graphs to be displayed is the value which is set by the attribute value Num. The window of the output destination is the one corresponding to the number which is displayed as the attribute value Window. The number of items to be displayed is the value which is set by the attribute value To. When the total number of items is larger than the value set by To, all the other items are displayed the lowest line in the chart. The window of the output destination is the one corresponding to the number which is displayed as the attribute value window.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select s and &.

【Caution】 The processing amount of the pattern available for the ParaGraph is limited to 8. When more than 8 items are set to be displayed, the same patterns are to be repeatedly used.

When × Where

Displays When, Where, and How much the process has been performed in the gradation graph. The amount processed (such as the number of reductions) is shown by the depth of the color. The axis of abscissas shows the cycle number and the axis of the ordinates shows the node number. The depth of the color (pattern) of the gradation graph shows the processed amount of each node. The correspondence between each pattern and the processed amount is displayed at the right bottom of the window. The amount processed is displayed by divided into 10 levels according to the maximum value of the processed amount in all the cycles by type. Note that the maximum value in the case of the operation status of the node is fixed to 100%.

- **r :** Displays the following items according to the type of the execution status information. Operation status of the node: ratio (%) of each operation status to the operation time of the node. Generation status of the communication between nodes: times of receiving each message for communication. Execution status of the program: number of reductions of each goal.
- **s :** display the following items according to the type of the execution status information by the individual pattern. Generation status of the communication between nodes: times of transmitting the message for communication. Execution status of the program: number of suspensions of the goal.

The window of the output destination is the one corresponding to the number which is displayed as the attribute value Window.

【Caution】 When the display format When × Where is selected, it is impossible to set the attribute values Num and To.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select s and &.

Result

Displays the execution status information by the list and the bar graph in the display format of Result. Displays the following items according to the type of the execution status information. Operation status of the node: average value of each operation status as for each node. Generation status of the communication between nodes: total times of receiving and transmitting messages for communication Execution status of the program: total number of reductions and suspensions. The number of total reductions and suspensions in each goal are listed in the table at the left side and the proportion of each item to the entire is indicated in bar graphs at the right. The axis of the ordinates shows the rank of each item in the order of the amount processed. The average value of each operation status (%) and times of receipt or number of reductions are displayed in the same graph by the black pattern light_paint (blue in color mode), and the times of transmissions or number of suspensions is indicated by the dotted pattern (green in color mode). Moreover, the hatched pattern right_hatch (light blue in color mode) indicates the part where both of them are overlapped. The window of the output destination is the one corresponding the number which is displayed by the attribute value Window.

【Caution】 In the Result display format, the button for scroll is to be clicked by using the mouse only for the list part at the left of the screen. Scrolling is not available for the bar graph at the right part of it.

【Caution】 When the Result display format is selected, it is impossible to set the types r, s, & and the attribute values Num, To.

do_it

Displays the execution status information in the current window according to the display format specified on the View selector. When no window is created, a new window is created before the information is displayed.

【Caution】 If no execution status information exists (empty), nothing is to be displayed on the screen.

new_win

Creates a new window, and displays the execution status information in the display format specified on the View selector.

【Caution】 If no execution status information exists (empty), nothing is to be displayed on the screen.

What

Displays the What manipulator on the screen.

【Caution】 If no execution status information exists (empty), it is not possible to open What manipulator.

History

Displays the History manipulator on the screen.

【Hint】 It is possible to open History manager even when no execution status information exists (empty), to allow the execution status information file is loaded.

(Rectangular) Button for exit

Deletes all the windows and View selector currently displayed on the screen and exit ParaGraph. Since the execution status information is kept in a pool for data storage which is equipped with the listener, it can be referred to by activating the ParaGarph again even when the ParaGraph has been terminated once.

r

Displays the following items according to the type of the execution status information. Operation status of the node: ratio (%) of each operation status to the operation time of the node Generation status of the communication between nodes: times of receiving each message for communication Execution status of the program: number of reductions of the goal.

【Caution】 When the display format Result is selected, the item r can not be selected.

s

Displays the following items according to the type of the execution status information by the individual pattern. Generation status of the communication between nodes: times of transmitting the message for communication Execution status of the program: number of suspensions of the goal.

【Caution】 When the display format Result is selected, it is impossible to select the item s.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select the item s.

&

Displays the following items according to the type of the execution status information. Operation status of the communication between nodes: correlation between times of receipt and times of transmission regarding each message for communication Execution status of the program: correlation between number of reductions and number of suspensions regarding each goal

【Caution】 When the display format When × Where or Result is selected, it is impossible to select the item &.

【Caution】 When the execution status information is the operation status of the node, it is impossible to select the item s.

Num

In the display format of What × When and What x Where, specifies the number of items (What) to be displayed in the window. The value initially set to the number of items is 5.

【Caution】 Clicking by using the mouse can be omitted when another display format is selected.

【Hint】 Up to 15 graphs can be displayed in the window.

To

In the display format of Overall What × When and Overall What x Where, specifies the number of items to be individually displayed in the window. The value initially set to the number of items is 5.

【Caution】 Clicking by using the mouse can be omitted when another display format is selected.

【Hint】 Up to 15 graphs can be displayed in the window.

Window

Displays the window number (number which is assigned to the window) of the output destination. The window of the output destination is the current window. When another window is selected, the number is updated. The window number is displayed at the head of the title for each window.

5.2.4.2 Operation Commands for What manipulator

set

Sets the item selected on the What manipulator as the display objectives, and displays * marks at the head of the item name.

set*

Sets all the items as the display objectives and displays them with * marks at the head of the item names.

shell

Displays the shell for the What manipulator. Whether an item (What) is to be displayed/not to be displayed can be set by a command input from the keyboard. When the shell is opened, the message is displayed according to the execution status information. The commands available are listed below.

- **set**
  Sets all the items registered in the What manipulator to be displayed.
- **set item name**
  Sets the specified item, among the ones registered in What manipulator, to be displayed. A wild card (*) can be used as the item name. (Example: set pimos*.)
- **reset**
  Resets the display settings of all the items registered in the What manipulator.
- **reset item name**
  Resets the display settings of the specified item among the ones registered in What manipulator. A wild card (*) can be used as the item name. (Example: reset pimos*.)
- **show**
  Displays all the items registered in the What manipulator.

- **list**
  Displays the list of all the items set to be displayed.
- **head**
  Display the item based on the value set to the environmental variable. The item at the head is specified by the environmental variable $from, and the number of items is specified by the environmental var$
- **setenv environmental variables value** Sets the values (`integer>0`) to the environmental variables ($from, $num). (Example: setenv $num 10) The default values of the environmental variables are $from=1, $num=30.
- **getenv** Displays the values set to all the environmental variables.
- **help (?)**
  Displays the list of commands available.
- **exit**
  Closes the shell. Returns to the top level of What manipulator by this command.

reset

Resets the setting condition of the item selected on What manipulator, and deletes the * mark at the head of the item name.

reset*

Resets the setting conditions of all the items, and deletes all the *marks displayed at the head of item names.

exit

Closes the What manipulator. The item(s) set is (are) regarded as display objectives. When a single item is not set, the What manipulator cannot be closed.

【Caution】 Clicking by using the mouse in the View selector and the button on the window when the What manipulator was opened are reserved until the time when the What manipulator is closed.

5.2.4.3 Operation Commands for History manipulator

renew

Updates the item in the History manipulator to the latest content. Mainly used to execute the paragraph command in the background. Enables the latest execution status information to be obtained even the ParaGraph is executed in the background.

remove

When the item remove is selected, the execution status information currently selected hs deleted from the menu and also deleted from the pool for data storage which is provided by the listener. Once it is deleted, it can not be recovered.

load

Load the execution status information file. Clicking the item load by using the mouse displays the window for inputting the file name. The load is then started by pressing the return key. When the loading process is completed, renewal is performed internally to update the menu, and the execution status information is registered in the pool for data storage which is provided by the listener at the same time. If the return key is pressed without any input, it is regarded as an abort and the file name input window is closed. The absolute path name specification and the relative path name specification are available for the file name.

【Caution】 When a file name which does not exist or a file have other than the execution status information file is input, a process similar to abort occurs.

【Example】 Specification of absolute path name

```
FileName: >sys>user>me>nqueen.log
```

【Example】 Specification of relative path name

```
FileName: nqueen.log
```

save

Saves the selected execution status information in the file. When the item save is clicked by using the mouse, the window for file name input is displayed on the screen. The execution status information is saved in the specified file by pressing the return key after the file name input. If the return key is pressed without any input, it is regarded as an abort and the file name input window is closed. The absolute path name specification and the relative path name specification are available for the file name.

【Example】 Specification of absolute path name

```
FileName: >sys>user>me>bmtp32.log
```

【Example】 Specification of a relative path name

```
FileName: bmtp32.log
```

【Caution】 If the name of a directory which @oes not exist, process similar to abort occurs.

【Hint】 It is possible to load the execution status information file saved on the real machine (Multi PSI) by Pseudo.

plot

Clicking the item plot by using the mouse creates a subwindow. The corresponding relationship of the execution time (axis of abscissas) and the data identification number (axis of the ordinates) in bar graph at the left side of the screen, and the correspondence of the number of nodes (axis of abscissas) and the execution time (axis of the ordinates) is displayed in a dotted chart at the right side of the screen. The data identification number is plotted in the dotted chart. All the execution status information registered in the pool for data storage which is equipped with the listener is displayed by a plot command. Clicking the EXIT button on the subwindow by using the mouse returns it to the top level of the History manipulator.

exit

Clicking the item exit by using the mouse closes the History manipulator. The execution status information selected at this moment is regarded as the display objective. If the execution status information is not selected, the History manipulator can not be closed until the time it is selected. In the case where no execution status information exists (empty), the History manipulator can be closed as it is.

付録 A

Setting of Network Environment

This chapter outlines the setting method for the network environment required when the user who is using the PSI on the remote site to login to the PIMOS by utilizing the network.

The following network protocols are currently supported by the PIMOS.

- TCP/IP
- LIA network

A.1 TCP/IP

1. Changes the file `>sys>tcpip>net_config.dat` on the SIMPOS to the content shown below.

   Before change

```
 ......
@spawn
  @@process
      5,simpos##talk_input;
      6,simpos##talk_output;
      8,simpos##x_server;
      9,simpos##remote_printer;
   2000,spawn;
 ......
```

   After change

```
 ......
@spawn
  @@process
      5,simpos##talk_input;
      6,simpos##talk_output;
      8,simpos##x_server;
      9,simpos##remote_printer;
   2000,spawn;
   2010,pimos_remote_login_server;
 ......
```

   【Caution】 Setting `2010,pimos_remote_login_server;` allows this function even when the specification of the SIMPOS is changed.

2. Retry the activation of the `TCP/IP`. (Restart the PSI.)

   【Caution】 Other than the above- stated condition to realize this function, the network address of the FEP which is connected to the Multi-PSI to be used should be set. Specifically, confirm the content of `>sys>tcpip>host.dat`. For further details refer to Chapter 5 TCP/IP Network Operation Management in the "Down-sized PSI/SIMPOS TCP/IP Network System Manual".

A.2 LIA-net

1. Change the file `>sys>net>process>mylia.dat` on the SIMPOS to the content shown below.

   `@process`

   is written, in the following format.

   Node name of the FEP::`pimos_remote_login_server`,*Net_No*,*LIA_No*,*Terminal_No*,40;

   Add the setting of the network to the next line of the place where The part of *Net_No*,*LIA_No*,*Terminal_No*
   in the above format should be the same as the node name of the *FEP* for the file `>sys>net>process>` mylia.dat.

   【Caution】 If the file `>sys>net.process>mylia.dat` does not exist, it is necessary to newly create it with reference to the example.

2. Retry the activation of the `LIA-net`. (Usually restart the PSI.)

   【Caution】 Other than the above- stated condition to realize this function, the network address of the FEP which is connected to the Multi-PSI to be used should be set. For further details refer to Chapter 5 TCP/IP Network Operation Management in the "Down-sized PSI/SIMPOS PSI Network System Manual".

Example of Setting

An example of the setting which allows the user to remotely login the node number icpsi108 as the FEP from his PSI.

Assume that the following description is included in the file `>sys>net.process>nclia.dat`,

```
 ......
icpsi108::spawn0,89,7C,01,00;
icpsi108::spawn ,89,7C,01,01;
 ......
```

Net_No,*LIA_No*,*Terminal_No* become as follows individually.

Net_No: 89

LIA_No: 7C

Terminal_No: 01

Using the above settings, write the following content to `>sys>net>process>mylia.dat`.

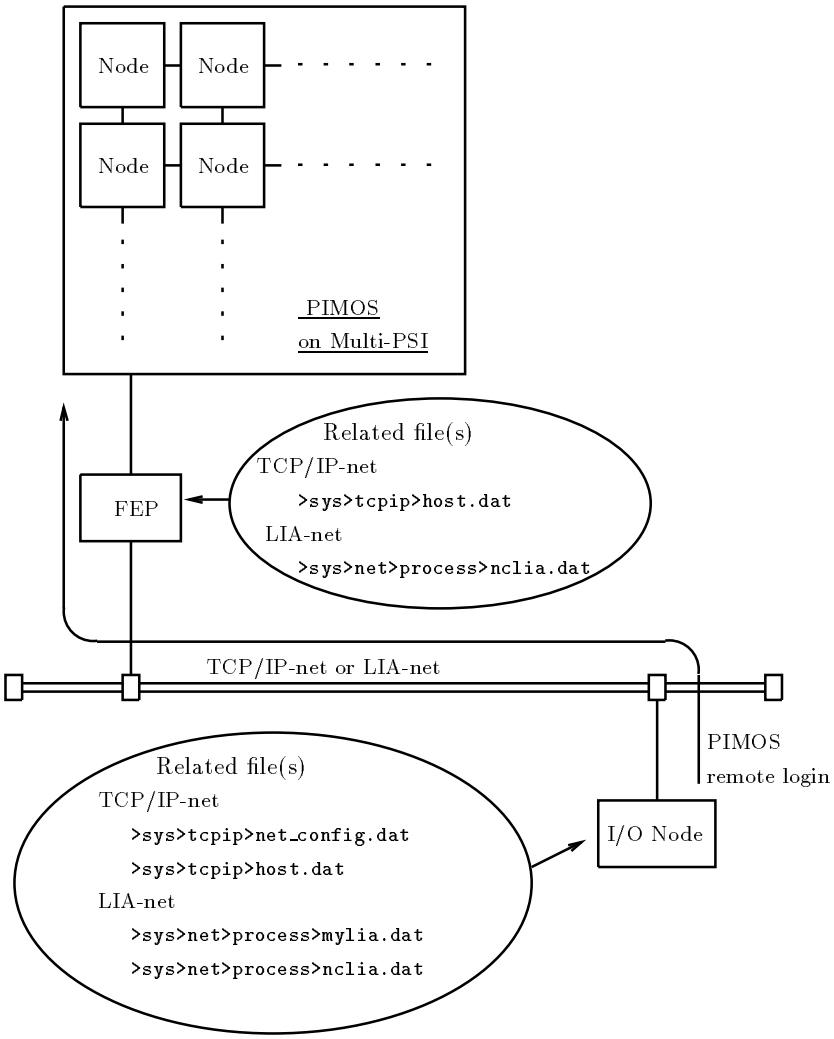

Figure A.1: Files related to the settings of a network

```
@process
    icpsi108::pimos_remote_login_server,89,7C,01,40;
     ......
```

付録 B

How to Know the Version of PIMOS

In inquiring of the PIMOS development group about a bug in PIMOS, information about the version of PIMOS now you are using is important. Information on how to get PIMOS version information is described below.

B.1 Version Displayed to CSP

The PIMOS consists of the multiple software modules, and the individual version information[1] exists for each software module. The version information is mainly classified as follows.

1. IPL
2. KL0/KL1 processing System
3. Cross compiler (hereafter described CAL)
4. Console System (CSP) or Supervisor Processor (SVP) Management Program[2] (hereafter described (generally named) CSP) 5. PIMOS Basic Unit
5. ParaGraph Display Program
6. FEP Management Program (hereafter described FEP)

Among the above information, the version information other than the FEP[3] is displayed on the CSP screen when the PIMOS is booted.

The versions of the IPL, CAL, KL0/KL1 processing system, CSP, and the ParaGraph display program are displayed by CSP command version.

【Example1】 Display example of Multi-PSI, PIM/m

```
$version
          PIMOS/ CAL/ CSP/  KL1 /  KL0 /ParaG
 Version : 3000/0193/0332/012100/019200/3000
$
```

[1] Normally expressed by number

[2] As for the CPU of the Multi-PSI, PIM/M, refer to "Multi-PSI/V2 Console System Operation Manual (Version 3)"

[3] The version information on the FEP of the next version is also scheduled to be displayed on the CSP screen at the boot of the PIMOS.

The version of the PIMOS basic unit is further subdivided, and the version of the set program at the time of IPL processing is displayed.[4] such information is displayed to the CSP between the messages "IPL loading" and "Link Module ..."

【Hint1】 The columns where the version information is to be filled in are prepared on the PIMOS inquiry form. Do not forget to note the points in filling in the PIMOS Query Form.

【Hint2】 When the PIMOS hangs up as a result of the occurrence of the bug, there is a high possibility that the important information is displayed on the CSP, as normal. It is recommended to save the screen information of the CSP.

【Example2】 Display Example of Multi-psi, PIM/m

```
PE00(00) @ DC> IPL loading
PE00(00) @ DC> BIOS    : 2990 : 31-Oct-91 :
PE00(00) @ DC> Debugger: 2990 : 31-Oct-91 :
PE00(00) @ DC> Library : 2990 : 31-Oct-91 :
PE00(00) @ DC> Utility : 2990 : 31-Oct-91 :
PE00(00) @ DC> Kernel  : 2990 : 31-Oct-91 :
PE00(00) @ DC> Server  : 2990 : 31-Oct-91 :
PE00(00) @ DC> Shell   : 2990 : 31-Oct-91 :
PE00(00) @ DC> Macro   : 2990 : 31-Oct-91 :
PE00(00) @ DC> patch1  : 2990 : 01-Dec-91 : BIOS patch
PE00(00) @ DC> Link Modules ...
```

B.2 Utility to Get Version B.3 Version of FEP Program

To know the version information on the PIMOS basic unit the method to use the utility program from such as the shell of PIMOS is available other than confirming the display on the CSP screen. The method to use this utility is shown below.

ipl_versions

The version information on the program which the IPL processing has performed is displayed on the screen. The display format for it is as follows.

"IPL file name"
{Blank} "version information"

When the version information is not added to the IPL file, the version information is not displayed but only the file name. In the case where the version information exists, it is to be displayed in the next line of the ipl file with the beginning letter lowered.

B.3 Version of FEP Program

You can know the version of the FEP program by using the debugger of the PIMOS to call the following method.

```
:pimfep_version(#pimfep##pimos_scsi_front_end,^Version,^Date,^Comment)
```

Version: Version of the FEP program (16 bits string)

[4] See Section 6.1 Setting IPL (Page 203) as for the information on how to set the IPL.

Date: Date when the package template of the FEP program was created (16 bits string).

Comment: Comment (16 bits string)

【Caution】 It is necessary for pimfep##pimos_scsi_front_end to have been loaded when this mode is called.

【Hint】 It is also possible to call it from the ESP program other than from the debugger.

Chapter 7

KL1 LANGUAGE SPECIFICATION

The concurrent logic kernel language KL1 is used as the interface between hardware and software on the computers on which PIMOS runs. PIMOS itself and application programs are written in KL1, either directly or through a higher level language.?

KL1 is based on Flat GHC (referred to below as FGHC), which is a subset of the concurrent logic language GHC, but also supports many extensions to FGHC. The main extensions are an extended syntax that eases coding, built-in data types and built-in predicate functions to improve efficiency, and the "Shoen" mechanism for meta-level control such as exception handling.

This chapter only describes the differences between KL1 and FGHC. Refer to other reference documents for a description of FGHC.

7.1 Notation of the I/O Mode

The functions of the PIMOS can be used either by calling predicates provided by PIMOS or by sending messages to PIMOS. There are two types of arguments used in these predicates and messages: input arguments, which are set by the user, and output arguments, which are set by PIMOS or the KL1 language functions. This document is written in a notation that allows this distinction to be seen clearly. This notation, which specifies the I/O relationship, is called "mode display".

Mode display follows the following principles:

- Mode display indicates whether an object term and the instantiation of the next outer level of that term belong to the same side.
- When both are of the same side, no special display is used, but when they differ, a caret ("^") is attached to the front of the term.
- When it is not known where the term is instantiated, e.g., the first argument to the built-in predicate "unbound" or the message "putt", a question mark is attached to front of the term.
- When the input stream to the built-in predicate "merge/2" (see page 297) has been unified, an asterisk ("*") is attached following the object variable.

【Example1】 normal(^Abort, Attention, ID)

This is the result of sending a reset message to a device. PIMOS instantiates the top level of this term. The user instantiates the variable "Abort", and PIMOS instantiates the variables "Attention" and "ID". Therefore, only "Abort" is instantiated at a different level, then only "Abort" is marked with a caret.

【Example2】 open(Pathname, read(^Result))

This is a file open message to the file manager. Since it is the user side (the side that sends the message) that instantiates the structure consisting of "Pathname" and "Read(_)" in this message, no carets are inserted. However, since PIMOS instantiates "Result", and caret is attached.

【Example3】 add(Integer1, Integer2, ^NewInteger)

This is an example of a built-in predicate. In the case of built-in predicates, it is assumed that the top level instantiation will be performed by the caller. Since the input variables "Integer1" and "Integer2" are instantiated by the same side, no caret is used, however, since "NewInteger" will be instantiated in the built-in predicate, a caret is used.

【Example4】 unbound(?X, ^Result)

In this example of a built-in predicate, "unbound" determines whether "X" is an unbound variable. As in the previous example, it is assumed that the top level instantiation will be performed by the caller. However, neither side instantiates the variable "X", so a question mark is used.

【Example5】 pool:keyed_bag(S*)

This predicate generates a keyed bag process. A stream to the process is unified with the variable "S". This stream is supplied with a merger. Therefore, an asterisk is attached. There is usually no need for the user to attach a merger to the stream when a stream with this notation is used.

7.2 Syntax

KL1 provides many extended functions to those in FGHC, and the syntax has been extended as well. This section describes the major extensions. The major syntax extensions include:

- Module structure
- Notation of asymmetrical clauses
- Execution priority specification
- Processor specification
- Macro notation

The macro notation is described separately in 7.3 "Macro Notation" on page (241).

7.2.1 Module structure

KL1 programs are coded in units referred to as **"modules"**.

7.2.1.1 Module definition

A module definition begins with

```
:-module <module name>.
```

which declares the name of the module. Predicates that are defined in the module and can be called from other modules must be declared in the form:

```
:-public <predicate name>/<arity>, <predicate name>/<arity>, ... .
```

The "public" declaration can be used multiple times between the "module" declaration and the first predicate definition. Multiple clauses belonging to the same predicate must be coded contiguously. If clauses are separated, even if they have the same predicate name and the same number of arguments, they will be handled as definitions of different predicates and generate multiple definition errors at compile time.

7.2.1.2 Inter-module call

Call to a predicate in the same module is the same as in FGHC. In the inter-module calling format, "goal" is preceded by the module name followed by a colon: "module name:goal".

【Example】 Inter-module call

```
pool:bag(S)
```

If a user-defined predicate which has a same name with a built-in predicate is defined and the predicate is called from the same module, then the user-defined predicate will be called.

【Example】 A predicate with the same name as a built-in predicate

```
:- module test.
:- public go/2.

add(X, Y, Z) :- true | ...
go(X, Y) :- true | add(X, 1, Y).
% The predicate add/3 defined above will be called.
```

【Hint】 To call the shadowed built-in predicate, simply attach ``builtin:'' to the call.

【Caution】 The module "builtin" doesn't actually exist. Therefore it is not possible to, for example, pass objects to the arguments of predicate_to_code or to request module information from PIMOS.

【Example】 An explicit specification of a built-in predicate

```
:- module test.
:- public go/2.

add(X, Y, Z) :- true | ...
go(X, Y) :- true | builtin:add(X, 1, Y).
% This calls the built-in predicate add/3.
```

7.2.2 Treatment of asymmetric clauses

In GHC, the coding order of clauses is not meaningful. This is also basically true for KL1. However, there are incidents in which multiple clauses comprising a single predicate need to be unequally treated.

In one example, assume one wants to set execution priority among the clauses. If multiple clauses are selectable (each of which has data required for reduction and execution of the guard part is successful), and if there is priority among the clauses, the clause with higher priority is selected. This is implemented by the "alternatively" function in KL1 which specifies the priority among clauses.

Another example is the simplified coding of negation. The condition for the selection of clauses may be completely written in the "guard" part. But sometimes "these cases and other cases" style is preferable. "otherwise" is a simplified notation of the negative condition provided by KL1.

7.2.2.1 Specification of priority among clauses (alternatively)

In the definition of a predicate, "alternatively" is coded between clauses so that the clause (group) preceding "alternatively" declares that this clause has precedence over the succeeding clauses (groups). That is, the clauses (groups) following "alternatively" may be selected only on the condition that it is known that all predicates before "alternatively" cannot be selected due to failure in the head unification or the guard part, or selection is impossible because the data required for reduction is not available yet.

Whether data is instantiated or not is checked only on the executing node [1] In other words, if the data required for reduction is contained in the other nodes and there is a clause which can be executed without suspension by accessing the data on the other node before "alternatively", a clause (group) which is executable with only data in the same node after "alternatively" may be executed. Even in such case, access to the other node for the data which is the reason of the suspension starts. Consequently, waiting for the same data makes it possible to execute the clause which uses that data sooner or later.

In general, the priority of predicates specified with "alternatively" is not always observed, so that a correct program including "alternatively" must be a correct program even in the absence of "alternatively". In other words, "alternatively" is not concerned with the meaning of the program but is a guideline for improved efficiency.

【Example】 Priority merger

[1] The KL1 load distribution specification unit. See section ??, "Node specification clause", on page (??).

```
        pm([], Y, Z) :- Z = Y.
        pm([W|X], Y, WZ) :- WZ = [W|Z], pm(X, Y, Z).
    alternatively.
        pm(X, [], Z) :- Z = X.
        pm(X, [W|Y], WZ) :- WZ = [W|Z], pm(X, Y, Z).
```

7.2.2.2 Simplified notation of negation (otherwise)

If it is determined that no clause before "otherwise" can be selected (the guard parts fail) by coding "otherwise" between the clauses in the definition of predicate, one can specify the clauses (groups) following "otherwise" to be the candidate for selection. Different from "alternatively", the clauses following "otherwise" are not assumed to be the candidate for selection as long as suspended clauses are present before "otherwise".

This results in the same effect as inserting negation of all guard conditions of the clauses before "otherwise" into the guard parts of all clauses after "otherwise".

```
        foo([X|XX]) :- X=a | pa(X,XX).
        foo([X|XX]) :- X=b | pb(X,XX).
        ...
    otherwise.
        foo(X) :- true | q(X).
        ...
```

【Caution】 Execution of the guard part in KL1 proceeds from left to right. Even when this kind of ordering is applied, the clause that should succeed will always succeed. However, if there are multiple conditions on the guard, it is possible that due to this ordering the execution of the first condition makes the predicate suspended, thus the failure of the other conditions are not checked. For example if a program such as the one shown below is called by `foo(_,1)`, since execution will be suspended at the unification of the first argument with `a` in the first clause, the unification of the second argument will not be executed and execution will be suspended without detection of its failure, and execution will not proceed to the execution of the second clause following the otherwise.

```
        foo(a, 0) :- true | bar.
    otherwise.
        foo(X, Y) :- true | baz(X, Y).
```

7.2.3 OR within a guard

Conditions which hold when either branch is true ("or" conditions) can be specified inside a guard with the ';'/2 notation.

【Example】 The following two code sections have the same meaning.

```
    foo(X) :- atom(X) | bar.
    foo(X) :- integer(X) | bar.
```

```
    foo(X) :- (atom(X); integer(X)) | bar.
```

The ';'/2 notation can be used any number of times within a guard, and can be nested.

【Caution】 This notation cannot be used with PDSS and CAL.

【Caution】 The guard internal "or" function in the current compiler is implemented by actually expanding the "or" into code similar to that shown in the example above at the source code level. No optimization for minimizing code size, such as "merging" into a single body, is performed.

7.2.4 Other directives

The following directives are valid in the parser of the compiler.

- Operator specification directive (:-op/3)
- Directive for "including" a file (:-include/1)

【Caution】 These two directives cannot be used with PDSS and CAL.

7.2.4.1 Operator specification directive

User defined operators can be specified by using the following directive in a source program.

```
:- op(Prec, Type, OpName).
```

Prec : An integer expressing the precedence (1 to 1200)

Type : An atom that expresses the operator association type (xfx, xfy, yfx, yfy, fx, fy, xf, yf)

OpName : The operator atom

7.2.4.2 Directive for "including" a file

A separate file can be read in by specifying the following directive in a source program.

```
:- include(<file name>)
```

The ¡file name¿ parameter is the file name of the file to be included, specified as a character string.

【Hint1】 In the PIMOS self-compiler, the file to be read is determined by the "path" specified to the compiler. For details, refer to the description of the compiler environment variables in section 3.1.4 on page (50).

【Hint2】 Operator specifications specified in the included file are also valid in the including file.

【Example】 Consider a file "myop.h" that includes the following content.

```
:-op(500, xfx, my_operator)
```

When this file is included, the atom my_operator will be handled as follows.

```
foo(X) :- true |
    X = a my_operator b.  % Syntax error
.....
....
:- include(''myop.h'').     % Read-in of myop.h
.....
...
bar(X) :- true |
    X = a my_operator b.  % Correctly read.
```

【Caution】 This directive cannot be used with PDSS and CAL.

7.2.5 Data notation

As a rule, the notation of KL1 programs/data is based on operator precedence grammar similar to Edinburgh Prolog.

This section describes only those notations which can be handled by or are meaningful when used by ordinary users. Note that in the following "Japanese characters" refers to characters for which there is no corresponding character in the ASCII character set, e.g., kana and kanji.

Variable A sequence of alphanumerics, Japanese characters, and the underline character ("_") that starts with either an upper case letter or the underline character. Variables with the same name in a single clause express the same variable. However, variables consisting of only the underline character are seen as separate individual variables even if they appear in the same range.

Integer As a rule, it is coded as decimal numbers. It may also be coded with binary to 36-adic numbers. For details, see the section on macro notation. Integers may included a sign.

Floating point number Floating point numbers are expressed in the form:

[sign number+ decimal point number+ [`e` or `E` [exponent sign] number+]

where, [] means that what is contained in it is optional, and number + denotes the digits of one or more characters. This notation cannot contain spaces. Similar to integer, a sign is included in floating point numbers.

【Hint】 In Multi-PSI, inner representation of floating point numbers uses single precision (32 bits) in accordance with IEEE.

【Caution1】 Although ± 0.0 may be coded in the source text, ± Infinity denoting infinity or NAN (not a number) cannot be coded.

【Caution2】 Unification of floating point numbers is performed by comparing the bit patterns of the inner representation. Thus, if the two numbers look the same when displayed, the unification between these number may fail. In general, it is recommended to avoid unification between floating point numbers.

Atom Essentially the same as Edinburgh Prolog:

1. Sequences of alphanumerics, Japanese characters, and the underline character that starts with either a lowercase letter or a Japanese character.

   【Caution】 Special characters cannot be used.

2. Sequences of the following special characters: `~+-*/\^<>=:.?[@]#$&`
3. Character strings surrounded at the ends by a pair of single quote characters ("'"). When a character string contains a single quote character, it is needed to add one more single quote character to the existing one.
4. ! ; []

Table 7.1: KL1 Data Notation

Variable	`A, B, C, _, _abc`
Integer	`123, 16#``ACE'', 8#``37''`
Floating point number	`1.23, 1.0e10, 3.0E-30, -2.0`
Atom	`abc, 'ABC'`
List	`[1,2,3], [1|X]`
Vector	`{a,Y,b}, f(X), {}`
String	`` ``abc'', ``'', string#``abc'' ``
Module	`module#foo, module#bar`

Vector As a rule, it is written in braces (`{}`) with each element delimited by a comma. In addition, there is a functor format similar to Edinburgh Prolog (as alternative to the notation of vectors which have an atom as the 0th element) in which the functor symbol specifies the atom of the 0th element, and the argument specifies the 1st element and on. For example, `f(a)` and `{f,a}` are the same.

【Caution】 The object `{}`, which has zero elements, is also a vector.

List As in Edinburgh Prolog, it is enclosed with brackets (`[]`) with the elements delimited by commas. Note that it is also the same with Edinburgh Prolog that vertical line (`|`) is used to delimit the element if the terminator of the list is other than atom `[]`.

String The string containing the element of the default bit width is expressed as `` ``abc'' `` or `string#``abc''`. If a double quotation mark itself needs to be an element, repeating two double quotation marks in a row makes one double quotation mark specified. For Multi PSI, default is 16 bit width and the JIS code system is used. ASCII code string with 8 bit width is expressed as `ascii#``abc''`.

【Hint】 There are strings with 1-bit and 32-bit elements for which no notation as constant is available, though it can be dynamically created at execution time for use.

【Caution】 In KL1, unlike in Edinburgh Prolog, a string is an independent data type and does not serve as notation for a list which contains the character code as elements.

【Caution】 Unification between strings is performed successfully when both strings have the same element size and length, and all elements contained in the strings are the same. This is also true for empty string, thus 16-bit empty string and 8-bit empty string cannot be unified.

module The object code corresponding to a module which is a coding unit of a program may be treated as modular data. A module is expressed in the format: `module#module-name`.

【Caution】 Among the notations described above, a notation including `#` is not a basic syntax but is a macro (explained later in this manual). Therefore, `#` bears the meaning as noted here if coded in a program, but notations with `#` simply read is taken as a vector with `#` as the 0th element.

Table 7.1 shows an example of the notation of data in KL1.

7.2.6 Execution priority specification

KL1 provides the function to specify goal execution priority (hereinafter simply called priority) for each goal. If reduction for number of goals in a single node[2]

【Caution1】 In Multi-PSI, priority cannot be specified to built-in predicates.

【Caution2】 Priority is valid only within a node and is not guaranteed outside individual nodes. Automatic load distribution to a certain extent is necessary in the future. If it is feasible, it may reasonably be presumed that priority based goal execution is not generally guaranteed because the user may not determine which goals are present in the same node. To ensure execution priority, it should be determined based on the interdependence of the data required for execution of the guard part, and priority should be used as a guide measure for improved efficiency.

Priority includes logical priority and physical priority and each goal is assigned individual logical priority. There are predetermined number of levels of physical priority which depends on a implementation, and before execution, the scheduler converts the logical priority into physical priority. If the range of physical priority is limited, it may not necessarily be changed even if the user changes logical priority. The upper and lower bounds of the priority assigned to a shoen (described later) also are in logical priority.

To specify the priority of a goal, the user uses the value relative to the logical priority of the parent goal or the upper and lower bounds of the shoen to which the goal belongs. The former is called "self-relative specification within its shoen" and the latter is "Rate specification within the assigned shoen".

7.2.6.1 Rate specification within shoen

The rate within the range determined by the upper and lower bounds of the shoen to which the specified goal belongs is used to specify the priority of a goal in the format:

Goal @ priority(Rate)

where the logical priority of the goal is determined by:

$$\text{lower bound value} + (\text{upper bound value} - \text{lower bound value}) \times \frac{\text{Rate}}{4096} \qquad (0 \leq \text{Rate} \leq 4096)$$

【Hint】 This has the same meaning as the

Goal @ priority(*, Rate)

form in the previous version of PIMOS. Although this form, which is provided for compatibility, is also valid, the new form is recommended.

7.2.6.2 Self-relative specification within shoen

Value relative to the logical priority of the calling goal is used to specify the logical priority of the goal. This value cannot exceed the upper and lower bound values of the logical priority of the shoen. The coding format is as follows:

Goal @ relative_priority(Rate)

[2] The KL1 load distribution specification unit. See section ??, "Node specification clause", on page (??). is possible, reduction normally begins with the goal assigned the highest priority.

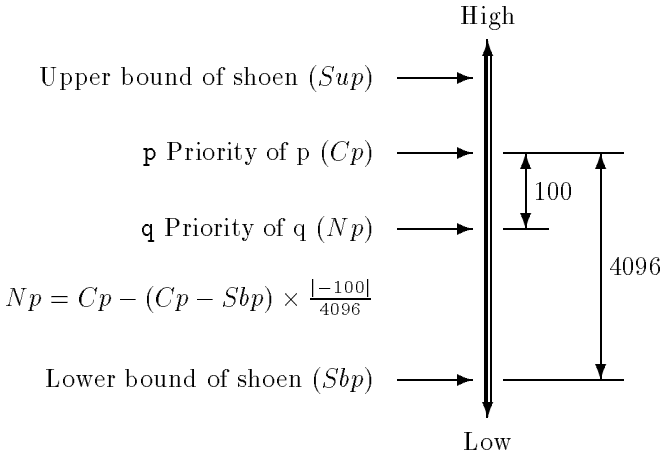

Figure 7.1: Execution Priority Calculation

Assuming the logical priority of the called goal is Cp, the logical priority of the goal "Goal" is determined by

$$Cp + (\text{upper bound value} - Cp) \times \frac{|\text{Rate}|}{4096} \qquad (0 \leq \text{Rate} \leq 4096)$$

or

$$Cp - (Cp - \text{lower bound value}) \times \frac{|\text{Rate}|}{4096} \qquad (-4096 \leq \text{Rate} < 0)$$

【Hint】 This has the same meaning as the

Goal @ priority($, Rate)

form in the previous version of PIMOS. Although this form, which is provided for compatibility, is also valid, the new form is recommended.

【Example】 Assuming the logical priority of "Goal" is Cp in the following program, logical priority Np of goal `q` is 7.1

```
Goal:      ?- p.
Clause:    p :- true | q @ relative_priority(-100).
```

7.2.7 Node specification

In KL1, the node to run each goal may be specified. A node is a set of 1 or more processors, such as a "cluster" (connected by shared memory). In Multi-PSI a single node is a single processor. A node is specified with a node number in the format:

Goal @ node (node-number)

In particular, a call of a goal which does not specify node is executed with the same node as that of the parent goal (which called that goal).

【Caution】 In Multi-PSI node specification cannot be given to built-in predicates.

【Caution】 In Multi-PSI it is currently not possible to directly call predicates defined in an external module with priority and node specification. Therefore, when such a notation is used in a

source program the compiler automatically generates an intermediate predicate similar to the following.

```
Notation in the source program:
   ..., module:goal@node(N), ...
Interpretation in the compiler:
   ..., '$$goal/0_1'@node(N), ...

   '$$goal/0_1' :- true | module:goal.
```

【Caution】 When the node and priority are specified at the same time to the same goal, the code will be expanded as follows:

```
Notation in the source program:
   ..., goal@priority(1024)@node(10), ...
Interpretation in the compiler:
   ..., '$$goal/0_0'@node(10), ...

   '$$goal/0_0' :- true | goal@priority(1024).
```

The order of expansion of each specification `priority(1024)` and `node(10)` in the example above) is determined only by the order of specification without regard for whether they are node or priority specifications, and code that "executes" in order from the rightmost specification will be output. Also, when the same type of specification appears multiple times in the same goal, the compiler simply expands those specifications into multiple predicates as described above.

```
Notation in the source program:
   ..., goal@node(0)@node(1)@...@node(10), ...
Interpretation in the compiler:
   ..., '$$goal/0_0'@node(10), ...

   '$$goal/0_0' :- true | '$$goal/0_1'@node(10).
   '$$goal/0_1' :- true | ...
   ...
   '$$goal/0_9' :- true | '$$goal/0_10'@node(1).
   '$$goal/0_10' :- true | goal@node(0).
```

7.3 Macro Notation

7.3.1 Overview

The KL1 language provides a macro expansion function to facilitate program coding. Macros in KL1 can be classified as follows:

System defined macros : The following macros are available for use with the PIMOS provided macro expander and compiler.

- Constant notations
- Unification
- Mathematical expression macros
- Conditional branches
- Implicit arguments
- Macros expanded by built-in predicates
- Conditional compilation

User defined macros : Macros defined by the user for his/her own requirements. Some of the system defined macros described above are defined using the same mechanism as that used by user defined macros.

Note that whereas there is a macro expansion function for built-in predicates, this built-in predicate function is described in ??, "Built-in Predicate", on page (275).

【Caution】 Note that of the system defined macros described above, the following use the macro module `pimos` definitions with the user defined macro function described in 7.3.4 (256) in the self compiler.

- Constant notations
- Unification
- Mathematical expression macros
- Macros expanded by built-in predicates

While the following are the main extensions and differences, except for these the specifications for macro definitions are the same as for versions 2.0 and earlier.

- These macros are expanded top down, regardless of their type.
- The suppression notation (`‘(<term>)`, and `‘‘(<term>)`) used in expression macros suppresses all of these macro expansions.
- The user can redefine part or all of these macros.

【Caution】 The macros expanded in built-in predicates in PDSS and CAL are defined using the user defined macro mechanism. All other system defined macros are defined with a framework separate from that of user defined macros.

7.3.2 Suppression of macro expansion

Expansion of user defined macros and system macros defined using the user macro definition mechanism can be suppressed using the following syntax.

- ``(<term>)
  Expansion of all macros contained in term is suppressed, and the meaning is "<term>" itself.
- `(<term>)
  If term is a structured object, expansion of the macros at the top level of ¡term¿ is suppressed. However, all macros nested one or more level within ¡term¿ will be expanded.

Since a different expansion mechanism is used for conditional branches, implicit arguments, and conditional compilation, such expansions cannot be suppressed using this syntax.

【Caution】 System defined macro expansions cannot be suppressed in PDSS and CAL.

7.3.3 System defined macros

7.3.3.1 Constant description

These macros are used to create constants.

`base#``character string''`
  Expands to the string converted to the integer value based on the base. Base in a range between 2 and 36 is allowable. The characters in the string include "0", " 1", ..., "9", "a"/"A", "b"/"B", ..., "z"/"Z" in that order according to the bases. There is no case sensitivity for the characters.

`string#``string''`
  The string is of the default type character code (JIS kanji code in Multi-PSI, or ASCII code in PDSS) corresponding to the string.

`ascii#``string''`
  8-bit string consisting of ASCII mode corresponding to the string.

`#``character''`
  Default type character code (JIS kanji code in Multi-PSI or ASCII code in PDSS) corresponding to the character.

`c#``character''`
  ASCII code denoting the character.

`ascii#character`
  ASCII code denoting the character. The character is an atom with one-character name (i.e., `\ascii#'[')`.

  【Caution】 It is possible that this will not be supported in future versions of PIM.

`key#lf`
  ASCII code (10) which denotes line feed.

`key#cr`
  ASCII code (13) which denotes carriage return.

`wrap#term`

This macro is converted to "wrap term" (9.2.1, on page 355) acquired by the message getwt/1 to the I/O utility buffer (see section ?? on page ??). However, variables in the term are not converted.

【Example】 Term conversion

```
foo         →  atom(foo)
1           →  integer(1)
[foo,1|A]   →  list([atom(foo), integer(1)|A])
foo(1, A)   →  vector({atom(foo), integer(A), A})
```

`macro#module_name`

Expands to the name of the module currently being compiled.

`macro#predicate`

This macro expands to a term that consists of the currently compiling predicate, the arity, and the order (a zero-origin integer) of the clause in the predicate separated by slashes ("/").

【Example】 A macro that acquires the module and predicate names.

```
Before expansion:
   :- module test.
   ...
   foo(1, X) :- true | X = macro#module_name.
   foo(2, X) :- true | X = macro#predicate.

After expansion:
   :- module test.
   ...
   foo(1, X) :- true | X = test.
   foo(2, X) :- true | X = foo/2/1.
```

`macro#switch(Param, ParamList, Default)`

This macros specifies a term to be expanded according to the value of the conditional compilation parameter Param. 7.3.3.6 on page (255 for details.

【Caution】 macro#module_name, macro#predicate, and macro#switch cannot be used with CAL and PDSS.

7.3.3.2 Unification

`Left side = right side`

Expands to the unification of the left and right sides. This can be used in both the guard part and the body part.

`Left side \= right side`

Expands to diff(left side, right side). This may be used only in the guard part.

【Caution】 In the current Multi-PSI diff/2 cannot take a structure as an argument.

Table 7.2: Arithmetic Comparison Macros

Priority	Operator	Pattern after expansion
700	`X < Y`	less_than(X,Y)
	`X > Y`	less_than(Y,X)
	`X =< Y`	not_less_than(Y,X)
	`X >= Y`	not_less_than(X,Y)
	`X =:= Y`	equal(X,Y)
	`X =\= Y`	not_equal(X,Y)
700	`X $< Y`	floating_point_less_than(X,Y)
	`X $> Y`	floating_point_less_than(Y,X)
	`X $=< Y`	floating_point_not_less_than(Y,X)
	`X $>= Y`	floating_point_not_less_than(X,Y)
	`X $=:= Y`	floating_point_equal(X,Y)
	`X $=\= Y`	floating_point_not_equal(X,Y)

7.3.3.3 Numerical expression macros

Unification

`Left side := right side`

Expands to the unification of the left and right side members. Because macro expansion of integer arithmetic applies to the right side, the result of the arithmetic operation is unified to the left side. It may be used in both the guard part and the body part. It corresponds to "is" in Prolog.

`Left side $:= right side`

Expands to the unification of the left and right sides. Because macro expansion of floating point arithmetic applies to the right side, the result of the arithmetic operation is unified to the left side. It may be used in both the guard part and the body part. It corresponds to "is" in Prolog.

Arithmetic comparison

The macro for arithmetic comparison uses operators such as `>`, `<`, `=:=`, and may be coded in place of the built-in predicate in the guard. Different from built-in predicates, macro expansion of arithmetic operation applies to both sides of the macro, so that results of the arithmetic operations are compared. Available comparison operators are listed in Table 7.2.

Arithmetic operation

Macros for arithmetic operation use operators `+`, `-`, `*`, or `/` and macro expansion applies to the following parts:

- The right side of `:=`, `$:=` macro (corresponds to "is" in Prolog). The result of arithmetic operation is unified to the left side. `:=` is for integer and `$:=` is for floating point arithmetic.
- Both sides of arithmetic comparison macro. Results of arithmetic operation are compared. The macro without `$` applies to integer arithmetic and the macro with `$` applies to floating point arithmetic.

Table 7.3: Arithmetic Operation Macro

Priority	Operator	Pattern after expansion Pattern	Corresponding built-in predicate	
			Within integer expression	Within floating point expression
500	`X + Y`	Z	add(X,Y,Z)	floating_point_add(X,Y,Z)
	`X - Y`	Z	subtract(X,Y,Z)	floating_point_subtract(X,Y,Z)
	`- X`	Z	subtract(0,X,Z)	
	`X \/ Y`	Z	or(X,Y,Z)	
	`X /\ Y`	Z	and(X,Y,Z)	
	`X xor Y`	Z	exclusive_or(X,Y,Z)	
400	`X * Y`	Z	multiply(X,Y,Z)	floating_point_multiply(X,Y,Z)
	`X / Y`	Z	divide(X,Y,Z)	floating_point_divide(X,Y,Z)
	`X << Y`	Z	shift_left(X,Y,Z)	
	`X >> Y`	Z	shift_right(X,Y,Z)	
300	`X mod Y`	Z	modulo(X,Y,Z)	
In term form	`\(X)`	Z	complement(X,Z)	
	`int(F)`	I	floating_point_to_integer(F,I)	
	`float(I)`	F		integer_to_floating_point(I,F)

【Caution】 `\(X)` may be converted to `exclusive_or(X,-1,Z)` in future versions of PIM.

- The right side of `<=`, `$<=` of the implicit argument macro (explained later in this manual). The result of the arithmetic operation is set to the argument specified on the left side. `:=` is for integer arithmetic and `$:=` is for floating point arithmetic operation.
- If expansion is explicitly indicated with `~(expression)`, `$~(expression)`, the result of arithmetic operation replaces `~(expression)` or `$~(expression)`. `~(expression)` is for integer, and `$~(expression)` is for floating point arithmetics.

【Example】 `p(~(X+Y+1))` is expanded to `add(X,Y,A), add(A,1,B), p(B)`.

If computation is possible (if all the input arguments are constants) when macro expansion takes place, computation is performed at the time of macro expansion.

Operators listed in Table 7.3 are provided. The smaller the value of priority, the stronger the connection, so that if another order of computation is required, enclose it with parentheses `( )`.

7.3.3.4 Conditional branch

This method is used to describe a conditional branch within a definition of a clause. Conditional branch may be specified in the form of:

```
guard -> body
```

and any number of branches delimited by a semicolon may be coded. The element which can be written in the **guard** is the same as that may be written in the guard part of a clause, and the element which can be written in the **body** is the same as that may be written in the body part of a clause.

Elements each of which is delimited by a semicolon may include "alternatively" or "otherwise" in addition to the above format. Their meaning is the same as that for a clause.

【Example】 Conditional branch macro

```
foo(X,Y) :- true |
    ( X=:=0 -> p(Y,Z);
      X > 0 -> q(Y,Z);
    otherwise;
      true  -> r(Y,Z) ),
    s(X,Z).
```

This program is equal to the following program:

```
foo(X,Y) :- true |
    '$foo/2/0'(X,Y,Z),
    s(X,Z).

'$foo/2/0'(X,Y,Z) :- X=:=0 | p(Y,Z).
'$foo/2/0'(X,Y,Z) :- X > 0 | q(Y,Z).
otherwise.
'$foo/2/0'(X,Y,Z) :- true  | r(Y,Z).
```

【Caution】 The preprocessor of the current compiler actually performs the above expansion. Predicate `'$foo/2/0'` is a predicate newly created by the preprocessor which expands macros. The name of the predicate to be created always begins with `$` (it is generally recommended that the user should avoid using atom which begins with `$`).

Although the ';'/2 section of this macro is spliced in as a predicate, the variables that are passed to this predicate as arguments are the "product set" of the variables in the "exterior" of the ';'/2 and the variables in the "interior".

【Example】 Consider the following example.

```
p(A, B, C, D) :- true |
   q(A, D),
   ( A =:= 0 -> r1(B, E), r2(E, C);
   otherwise;
     true -> r3(B, E), r(E, C)).
```

Here;

- The "exterior" variables are: A, B, C, and D
- The "interior" variables are: A, B, C, and E.

Therefore, the variables passed in are A, B, and C, and the example expands as shown below.

```
p(A, B, C, D) :- true | q(A, D), '$p/2/0'(A, B, C).

'$p/2/0'(A, B, C) :- A =:= 0 | r1(B, E), r2(E, C).
otherwise.
'$p/2/0'(A, B, C) :- true | r3(B, E), r4(E, C).
```

【Caution】 In CAL and PDSS conditional branch expressions cannot be nested, and multiple conditional branch expressions cannot be written in the same clause.

7.3.3.5 Implicit argument

Since KL1 has no function for global data such as "global variable", any data to be accessed must be passed as an argument. While this limitation has an advantage of clearly defining the potential range to which access from the program is made, coding tends to be complicated. Assuming that all predicates have an argument for the stream for error report, the description of the merge of error report stream each time the process is forked causes the program to be complex and, in turn, may result in poor perspective. To overcome such disadvantage, the macro function is provided in order to allow arguments which number of predicates have in common to be implicitly coded instead of being explicitly coded.

Implicit arguments include global declaration valid for an entire module and local declaration valid for a part of it. Each of them is coded in the form of:

```
:- implicit argument-name : type { , argument-name : type, ...}.

:- local_implicit argument-name : type { , argument-name : type, ...}
```

"argument-name" is the name used for referencing an implicit argument and is specified as an atom. "type" is used for specifying argument type by which the mode of expansion for the implicit argument in the predicate definition is determined. Currently available argument type includes "shared", "stream", "oldnew" and "string".

Shared In principle, the same data is passed to the forked goal. This is convenient for passing values that will be referenced in common by each goal.

Stream Branching streams that are merged into a single stream are passed to the forked goal. This is convenient for collecting the results from each goal in an arbitrary order from the goals that acquired a result.

Oldnew Convenient for passing the old and new versions of a data which is expanded to two arguments and has one single forked goal sequentially at each time of updating.

String Almost the same as "oldnew". The different point is that the elements of a structure are accessed as character strings, though "oldnew" accesses as vectors.

The declaration of a global implicit arguments may appear only once after :-module and before the :-local_implicit declaration or the first predicate definition. This declaration cannot be changed afterwards.

Local implicit arguments may be declared as many times as needed. In this case, the previous declaration is invalidated and the new declaration becomes valid. If no argument is specified as follows:

```
:- local_implicit.
```

no local implicit argument is declared in what follows. An argument of the same name cannot be declared in both "implicit" declaration and "local_implicit" declaration.

Whether or not the expansion of implicit argument should be applied may be determined for each clause. If a predicate has an implicit argument (requiring expansion), `-->` is used in place of `:-` for the clause definition. In the predicate of the clause defined by `-->`, the arguments declared implicit are added in the order of their declaration before the explicitly written arguments. The arguments are arranged in the order of:

1. global implicit arguments
2. local implicit arguments

3. explicit arguments

【Example】 Valid scope of an implicit argument declaration

```
:- module test.
:- public ...
:- implicit a:oldnew, b:shared.
   %% At this point, implicit arguments include only a and b.
:- local_implicit d:oldnew.
   %% At this point, implicit arguments include a, b and d.
:- local_implicit d:shared, e:stream.
   %% At this point, implicit arguments include a, b, d and e.
   %% Note that the type of argument name d is changed to "shared".
:- local_implicit.
   %% At this point, implicit arguments include only a and b.
```

To reference an implicit argument by name, place the prefix operator `&` before the argument name. Notation using the infix operators `<=`, `$<=` and `<<=` is intended for updating (or unification) of argument values. "`&argument_name(position)`" may be used for updating elements of a vector or string. Position means the element number of the targeted vector or string. Coding details vary according to the type of implicit argument. Meaning and use of individual types of argument are listed below.

Shared type argument

This type of argument shares the same value with other arguments of the same name in the predicates of the body part. To change the value of an argument referenced by argument name, use `<=` and `$<=` and specify as follows:

```
&argument_name <= new_value
&argument_name $<= new_value
```

As a result, subsequent values (to the right or below in the clause definition) to be referenced are updated. The right-side member of `<=` undergoes the macro expansion for integer arithmetic, and the right-side member of `$<=` undergoes the macro expansion for floating point arithmetic.

【Example】 Passing the counter

```
Declaration:    :- implicit counter:shared.

Before expansion:  p --> true | q, r.
After expansion:  p(Cnt) :- true | q(Cnt), r(Cnt).

Before expansion:  p --> true | q, &counter <= &counter + 1, r.
After expansion:  p(Cnt) :- true | q(Cnt), add(Cnt,1,Cnt1), q(Cnt1).
```

The following notation is available for cases where the value of the argument is a vector.

Element update

```
&argument_name(position) <= new_element
&argument_name(position) $<= new_element
```

Element reference (for cases other than the left side of the <= and $<= operators)

```
&argument_name(position)
```

The update form can only be used in the body part, and is expanded to the built-in predicate set_vector_element/5. Expansion of the reference form depends on the position the notation appears, with the reference form expanding into vector_element/3 in the guard, and set_vector_element/5 in the body.

```
Declaration:    :- implicit table:shared.

Before expansion:  p(Index, New) --> true |
            q(&table(3)), &table(Index) <= New, r.
After expansion:  p(Table, Index, New) :- true |
            set_vector_element(Table, 3, Elem, Elem, Table1),
            q(Table1, Elem),
            set_vector_element(Table1, Index, _, New, Table2),
            r(Table2).
```

Stream type argument

This type of arguments are merged and connected to the corresponding argument in the head part, if there are two or more implicit arguments of this type with the same name among the predicates in the body part. To put elements on the stream, connect the target elements in a list and code them on the right side of <<= in the form:

```
&argument_name <<= [element1, element2, ... ]
```

【Example】 Passing the stream

```
Declaration:    :- implicit window:stream.



Before expansion:  p --> true | q, r.
After expansion:  p(Win) :- true | Win = {In1, In2}, q(In1), r(In2).

Before expansion:  p --> true | &window <<= [putb("gazonk")], r.
After expansion:  p(Win) :- true | Win = [putb("gazonk")|Win1], r(Win1).
```

References (&argument_name) to stream type implicit arguments express a single stream that merges to the base stream.

【Example】 Explicit reference to a stream type implicit argument

```
Declaration:    :- implicit window:stream.

Before expansion:  p --> q, {{ r(&window), s(&window) }}.
After expansion:  p(W) :- true | W = {W1, W2, W3}, q(W1), r(W2), s(W3).
```

【Caution】 A "merger" is not automatically inserted into the implicit argument structure. The user must apply merger explicitly at the suitable position (usually only once at the beginning). (7.4.19 "Stream support", on page 297.)

Oldnew type argument

This type of argument expresses a pair of arguments and is similar to DCG (definite clause grammars) function in Prolog. With DCG, the two arguments indicate "difference list". In KL1, extended function allows data before and after updating to be expressed, and more than one pair of implicit arguments can be coded.

In KL1, the oldnew type argument is very useful in cases where vectors are used as a table which can be easily updated, and are usually programmed so that the number of references to vectors is always 1 for reasons of efficiency.

If used as a difference list, elements may be added to the following list as with stream type argument.

```
&argument_name <<= [element1, element2, ... ]
```

If the value of the argument is a "number", `<=` and `$<=` are used to code in the form:

```
&argument_name <= new_value
&argument_name $<= new_value
```

as similar to shared type. The right side of `<=` undergoes the macro expansion for integer arithmetic, and the right side of `$<=` undergoes the macro expansion for floating point arithmetic. If argument value is a vector, the following notation may be used.

Element update (1)

```
&argument_name(position) <= new_element
&argument_name(position) $<= new_element
```

Reference of element (for cases other than the left side of the `<=` and `$<=` operators)

```
&argument_name(position)
```

Element update (2)

```
&argument-name (position) <<= [element1, element2, ...]
```

(both 1 and 2) Both the update forms of 1 and 2 can only be used in the body part, and are expanded to the built-in predicate set_vector_element/5. Expansion of the reference form depends on the position the notation appears, with the reference form expanding into vector_element/3 in the guard, and set_vector_element/5 in the body. The arguments with the same name are connected from left to right and from top to bottom in that order as in DCG. Difference between the updates is that in update (1), an element which is indicated by "position" of the vector is replaced by the right side of `<=` or `$<=`, while update (2) assumes the position's column of the vector to be the end of "difference list", so that the elements of the list on the right side of `<<=` are added, and a new end is set to the element indicated by position of the vector.

The oldnew type argument also provides the following notation in order to reference the current "old" value (it is very useful, for example, in cases where the counter value is referenced at convenient times while counting up the counter).

```
&argument_name(old)
```

【Example】 The following examples show a variety of expansions for cases where the following oldnew type implicit argument declarations are used.

```
Declaration:    :- implicit mutter:oldnew.
```

Normal argument passing:

```
Before expansion:  p --> true | q, r.
After expansion:  p(Old,New) :- true | q(Old,Mid), r(Mid,New).
```

Update:

```
Before expansion:  p(X) --> true | q, &mutter <= X, r.
After expansion:  p(Old,New,X) :- true | q(Old,_), r(X,New).
```

Reference and update:

```
Before expansion:  p --> true | q, &mutter <= &mutter+1, r.
After expansion:  p(Old,New) :- true |
            q(Old,Mid1), add(Mid1,1,Mid2), r(Mid2,New).
```

Update as a difference list:

```
Before expansion:  p --> true | q(X), &mutter <<= [X], r.
After expansion:  p(Old,New) :- true | q(Old,Mid1,X), Mid1=[X|Mid2], r(Mid2,New).
```

Reference of element:

```
Before expansion:  p --> true | r(&mutter(3)).
After expansion:  p(Old,New) :- true |
            set_vector_element(Old,3,Elem,Elem,Mid), r(Mid,New,Elem).
```

Element update:

```
Before expansion:  p --> true | q(X), &mutter(3) <= X, r.
After expansion:  p(Old,New) :- true |
            q(Old,Mid1,X),
            set_vector_element(Mid1,3,_,X,Mid2), r(Mid2,New).
```

Update as an element difference list:

```
Before expansion:  p --> true | &mutter(2) <<= [d_n,gee,gosh], r.
After expansion:  p(Old,New) :- true |
            set_vector_element(Old,2,[d_n,gee,gosh|Cdr],Cdr,Mid),
            r(Mid,New).
```

Reference of the old value:

```
Before expansion:  p(X) --> true |
            X = [&mutter(old)|XX], &mutter <= &mutter+1, p(XX).
After expansion:  p(Old,New,X) :- true |
            X=[Old|XX], add(Old,1,Mid), p(Mid,New,XX).
```

String type argument

This type of argument is the same as oldnew type, but differs in that built-in predicates to be used are string_element/3 and set_string_element/4. That is, it is the oldnew type for strings instead of vectors.

Automatic termination

If the clause with implicit argument does not includes the call to the goals to be expanded in the body part, the following processings are performed according to the declared argument type.

Shared type	does nothing.
Stream type	unifies with atom `[]`.
Oldnew type	unifies Old and New.
String type	unifies Old and New.

Expansion control over implicit argument

To call a predicate without implicit argument from the body part of a clause declared to have implicit argument, the goal must be enclosed with a pair of double braces (`{{` and `}}`), so that automatic argument expansion is suppressed.

```
:- module test.
:- public go/0.
:- implicit input:stream, output:oldnew, counter:shared.
...
  loop(_) --> &counter =< 0 | true.
otherwise.
  loop(A) --> true |
    &counter <= &counter-1, &input <<= [getc(X)],
    {{ check(X, &output, &counter) }},
    loop(A).
...
```

In the example above, three implicit arguments, input, output, and counter, are globally declared each being the stream, oldnew, and shared types. With the above declaration, the clauses with the necks expressed as `-->` are assumed to have four implicit arguments (because one oldnew type is counted as two) and so converted when macro expansion takes place. For example, the above predicate definition "loop" is expanded as follows:

```
  loop(In, Oh,Ot, Cnt, A) :- Cnt =< 0 | In=[], Oh=Ot.
otherwise.
  loop(In, Oh,Ot, Cnt, A) :- true |
    Cnt1 := Cnt-1, In=[getc(X)|In1],
    check(X, Oh,Om, Cnt1),
    loop(In1, Om,Ot, Cnt1, A).
```

Note that predicate "check/4" enclosed with `{{` and `}}` is expanded so that it does not have implicit arguments. To pass the value of an implicit argument to such predicate, `&argument_name` as described above must be used. Care should be taken since the oldnew type output is expanded into two arguments.

【Caution】 Implicit arguments may have any value irrespective of argument type. Macro simply expands in accordance with its description, and, in most cases, does not do so much as calling programmer's attention to an obviously incorrect description (such that the list is unified with an argument whose elements are updated elsewhere).

7.3.3.6 Conditional compilation function

Small differences, mainly in low level functions that use the operating system, have appeared between the various KL1 processing systems (including systems under development). A conditional compilation function has been introduced to prevent proliferation of source code versions due to processing system differences, and to improve portability.

This function allows specified sections of the source code to be enabled or disabled according to values of conditional compilation parameters specified in the source code [3] or specified to the compiler top level. These perform similar functions to the #define, #ifdef, and other commands provided by the C language preprocessor "cpp".

[3] Including "include files". 7.2.4.2 on page (235)

【Caution】 The conditional compilation function cannot be used with PDSS and CAL.

【Hint1】 The standard macro preprocessor (kl1_macro_expander) implements this conditional compilation function.

【Hint2】 This conditional compilation function can be used in macro definition files as well as in KL1 programs.

Parameter definition

Parameters are the terms that are the object of the conditional compilation function's "conditionalization". Currently only atoms are recognized as these parameters. Parameters are defined, along with their values, by the method shown below. The values are arbitrary terms that do not included variables.

- If the following type of clause appears in the source code the parameter will be valid in the source program following that clause.

```
:- define(parameter, value).
```

  【Hint】 This definition can also appear in a file read in by :- include. (7.2.4.2, on page 235.)

- Parameters can be defined by entering the following command at the top level of the self compiler.

```
!define parameter value
```

  This results in an effect identical to inserting the above declaration at the beginning of source programs to be compiled.

Parameter "invalidation"

Parameters can be made invalid by the follow methods.

- If the following type of clause appears in the source code the parameter will be invalid in the source program following that clause.

```
:- undefine(parameter).
```

  【Hint】 The only action taken will be to issue a warning message if a parameter that has not been defined is declared invalid.

- The effect of !define can be invalidated by entering the following command at the top level of the self compiler. That is, this command stops the generation of the effect identical to including a :- define clause in the KL1 programs being compiled. (3.1.3.4, on page 46.)

```
!undefine parameter
```

Conditional compilation macros

Specification clauses that determine the sections that become the objects of compilation based on parameter states can be inserted in source programs.

These clauses can be roughly divided into two classes.

- :- if_XXX @ :- else @ :- end_if.
- :- switch @ :- case @ :- end_switch.

Boolean conditions

KL1 provides the following as a ":- if_XXX" family of specification clauses for conditional compilation based on boolean conditions. All of these obey the following rules. Note that the :- else clauses can be omitted.

- If the condition specified by the :- if_XXX is true, the source code up to the following :- else or :- end_if (when there is no :- else preceding the :- end_if) becomes valid.
- If the condition specified by the :- if_XXX is false, the source code from the :- else to the :- end_if becomes valid. If there is no :- else, the code up to the :- end_if is omitted.

Specifically, the following notations are possible.

:- if_equal(parameter, comparison value) : True if the value of the parameter and the comparison value can be unified, otherwise false.

:- if_not_equal(parameter, comparison value) : False if the value of the parameter and the comparison value can be unified, otherwise true.

:- if_defined(parameter) : True if the parameter is defined, false otherwise.

:- if_not_defined(parameter) : False if the parameter is defined, true otherwise.

【Example】 The examples below show the expansions when the following definition has been made.

```
:- define(machine, mpsi).
```

```
Before expansion: :- if_equal(machine, mpsi).
          p1(X) :- true | X=mpsi.
        :- else.
          p1(X) :- true | X=pim.
        :- end_if.

After expansion:   p1(X) :- true | X=mpsi.
```

```
Before expansion: :- if_not_equal(machine, pim).
          p2(X) :- true | X=pim.
        :- else.
          p2(X) :- true | X=mpsi.
        :- end_if.

After expansion:   p2(X) :- true | X=mpsi.
```

```
Before expansion: :- if_defined(machine).
          p3(X) :- true | X=defined.
        :- else.
          p3(X) :- true | X=undefined.
        :- end_if.

After expansion:   p3(X) :- true | X=defined.
```

```
Before expansion: :- if_not_defined(machine).
          p4(X) :- true | X=undefined.
        :- else.
          p4(X) :- true | X=defined.
        :- end_if.
After expansion:   p4(X) :- true | X=defined.
```

Selection by value

In the :- switch(parameter) notation, if the parameter can be unified with one of the comparison values value $_{1...N}$ in a following :- case(comparison value $_1$, comparison value $_2$ ... comparison value $_N$) clause, then the code from that clause to the next :- case clause or :- end_switch clause becomes valid. If there are multiple case statements that include comparison values that can be unified, then all become objects for expansion.

The clause :- default can be used instead of :- case(...) to express the default condition. This becomes the object of expansion only in the case where there is not even one case clause that became an expansion candidate.

【Example】 Conditional compilation using switch

```
:- switch(int).
:- case(0).
  foo(X) :- X = zero.
:- case(1,2,3,4,5,6,7,8,9).
  foo(X) :- X = little_integer.
:- default.
  foo(X) :- X = big_integer.
:- end_switch.
```

Parameter controlled macro expansion

The macro module "pimos" provides a macro that allows the results of expansion to be changed based on the value of a parameter, using the user defined macro function described in (7.3.4, "User defined macros", on page 256). This macro can be used in any place where a KL1 program arbitrary term can be used.

```
macro#switch(Param, ValueList, Default)
```

Param : The object parameter.

ValueList : A list of vectors that have two elements consisting of a value and the term following expansion when that value matches.

$$[\{\text{value}_1, \text{term}_1\}, \{\text{value}_2, \text{term}_2\}, \ldots, \{\text{value}_N, \text{term}_N\}]$$

Default : The term to be expanded when none of the values in ValueList match.

【Example】 Parameter controlled macro expansion

```
          :- define(machine, pim).
Before expansion:  foo(X) :- true |
            X=macro#switch(machine, [{mpsi,10}, {pim, 20}], 30).
After expansion:  foo(X) :- true | X=20.
```

【Caution】 This macro cannot be used with PDSS and CAL.

7.3.4 User defined macros

There are currently two major types of user defined macros provided by KL1.

- Macros that were used through PIMOS version 2.0. These macros can be used with the CAL and PDSS self compilers, and only support the replacement of simple terms. These are referred to as **old version macros**.

  Since the old version macros are provided to maintain compatibility, maintenance and functionality expansions will be mainly concerned with the new version macros. New macro definitions should be coded according to the new specifications.

- Macros that can be used with PIMOS version 2.5 and later. Although these macros can only be used with the self compiler, they support a wide range of expansion rules other than term replacement. These are referred to as **new version macros**.

This section describes the new version macros. 7.3.5 on page (273) for details on the old version macros.

The KL1 macro expansion facility supports the specification of the following types of manipulations on the pattern which matches to the expansion object input pattern that are specified in macro definitions that appear in source programs.

Replacement Replacement of an input pattern with a different pattern. In this operation, it is also possible to extract part of the original pattern and insert it in the expanded pattern.

Insertion Insertion of an arbitrary pattern in the area around the input pattern, e.g., if the pattern appears in a goal's arguments, before or after those arguments or before or after that goal. (This is called a splicing macro.) Part of the original pattern can be inserted in this case as well.

This operation is referred to as macro expansion. Since macro expansion is performed at compile time, there will be no difference in the execution time between using a macro notation and writing the result of the expansion directly.

【Caution】 PIMOS standardly uses the following format macros, which use a sharp sign. Therefore, we recommend that users not define macros with these formats.

`fep#???`	`shoen#???`	`builtin#???`
`pimos_tag#???`	`shell_tag#???`	`listener_tag#???`
`exception_code#???`	`exception_tag#???`	

Here ``???'' refers to an arbitrary term.

7.3.4.1 Procedures for using the macro expansion function

When using user defined macros, first the macros are defined and registered, then the program using those macros is compiled and executed or executed interpretively. The following procedure is used to define a macro.

1. The user writes a macro definition file using the macro notation language, and completes with the macro compiler.
2. The macro compiler processes the contents of the macro definition file and generates a KL1 program that performs the expansion of the defined macro.
3. The user then compiles the KL1 program and registers that program in the PIMOS module management process.

Since steps 2 and 3 of this process are performed automatically by the self compiler pre-processor function, normally users will not need to be aware of these steps.

When compiling using macro definitions registered in this manner simply include the following declaration in the program in which the macro expansion is to be performed, and then compile normally.

```
:- with_macro <appropriate macro module name>
```

Use "change_macro" (4.1.2.4 on page 64) to use macro expansion in interpretive execution with the listener.

Macro expansion is performed by calling the macro expansion program that is registered as a KL1 program using dynamic links. The macro expansion process that is invoked in this manner operates as a subprocess of the compiler, listener, or other application program. After the required expansion is performed, the result is compiled or executed interpretively.

7.3.4.2 Macro expansion order

In macro expansion, the system first determines if there is a rule defined in advance that can expand the input terms that will be the object of macro expansion. If there is, it expands the macro according to that rule. If there are multiple rules it applies one of those rules. Which rule is used is not guaranteed.

Term expansion is repeatedly applied to the results of expansion until no further expansion can be performed. When no further expansion can be performed, the system attempts to apply expansion recursively to the result if the result was a structure. That is, macro expansion is performed top down, from the shallower levels to the deeper levels of a structure (i.e. from the outside to the inside).

【Example】 The term `wrap#(16#''FFFF'')` expands to:

```
    wrap#(16#''FFFF'')
→   vector({atom((#)),integer(16), string("FFFF")})
```

Since the section "`16#''FFFF''`", which appears to express a hexadecimal value, has its form changed by the result of the external macro expansion due to "wrap#", it is not expanded as a hexadecimal value.

7.3.4.3 Macro expansion rules

Macro rules are collections of terms expressed in the following notation.

```
<macro rule>            ::=  <macro rule predicate>
                         |   <macro rule predicate> <macro rule>
<macro rule predicate>  ::=  <macro rule clause> '.'
                         |   <macro rule clause> '!' <macro rule predicate>
```

The syntax and function of each of these parts is described below.

Macro rule clause

A macro rule clause is written as either:

```
<pre-expansion pattern> => <post-expansion pattern> :- <guard> | <body>
```

Or:

```
otherwise
```

The gist of the first form is "If the input parameter matches the pre-expansion pattern and the guard condition is fulfilled, perform the expansion by in addition to executing the body, transforming to the post-expansion pattern." (A detailed description of each section is provided later in this section.)

【Hint】 The guard and the body can be abbreviated with the following syntax. Both are handled as though specified by "true" when abbreviated.

```
% Guard abbreviation:
<pre-expansion pattern> => <post-expansion pattern> :- <body>.
% Guard and body abbreviation:
<pre-expansion pattern> => <post-expansion pattern rule>.
```

The later form, "otherwise", means that if no previous macro rule clause could be applied, then the following macro rule clauses can be applied.

【Caution】 The "otherwise" clause cannot be used as either the first or the last clause. Also, "otherwise" clauses cannot be used in succession.

Pre-expansion pattern

The pre-expansion pattern roughly determines what kind of patterns are the object of expansion, and when the current term pattern matches with the input term the guard is tried next, and a rule in which the guard also succeeds is used in the expansion. That is, the pre-expansion pattern corresponds to the "head" that is used in normal KL1 programs. It must be one of the following:

- A wrapped term (See section 000on page 000.)
- The atom "begin" or "end"

The atoms "begin" and "end" are special patterns, and are selected at the start and completion of the expansion of a single source file respectively. (For details see page 265.)

【Hint】 It is possible to use wrap# macros either by declaring :- with_macro pimos at the start of a macro module, or by specifying, with the :- with_macro declaration, a macro module that inherits the pimos macro module. These methods reduce the efforts associated with "wrap". (See page 243.)

【Example】 The following two examples have the same meaning when using the pimos macro module.

```
vector({atom((#)), atom(macro), atom(mymacro)}) => ...
wrap#(macro#mymacro) => ...
```

Guard

The guard is a goal sequence that is executed from left to right when the input term and the pre-expansion pattern match. In addition to the usual KL1 guard built-in predicates, the following pseudo-predicates can be used to determine what kind of object the object of expansion is, or where it appeared. .

- Expansion object classification

  as_a_clause :: G

  The input term succeeded when it was a term at the clause level.

  as_a_goal :: G

  The input term succeeded when it was a term at the goal level.

  as_an_argument :: G

  The input term succeeded when it was a term at the goal argument level.

- Expansion object appearance location

  in_the_head :: G

  The input term succeeded when it was in the head.

  in_the_guard :: G

  The input term succeeded when it was in the guard.

  in_the_body :: G

  The input term succeeded when it was in the body.

  These pseudo-predicates concerned with the appearance location will only succeed when the input term was an object at a level deeper than the goal level. (They will not succeed when the input term is at the clause level.)

Expansion is performed by applying a single macro definition clause for which the pattern matched and all the goals written in the guard succeeded.

Post-expansion pattern

The post-expansion pattern has the following syntax:

```
<post-expansion pattern>  ::=  <replacement term> <splicing rule>
<replacement term>        ::=  <wrapped term>
                          |    <list of wrapped terms>
<splicing rule>           ::=  [ 'when' <splicing term>] [ 'where' <splicing term>]
                          |    [ 'where' <splicing term>] [ 'when' <splicing term>]
```

The input term is replaced with a "post-expansion term". "Splicing" is performed at this time. (See page 262.)

If the replacement term is simply a wrapped term, the input term is replaced in its original position. When the replacement term is a list of wrapped terms, the input term is replaced in its original position by an object which is a list of elements as terms at the same level listed up with commas.

【Example】 Replacing a single clause with multiple clauses:

```
Macro definition: wrap#(foo :- true | true) =>
                  [wrap#(bar1 :- true | true),
                   wrap#(bar2 :- true | true)]
                :- true | true.
Pre-expansion: foo :- true | true.
Post-expansion: bar1 :- true | true.
        bar2 :- true | true.
```

【Example】 The following two macros produce the same result when expanded as goal level terms.

```
wrap#goal => wrap#(goal1,goal2) :- true | true.
wrap#goal => [wrap#goal1, wrap#goal2] :- true | true.
```

【Example】 If [] is specified as the replacement term, the input term can be deleted.

```
wrap#(foo) => [] :- true | true.
```

【Caution】 In the current macro expansion function, when macros are expanded for argument level terms, an error is detected if the post-expansion term is a list.

Body

The body of a macro rule (among other things) generates the expansion result by calling normal KL1 predicates including built-in predicates.

Body predicate calling performs argument expansion as though the following sort of implicit argument declaration had been performed.

```
:- local_implicit info:shared, stream:oldnew.
```

The following types of information are passed to the respective arguments.

info A vector that holds information concerning the clause currently undergoing macro expansion. In the current implementation the following two types of information can be acquired.

Structure level The structure level of the term currently undergoing expansion is stored in the zeroth element of &info, expressed as an atom. The atoms that can appear are as follows:

- `clause` : clause level
- `goal` : goal level
- `argument` : argument level

Structure position The structure position of the term currently undergoing expansion is stored in the first element of `&info`, expressed as an atom. The atoms that can appear are as follows:

- `head` : head
- `guard` : guard
- `body` : body
- `clause` : None of the above. That is, the structure level is the clause.

【Caution】 Currently this is a two element vector, but additional elements may be added in the future.

stream A stream for communication with the process provided by the macro processing system. The following messages can be used.

error(FormatString, ArgList) error(FormatString, ArgList): The arguments are passed without modification to putf/2 and output to standard message output. If even one instance of this message is sent, the system determines that there was an error in macro expansion, and the result of the expansion is discarded after compilation, and is not registered in PIMOS as a module. The compiler counts the number of messages of this type sent and reports that number to standard output.

warning(FormatString, ArgList) The arguments are passed without modification to putf/2 and output to standard message output. No other actions are taken. The compiler counts the number of messages of this type sent and reports that number to standard output.

info(FormatString, ArgList) The arguments are passed without modification to putf/2 and output to standard message output. No other actions are taken.

new_variable(^Var) A unique wrapped variable that is not included in the clause that is the object of expansion is unified with Var.

new_atom(String, ^Atom) An atom which has the 16-bit character string String as its print name is unified with Atom. If there is no atom corresponding to String, a new atom is created.

Messages allowed with pool:keyed_bag A pool process defined by pool:keyed_bag is attached to a stream expressed by the implicit argument stream. Registration and reference of the arbitrary values required for macro expansion are supported with a message passing interface. (See section 8.3.5 "keyed bag" on page 323.) Although the internal state of this processes is system initialized for each program module that is the object of expansion, all other internal state changes are completely left to the user macro definition processes.

Since this stream's macro arguments are declared to be of type oldnew, the order in which those messages are transmitted will be according to the order in which the macros are expanded, i.e., the order in which macro expansion rules are applied.

The above implicit arguments can be suppressed or used partially with the same syntax that is used for normal implicit argument macros. (See page 252.)

Unless there is an explicit specification, user predicates called in the body call predicates in the module specified by `:- macro_module` in the package (usually the default package) specified to the compiler at that time. Explicit specification conforms to the normal KL1 calling syntax, i.e., ¡package name¿::¡module name¿:¡goal¿.

Macro rule predicates

Conceptually, a macro rule predicate is a set of macro rule clauses written for a single macro pattern.

【Example】 A macro rule predicate that includes an otherwise

```
Macro definition:
   vector({atom(integerOrNot), integer(_)})}) => atom(yes)
      :- true | true !
  otherwise !
   vector({atom(integerOrNot), _}) => atom(no).
Application example:
   Before expansion: integerOrNot(1) After expansion: yes
   Before expansion: integerOrNot([]) After expansion: no
```

【Hint】 The 'macro rule predicate' concept was introduced to bring in the 'otherwise' function. Macro rule predicates that do not include an 'otherwise' are compiled into the same code as a simple sequence of macro rule clauses.

【Example】 The following two macro notations are compiled into the same code.

- A definition using a single macro rule predicate composed of two rule clauses.

```
vector({atom(foo), X}) => atom(bigger)
    :- X=integer(Int), Int > 1000 | true !
vector({atom(foo), X}) => atom(less)
    :- X=integer(Int), Int =< 1000 | true.
```

- A definition using two macro rule predicates each composed of a single rule clause.

```
vector({atom(foo), X}) => atom(bigger)
  :- X=integer(Int), Int > 1000 | true.
vector({atom(foo), X}) => atom(less)
  :- X=integer(Int), Int =< 1000 | true.
```

Splicing macro rules

`...,a(~(X+1)),...` ⇒ `...,X1:=X+1,a(X1),...`

Splicing macro rules are a notation used to create the results of a macro expansion at a structural level different from the original position of the term that was the object of macro expansion, as shown in the example below.

Syntax for splicing macros The following operators are used in splicing macros.

when: Insert the immediately following term to the left of (before) the term that was the object of macro ex

where: Insert the immediately following term to the right of (after) the term that was the object of macro ex

The splicing term immediately following one of these splicing macro operators must be a wrapped term.

【Example】 Example of splicing.

```
Macro expansion rule: wrap#a => wrap#b when wrap#g1 where wrap#g2.
Before expansion: foo(a)
After expansion: g1, foo(b), g2
```

However, when arguments in a head are being expanded, the following rules are applied.

when: Insert the immediately following term from the right of the guard (at the tail).

where: Insert the immediately following term from the left of the guard (at the head).

Splicing level The splicing function can only be applied to terms at the goal argument level or at the goal level. Spliced terms always are goal level [4] terms. An attempt to expand a definition that has the splicing syntax for a term at the clause level generates an error.

【Hint】 The following two rules have the same behavior when applied to goal level terms.

```
wrap#(macro#a) => wrap#(macro#b, macro#c).
wrap#(macro#a) => wrap#(macro#b) where wrap#(macro#c).
```

Preprocessing of clause level expansions

During macro expansion, the macro expander performs macro expansion after first expanding clause level terms into the following standard form.

```
<header> :- <guard> | body.
```

For example, if the following is written at the clause level in a KL1 source program that is the object of expansion,

```
foo.


foo :- true | true.
```

then it will first be supplemented with a guard and body as follows. Therefore, macro expansion rules, such as rules which have the atom `foo` as their pre-expansion pattern, cannot be used for clause level expansion.

Variables in post-expansion patterns

Since the post-expansion pattern is passed to the compiler without modification, it must consist of ground values that do not include undefined variables, and all variables must be in the wrapped format (variable(character string) or variable(integer)). If a variable that is not wrapped is passed

[4] Guard goal if they are goal arguments within a guard, and body goal if they are goal arguments in a body. Also, in the head argument case, such terms are expanded into guard goals.

to the compiler, the compiler input filter will be permanently suspended. (This will be reported as deadlock at GC time.)

Since a term unified with the pre-expansion pattern is a term in a source program that is the object of expansion, all variables will have been wrapped. However, variables that only appear in the post-expansion pattern end up being variables that have not been instantiated at that point, and must be unified with appropriately wrapped variables.

The macro language processing system provides two methods for dealing with this problem: automatic generation of wrapped variables, and explicit generation during macro definition.

Automatic generation For variables that only exist in the post-expansion pattern, that is, do not appear in the pre-expansion pattern, body, or guard positions, this function outputs code which automatically replaces such variables by unique wrapped variables in the macro language processing system.

【Example】

```
wrap#a => wrap#(b(X), c(X)).
```

Since the variable X only appears in the post-expansion pattern, code, which unifies X and the wrapped term in the macro language processing system, is output.

【Hint】 Since this rule exists, the old version user defined macro syntax can only be used to make pre-expansion and post-expansion patterns into a wrap term.

【Example】 Example of conversion from the old specifications to the new specifications

- Old version syntax

```
builtin#set_substring(S1,P1,L,S2,NewS1)
  => builtin_utility:copy_string_elements(S1,P1,L,S2,0,NewS1,_).
```

- New version syntax

```
% The anonymous variable _ is automatically instantiated
% as a wrap term.
wrap#builtin#set_substring(S1,P1,L,S2,NewS1)
  => wrap#builtin_utility:
        copy_string_elements(S1,P1,L,S2,0,NewS1,_).
```

【Hint1】 It is possible to compile macro definition files written in the old version macro definition syntax with the current compiler.

【Hint2】 Macro definition files written in the old version macro definition syntax can be converted to macro modules written in the new version macro definition syntax by using a macro converter program. (See page ??.)

Explicit generation It is also possible to explicitly create and unify wrapped variables using user defined KL1 predicates called from the macro expansion rule. A unique wrapped variable can be created by simple sending a new_variable/1 message to the `&stream` implicit argument.

【Example】 The following macro definition is equivalent to the example given above.

```
% Since the variable ExpandedTerm appears in the body,
% the macro language processing system does not
% automatically create a variable
wrap#a => ExpandedTerm
  :- true | pred(ExpandedTerm).

% The new_variable/1 message is sent to &stream.
pred(ExpandedTerm) --> true |
  &stream <<= [new_variable(X)],
  ExpandedTerm = wrap#(b(X),c(X)).

% Implicit argument expansion gives the following:
pred(_, Stream0, Stream, ExpandedTerm) :- true |
  Stream0=[new_variable(X)|Stream],
  ExpandedTerm = wrap#(b(X),c(X)).
```

Special pre-expansion patterns

When the unwrapped atoms begin or end are used as the pre-expansion pattern, the macro processing system attempts that macro's expansion at the beginning or end of macro expansion for each file.

Each of these patterns must not appear more than once in a given macro module. Also, the result of the expansion is ignored, and never included in the post-expansion program. That is, only operations on &stream are meaningful.

【Example】 This example generates the pool process (referred to as 'keyed_sorted_set' in the following example) required for macro expansion when macro expansion starts, and registers it in the macro system supplied keyed_bag. When macro expansion ends, it sends a close to the pool stream. (A deadlock would occur if the stream were not closed.)

```
begin => [] :- true |
  pool:keyed_sorted_set(Stream),
  &stream <<= [put(my_pool, Stream)].
end => [] :- true |
  &stream <<= [get(my_pool, [])].
```

Note that when inheritance is performed, the begin/end special pattern expansions are tried in the following order.

begin : Expansion is tried in the order from "child" to "parent".

end : Expansion is tried in the order from to "parent" to "child".

KL1 predicate definitions in a macro definition file

KL1 predicates can be defined in a macro definition file. KL1 predicates defined in a macro definition file can be called from macro definition bodies written in the same file without any special package name or module name specification.

【Caution】 KL1 programs that are defined in a macro definition file do not have the macros defined in that file applied to them. Currently, it is not possible to have macros written in the same

file applied to these programs. [5]

KL1 predicate groups written in a macro definition file become part of the module which has as its name the atom specified in a macro module declaration (:- macro_module). That is, they go into the same module as the KL1 predicates generated as a result of macro compilation. This module is registered in default package. As long as the user does not declare them public, these predicates become local predicates. (The :- public declaration is sufficient to declare such predicates public.)

7.3.4.4 Macro module structure

A set of macro expansion rules that is handled as a single unit in the PIMOS macro preprocessor is called a macro module. Before a given macro module is used for actual macro expansions, it is converted to KL1 and handled as a single KL1 module.

A macro module source program consists of the following two sections, and must be written in this order even in the middle of a source program.

- Declaration section
- Macro rule section

In addition to macro expansion rules (see section 7.3.4.3 on page 258), normal KL1 programs can also be written in the macro rule section. These programs can be called from macro definition bodies and play an auxiliary role in macro expansion. (See page 260.)

The remainder of this section describes the macro module declaration section.

Macro module declaration

While macro modules are handled as normal KL1 modules, this declaration specifies that module's name (atom). This declaration must be made at the head of the macro definition file. The macro module declaration has the following syntax.

```
:- macro_module '<macro module name>'
```

Macros are defined by registering in PIMOS KL1 programs that have the atom specified by this declaration as their module name.

Application macro module declaration

Macro expansion can be performed even when the macro module source program is converted to a KL1 program. This declaration specifies the macro module name used at this macro expansion time.

```
:- with_macro '<application macro module name>', '<application macro module name>', ...
```

Macro modules are applied in the left to right order specified here at this macro application time.

When a macro module with no application macro module declaration is converted to a KL1 program, the macro module pimos is used as the default. Use the following statement if no macro expansion whatsoever is desired.

```
:- with_macro.
```

[5] In the future we would like to introduce this ability, but we think that even when introduced, it will be difficult to have them both coexist.

【Caution】 In the current specification this declaration can only be used once within a single macro module.

【Caution】 Macros written within the same macro modules cannot be used in the process of expanding that macro module itself into KL1.

【Example】 Macro expansion rule inside the same macro module are not applied.

```
:- macro_module test.
:- with_macro pimos.

% A macro that expands "macro#this" to "that".
wrap#(macro#this) => atom(that)
    :- true | true.
...
% "macro#this" is not expanded.
f(macro#this) :- true | true.
```

【Example】 If these are divided up into several macro modules (i.e., separate files) as follows, then the macro rules can be applied.

```
:- macro_module macro_test.
:- with_macro pimos.

% "macro#this" expands to "that".
vector({atom((#)), atom(macro), atom(this)}) => atom(that)
    :- true | true.
```

In a separate file:

```
:- module test.
:- with_macro macro_test.

% Expanded to "f(that) :- true | true.".
f(macro#this) :- true | true.
```

Inheritance macro module declaration

When an expansion object term does not match even a single object pattern in a corresponding macro module, it is possible to specify that expansion be tried using rules in another macro module. This is called macro module inheritance, and the inheritance macro module declaration declares the macro module that should be tried next.

```
:- inherit_macro '<inheritance macro module name>'
```

Only unitary inheritance is possible in macro inheritance, and it is not possible to include multiple inheritance macro module declarations for a single macro module declaration. Macro modules in which this declaration does not appear inherit pimos as the default. Use the following statement if no macro inheritance whatsoever is desired.

```
:- inherit_macro.
```

It is possible to include notations identical to those in normal KL1 modules, such as the public declaration in the declaration section.

7.3.4.5 Compilation into KL1 programs

The macro rules described above are compiled into KL1 predicates as follows.

The post-compilation head The head becomes the following sort of term (expand_macro/6) after compilation.

```
:- local_implicit info:shared, stream:oldnew.
expand_macro(MacroCode, InputPattern, ^OutputPattern)
```

Since there are implicit arguments, there are actually 6 arguments.

- `info`, `stream`: ?? on page (260).
- `MacroCode`: A pointer to the expand_macro/6 predicate in the macro module that is referenced directly in the `:-with_macro` definition in the program that is the object of expansion (This is the starting point when macro expansion is iterated.)
- `InputPattern`: A term written as the "pre-expansion pattern"
- `OutputPattern`: The output argument that is unified with the macro post-expansion pattern

Post-compilation guard The predicate group that is written in the guard in the macro rule becomes the post-expansion clause guard without change. However, the pseudo-predicates discussed in section 7.3.4.3 (page 259) are expanded as follows.

```
as_a_clause      vector_element(&info, 0, clause)
as_a_goal        vector_element(&info, 0, goal)
as_an_argument   vector_element(&info, 0, argument)
in_the_head      vector_element(&info, 1, head)
in_the_guard     vector_element(&info, 1, guard)
in_the_body      vector_element(&info, 1, body)
```

Post-compilation body The body written by the user is expanded as though the following implicit argument declaration was made.

```
:- local_implicit info:shared, stream:oldnew.
```

Also, spliced terms specified by `when` and `where` in the post-expansion pattern are transferred to `&stream` by an appropriate message.

Variables that only appear in the post-expansion pattern are instantiated during macro expansion by automatically adding the following goal:

```
&stream <<= [new_variable(<variable>),...]
```

Handling of post-expansion terms differs between the case where the specified term is a wrapped term and the case where the term is an unwrapped list (or empty list).

- A wrapped term was specified as the post-expansion term
  The code indicated by the MacroCode argument (a pointer to the expand_macro/6 predicate in the macro module directly referenced by the :- with_macro declaration in the program that is the object of expansion) is called recursively with the following arguments.

  ```
  {&info, &stream, MacroCode, <macro post expansion pattern>,
    ^OutputPattern}
  ```

  This call results in iterating the macro expansion to the extent that expansion is possible.

- A list or empty list was specified as the post-expansion term
  The post-expansion term is unified with the output variable (OutputPattern in the examples above) without modification. (Afterwards, the caller will perform iterative expansion on the list elements.)

Clauses for inheritance and expansion termination After the above compilation has been performed for all the macro rules, the following sorts of clauses that accompany 'otherwise' statements are output.

- When there are macros that use inheritance

```
otherwise.
  expand_macro(MacroCode, InputPattern, OutputPattern) -->
    true |
    '<inheritance macro module name>':
       expand_macro(MacroCode, InputPattern, OutputPattern).
```

  This corresponds to attempting expansion in the inheritance module when there is no appropriate expansion rule in the concerned macro module.

- When there is no inheritance macro

```
otherwise.
  expand_macro(_, InputPattern, OutputPattern) --> true |
    InputPattern = OutputPattern.
```

  This corresponds to the input term becoming the expansion result without modification in the case where there is no appropriate expansion rule in the concerned macro module and there is no inheritance module.

【Caution】 As described above, macro inheritance is implemented with a static link. Therefore, it is necessary to update the inter-macro module links by re-linking when a macro module in the upper levels of the inheritance hierarchy is updated. (See section 3.2 "Re-linking", on page 53).

【Example】 Consider two macro modules declared as follows.

1. Upper level module

```
:- macro_module god.
:- inherit_macro .
...
```

2. Lower level module

```
:- macro_module human.
:- inherit_macro god.
...
```

Even if only the `god` module is updated, the `human` module will continue to reference the pre-update version of the `god` module. As a result, the contents of the module referenced by inheritance from the `human` module will continue to be the pre-update contents.

To refresh this link, it is necessary to enter

```
relink(human, god)
```

from the shell or some other point and update the reference relations.

7.3.4.6 Handling of old version macros

The old version macro syntax (see section 7.3.5, page 273) from PIMOS version 2.0 and earlier, and which can be used with the cross compiler, can also be used with the new version macro compiler.

The macro compiler determines that the remainder of a file is in the old version macro definition syntax when it finds a declaration of the form "`:- user_macro <macro module>`". That is, compilation is performed as described in this section after the conversion described below is performed.

- ":- user_macro" is converted to ":- macro_module". That is, a module which has as its name the atom specified with :- user_macro is generated.
- The following declarations are added.

  :- with_macro pimos.

  :- inherit_macro pimos.
- The arguments to `'=>'/2` are wrapped.
- The above three steps are the whole "conversion" process.

It is also possible to convert a file to the new version macro definition syntax and save it as a file using the conversion program (page 274).

7.3.4.7 Macro tools

Currently the following tools can be used when using macros.

- Macro definition syntax conversion program
- Macro inheritance relationship checker

Macro definition syntax conversion program This is a program for converting macro definitions written in the old version syntax, which had been supplied earlier as a provisional "user defined macro" facility, to the new version macro syntax described in this chapter. See section 7.3.5 on page 273 for details.

Macro inheritance relation checker This program displays the inter-macro module inheritance relations on standard output. If there is a loop in the inheritance relations, it outputs an indication to that effect.

macro(EntryModule)

EntryModule is an atom that specifies the name of the lowest level macro module to be investigated.

【Example】

```
SHELL> macro(mymacro)
mymacro --> pimos
```

The output in the above example expresses the fact that the mymacro module inherits from the pimos module.

7.3.4.8 Macro definition examples

pimos.mac

This section presents some of the macros defined in pimos.mac.

```
:- macro_module pimos
% Macro expansion is not performed.
:- with_macro .
% Macro inheritance is not performed.
:- inherit_macro .

%
% Atomic data to atomic data conversions
%  Since macro expansion cannot be performed on this file itself, macros cannot be used in the pre-expansion patte
%  Since post-expansion patterns are performed dynamically and recursively when these macro definitions are refere
vector({atom((#)), atom(pimos_tag), atom(task)}) =>
        vector({atom((#)), integer(16), string("100")}).
% pimos_tag#pimos -> 16#"200"
vector({atom((#)), atom(pimos_tag), atom(pimos)}) =>
        vector({atom((#)), integer(16), string("200")}).
%
% For the case when variables appear in the expansion pattern.
%
% builtin#set_substring(S1,P1,S2,NewS1)
%   => string(S2,L,_,S22),
%     builtin_utility:copy_string_elements(S1,P1,L,S22,0,NewS1,_).
vector({atom(#}, atom(builtin),
        vector({atom(set_substring), S1, P1, S2, NewS1})}) =>
    vector({atom((',')),
            vector({atom(string), S2, L, _, S22}),
            vector({atom((:)), atom(builtin_utility),
                   vector({atom(copy_string_elements),
                           S1, P1, L, S22, integer(0), NewS1, _})})}) .
%
% Arithmetic operation macros
%
% When ~(<arithmetic operation>) appears at the argument level, it is replaced w%ith a new variable Z, and the exp
%The arguments ~X and ~Y are recursively converted.
%Objects other than arithmetic operations enclosed by ~(...) are converted to th%e object enclosed itself.
vector({atom((~)), vector({atom(+), X, Y})}) => Z
        when vector({atom(add), vector({atom((~)), X}),
                             vector({atom((~)), Y}), Z})
  :- as_an_argument | true !
vector({atom((~)), vector({atom(-), X, Y})}) => Z
       when vector({atom(subtract), vector({atom((~)), X}),
                                  vector({atom((~)), Y}), Z})
  :- as_an_argument | true !
vector({atom((~)), vector({atom(int), X})}) => Z
       when vector({atom(floating_point_to_integer),
% Converted to a floating point calculation
                    vector({atom(($~)), X}), Z})
  :-as_an_argument | true !
otherwise !
vector({atom((~)), X}) => X :- as_an_argument | true.
%
% The "X := <arithmetic operation>" at the goal top level is expanded.
```

```
%
vector({atom((:=)), X, Exp}) =>
    vector({atom((=)), X, vector({atom((~)), Exp})})
  :- as_a_goal | true.
```

Macros converted to standard input and standard output

Macro definition

```
:- macro_module stdio.
:- with_macro pimos.
:- inherit_macro pimos.

wrap#(stdio#stdin) => Stream
    when wrap#(shoen:raise(pimos_tag#shell, get_std_in, Stream))
    :- as_an_argument, in_the_body | true.
wrap#(stdio#stdout) => Stream
    when wrap#(shoen:raise(pimos_tag#shell, get_std_out, Stream))
    :- as_an_argument, in_the_body | true.
```

Usage example

Source program:

```
:- module stdio_test.
:- public go/0.
:- with_macro stdio.

go :- true |
    stdio#stdin = [getwt(Command)|StdIn],
    commandLoop(Command, StdIn, [], stdio#stdout, []).
...
```

Result of expansion:

```
:- module stdio_test.
:- public go/0.

go :- true |
    shoen:raise(16384, get_std_in, GenVar0),
    GenVar0=[getwt(Command)|StdIn],
    shoen:raise(16384, get_std_out, GenVar1),
    commandLoop(Command, StdIn, [], GenVar1, []).
...
```

Arithmetic calculation

Macro definitions

```
:- macro_module arithmetic.
:- with_macro pimos.
:- inherit_macro pimos.

wrap#($~(sin(X))) => Y
%% Since the wrap# is expanded first the $~ is not expanded
%% at macro compile time.
    when wrap#(arithmetic_library:sin($~(X), Y))
    :- in_the_body | true .

% Calculated at macro expansion time if it is a constant.
wrap#(sin(X)) => floating_point(Y)
    :- X = floating_point(Float), as_an_argument, in_the_body |
  {{arithmetic_library:sin(Float, Y)}} !
otherwise !
wrap#(sin(X)) => Y
    when wrap#(arithmetic_library:sin(($~(X)), Y))
    :- as_an_argument, in_the_body | true.
```

Usage example

Source program:

```
:- module arith.
:- public calc/2.
:- with_macro arithmetic.

calc(X, Y) :- true | Y $:= sin(X) * sin(X).
```

Results of expansion:

```
:- module arith.
:- public calc/2.

calc(X, Y) :- true |
  arithmetic_library:sin(X, GenVar0),
  arithmetic_library:sin(X, GenVar1),
  floating_point_multiply(GenVar0, GenVar1, GenVar2),
  Y = GenVar2.
```

7.3.5 Old version macros

【Caution】 The function described in this section is the function provided by PIMOS up through version 2.0, and is functionally included in the new version macros (versions 2.5 and later)described in section 7.3.4, page (256). New macro definitions should be coded according to the new specifications. A program is provided to convert old version macros to the new specifications. (See page (274.)

In old version macros the user may define the macro which replaces a pattern in the source program with another pattern. To define the macro, a macro definition file must be created in the following format:

```
:- user_macro <macro definition name>.
macro-name1#macro-name2 => format after expansion.
```

In the current version, the pattern on the left side must be used a sharp sign such as **xxx#yyy**, and macro-name1 must be an atom.

【Example】

```
:- user_macro io.
   io_status#normal      => 0.
   io_status#read_error  => 1.
   io_status#write_error => 2.
   io_command#read(X)    => io(read,X).
   io_command#write(X)   => io(write,X).
```

To use in the source program, user must declare the macro at the beginning of the module definition in the following format:

```
:- with_macro <macro definition name>.
```

where macro-definition-name is an atom.

If the pattern on the left side of `=>` is written in the source program, it is expanded to the pattern on the right side at compile time.

【Hint1】 To use the cross compiler system (CAL) for multi PSI, the macro definition file must be loaded (with LMac command) before compiling the KL1 source program which uses macro. See CAL Reference Manual for details.

【Hint2】 In the self compiler, macro definition names are interpreted to be and referenced as KL1 modules that specify the macro expansion method. See section 7.3.4 (page 256) for details.

【Hint3】 The new version macro function that will be provided by PIMOS in versions from PIMOS 3.0 and later, will be able to handle macros in the old specifications.

【Caution】 The macro file read-in function that uses environment variables (macro:auto_initialize, macro:initial_file) that are supplied by the self compiler in PIMOS version 2.1 is no longer necessary due to functional extensions provided by new version macros, and has been discarded. A program (convert_macro) that converts old version macros to the new specifications is provided. This program performs conversions for the following two aspects:

- `:- user_macro` is converted to `:- macro_module`.
- The pre-expansion pattern and the post-expansion pattern are wrapped.

convert_macro(OldFile, NewFile)

OldFile: Inputs the name of the macro file written in the old specifications as a 16-bit string.

NewFile: Inputs the name of the macro file that should be written in the new specifications as a 16-bit string.

【Caution】 Module names have been changed from the old version.

7.4 Built-in Predicate

While this section describes built-in predicates provided by Multi-PSI and PIM/m, predicates with no special indication are also provided by other PIM machines. Each built-in predicate is shown in the following format:

vector(X, ^Size) :: **G**

↑ Calling format ↑ Location

Location: Appearances of some built-in predicates are restricted due to their individual characteristics. If **G** is specified for the location, it means the predicate can appear in the guard part only. **B** means that the predicate can appear in the body part only. **GB** means that the predicate may appear in either the guard or body part.

Mode of argument: An argument with ^ mark indicates the argument to be unified, which may be called "output argument" later. Other arguments are input arguments. When input argument has not been instantiated at the execution time, if the built-in predicate appears in the guard part, the execution of the clause including the predicate is suspended and another clause is tried. If the built-in predicate appears in the body part, the execution of that built-in predicate is suspended (it never affects the execution of other goals in the same body part).

"Fail/exception" means that the execution fails if the predicate is in the guard part, and the execution is treated as an exception if the predicate is in the body part.

Note that KL1 provides macros that allow arithmetic operations to be specified conveniently without using the raw expression of built-in predicates. See section 7.3.3.3, "Numerical expression macros", on page (244) for details.

7.4.1 Type check

wait(X) :: G

Succeeds if the top level of X is instantiated.

Suspend: If the top level of X is uninstantiated.

Fail: Never fails.

atom(X) :: G

Succeeds if X is an atom.

Suspend: If X is uninstantiated.

Fail: If X is anything other than an atom.

integer(X) :: G

Succeeds if X is an integer.

Suspend: If X is uninstantiated

Fail: If X is anything other than an integer.

floating_point(X) :: G

Succeeds if X is a floating point number.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a floating point number.

list(X) :: G

Succeeds if X is a list.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a list.

vector(X, ^Size) :: G

Unifies the size of the vector X with Size.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a vector.

string(X, ^Size, ^ElementSize) :: G

Unifies the size of the string X with Size and unifies the element size with ElementSize.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a string.

module(X, ^Size) :: G

Unifies the size of the module X with Size.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a module.

code(X, ^Arity, ^Module, ^Position) :: G

Unifies the number of arguments of the code X with Arity, unifies the module to which the code belongs with Module and unifies the position from the beginning of the module with Position.

Suspend: If X is uninstantiated.

Fail: If X is anything other than a code.

unbound(X, ^Result) :: B

If X is an undefined variable at the time when unbound/2 is executed, a vector {Id, Address, NewX} is unified with Result. Where Id is an identifier attached to each variable, Address is the address of the variable, and NewX is the result after unifying X. If X has been instantiated, a vector composed of only one element {NewX} is unified with Result.

Suspend, Exception: None

【Caution】 In the concurrent language KL1, different from in the sequential language Prolog, a test whether a variable is uninstantiated or not is not reliable. Such a variable may have been instantiated (by another concurrently running process) immediately after testing that it is uninstantiated. Therefore, this built-in predicate must be used with special care.

7.4.2 diff

diff(X, Y) :: G

Succeeds if it is assured that X is not equal to Y.

【Caution】 X or Y must be an atomic term (i.e. data expressed in 1 word such as an integer, atom or an empty vector) other than an undefined variable.

Suspend: If either X or Y is uninstantiated.

Fail: If it is assured that X is equal to Y.

Normally, the following macro notation is used.

```
X \= Y  =>  diff(X, Y).
```

In PIM/m or Multi-PSI,

```
p(X, X) :- q(X).
p(X, Y) :- X\=Y | r(X, Y).
```

The above program should preferably be written as follows:

```
    p(X, X) :- q(X).
otherwise.
    p(X, Y) :- r(X, Y).
```

In systems other than PIM/m and Multi-PSI, there are different predicates for which, inversely, an optimal code is generated if otherwise is not used, and there are other different predicates that can handle arguments other than atomic terms.

Note that if either X or Y is an undefined variable or not an atomic term, the message "Illegal DIFF!!!" will be displayed in the CSP window. If s is entered from the CSP window, the predicate will succeed, if f is entered it will fail, and if b is entered, the system will break and return to the CSP top level.

7.4.3 Comparison of integers

less_than(Integer1, Integer2) :: G

Succeeds if Integer1 *is less than* Integer2.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail:
- If either Integer1 or Integer2 is not an integer.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X < Y  =>  less_than(X, Y).
X > Y  =>  less_than(Y, X).
```

not_less_than(Integer1, Integer2) :: G

Succeeds if Integer1 *is not less than* Integer2.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail:

- If either Integer1 or Integer2 is not an integer.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X >= Y  =>  not_less_than(X, Y).
X =< Y  =>  not_less_than(Y, X).
```

equal(Integer1, Integer2) :: G

Succeeds if Integer1 is equal to Integer2.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail:

- If either Integer1 or Integer2 is not an integer.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X =:= Y  =>  equal(X, Y).
```

not_equal(Integer1, Integer2) :: G

Succeeds if Integer1 is not equal to Integer2.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail:

- If either Integer1 or Integer2 is not an integer.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X =\= Y  =>  not_equal(X, Y).
```

7.4.4 Integer operations

add(Integer1, Integer2, ^NewInteger) :: GB

Adds Integer1 and Integer2 and unifies the result with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.
- If an overflow occurs.

Normally, the following macro notation is used.

```
Z := X + Y  =>  add(X, Y, Z).
```

subtract(Integer1, Integer2, ^NewInteger) :: GB

Subtracts Integer2 from Integer1 and unifies the result with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.
- If an overflow occurs.

Normally, the following macro notation is used.

```
Z := X - Y  =>  subtract(X, Y, Z).
Z := - X    =>  subtract(0, X, Z).
```

multiply(Integer1, Integer2, ^NewInteger) :: GB

Multiplies Integer1 by Integer2 and unifies the result with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.
- If an overflow occurs.

Normally, the following macro notation is used.

```
Z := X * Y  =>  multiply(X, Y, Z).
```

divide(Integer1, Integer2, ^NewInteger) :: GB

Divides Integer1 by Integer2 and unifies the quotient with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.
- If Integer2 is 0.
- If an overflow occurs. [6]

Normally, the following macro notation is used.

```
Z := X / Y  =>  divide(X, Y, Z).
```

modulo(Integer1, Integer2, ^NewInteger) :: GB

Divides Integer1 by Integer2 and unifies the remainder with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.
- If Integer2 is 0.
- If an overflow occurs.

Normally, the following macro notation is used.

```
Z := X mod Y  =>  modulo(X, Y, Z).
```

[6] An overflow occurs when the negative number with maximum absolute value is divided by -1.

shift_left(Integer, ShiftWidth, ^NewInteger) :: GB

Logically shifts Integer to the left by ShiftWidth bits and unifies the result with NewInteger.

Suspend: If either Integer or ShiftWidth is uninstantiated.

Fail, Exception:

- If Integer1 is not an integer.
- If ShiftWidth is not an integer from 0 to 31.

Normally, the following macro notation is used.

```
Z := X << Y  =>  shift_left(X, Y, Z).
```

shift_right(Integer, ShiftWidth, ^NewInteger) :: GB

Logically shifts Integer to the right by ShiftWidth bits and unifies the result with NewInteger.

Suspend: If either Integer or ShiftWidth is uninstantiated.

Fail, Exception:

- If Integer1 is not an integer.
- If ShiftWidth is not an integer from 0 to 31.

Normally, the following macro notation is used.

```
Z := X >> Y  =>  shift_right(X, Y, Z).
```

and(Integer1, Integer2, ^NewInteger) :: GB

Applies bitwise AND operation to Integer1 and Integer2, and unifies the result with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.

Normally, the following macro notation is used.

```
Z := X /\ Y  =>  and(X, Y, Z).
```

or(Integer1, Integer2, ^NewInteger) :: GB

Applies bitwise OR operation to Integer1 and Integer2, and unifies the result with Newinteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.

Normally, the following macro notation is used.

```
Z := X \/ Y  =>  or(X, Y, Z).
```

exclusive_or(Integer1, Integer2, ^NewInteger) :: GB

Applies bitwise XOR operation to Integer1 and Integer2, and unifies the result with NewInteger.

Suspend: If either Integer1 or Integer2 is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.

Normally, the following macro notation is used.

```
Z := X xor Y  =>  exclusive_or(X, Y, Z).
```

complement(Integer, ^NewInteger) :: GB

Inverts each bit of Integer and unifies the result with NewInteger.

Suspend: If Integer is uninstantiated.

Fail, Exception:

- If either Integer1 or Integer2 is not an integer.

Normally, the following macro notation is used.

```
Y := \(X)  =>  complement(X, Y).
```

【Caution】 If this predicate is currently used in a body part, it is equivalent to taking the exclusive_or with -1.

7.4.5 Comparison of floating point numbers

floating_point_less_than(Float1, Float2) :: G

Succeeds if Float1$is less than$Float2.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail:

- If either Float1 or Float2 is not a floating point number.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X $< Y  =>  floating_point_less_than(X, Y).
X $> Y  =>  floating_point_less_than(Y, X).
```

floating_point_not_less_than(Float1, Float2) :: G

Succeeds if Float1$is not less than$Float2.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail:

- If either Float1 or Float2 is not a floating point number.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X $>= Y  =>  floating_point_not_less_than(X, Y).
X $=< Y  =>  floating_point_not_less_than(Y, X).
```

floating_point_equal(Float1, Float2) :: G

Succeeds if Float1$is equal to$Float2.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail:

- If either Float1 or Float2 is not a floating point number.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X $=:= Y  =>  floating_point_equal(X, Y).
```

floating_point_not_equal(Float1, Float2) :: G

Succeeds if Float1 is not equal to Float2.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail:

- If either Float1 or Float2 is not a floating point number.
- If the success condition above is not fulfilled.

Normally, the following macro notation is used.

```
X $=\= Y  =>  floating_point_not_equal(X, Y).
```

7.4.6 Floating point arithmetic

floating_point_add(Float1, Float2, ^NewFloat) :: GB

Adds Float1 and Float2 and unifies the result with NewFloat. If an overflow occurs, infinity is unified with NewFloat.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail, Exception:

- If either Float1 or Float2 is not a floating point number.
- If $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$ is executed.

Normally, the following macro notation is used.

```
Z $:= X + Y  =>  floating_point_add(X, Y, Z).
```

floating_point_subtract(Float1, Float2, ^NewFloat) :: GB

Subtracts Float2 from Float1 and unifies the result with NewFloat. If an overflow occurs, infinity is unified with NewFloat.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail, Exception:

- If either Float1 or Float2 is not a floating point number.
- If $(+\infty) - (+\infty)$ or $(-\infty) - (-\infty)$ is executed.

Normally, the following macro notation is used.

```
Z $:= X - Y  =>  floating_point_subtract(X, Y, Z).
```

floating_point_multiply(Float1, Float2, ^NewFloat) :: GB

Multiplies Float1 by Float2 and unifies the result with NewFloat. If an overflow occurs, infinity is unified with NewFloat.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail, Exception:

- If either Float1 or Float2 is not a floating point number.
- If $(\pm 0.0) * (\pm\infty)$ or $(\pm\infty) * (\pm 0.0)$ is executed.

Normally, the following macro notation is used.

```
Z $:= X * Y  =>  floating_point_multiply(X, Y, Z).
```

floating_point_divide(Float1, Float2, ^NewFloat) :: GB

Divides Float1 by Float2 and unifies the quotient with NewFloat. If an overflow occurs, infinity is unified with NewFloat. If Float1 is not @0.0 and Float2 is @0.0, infinity is unified with NewFloat.

Suspend: If either Float1 or Float2 is uninstantiated.

Fail, Exception:

- If either Float1 or Float2 is not a floating point number.
- If $(\pm 0.0)/(\pm 0.0)$ or $(\pm\infty)/(\pm\infty)$ is executed.

Normally, the following macro notation is used.

```
Z $:= X / Y  =>  floating_point_divide(X, Y, Z).
```

7.4.7 Conversion of integer-floating point number

floating_point_to_integer(Float, ^Integer) :: GB

Converts Float to an integer (ignoring decimal) and unifies it with Integer.

Suspend: If Float is uninstantiated.

Fail, Exception:

- If Float is not a floating point number.
- If an overflow occurs.

Normally, the following macro notation is used.

```
Y := int(X)  =>  floating_point_to_integer(X, Y).
```

integer_to_floating_point(Integer, ^Float) :: GB

Converts Integer to a floating point number and unifies it with Float.

Suspend: If Integer is uninstantiated.

Fail, Exception: If Integer is not an integer.

Normally, the following macro notation is used.

```
Y $:= float(X)  =>  integer_to_floating_point(X, Y).
```

7.4.8 Double precision integer operations

add_extended(X, Y, ^Zh, ^Zl) :: B

X and Y are regarded as 32-bit unsigned integers, and the lower 32 bits of their sum is unified with Zl and the carry is unified with Zh.

Suspend: If either X or Y is uninstantiated.

Exception: If either X or Y is not an integer.

subtract_extended(X, Y, ^Borrow, ^Z) :: B

X and Y are regarded as 32-bit unsigned integers and Y is subtracted from X. If the result is non-negative, Z is unified with the absolute value and Borrow is unified with 0, or if the result is negative, Z is unified with the complement of 2^{32} and Borrow is unified with 1.

Suspend: If either X or Y is uninstantiated.

Exception: If either X or Y is not an integer.

multiply_extended(X, Y, ^Zh, ^Zl) :: B

Taking X and Y to be 32-bit unsigned integers, the upper 32 bits of the product are unified with Zh and the lower 32 bits are unified with Zl.

Suspend: If either X or Y is uninstantiated.

Exception: If either X or Y is not an integer.

devide_extended(Xh, Xl, Y, ^Z, ^Mod) :: B

Taking Xh, Xl, and Y to be 32-bit unsigned integers, the result of dividing the unsigned 64-bit integer of which the upper 32 bits are Xh and the lower 32 bits are Xl by Y is unified with Z, and the remainder is unified with Mod.

Suspend: If Xh, Xl or Y is uninstantiated.

Exception:

- If either Xh, Xl or Y is not an integer.
- If the result of the operation cannot be represented by a 32-bit integer.
- If Y is 0.

7.4.9 Comparison of multiple precision integers

The multiple precision integer expressions that are the arguments to the following built-in predicates are 32-bit strings, and are handled as though they represented unsigned integer values. The MSB is the zeroth element and is zero suppressed.

bignum_less_than(Bignum1, Bignum2) :: G

Succeeds if Bignum1 is less than Bignum2 when Bignum1 and Bignum2 are seen as multiple precision integers. Bignum1 < Bignum2 であれば成功.

Suspend: If either Bignum1 or Bignum2 is uninstantiated.

Fail:

- If either Bignum1 or Bignum2 is anything other than a 32-bit string multiple precision integer.
- If the success condition is not fulfilled.

bignum_not_less_than(Bignum1, Bignum2) :: G

Succeeds if Bignum1 is not less than Bignum2 when Bignum1 and Bignum2 are seen as multiple precision integers. Bignum1 $\geq$ Bignum2 であれば成功.

Suspend: If either Bignum1 or Bignum2 is uninstantiated.

Fail:

- If either Bignum1 or Bignum2 is anything other than a 32-bit string multiple precision integer.
- If the success condition is not fulfilled.

7.4.10 Multiple precision integer operations

The multiple precision integer expressions that are the arguments to the following built-in predicates are 32-bit strings, and are handled as though they represented unsigned integer values. The MSB is the zeroth element and is zero suppressed.

add_bignum(BigX, BigY, ^BigZ) :: B

Adds BigX and BigY and unifies the result with BigZ.

Suspend: If either BigX or BigY is uninstantiated.

Exception: If either BigX or BigY is anything other than a 32-bit string multiple precision integer.

subtract_bignum(BigX, BigY, ^Borrow ^BigZ) :: B

The difference between BigX and BigY is calculated and the absolute value of that result is unified with BigZ. If the result was non-negative 0, otherwise 1, is unified with Borrow.

Suspend: If either BigX or BigY is uninstantiated.

Exception: If either BigX or BigY is anything other than a 32-bit string multiple precision integer.

multiply_bignum_by_word(BigX, Y, ^BigZ) :: B

The result of multiplying BigX by Y seen as a 32-bit unsigned integer is unified with BigZ.

Suspend: If either BigX or Y is uninstantiated.

Exception:

- If BigX is anything other than a 32-bit string multiple precision integer.
- If Y is not an integer.

【Hint】 We are planning to supply a module that implements a variety of multiple precision integer operations using the basic built-in predicates for multiple precision integer comparison and calculation described in this section. This module is currently under development. The module name will be bignum. We are also planning to provide macros for multiple precision integer arithmetic expressions. These modules are expected to support operators of the form "^:=" and "^<".

7.4.11 Vector

vector(X, ^Size, ^NewVector) :: B

The size of the vector X is unified with Size, and X is unified with NewVector.

Suspend: If X is uninstantiated.

Exception: If X is anything other than a vector.

new_vector(^Vector, Size) :: B

Generates a vector whose number of elements is Size and unifies it with Vector. All elements of the vector are initialized with integer 0.

Suspend: If Size is uninstantiated.

Exception: If Size is anything other than a non-negative integer.

vector_element(Vector, Position, ^Element) :: G

Unifies the Position'th element of Vector with Element.

Suspend: If either Vector or Position is uninstantiated.

Fail:

- If Vector is anything other than a vector.
- Either if Size is anything other than a non-negative integer or larger than the number of elements.

vector_element(Vector, Position, ^Element, ^NewVector) :: B

Unifies the Position'th element of Vector with Element and unifies Vector with NewVector.

Suspend: If either Vector or Position is uninstantiated.

Exception:

- If Vector is anything other than a vector.
- Either if Size is anything other than a non-negative integer or larger than the number of elements.

set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B

Unifies the Position'th element of Vector with OldElem, and generates a new vector by replacing Position'th element of Vector with NewElem and unifies the result with NewVect.

Suspend: If either Vector or Position is uninstantiated.

Exception:

- If Vector is anything other than a vector.
- Either if Size is anything other than a non-negative integer or larger than the number of elements.

【Hint】 NewElem is not necessarily defined and may be an undefined variable when this built-in predicate is executed.

7.4.12 String comparison

The following method is used for string comparison. If the bit widths of the elements differ, the string with the wider elements is "larger". Otherwise the strings are compared element by element starting at the first character. During the comparison, if it is found that one string is longer than the other, the longer string is "larger".

string_less_than(String1, String2) :: G

Succeeds if String1*islessthan*String2.

Suspend: If either String1 or String2 is uninstantiated.

Fail:

- If either String1 or String2 is anything other than a string.
- If the success condition is not fulfilled.

string_not_less_than(String1, String2) :: G

Succeeds if String1*isnotlessthan*String2.

Suspend: If either String1 or String2 is uninstantiated.

Fail:

- If either String1 or String2 is anything other than a string.
- If the success condition is not fulfilled.

7.4.13 String manipulation

string(X, ^Size, ^ElementSize, ^NewString) :: B

The size of string X is unified with Size, the element bit width is unified with ElementSize, and X is unified with NewString.

Suspend: If X is uninstantiated.

Exception: If X is anything other than a string.

new_string(^String, Size, ElementSize) :: B

Generates a new string whose number of elements is Size and bit width is ElementSize and unifies the result with String. All the elements of newly generated string are initialized with integer 0.

Suspend: If either Size or ElementSize is uninstantiated.

Exception:

- If Size is a non-negative integer.
- If ElementSize is an integer other than 1, 8, 16, 32.

string_element(String, Position, ^Element) :: G

Assumes the Position'th element of String as an unsigned integer and unifies it with Element.

Suspend: If either String or Position is uninstantiated.

Fail:

- If String is anything other than a string.
- Either if Position is other than a non-negative integer or larger than the number of elements of String.

string_element(String, Position, ^Element, ^NewString) :: B

Unifies the Position'th element of String with Element and unifies String with NewString simultaneously.

Suspend: If either String or Position is uninstantiated.

Exception:

- If String is anything other than a string.
- Either if Position is other than a non-negative integer or larger than the number of elements of String.

set_string_element(String, Position, NewElement, ^NewString) :: B

Generates a new string by replacing Position'th element of String with NewElement and unifies the result with NewString.

Suspend: If String, Position or NewElement is uninstantiated.

Exception:

- If String is anything other than a string.
- Either if Position is other than a non-negative integer or larger than the number of elements of String.
- If NewElement is anything other than a positive integer of a size that can be held in an element of String. (Excepts for 32-bit strings: NewElement can be a negative integer in this case.)

change_element_size(String, NewElemSize, ^NewString) :: B

Unifies NewString with an object that is the conversion of the bit width of the elements of String to NewElemSize. NewElemSize must be 1, 8, 16 or 32.

If the bit width is increased, multiple elements of String will correspond to a single element in NewString, and if it is decreased, a single element of String will correspond to multiple elements in NewString. The correspondence between multiple elements and a single element is that the smallest index element occupies the highest order bits in the single element. For example, if String is an 8-bit string and NewElemSize is 16, element 0 in String will become the upper 8 bits of element 0 in NewString, and element 1 in String will become the lower 8 bits of element 0 in NewString.

When converting from narrow bit widths to wider bit widths, if the number of elements in String do not result in an alignment with the boundaries of the elements of NewElemSize, extra elements are appended to the end up to the point of agreement with the NewElemSize boundary, and those elements are filled with zeros.

Suspend: If either String or NewElemSize is uninstantiated.

Exception:

- If String is not a string.
- If NewElemSize is anything other than 1, 8, 16, 32.

7.4.14 String element search

search_character(String, Start, End, CharacterCode, ^Position, ^NewString) :: B

Searches for a character identical to CharacterCode in the range from Start to End (including the element End) in String, unifies the position of the first such element found with Position, and unifies String with NewString. If End*islessthan*Start, it searches from the rear. If a character identical to CharacterCode is not found between Start and End, -1 is unified with Position.

Suspend: If String, Start, End or CharacterCode is uninstantiated.

Exception:

- If String is anything other than a string.
- If Start is not an integer such that: $0 \leq$ Start $<$ the number of elements in the string.
- If End is not an integer such that $0 \leq$ End $<$ the number of elements in the string.
- If CharacterCode is anything other than a positive integer of a size that can be held in an element in String. (Excepts for 32-bit strings: CharacterCode can be negative in this case.)

search_characters
(String, Start, End, Mask, Table, ^Position, ^NewString, ^NewTable) :: B

Searches the elements of String in the range from Start to End (including the element End) for an element, say ELEM, such that the logical AND of Mask (integer) and the element of Table indexed by ELEM is non-zero, unifies the position of the first such element of String with Position, and unifies NewString and NewTable with objects identical to String and Table respectively.

Table must be one of the following:

- If String is an 8-bit string: an 8-bit string with 256 elements.
- If String is a 16-bit string: an 8-bit string with 65536 elements.

【Caution】String must be either an 8-bit or a 16-bit string.

Suspend: If String, Start, End, Mask or Table is uninstantiated.

Exception:

- If String is anything other than an 8-bit string or a 16-bit string.
- If Start is not an integer such that $0 \leq$ Start $<$ the number of elements in the string.
- If End is not an integer such that $0 \leq$ End $<$ the number of elements in the string.
- If Mask is not an integer.
- If Table is not an 8-bit string.

7.4.15 General string operations

binary_string_operation
(Op, String1, Position1, Length, Position2, String2, ^NewString1, ^NewString2) :: B

Performs the operation specified by Op (Table 7.4) on Length elements starting at Position1 in String1 and on corresponding elements in String2 starting at Position2, and unifies NewString1 with a string that results from replacing the Length elements starting at Position1 in String1 with the results of that operation. String2 is unified with NewString2 without change.

Suspend: If Op, String1, Position1, Length, Position2 or String2 is uninstantiated.

Exception:

- If Op is not an integer from 0 to 15.
- If neither String1 nor String2 is a string.
- If String1 and String2 do not have the same bit width.
- If none of Position1, Position2 and Length is a non-negative integer.
- If Position1 + Length*isgreaterthanthe*(number of elements in String1).
- If Position2 + Length*isgreaterthanthe*(number of elements in String2).

【Caution】 Even when the operation specified by Op does not require the String2/Position2 data, the values of String2 and Position2 are checked.

shifting_string_operation
(Op, String1, Position1, Length, Position2, ^NewString1) :: B

Performs the operation specified by Op (Table 7.4) on Length elements starting at Position1 in String1 and on corresponding elements in String1 starting at Position2, and unifies NewString1 with a string that results from replacing the Length elements starting at Position1 in String1 with the results of that operation.

Suspend: If Op, String1, Position1, Length or Position2 is uninstantiated.

Exception:

- If Op is not an integer from 0 to 15.
- If String1 is not a string.
- If none of Position1, Position2 and Length is a non-negative integer.
- If Position1 + Length*isgreaterthan*(the number of elements in String1).
- If Position2 + Length*isgreaterthan*(the number of elements in String2).

【Caution】 Even when the operation specified by Op does not require the Position2 data, the value of Position2 is checked.

unary_string_operation(Op, String1, Position1, Length, ^NewString1) :: B

Performs the operation specified by Op (Table ??) on Length elements (paired) starting at Position1 in String1, and unifies NewString1 with a string that results from replacing the Length elements starting at Position1 in String1 with the results of that operation.

Suspend: If Op, String1, Position1 or Length is uninstantiated.

Exception:

Table 7.4: General String Operations

No.	macro	Output for inputs S1 and S2
0	string_operation#clear	0
1	string_operatin#and	S1 AND S2
2	string_operation#s2_and_not_s1	(not S1) AND S2
3	string_operation#s2	S2
4	string_operation#s1_and_not_s2	S1 and (not S2)
5	string_operation#s1	S1
6	string_operation#xor	S1 XOR S2
7	string_operation#or	S1 OR S2
8	string_oprtation#nor	S1 NOR S2
9	string_operation#not_xor	S1 not XOR S2
10	string_operation#not_s1	not S1
11	string_operation#s2_or_not_s1	(not S1)O R S2
12	string_operation#not_s2	not S2
13	string_operation#s1_or_not_s2	S1 OR (not S2)
14	string_operation#nand	S1 NAND S2
15	string_operation#set	1

- If Op is not an integer from 0 to 15.
- If String1 is not a string.
- If neither Position1 nor Length is a non-negative integer.
- If Position1 + Length$isgreaterthan$(the number of elements in String1).

【Hint1】 It is convenient to specify the argument Op with the macros shown in Table 7.4. These macros are defined in the pimos module.

【Hint2】 The string macro utilities described in the following section have been re-implement in the built-in predicates described in this section.

7.4.16 String macro utility

The following predicates, described using macros, can be specified to call the string-related utilities provided by PIMOS. The following macros are defined in the pimos module.

builtin#substring(String, Position, Length, ^SubString, ^NewString) :: B

Generates a new string by copying the length "Length" from the Position'th location of String and unifies the result with SubString, and unifies String with NewString simultaneously.

Suspend: If String, Position or Length is uninstantiated.

Exception:

- If String is anything other than a string.
- If Position is not an integer from 0to(the number of elements in String).
- If Length is not an integer from 0to(the number of elements in String)$minus$Position.

builtin#set_substring(String, Position, SubString, ^NewString) :: B

Generates a string by replacing the latter elements starting at the Position'th position in String with the string indicated by Substring, and unifies it with NewString.

Suspend: If String, Position or SubString is uninstantiated.

Exception:

- If neither String nor SubString is a string.
- If String and SubString do not have the same bit width.
- If Position is not an integer such that 0 ≤ Position ≤ (the number of elements in String)*minus*(the nu

builtin#set_substring(String, Position, Length, SubString, ^NewString) :: B

Generates a string by replacing Length elements starting at the Position'th element in the String with Length elements starting at the 0th element indicated by SubString, and unifies that result with NewString.

Suspend: If String, Position, Length or SubString is uninstantiated.

Exception:

- If neither String nor SubString is a string.
- If String and SubString do not have the same bit width.
- If Position is not an integer such that 0 ≤ Position ≤ (the number of elements in String).
- If Length is not an integer such that 0 ≤ (the number of elements in SubString).
- If Length is not an integer such that Length ≤ the number of elements in String*minus*Position.

builtin#set_substring
(String, Position, Length, SubString, ^NewString1, ^NewString2) :: B

Generates a string by replacing the Length elements starting at the Position'th position in String with Length elements starting at the 0th position in the string indicated by SubString, and unifies that result with NewString1. String is unified with NewString2 without change.

Suspend: If String, Position, Length or SubString is uninstantiated.

Exception:

- If neither String nor SubString is a string.
- If String and SubString do not have the same bit width.
- If Position is not an integer such that 0 ≤ Position ≤ (the number of elements in String).
- If Length is not an integer such that 0 ≤ *Length* ≤ (the number of elements in SubString).
- If Length is not an integer such that Length ≤ (the number of elements in String)*minus*Position.

builtin#copy_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

Generates a string by replacing the Length elements starting at the Position'th position in String1 with Length elements starting at the Position2 in String2, and unifies that result with NewString1. String2 is unified with NewString2 without change.

Suspend: If String1, String2, Position1, Position2 or Length is uninstantiated.

Exception:

- If neither String1 nor String2 is a string.
- If String1 and String2 do not have the same bit width.
- If Position1 is not an integer such that $0 \leq$ Position1 $\leq$ (the number of elements in String1).
- If Position2 is not an integer such that $0 \leq$ Position2 $\leq$ (the number of elements in String2).
- If Length is not an integer such that $0 \leq$ Length $\leq$ (the number of elements in String1)*minus*Position1
- If Length is not an integer such that $0 \leq$ Length $\leq$ (the number of elements in String2)*minus*Position2

builtin#or_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#xor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#and_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#nand_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#nor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#not_xor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

These macros generate a string by replacing Length elements starting at the Position'th position in String1 with the logical OR, XOR, AND, NAND, NOR, or XOR of the corresponding Length elements starting at Position2 in String2, and unify that result with NewString1. String2 is unified with NewString2 without change.

Suspend: If String1, String2, Position1, Position2 or Length is uninstantiated.

Exception:

- If neither String1 nor String2 is a string.
- If String1 and String2 do not have the same bit width.
- If Position1 is not an integer such that $0 \leq$ Position1 $\leq$ (the number of elements in String1).
- If Position2 is not an integer such that $0 \leq$ Position2 $\leq$ (the number of elements in String2).
- If Length is not an integer such that $0 \leq$ Length $\leq$ (the number of elements in String1)*minus*Position1
- If Length is not an integer such that $0 \leq$ Length $\leq$ (the number of elements in String2)*minus*Position2

builtin#complement_string_elements(String, Position, Length, ^NewString) :: B

Generates a string by replacing Length elements starting at the Position'th position in String with their complement, and unifies that result with NewString.

Suspend: If String, Position or Length is uninstantiated.

Exception:
- If String is not a string.
- If Position is not an integer such that $0 \leq$ Position $\leq$ (the number of elements in String).
- If Length is not an integer such that $0 \leq$ Length $\leq$ (the number of elements in String)*minus*Position1.

builtin#move_string_elements
(String, Position, Length, ShiftCount, ^NewString) :: B

Generates a string by shifting Length elements starting at element Position'th position in String by exactly ShiftCount positions, and unifies that result with NewString. When ShiftCount$is greater than zero$, this macro shifts to the right, and when $less than zero, to the left$.

Suspend: If String, Position, Length or ShiftCount is uninstantiated.

Exception:

- If String is not a string.
- If Position is not an integer such that 0 ≤ Position ≤ (the number of elements in String)$minus$ShiftCo
- If Length is not an integer such that 0 ≤ Length ≤ (the number of elements in String)$minus$Position1
- If ShiftCount is not a non-negative integer.

builtin#string_tail(String, Position, ^NewString) :: B

Unifies NewString with a string that consists of the elements of String from the Position'th position to the end of String.

Suspend: If either String or Position is uninstantiated.

Exception:

- If String is not a string.
- If Position is not an integer such that 0 ≤ Position ≤ (the number of elements in String).
  の整数でない場合

builtin#append_string(String1, String2, ^NewString) :: B

Generates a new string that consists of the elements of String2 appended to the end of String1, and unifies that result with NewString.

Suspend: If String1 and String2 are uninstantiated.

Exception:

- If neither String1 nor String2 is a string.
- If String1 and String2 do not have the same bit width.

builtin#append_string([String1, String2, ..., StringN], ^NewString) :: B

Generates a new string by concatenating the elements of String1 through StringN, and unifies that result with NewString.

Suspend: If any one of the variables String1 to StringN is uninstantiated.

Exception:

- If any one of String1 to StringN is not a string.
- If the bit widths of String1 to StringN are not all the same.
- If the first argument is either the empty list or not a list.

【Hint】 In version 2.5, if the first argument was an empty list, then a length 0 16-bit string was unified with NewString.

7.4.17 Atom

new_atom(^Atom) :: B

Generates a new atom and unifies the result with Atom.

Suspend, Exception: None

【Caution】 In PIMOS/KL1, it is PIMOS that is responsible for providing correspondence between an atom and a name string, and the atom created with this built-in predicate has no correspondence with the name string. To create a named atom, use the atom name database described later in the section (10.1 on page 404).

7.4.18 Code and module

Caution is required using the built-in predicates in this section since there are many machine dependent characteristics related to code and modules in each PIM and Multi-PSI implementation.

module_element(Module, Position, ^Element) :: B

Unifies the Position'th element in Module with Element.

Suspend: If either Module or Position is uninstantiated.

Exception:

- If Module is not a module.
- If Position is not an integer such that $0 \leq$ Position $<$ (the number of elements in Module).

predicate_to_code(Module, Name, Arity, ^Code) :: B

Unifies a code denoting module Module, predicate name Name and argument count Arity with Code. If a corresponding predicate is absent in the module (because undefined or not declared in public), unifies a 3-element vector that is {`Module, Name, Arity`} with Code.

Suspend: If Module, Name or Arity is uninstantiated.

Exception:

- If Module is anything other than a module.
- If Name is anything other than an atom.
- If Arity is anything other than a non-negative integer.

code_to_predicate(Code, ^Module, ^Name, ^Arity, ^Info) :: B

Unifies a module to which Code belongs with Module, a predicate atom with Name, argument count with Arity, and 1-word information (contents of <Code address -3>) for software with Info.

Suspend: If Code is uninstantiated.

Exception: If Code is anything other than a code.

【Caution】 The specifications of this predicate are machine dependent, and those specifications are for the Multi-PSI and PIM/m.

apply(Code, Args) :: B

Calls a predicate corresponding to Code with argument environment Args. Code に対応する述語を引数環境 Args で呼び出す.

Suspend: If either Code or Args is uninstantiated.

Exception:

- If Code is anything other than a code.
- If Code is a destination code of a predicate to be spied.
- If Args is anything other than a vector.
- If the number of arguments to Code and the number of elements in Args do not agree.

apply_tracing(Code, Args, ID) :: B

Calls a predicate corresponding to Code with argument environment Args with tracing facilities. ID is used to identify this goal (optional data type is acceptable), and is returned when trace exception occurs.

Suspend: If either Code or Args is uninstantiated.

Exception:

- If Code is anything other than a code.
- If Args is anything other than a vector.
- If the number of arguments to Code and the number of elements in Args do not agree.
- If called from within the execution of the built-in predicate apply_spying.

apply_spying(Code, Args, Spied_code_vector, ID) :: B

Calls a predicate corresponding to Code with argument environment Args with spying facilities. ID is used to identify this goal (optional data type is acceptable), and is returned when spy exception occurs. Spied_code_vector is a vector for specifying a predicate to be spied, and has iteration sequence of module_name_atom, predicate_name_atom, and arity as elements. The wildcard of module_name_atom and predicate_name_atom is integer 0, and the wildcard of arity is [] (NIL). Spied_code_vector has the following format.

```
{ Mod1, Pred1, Arity1, Mod2, Pred2, Arity2, ... , ModN, PredN, ArityN }
```

Suspend: If Code, Args, Spied_code_vector or ID is uninstantiated.

Exception:

- If Code is anything other than a code.
- If Code is a destination code of a predicate to be spied.
- If Args is anything other than a vector.
- If the number of arguments to Code and the number of elements in Args do not agree.
- If the elements of Spied_code_vector are undefined.
- If the format of Spied_code_vector is incorrect.
- If called from within the execution of the built-in predicate apply_spying.

7.4.19 Stream support

merge(In, ^Out) :: B

Creates a built-in constant time merger and unifies the input stream for that merger with In, and the output stream with Out. When unifying a vector with the input stream, the merger handles all elements in the vector as new input streams.

Suspend, Exception: None

Logical definition of the merger can be defined as the predicate which has the number of arguments from 2 to infinity as shown below.

```
merge([], O) :- true | O=[].
merge([A|I], O) :- true | O=[A|NO], merge(I, NO).
merge({}, O) :- true | O=[].
merge({I}, O) :- true | merge(I, O).
merge({I1, I2}, O) :- true | merge(I1, I2, O).
merge({I1, I2, I3}, O) :- true | merge(I1, I2, I3, O).
                 • • •
merge([], I2, O) :- true | merge(I2, O)
merge(I1, [], O) :- true | merge(I1, O).
merge([A|I1], I2, O) :- true | O=[A|NO], merge(I1, I2, NO).
merge(I1, [A|I2], O) :- true | O=[A|NO], merge(I1, I2, NO).
merge({}, I2, O) :- true | merge(I2, O).
merge(I1, {}, O) :- true | merge(I1, O).
merge({I3, I4}, I2, O) :- true | merge(I3, I4, I2, O).
merge({I3, I4, I5}, I2, O) :- true | merge(I3, I4, I5, I2, O).
                 • • •
                 • • •
```

7.4.20 Special input/output

【Caution】 The built-in predicates for input and output described here are provided for the low level debugging support, and are not suitable for ordinary I/O use in terms of both functions and performances. Also, some of these functions are machine dependent.

read_console(^X) :: G

A data input from CSP window is read in and unified with X. The input is in the form of "tag part [blank] value part" each part being entered with hexadecimal numbers. The tag part may be omitted and is assumed to be an integer.

Suspend, Fail: None

【Caution】 All processing nodes halt when waiting for input. The specifications may differ in implementations other than PIM/m and Multi-PSI.

display_console(X) :: G

The current value of X is displayed in the CSP window and LF is also entered. If X is an 8-bit string, it is assumed to be an ASCII string, and if X is a 16-bit string, it is assumed to be a

JIS string and the contents of that string is written to the CSP window. If X is a vector, the contents of each element is written to the CSP window. If X is other data type, the tag part and the value part are displayed.

Suspend, Fail: None

【Caution】 All nodes halt when outputting. The specifications may differ in implementations other than PIM/m and Multi-PSI.

get_console(^X) :: G

One character input from the CSP window is read in, and the ASCII code for that character is unified with X.

【Caution】 All nodes halt when waiting for input.

put_console(X) :: G

If X is an integer, it is assumed to be an ASCII code and the corresponding character is displayed in the CSP window. At this time, no LF is entered. If X is an 8-bit string, it is assumed to be an ASCII string, and the contents of that string is displayed in the CSP window with no LF entered. If X is other data type, no operation is guaranteed.

Suspend, Fail: None

【Caution】 All processors halt when outputting. The specifications may differ in implementations other than PIM/m and Multi-PSI.

7.4.21 Others

current_priority(^Priority, ^Min, ^Max) :: B

The logical priority of this predicate is unified with Priority, the minimum priority acceptable in shoen to which the predicate belongs is unified with Min, and the maximum priority is unified with Max.

Suspend, Exception: None

calculate_priority_from_current(X, ^Y) :: B

When the self-relative priority X is specified, the logical priority corresponding to X is calculated from the logical priority of this predicate, and unified with Y.

Suspend: If X is uninstantiated.

Exception: If X is not an integer from -4096 to 4096.

calculate_priority_from_minimum(X, ^Y) :: B

When the relative priority within the attached shoen is specified, the logical priority corresponding to X is calculated from the logical priority range for the shoen belonging to this predicate, and unified with Y.

Suspend: If X is uninstantiated.

Exception: If X is not an integer from 0 to 4096.

current_node(^NodeNumber, ^TotalNode) :: B

NodeNumber is unified with the number of node on which this predicate is running and TotalNode is unified with the total number of nodes. The node number will be a value between 0 and (TotalNode - 1) inclusive. TotalNode − 1 までふられている.

Suspend, Exception: None

get_heap_size(^Size, ^Used) :: B

This predicate returns the memory usage state for the node on which the predicate is running. Size is unified with the available memory size, and Used is unified with the currently used memory, as integer values. The units of these values are system dependent.

Suspend, Exception: None

【Hint】 These values are in word units (the memory unit that can hold a pointer or an integer value) on Multi-PSI and PIM/m.

hash(X, ^Value, ^NewX) :: B

Generates an integer (hash value) and unifies the result with Value. NewX is unified with X. X may be an optional type.

The following integers are generated depending on data type.

Integer: Value part (as it is)

Atom: Value part (atom number)

Other atomic data: Value part

List: −1

Vector: Length of vector

String: Calculated from string length and first and last elements.

Others: 0

Suspend: If X is uninstantiated.

Exception: None

【Caution】 The specification of the value returned by the hash function may be modified in the future. In principle, it is guaranteed that the same hash value should be returned (as long as computation is completed) even if hash function is repeatedly computed during a single execution for the same data (even if its structure includes undefined part while calling). Hash value never changes even in cases where the pending hash is completed because of the instantiation of variable.

【Hint】 The standard PIMOS hashing function is not this built-in predicate but the hashing function provided by the "hasher" utility. (See section 8.2, "Hashing", on page 319)

system_timer(^X1, ^X2) :: G

The upper 16 bits of the time (in 1/16 msec units) that has passed since the processor on which this predicate is running was booted is unified with X1, and the lower 32 bits are unified with X2.

Suspend, Exception: None

【Caution】 The specifications of this predicate are hardware dependent, and in systems other than PIM/m and Multi-PSI the unit depends on the machine type. Also, there are models that do not have a timer. Therefore, we recommend to access this function through the utility described in section 9.2.3 on page (368) in ordinary use.

7.4.22 Reference count management

Reference counts are managed using the MRB method on current KL1 language processors in order to implement efficient incremental garbage collection. With this mechanism, the allocation of the actual memory area can be avoided quite often, even though the specifications of built-in predicates seem to require the memory allocation.

For example, the specifications for the built-in predicate set_vector_element/5 seem to require the generation of a new vector that differs from the original vector in only a single element. However, when the reference count management guarantees that the original vector is not used anywhere else, the operation can be done without new memory allocation by updating the original vector destructively. This not only reduces memory usage, but also can reduce the order of the execution time of the operation, since the time required for copying the elements is unnecessary. The same strategy can also be applied to other built-in predicates, such as set_string_element, binary_string_operation, unary_string_operation, shifting_string_operation, each of which creates a new structure by over-writing part of the elements of an existing structure, as well as to macros that are expanded into these operations.

Since the MRB mechanism does not require extra memory accesses, it is an efficient reference count management technique for parallel processing. However, once a data reference path is duplicated during the execution, there is no way to detect the decrease of the reference paths with the MRB mechanism, hence the destructive update can never be performed from that time on. As a result, it is desirable to adopt the following coding style: keep the data single-referenced so long as it is possible.

In the example below, there two reference paths to Table are created, the one when Table is passed to vector_element/4, and the one when Table is passed to set_vector_element/5, although the reference from vector_element/4 is immediately discarded after the element is taken. As a result, the update cost due to set_vector_element/5 is proportional to the size of the vector since new vector data must be created by copying all the elements from Table data.

```
increment_element(Table, K, NewTable) :- true |
    vector_element(Table, K, Elem, _),
    NewElem := Elem + 1,
    set_vector_element(Table, K, _, NewElem, NewTable).
```

The data can be kept single-referenced and the operation can be performed in a constant time independent of the vector size, if the program is written as shown below.

```
increment_element(Table, K, NewTable) :- true |
    set_vector_element(Table, K, Elem, NewElem, NewTable),
    NewElem := Elem + 1.
```

Although the value of NewElem is not determined on the execution of set_vector_element in this program, it operates correctly since the variable NewElem, which will take on the result of the sum, is stored as an element of the NewTable.

Even when the built-in predicates are not called directly operation can be done in the same manner. For example, consider a program that uses the predicate "compute" to perform some computation using a table, and then uses the predicate "update" to store the result in the table. If this is coded in this manner:

```
do(Table, NewTable) :- true |
    compute(Table, Values),
    update(Table, Values, NewTable).
```

then there are two references to the variable Table, and efficiency might become a problem. If this is re-coded as follows:

```
do(Table, NewTable) :- true |
    compute(Table, Values, Table1),
    update(Table1, Values, NewTable).
```

where the table that "compute" has finished reading is returned as a new variable, and then passed to "update", then the table can be kept single-referenced.

We recommend this coding manner, in which a single reference path is kept, especially for cases where structure data is frequently updated. However, it is not necessary to always follow this rule. Maintaining a single reference path can sometimes restrict the parallelism, since the next predicate cannot use necessary dta until the previous predicate has completed its execution. Thus programmers are responsible for determining whether parallelism should be gained at the risk of increasing the cost due to multiple references or whether single reference paths should be maintained. Decisions should be made by taking account of the profit and loss resulted from multiple references and the limited parallelism.

7.5 Shoen Function

Module shoen of PIMOS provides the user with the shoen function provided by KL1 in the form of easy-to-use feature. This module also plays a role of showing the functions among those of KL1 that will survive the abuse of users.

【Hint】 PIMOS provides the task function which accommodates the management function for I/O devices in addition to the shoen function. See chapter 11, "Resource and Execution Management" (page 418) for more information on tasks.

7.5.1 Overview

Shoen encompasses part of KL1 program execution (top-level goals of shoen and a group of descendent goals derived from the top level goals). In general, the shoen comprises a tree structure in which shoens can be nested. Shoen has the following functions:

Execution control: Controls start, suspension, restart, and abort for the execution of the goals in the shoen (hereinafter simply called "shoen execution").

【Caution】 For efficient parallel processing, the specification is such that these operations are not always performed in a click of time.

Resource management: Manages the computer resources (hereinafter simply called "resources" in this section) including computing time or memory areas used in shoen execution. Specifically, as the consumption of resources comes close to the specified upper bound, it is reported. When resource consumption actually exceeds the upper bound, execution is suspended. The upper bound value of resource consumption can be increased later (though, once the upper bound value is set, it cannot be decreased). If the upper bound value is increased when execution is suspended due to trespassing the upper bound, shoen execution resumes.

The resources managed by the KL1 language shoen function are called "Language Definition Resources" in contrast to the resources managed by PIMOS.

【Caution】 The resource consumption currently managed only includes those corresponding to the number of reductions (1 for goal creation and 1 for termination of the reduction, thus 2 is taken for each reduction). In the future, computing time, memory consumption, and communication traffic between nodes will also be managed.

Exception management: Manages the exceptions taking place during shoen execution. Failed execution (such as absence of the candidate for the goal, and failed unification) is also treated as exception.

Each shoen has a mask pattern of integer used to determine what exception be managed. Each exception event also has a tag of integer according to its type, and the exception is reported to the innermost parent shoen for which AND of the exception tag and the mask pattern is not 0.

When an exception occurs, the information indicating what exception occurs is reported, and also the variables to specify the code (corresponding to the predicate) and the argument vector of the goal to be executed in place of the goal which caused the exception are passed to the exception manager. By instantiating these variables, the goal which caused the exception is replaced with the goal specified by the code and the argument vector. Since there are goals in the shoen awaiting the instantiation of these variables, shoen execution does not successfully terminate unless the exception management has been performed.

7.5.2 Creation of shoen

shoen:execute(Code, ArgV, MinPrio, MaxPrio, ExcepMask, Control, ^Report)

Creates a shoen.

The goal at the top level in the shoen to be created is specified by Code (i.e., the code type data corresponding to the top level predicate) and ArgV (i.e., the argument vector of the top level goal).

Execution control of the created shoen is performed from the stream Control, and report of the execution status is made from the stream Report. See section 7.5.3 (page 304) and section 7.5.4 (page 305) for details on control stream and report stream respectively.

The range of priority of the goals in a shoen is specified with MinPrio and MaxPrio. Priority is specified with integers in a range between 0 and 4096, of which 0 is the minimum priority within the outer shoens, and 4096 corresponds to the priority of the goal which called the creation predicate.

ExcepMask is the mask pattern which determines the exception event to be managed. See section 7.5.6 (page 314) for details on Exception tag.

【Caution】 When resources that are managed by PIMOS (i.e., OS defined resources) are used within a created shoen, those resources must be correctly terminated. ("Correctly terminate" means to close normally by closing all streams to that resource. Therefore, for example, discarding an execution within a shoen by aborting does not meet the definition of correct termination.) If a resource is not correctly terminated, PIMOS is not guaranteed to operate correctly thereafter. The task functions must be used when OS defined resources, such as I/O equipment, are used in a user created shoen. Therefore, use of the shoen module may be forbidden in the future.

shoen:profile(Code, ArgV, MinPrio, MaxPrio, ExcepMask, Control, ^Report)

Creates a shoen with the profiling function attached.

The arguments are the same as those to the `shoen:execute/7` predicate. The profiling function measures how many times each predicate is executed in the code executed in a user generated shoen.

Execution control of the created shoen is performed from the stream Control, and execution reporting is performed from the stream Report. Measurements can also be collected during shoen execution. See section 7.5.3 (page 304) and section 7.5.4 (page 305) for details on profiling related messages.

【Caution1】 After a profiling shoen is created, measurement cannot be stopped. Also, when a (child) shoen is created in a profiling shoen, that child shoen automatically becomes a profiling shoen. However, measured data is only reported for the parent shoen.

【Caution2】 As is the case with shoen:execute/7, profiling tasks must be used if resources that are managed by PIMOS (i.e., OS defined resources) are used within the created shoen. See chapter 11, "Resource and Execution Management" (page 418) for more information on profiling tasks.

7.5.3 Control stream

The control stream of a shoen is a message stream in the direction from the user to the shoen. Shoen execution and resource consumption can be controlled using the messages listed below:

【Hint】 A corresponding reception confirmation message is always associated with each message in a control stream.

start

Starts a shoen execution.

Immediately after created, a shoen is dormant, and it is made ready for an execution when this message is sent. This message is also used to restart the execution suspended by the "stop" message shown below.

stop

Suspends a shoen execution.

Execution can be restarted by the abovementioned "start" message.

abort

Aborts a shoen execution.

If an execution is aborted with this message, execution cannot be restarted later.

statistics

Indicates to make a report concerning the amount of resources consumed by an execution in a shoen.

The report is sent from the report stream.

add_resource(Amount_h, Amount_l)

Supplies a resource to a shoen execution.

A value of which the upper 32 bits are expressed in Amount_h and the lower 32 bits are expressed in Amount_l is added to the upper bound value of the resource consumption amount.

allow_resource_report

Allows a report to be sent as the amount of consumed resource has nearly reached its upper bound.

A report is normally ready for being sent, but once a report is sent to inform that the resource comes close to its upper bound, no report is sent any more. Only when this message is received, another report becomes ready for being sent. This mechanism is provided to prevent number of useless reports from being sent.

profile

Requests the total of the profiling data to date from a profiling shoen.

Whether or not the request was accepted can be determined from the reception confirmation message from the report stream.

7.5.4 Report stream

The report stream of a shoen is a message stream in the direction from the shoen to the user. The following two types of messages are routed through the report stream.

Event report message: This message is used to report the events occurring associated with a shoen execution

Receive ACK message: This message is put on the stream when a message from the control stream is received. Based on the order of these messages and the event report messages in the stream, the sequence of the events reported in the event report message and the management of the control message can be determined. For example, the time-dependent relation of "add_resource" (raising of the resource upper bound) and "resource_low" (the current consumption of the resource coming close to the resource upper bound) makes it possible to determine whether an additional resource is needed or not.

terminated

Reports the termination of a shoen.

Whether or not an "aborted" message described later is ahead of this message in the report stream makes it possible to determine whether a termination is a normal one upon the reduction termination of all the goals in a shoen or a forced abnormal one. Once this message has been reported, reduction or unification of the goals in the shoen is no longer performed.

resource_low

Reports that the consumed amount of the resource comes close to the upper bound.

【Caution】 How close the resource consumption has come to the upper bound when this report is made depends on the processing system.

statistics(Info_h, Info_l)

report Report on total resource consumption

Collecting operation starts when the control message "statistics" is received. Upon starting, statistics_started (described later) is returned, and this message is sent upon completion of collecting operation. Info_h is the upper 32 bits and Info_l is the lower 32 bits of the integer (that is, in the current specifications, twice of the number of reductions) which indicates the total amount of resource consumption up to now in a shoen (including the descendent shoens).

If statistics is sent upon completion of shoen execution, this message returns a correct value. If the shoen is in process, reported amount of consumption is incorrect, but it is guaranteed that the amount should not exceed the actual consumption, and two or more times of collecting operations should result in a monotonic increase.

return_resource(Amount_h, Amount_l)

Reports that the upper bound value of resource consumption was too high.

Amount_h and Amount_l are the upper and the lower 32 bits of the resource amount respectively. The remaining amount of the resource when the shoen execution terminated is reported with this message. Furthermore, this is also reported when the amount of the consumable resource (i.e., the upper bound of the resource consumption minus the consumed amount) exceeds the default value (2^{63} on PIM/m and Multi-PSI) during execution.

started

stopped

aborted

resource_added

resource_report_allowed

statistics_started

These are the receive ACK messages corresponding to the control messages start, stop, abort, add_resource, allow_resource_report, and statistics respectively.

【Caution】 Although the processing order can be inferred from the order of these messages, the extent of the progress of the processing cannot be inferred. For example, even if a "stopped" message has been reported, it cannot be inferred that the shoen internal processing has already been suspended at that point.

profiling_started

profiling_rejected

These are the receive ACK messages corresponding to the "profile" control message.

If the request was legal, then "profiling_started" is reported, and if the request was illegal, then "profiling_rejected" will be reported. The "profile" control message is illegal in the following cases.

1. The shoen execution has already terminated or has already been aborted.
2. The collection of intermediate data due to a previously issued "profile" message has not completed.
3. The shoen that corresponds to the executing shoen's parent is also a profiling shoen.
4. The shoen to which the "profile" message was sent was not a profiling shoen.

profiling_completed

Reports the fact that the collection of profiling data in response to a "profile" control message has completed.

profiling_information(Node, Type, Depth, Data)

Reports profiling data. The following data is reported.

Node: Node data. This is the same as the "Node" exception message.

Type: The data type. The following values are returned.

0: Data reported by foster parent completion. [7]

1: Data reported by the "profile" intermediate collection request.

2: Data reported when a counter is about to overflow. [8]

[7] When a goal is executed on a different node from the goal that called that goal (for example, with user specified load distribution), a new shoen is created on the goal's destination node, and the transferred goal is executed within that shoen. That shoen is called a "foster parent shoen". This is the relationship between a foster parent and a foster child. The transferred goal can be thought of as a foster child.

[8] In PIM/m and Multi-PSI, when the processing system counter becomes 16#"7FFFFFFF".

Depth: The depth of shoen nesting

Data: Count data

【Caution】 It is possible that the format of the count data "Data" will differ between PIMs.

【Hint】 In PIM/m and Multi-PSI, the format of the count data "Data" consists of the nested 5-element vector shown below.

User-defined predicates: { `Code, 0, Dequeue_count, Suspend_count, NextData` }

【Caution】 If relinking is incomplete, that is, if some module is pointing to an old module as it was, multiple predicates with the same package name, module name, predicate name, and arity will be reported separately.

Built-in predicates: { `Module, Offset, Dequeue_count, Suspend_count, NextData` }
Reporting for built-in predicates is only performed for suspended predicates. [9] The value of Suspend_count is always 0.

【Caution1】 If a merger execution is interrupted, it will not be reported as profiling data.

【Caution2】 In cases where the same built-in predicate is called multiple times in a given body part, the interruption of the execution of those built-in predicates will be reported separately.

【Hint】 There are cases where special built-in predicates are reported.

enqueue_goal_to_node : When a node specified goal is suspended.

enqueue_goal_with_priority: When an execution priority specified goal is suspended. The next predicate will be reported in the same way.

calculate_priority_from_minimum: Ratio specification in the associated shoen.

calculate_priority_from_current: Specification relative to the current process in the associated shoen.

special instructions: { `Opecode, 0, Dequeue_count, Suspend_count, NextData` }
The value of Suspend_count is always 0.

【Hint】 The following are the special instructions.

apply: Reported in the following cases:

1. When a task (shoen) is created and a goal executed.
2. When a node specified goal is suspended.
3. When an execution priority specified goal is suspended.

unify: Body unification. There are several cases. For example, when a pair of merger input arguments are unified.

unify_vector: Body unification of a pair of vectors. Reported the same times as the number of elements. This can be thought of as the execution of a program similar to the following.

```
unify_vector(0,V1,V2) :-
  vector_element(V1,0,E1), vector_element(V2,0,E2) | E1=E2.
```

[9] Of the built-in predicates, only Dequeue_count is reported. This corresponds to the number of times that built-in predicates have been suspended. (The suspension of a built-in predicate is counted as a single reduction.)

```
unify_vector(N,V1,V2) :- N>0 |
    set_vector_element(V1,N,OE1,[],NV1),
    set_vector_element(V2,N,OE2,[],NV2),
    OE1=OE2, N1 := N-1,
    unify_vector(N1,NV1,NV2).
```

Unify_list: Body unification of a pair of lists. Reported the same times as the number of elements. This can be thought of as the execution of a program similar to the following.

```
unify_list([X1|X2],[Y1|Y2]) :- true | X1=Y1, X2=Y2.
```

Data termination: []

Note that the Dequeue_count and Suspend_count in the data are indicated as follows.

Dequeue_count: The number of times the execution of each predicate was started or restarted [10]

Suspend_count: The number of times each predicate was suspended [11]

Therefore the actual number of goal reductions is given by the following formula.

```
(Reduction count) = (Dequeue_count) - (Suspend_count)
```

【Hint1】 Profile data in response to a "profile" message is reported for the interval between the point the "profiling_started" receive ACK message is reported and the point "profiling_completed" is reported.

【Hint2】 Profile data counting is performed for each node and for each shoen. Since all counters are cleared to zero when profile data is reported, it is necessary to sum up all the data reported until shoen termination to determine the execution counts for the whole shoen.

【Hint3】 The count data "Data" can be converted to a format that is easier to process by using a profiling utility described later in this section. See section 7.5.7.2 (page 316) for more information about this profile data conversion utility.

exception(Kind, Info, ^NewCode, ^NewArgV)

Reports the exception events taking place during a shoen execution.

Kind is an atom indicating the kind of the exception. Info is a data vector on the exception. NewCode and NewArgv are used for specifying the goal to be executed in place of the goal causing the exception, and each of them is the variable corresponding to the code and the argument vector.

The format of exception data (Info) for each type (Kind) of exception is shown below. See Table ??, "Exception Tag Bit Positions", (page 315) for a listing of the exception types. While the contents of exception data differs with the exception type, the following items are common to all exception types.

Node: Data for the node on which the goal that caused the exception existed. This data has the following format.

[10] The number of times dequeue family and execute family instructions were executed.

[11] The number of times suspend family instructions were executed.

`{<node number>, <processor number>}`

The node number is a value that is the cluster number reported from the processing system converted to a relative value, and the processor number is the processor number within the cluster indicated by the node number [12]

Predicate: Data for the predicate that caused the exception and data for the predicate that called that predicate. There are two types of data: "Code" and "Built-in".

Code: Code format data. KL1 defined predicates. Can be manipulated by code related built-in predicates such as predicate_to_code and apply. Predicate information such as the module name and the predicate name can be acquired by using the separately supplied exception processing utilities. See section 7.5.7.1 (page 315) for details on the exception processing utilities.

Builtin: Data on the built-in predicate that caused the exception. This data has one of the following types.

{Code, Caller}: KL1 defined built-in predicates (library predicates)

{Name, Caller}: Firmware defined predicates

{Name, Caller1, Caller2}: Firmware defined built-in predicates called from the library. Caller1 called the built-in predicate Name, and Caller2 called Caller1. That is, Caller1 is data related to the library, and Caller2 is data related to a user defined predicate.

Name: An atom that is the name of a built-in predicate.

Caller: {Module, Offset}

The Module and Offset indicate which KL1 defined predicate called the predicate that caused the exception. In particular, Module is the module that called that predicate, and Offset is the position of that predicate in the module, i.e., the position in the module relative to the beginning of the module.

【Hint】 The module name and the predicate name can be acquired by using the separately supplied exception processing utilities. See section 7.5.7.1 (page 315) for details on the exception processing utilities.

ArgV: Argument data vector for the predicate that caused the exception.

ArgPos: Data indicating the position of the predicate's argument that caused the exception. The number of an origin.

Table of Exception Phenomena

【Hint】 See section 4.1.8.3 (page ??) for examples of the generation of these exceptions.

illegal_input: Illegal input argument type

- {Node, Builtin, ArgV, ArgPos}

ArgPos'th argument is illegal ($1 \leq ArgPos$).

【Hint1】 On PIM/m and Multi-PSI systems, the following built-in predicate names will be returned in Name if an illegal value (e.g., an atom) is specified for the execution priority.

calculate_priority_from_minimum : Ratio specification in the associated shoen.

[12] Due to the structure of the processors, the processor number will always be zero on PIM/m and Multi-PSI systems.

calculate_priority_from_current : Specification relative to the current process in the associated shoen.

Also, the first argument to ArgV (the zeroth element) will be the user specified priority ratio.

【Hint2】 In PIM/m and Multi-PSI systems, the following built-in predicate name will be returned in Name if an illegal value (e.g., an atom) is specified for the execution node.

enqueue_goal_to_node: Corresponds to Goal @ node(Node).

Also, the first argument to ArgV (the zeroth element) will be the user specified node number, and the second argument (the first element) will be the code that called the predicate for which the node specification was made.

range_overflow: Illegal input argument range

- {Node, Builtin, ArgV}
- {Node, Builtin, ArgV, ArgPos}

The argument is not within the range of correct values.

【Hint1】 The following built-in predicate names will be returned in Name if an illegal value is specified for the execution priority.

calculate_priority_from_minimum: Ratio specification in the associated shoen.

calculate_priority_from_current: Specification relative to the current process in the associated shoen.

Also, the first argument to ArgV (the zeroth element) will be the user specified priority ratio.

【Hint2】 The following built-in predicate name will be returned in Name if an illegal value (e.g., an atom) is specified for the execution node.

enqueue_goal_to_node: Corresponds to Goal @ node(Node).

Also, the first argument to ArgV (the zeroth element) will be the user specified node number, and the second argument (the first element) will be the code that called the predicate for which the node specification was made.

out_of_bounds: Illegal index range

- {Node, Builtin, ArgV}
- {Node, Builtin, ArgV, ArgPos}

The structure index (ArgPos'th argument) is out of the range.

integer_overflow: Integer operation digit overflow

- {Node, Builtin, ArgV}

integer_zero_division: Division by zero

- {Node, Builtin, ArgV}

invalid_floating_point_operation: Floating point invalid operation

- {Node, Builtin, ArgV}

【Hint】 Invalid if the following types of operations are performed.

```
(+infinity) + (-infinity)  or   (-infinity) + (+infinity)
(-infinity) - (-infinity)  or   (+infinity) - (+infinity)
 (± 0.0)   * (± infinity)  or  (± infinity) *  (± 0.0)
 (± 0.0)   /   (± 0.0)     or  (± infinity) / (± infinity)
```

arity_mismatch: Wrong number of arguments to a high level predicate

- {Node, Builtin, ArgV}

undefined_module: A call to an undefined module

- {Node, UndefModInfo, ArgV}

UndefModInfo holds data for the undefined module predicate in the following format.

```
{{Package, ModName, Pred, Arity}, Caller}  % If the package differs from that of Call
{         {ModName, Pred, Arity}, Caller}  % If the package is the same as that of Ca
```

Package, ModName, Pred, and Arity are, respectively, the package name atom, the module name atom, the predicate name atom, and the arity of the undefined module predicate. Also, ArgV holds the arguments passed to the undefined module predicate.

undefined_predicate: A call to an undefined predicate

- {Node, UndefPredInfo, ArgV}

UndefPredInfo holds data for the undefined predicate in the following format.

```
{{Module, Pred, Arity}, Caller}
```

Module, Pred, and Arity are, respectively, the module name atom, the predicate name atom, and the arity of the undefined predicate. Also, ArgV holds the arguments passed to the undefined predicate.

illegal_merger_input: Incorrect input to a merger

- {Node, Builtin, Input, NewInput}

Input holds the illegal input data, NewInput holds the new merger input variable for recovery. The built-in predicate name that is reported will always be "merge".

reduction_failure: Failure of the guards in all candidate clauses

- {Node, Code, ArgV}
- {Node, {Code, Caller}, ArgV}

Code is the code for the predicate that failed.

unification_failure: Failure in body unification

- {Node, DataX, DataY}
- {Node, DataX, DataY, Caller}

DataX and DataY are the two data for the failed unification.

【Hint】 While the top level of DataX and DataY must always be instantiated (otherwise, unification never fails), the elements for a structure are not always instantiated.

【Caution】 Unification between two structures is separately performed for each element. So that the failure in unification of `[a, b, c]` and `[a, b, d]` is reported as the failure in the unification of c and d.

deadlock: A deadlock (perpetual suspension) occurred

- {Node, Predicate, ArgV, Type}

- {Node, Predicate, ArgV, Type, Collector}

Type is an integer that indicates the type of deadlock detection. Collector is data concerning the goal (the so-called "murderer") that produced the problem that caused the deadlock. In particular, this is data added when deadlock was detected by real-time GC, and is data for the predicate that tried to reclaim the HOOK variable for which the deadlocked goal was waiting for instantiation. The format is the same as that for "Predicate". Currently there are the following values for Type.

0: Detected by a full GC.

1: A goal in trace execution was detected by a full GC.

2: A goal in spy execution was detected by a full GC.

10: An attempt was made to reclaim a HOOKo (MGHOKo) variable. (collect_value instruction)

11: An attempt was made to suspend on a HOOKo (MGHOKo) variable. (suspend instruction)

12: An attempt was made to unify a pair of HOOKo (MGHOKo) variables. (unify instruction)

13: An attempt was made to suspend on a VOID variable. (suspend instruction)

14: An attempt was made to unify a a VOID variable and a HOOKo variable. (unify instruction)

15: For a HOOKO variable.

Other integer: A different pattern.

【Hint】 See section 4.1.5.3 (page 81) for a listing of deadlock types and detection examples.

merger_deadlock: A merger deadlock occurred.

- {Node, Builtin, Inputs, Type}
- {Node, Builtin, Inputs, Type, Collector}

Type is an integer that indicates the type of deadlock detection. Collector is data concerning the goal that produced the problem that caused the deadlock. Inputs is a vector that holds the discarded merger input variables. The size of the vector corresponds to the number of input variables that were discarded, and each element of the vector is an undefined variable. It is possible to connect an arbitrary stream to these variables, or to close them with [].

trace: A trace exception was detected.

- {Node, TraceIDList, SubGoalInfoList}
- {Node, TraceIDList, SubGoalInfoList, DebugInfo}

TraceIDList is a list of identifiers for identifying goals, and the identifiers are the objects specified as the third argument to the built-in predicate apply_tracing. SubGoalInfoList is a list of the subgoal data SubGoalInfo. DebugInfo is debugging data added by the compiler, and is currently used to report the close number (a positive integer) that was committed.

【Caution】 It is possible that different data will be added to DebugInfo in the future.

【Hint】 During tracing it is possible for goals to output committed close numbers. See section 4.1.7.5 (page 128) for details on the close number output method.

【Hint】 SubGoalInfo has the following format.

{KlbName, Code, ArgV}: User defined predicates

{KlbName, Code, ArgV, Priority}: Priority specification

{KlbName, Code, ArgV, ClsOrProc}: Cluster or processor specification

{unify, {DataX, DataY}}: Body unification

{Name, ArgV}: Body built-in predicate

KlbName is a processing system instruction name atom (a KL1-B instruction). Since the KL1-B instructions are machine dependent (especially the programmer specified ones), tracer developers must be well aware of each type of processing system and the meanings of their instructions. Here we will mention the instruction types used by the PIM/m (and Multi-PSI) processors.

{execute, Code, ArgV}: User defined predicate within the same module [13]

{execute_external, Code, ArgV}: User defined predicate from an external module [14]

{enqueue, Code, ArgV}: User defined predicate within the same module

{enqueue_with_priority, Code, ArgV, {Prio, Min, Max}}:
Priority specified user defined predicate from within the same module. The structure {Prio, Min, Max} expresses the logical priority specified by the user. Prio is the logical priority [15] specified by the user, Min and Max are the minimum and maximum priorities of the shoen to which that goal belonged.

{enqueue_to_processor, Code, ArgV, Node}: Node specified user defined predicate within the same module. Node is the user specified node number.

{enqueue_external, Code, ArgV}: User defined predicate from an external module

{unify, {DataX, DataY}}: Body unification. DataX and DataY are the two data items that are the object of unification.

{Name, ArgV}: Body built-in predicate

spy: A spy exception was detected.

- {Node, SpyID, SubGoalInfo, Caller}

SpyID is an identifier for identifying goals, and has been specified as the forth argument to the built-in predicate apply_spying. SubGoalInfo is subgoal data with the same format as that for trace exception.

【Caution1】 However, the priority reported as the argument to the enqueue_with_priority instruction is not the 3 element vector {Prio, Min, Max} of the trace exception case, but merely Prio.

【Caution2】 For the spy exception, when code given as an argument to a high level predicate such as apply is the spy object predicate, the name of that high level predicate and its arguments are reported in SubGoalInfo [16]

raised: An explicit exception occurred.

- {Node, Info, Data}

An explicit exception was caused by a call to the predicate shoen:raise/3. Info and Data are the objects given in the arguments to the built-in predicate raise. See section 7.5.5 (page 314) for details on the methods for raising explicit exceptions.

[13] The first executed goal in the subgoal.

[14] The first executed goal in the subgoal.

[15] This is an undefined variable. Prio is instantiated by the built-in predicate calculate_priority_from_XXX.

[16] These high level predicates are apply, apply_tracing, apply_spying, create_shoen, and create_profiling_shoen.

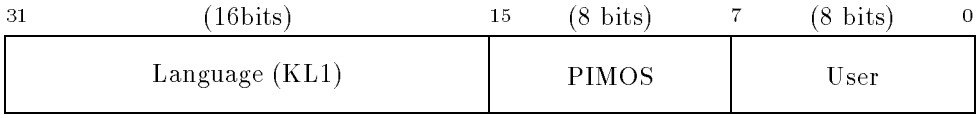

Figure 7.2: Exception Tag Bit Position Allocations

etc: An exception other than those listed above occurred.

- {ExpID, ExpInfo}

ExpID is an identifier that expresses the type of the exception, and ExpInfo is exception data that may include variables.

【Hint】 These exceptions are not generated in the current system.

7.5.5 Occurrence of exception

Exception events occur in built-in predicates and also can be generated intentionally using the following predicates.

shoen:raise(Tag, Info, Data)

Generates an exception. Tag is an exception tag. See section 7.5.6 (page 314) for exception tag.

If Info is a structure, or if its elements are structures, generation of an exception is suspended until all the elements have been completely instantiated. This restriction does not apply to Data.

【Hint1】 In usual cases, Data is not checked its contents during the exception management, and is directly used as the element of NewArgv in terms of the argument to the goal to be executed in place of "raise".

【Hint2】 Exceptions generated explicitly by shoen:raise/3 are reported as **raised** exceptions on the shoen report stream for the innermost shoen for which the logical AND of the exception tag Tag passed as an argument to raised and the master tag (that was given when the shoen was generated and which determines the exception processing) is 0. (See section 7.5.4 on page 313.)

7.5.6 Exception tag

Bit position of the exception tag is divided into three areas as shown in Figure 7.2, each being defined by Language, PIMOS, and the user. When user programs are communicated with each other using (raise), the exception tag's bit position must be in a range between 0 and 7.

Table 7.5 shows the bit position of the exception tag for the exception event of language definition.

【Hint】 No bit position of the exception tag is determined for "raised" because the exception tag is specified when raise is called.

【Caution】 The bit position of this exception tag is likely to be changed in the future, so that it is recommended to use the macro prepared by PIMOS for specifying the mask pattern of a

Table 7.5: Exception Tag Bit Positions

Exception type	Bit position	Meaning
illegal_input	16	Illegal input argument type
range_overflow	17	Illegal input argument range
out_of_bounds	18	Illegal index range
integer_overflow	19	Integer operation digit overflow
integer_zero_division	20	Division by zero
invalid_floating_point_operation	21	Floating point invalid operation
arity_mismatch	22	Wrong number of arguments
undefined_module	23	A call to a nonexistent module
undefined_predicate	23	A call to a nonexistent predicate
illegal_merger_input	24	Incorrect merger input
reduction_failure	25	Failure of all candidate clause
unification_failure	26	Failure of unification
deadlock	27	A deadlock detected
merger_deadlock	28	A merger deadlock detected
trace	29	Trace
spy	29	Spy
(reserved)	30	
etc	31	Other

shoen (or a task). This macro is converted to the mask pattern corresponding to each exception by coding "`exception_tag#`exception_event_name". To specify the mask pattern for multiple exceptions, each of them may be ORed by using the above macro notation. For these reasons, as long as the exception management side is checking the contents of Info alone, the exception management will not be suspended for reason of incomplete instantiation.

7.5.7 Exception processing utilities

7.5.7.1 Predicate data extraction

The following predicates allow the determination of which predicate caused an exception event from the Code and Caller in the Info structure returned when an exception event is reported.

module:get_predicate_info(CallerOrCode,^Package,^ModName,^Pred,^Arity,^Info)

When the Caller({Module, Offset}) data is given as the first argument, extracts information on the KL1 defined predicate that called the instruction (the built-in predicate that caused the exception) that is at the position Offset in the module Module. The data that is acquired is the package name atom (Package), the module name atom (ModName), the predicate name atom (Pred), and the arity (Arity).

Also, if the code (Code) is given as the first argument, it returns the package name for the module to which the code belongs, the module name, the predicate name, and the arity.

In either case, the data inserted by the compiler (although currently the integer 0 is always returned, it is possible that in future versions the corresponding position in the source file will be returned) is returned in the last argument, Info.

module:get_module_name(Module,^Package,^ModName)

Returns the package name atom (Package) and the module name atom (ModName) from the module (Module).

module:get_module_name(Module,^ModSpec)

Returns the module name with the package name attached in ModSpec from the module specified by Module. The format of ModSpec is {::, Package, ModName}.

7.5.7.2 Shoen profile data conversion utilities

exception_table:translate_profiling_data(Data,^NewData)

The data from the call profiling_information(Node, Type, Depth, Data) is converted to the following format.

- {Predicate, Dequeue_count, Suspend_count}
  User defined predicate, or library, or built-in predicate. The format of Predicate is the same as that as in the exception data.
- {Name, Dequeue_count, Suspend_count}
  Special instruction. Name is an instruction name atom.

Chapter 8

Utility

PIMOS has common utilities which are available for any application programs as well as PIMOS. These utilities provide the following functions:

- Comparing the values of arbitrary KL1 data
- Calculating a hash function on arbitrary KL1 data
- Database (pool) where arbitrary data is stored and extracted (pool)
- Filters for sorting and merging arbitrary data sets (sort/merge filter)
- Database where character string data is stored and extracted (infinite-length buffer)
- Filters for blocking and deblocking character string data (buffering filter)

Each of these utilities adopts one of two usage formats. You can use these utilities by calling the predicate of a module supplied by the PIMOS, or by sending a message to a stream obtained by calling such a predicate. This chapter describes all of these common utilities except the I/O utilities. See section 9.2, "I/O Utilities" (page 355), for details on I/O utilities.

【Caution】 Note that of the streams acquired by calling the predicates described in this chapter, an asterisk is attached to the right of variables that express streams that have a built-in predicate merger inserted in advance. (See section 7.4.19, "Stream support" on page 297.) For a detailed description of reading this display, see section 7.1, "Interpreting the I/O Mode Display" (page 231).

8.1 Comparison

Supplies a structure that can uniformly compare arbitrary KL1 data.

8.1.1 Standard comparison routines

comparator:sort(X, Y, ^S, ^L, ^Swapped)

In this predicate, first X and Y are compared. If these values are different, smaller one is returned in S and larger one in L. If these are the same, X is returned in S and Y is returned in L (this is called "stable"). Moreover, if X is larger than Y, yes is returned in Swapped, otherwise no. The definition of magnitude relationship will be presented later in this section.

comparator:descending_sort(X, Y, ^L, ^S, ^Swapped)

In this predicate, first X and Y are compared. If these values are different, the larger one is returned in L and the smaller one in S. Moreover, if these are the same, X is returned in L and Y is returned in S. If X is smaller than Y, yes is returned in Swapped, otherwise no. The definition of magnitude relationship will be presented later in this section.

8.1.2 Magnitude relationship on data in the standard comparison routines

The magnitude relationship between data determined by a call to the above predicates is defined as follows.

When the data types of the two arguments being compared are the same, the magnitude relationship is defined for each data type as follows.

Integer: Signed integer

Floating-point: Signed floating-point number

Atom: Atom number

Character string: For the strings of which the bit length of each element is different, the comparison is based on the bit length. For those of which the bit length of each element is the same, the comparison is based on the lexicographical order. [1]

List: Based on the the value of Car. If the Car values are the same, the comparison depends on the value of Cdr.

Vector: Based on the number of elements if the numbers of elements on both arguments are different. If they are the same, the first elements are compared; if the first elements are the same, then the second elements are compared, and so on.

Module: Based on the number of elements if the numbers of elements on both arguments are different. If they are the same, the first elements are compared; if the first elements are the same, then the second elements are compared, and so on.

Code: Based on the module to which the codes belong. If the modules are the same, the offset values are compared.

Others: Based on the data tags. If they are the same, the value parts of data are compared.

When the data types of the two arguments being compared are different, the magnitude relationships between each data type are defined in the following order.

integer < floating-point < atom < string < list < vector < module < code < others

However, the comparison of integer and floating point numbers is determined by a magnitude comparison between their actual numerical values. When the values being compared are the same, the integer type argument is assumed larger than the floating-point type.

【Caution】 The definition of the magnitude relationship between each data type may be changed in the future.

[1] Lexicographic order

8.1.3 Comparison routine specification

In many pools, element storage and extraction is performed based on an inter-element magnitude comparison. The routine used for comparison can also be specified in the program, and one of the following will be used in such cases.

ascending (atom): The standard ascending order comparison routine comparator: sort/5 (See section 8.1.1, "Standard comparison routines" on page 317.)

descending (atom): The standard descending order comparison routine comparator: descending_sort/5 (See section (8.1.1, "Standard comparison routines" on page 318.)

Code (code): A user defined comparison routine. However, Code must have the same interface as the standard comparison routine comparator:sort/5 (See section 8.1.1, "Standard comparison routines" on page 317.)

8.2 Hashing

PIMOS has a standard hash function:

hasher:hash(X, ^H, ^Y)

X is hashed with an appropriate hash function, and the resultant value is returned in H (a non-negative integer). At the same time, X is returned in Y as is.

【Caution】 This manual does not describe in detail the specification of the hash function because it will be changed in the future. But, it should be pointed out that the hash function can calculate the structure much deeper than the built-in predicate hash can do (See section 7.4.21, "Others" on page 299 in Built-in Predicate).

8.3 Pool

PIMOS has a mechanism (pool) that can generally store any KL1 data. Except those for search keys and data for comparison, even any undefined data can be stored in a pool.

There are two main types of pools: unkeyed pools which store data with the data itself as the search keys, and keyed pools which store data with keys. The following 4 types of unkeyed pools are provided.

Bag: The most basic pool. See section 8.3.1, "Bag", (page 320).

Stack: Identical to bag except that data addition and fetching is performed LIFO. See section 8.3.2, "Stack", (page 321).

Queue: Identical to bag except that data addition and fetching is performed FIFO. See section 8.3.3, "Queue", (page 322).

Sorted bag: Identical to bag except that data is fetched in the smallest first order. See section 8.3.4, "Sorted bag", (page 322).

There are two main types of pools that store data with an attached key and allow reference and fetching using that key. One type is the type in which exactly one key is stored for each datum, and the other type is the type in which multiple keys can be attached to a datum. The former are called "keyed pools", and the latter are called "multiply keyed pools". The following types of these pools are provided.

Keyed bag: The most basic keyed pool. See section 8.3.5, "Keyed bag", (page 323).

Keyed set: Identical to keyed bag except that overlapping keys are not allowed. See section 8.3.6, "Keyed set", (page 326).

Keyed sorted bag: Identical to keyed bag except that data is fetched in the smallest first order. See section 8.3.7, "Keyed sorted bag", (page 327).

Keyed sorted set: Identical to keyed bag except that overlapping keys are not allowed, and data is fetched in the smallest first order. See section 8.3.8, "Keyed sorted set", (page 328).

Multiply keyed bag: A multiply keyed pool. See section 8.4, "Multiply Keyed Bag", (page 330).

These pools are manipulated by activating a process by calling the predicate specified for each type, and then using message communication intermediated by streams to those processes. The remainder of this section describes the methods for generating each type of pool and the associated message protocols.

【Caution】 In the following message protocols notations, the symbols (e.g., ^ and ?) attached to the variables in the protocols follow the mode display method described in section 7.1. In general, except for those items used for key searches or comparisons, stored data is allowed to be in the undefined state in pools. Therefore, a ? will be used as the mode display for this type of data storage and fetching. For details, see section 7.1, "Interpreting the I/O Mode Display", (page 231).

8.3.1 Bag

Basic unkeyed pool which has only the basic functions such as input and output. You cannot fetch an element from this pool with leaving it in the pool: after you have got the element, it does not exist in the pool. A bag process is generated by calling the predicate described below and the bag is manipulated by sending messages to the process.

pool:bag(S*)

Creates a new object, and acquires a stream S to that data structure.

The messages that can be sent to the bag data type are as follows:

empty(^YorN)

Inquires whether a bag is empty or not. If the bag is empty, it sets YorN to yes; otherwise no.

put(X)

Adds an element X to a bag.

get(^X)

Fetches an element from a bag and returns the result in X. Which element in the bag will be gotten is uncertain. When you get an element, it disappears from the bag. If the bag is empty, this function fails.

get_if_any(^X)

Fetches an element from a bag, if any, and returns it to X in the { ¡value¿ } format. Which element in the bag will be gotten is uncertain. When you get an element, it disappears from the bag. If the bag is empty, it returns {} in X.

get_all(^O)

Fetches all the elements from a bag, and returns them in O. The format of O is a list of elements. The order of elements returned to the list is not determined in any particular manner. If the bag is empty, it returns `[]` in O.

carbon_copy(^O)

References all the elements in a bag, and returns them in O. The format of O is a list of elements. The order of elements returned to the list is not determined in any particular manner. If the bag is empty, it returns `[]` in O.

【Caution】 The MRB of elements referenced by executing this message is set to black. Users who wish to reference elements in white without modification can use the fetching (get/1, etc.) and storage (put/1) operations.

do(Stream*)

Processes consecutively the messages in Stream when a list containing messages has been passed to Stream.

【Hint】 This is useful when multiple processes are accessing a single pool and the user desires a single process to perform a series of manipulations consecutively.

8.3.2 Stack

Unkeyed pool which is basically the same as bag. However, stacks differ from bags in that the order in which elements are fetched is the reverse of the order in which they were added (LIFO). A stack process is generated by calling the predicate described below and the stack is manipulated by sending messages to the process.

pool:stack(S*)

Creates a new stack, and acquires a stream S to that data structure.

The messages that can be sent to the stack data type are as follows, and are identical to those for bag. However, when elements are fetched, they are fetched starting from the most recently added element. When all the elements in a stack are fetched or referenced, they are returned starting from the most recently added element.

empty(^YorN) (See page 320)

put(X) (See page 320)

get(^X) (See page 320)

get_if_any(^X) (See page 321)

get_all(^O) (See page 321)

carbon_copy(^O) (See page 321)

do(Stream*) (See page 321)

8.3.3 Queue

Unkeyed pool which is basically the same as bag. However, queues differ from bags in that the order in which elements are fetched is the order in which they were added (FIFO). A queue process is generated by calling the predicate described below and the queue is manipulated by sending messages to the process.

pool:queue(S*)

Creates a new queue, and acquires a stream S to that data structure.

The messages that can be sent to the queue data type are as follows, and are identical to those for bag. However, when elements are fetched, they are fetched starting from the element added first. When all the elements in a queue are fetched or referenced, they are returned starting from the element added first.

empty(^YorN) (See page 320)

put(X) (See page 320)

get(^X) (See page 320)

get_if_any(^X) (See page 321)

get_all(^O) (See page 321)

carbon_copy(^O) (See page 321)

do(Stream*) (See page 321)

8.3.4 Sorted bag

Unkeyed pool which is basically the same as bag. However, sorted bags differ from bags in that element fetching occurs in the smallest first order defined by the specified comparison routine. If the comparison routine is stable (the element order is not changed if the result of comparison is equal), elements which have the same value are fetched in the order as is.

【Caution】 It is possible that the search algorithm used by the sorted bag data type will be changed to an algorithm with a high degree of parallelism. It is possible that associated with that change the MRB of stored elements will be set to black.

A sorted bag process is generated by calling the predicates described below and the sorted bag is manipulated by sending messages to the process.

pool:sorted_bag(S*)

Creates a new sorted bag, and acquires a stream S to that data structure. The standard ascending order comparison routine comparator:sort/5 (See section 8.1.1, "Standard comparison routines" on page 317.) is used for element comparisons.

pool:sorted_bag(Comparator, S*)

Creates a new sorted bag, and acquires a stream S to that data structure. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comparator.

The messages that can be sent to the sorted bag data type are as follows. However, when elements are fetched, they are fetched starting from the smallest element of those. When all the elements in a sorted bag are fetched or referenced, they are returned starting from the smallest element.

empty(^YorN) (See page 320)

put(X) (See page 320)

get(^X) (See page 320)

get_if_any(^X) (See page 321)

get_all(^O) (See page 321)

carbon_copy(^O) (See page 321)

do(Stream*) (See page 321)

get_max_if_any(^X)

If there are any elements in the sorted bag, fetches the largest of those and returns it in X in the format { ¡value¿ }. If the comparison is stable and there are multiple largest elements, they are fetched in the order they were inserted. When an element is fetched, that element no longer appears in the sorted bag. If the sorted bag is empty, {} is returned in X.

get_min_if_any(^X)

If there are any elements in the sorted bag, fetches the smallest of those and returns it in X in the format { ¡value¿ }. If the comparison is stable and there are multiple smallest elements, they are fetched in the order they were inserted. When an element is fetched, that element no longer appears in the sorted bag. If the sorted bag is empty, {} is returned in X.

8.3.5 Keyed bag

Basic keyed pool which provides a mechanism capable of storing any KL1 data with an attached key and retrieving them by key. It is implemented by a hash table. Though the initial size of the hash table is a default value or determined by the argument specified when the table is generated, it is automatically increased when the number of elements are increased too much for the initial size. A keyed bag process is generated by calling the predicate described below and the keyed bag is manipulated by sending messages to the process.

pool:keyed_bag(S*)

Creates a new keyed bag, and acquires a stream S to that data structure. The standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319) is used for element hashing. The initial value for the size of the hash table is 1.

pool:keyed_bag(S*, Size)

Creates a new keyed bag, and acquires a stream S to that data structure. The standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319) is used for element hashing. The value of Size (a non-negative integer) is used as the initial value for the size of the hash table.

pool:keyed_bag(Hasher, S*, Size)

Creates a new keyed bag, and acquires a stream S to that data structure. The standard hashing function Hasher (code) is used for element hashing. However, Hasher must have the same interface as the standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319). The value of Size (a non-negative integer) is used as the initial value for the size of the hash table.

The messages that can be sent to the keyed bag data type are as follows.

empty(^YorN)

Inquires whether a keyed bag is empty or not. If empty, returns yes in YorN, otherwise no.

empty(Key, ^YorN)

Inquires whether or not a keyed bag contains an element of which the key corresponds to a specified one (Key). If there is no corresponding element, returns yes in YorN, otherwise no.

put(Key, ?X)

Adds an element of which the key is Key and the value is X to a keyed bag.

get(Key, ?X)

Fetches an element of which the key corresponds to Key (specified one), and returns it in X. If there are two or more keys corresponding to the specified one, it is undefined which key is fetched. The fetched key disappears from the keyed bag. If there is no corresponding key, this function fails.

get_if_any(Key, ^X)

Fetches any element of which the key corresponds to Key (specified one) from a keyed bag, and returns it in X in a format of `{} ?<value>}`. If there are two or more keys corresponding to the specified one, it is undefined which key is fetched. The fetched key disappears from the keyed bag. If there is no corresponding key, it returns {} in X.

get_all(^O)

Fetches all the elements from a keyed bag, and returns them in O. The format of O is a list of {¡key¿, ?¡value¿}. If the keyed bag is empty, returns `[]` in O.

get_all(Key, ^O)

Fetches all the elements of which the key corresponds to Key from a keyed bag, and returns them in O. The format of O is a list of values. If the keyed bag is empty, returns `[]` in O.

carbon_copy(^O)

References all the elements in a keyed bag, and returns them in O. The format of O is a list of {¡key¿, ?¡value¿}. If the keyed bag is empty, returns [] in O.

【Caution】 The MRB of the key and value referenced by the execution of this message turn black. Users who wish to reference elements in white as they are can use the two operations of fetching and storage (get/2, put/2, etc.).

carbon_copy(Key, ^O)

References all the elements of which the key corresponds to Key in a keyed bag, and returns them in O. The format of O is a list of values. If the keyed bag is empty, returns [] in O.

【Caution】 The MRB of the value referenced by the execution of this message turns black. Users who wish to reference elements in white as they are can use the two operations of fetching and storage (get/2, put/2, etc.).

get_and_put(Key, ?X, ?Y)

Fetches an element of which the key corresponds to Key from a keyed bag, and returns it in X, and stores a new element Y to the same location. If there is no corresponding element, the function fails.

get_if_any_and_put(Key, ^X, ?Y)

Fetches an element of which the key corresponds to Key from a keyed bag, if any, and returns it in X in a { ?¡value¿} format, then stores a new element Y in the same location. If there is no corresponding element, returns {} in X and adds Y.

get_and_put_if_any(Key, ^X, ?Y, ^Y1)

Fetches an element of which the key corresponds to Key from a keyed bag, if any, and returns it in X in a { ?¡value¿ } format. Then stores a new element Y in the same location, and returns {} in Y1.

If there is no corresponding element, returns {} in X, and returns { ?Y } in Y1, though nothing is performed for the keyed bag.

get_load_factor(^A)

Returns the hash table load factor A at the point the message is received. The load factor indicates the ratio of the number of entries in use to the total size of the hash table. It is expressed as a floating point number between 0.0 and 1.0.

【Caution】 Currently hash tables are implemented so that the size is automatically doubled at the point the load factor exceeds 0.75.

get_average_chaining_length(^L)

Returns the average chain length L of element chains in the hash table at the time the message is received. Hash table collision processing is implemented by separate chaining. This results in elements with the same hash value (i.e., words of the same class) being connected in the same chain. The average chain length indicates the average length of chains (of length 1 or more)

existing in the hash table. It is expressed as a floating point number greater than or equal to 1.0. However, when the hash table is empty, a number of 0.0 is returned for L.

【Hint】 This value can be used as an estimate of the extent of collisions in the hash table. For example, if the value of the average chain length is large even though the value of the load factor is small, the possibility of many collisions occurring in the table is high. We recommend changing the key or redefining the hash function in such a case.

do(Stream*)

Processes consecutively messages in Stream when a list of messages has been passed to Stream.

【Hint】 This is useful when multiple processes are accessing a single pool and the user desires a single process to perform a series of manipulations consecutively.

8.3.6 Keyed set

Keyed pool which is basically the same as keyed bag except that duplicate keys are not allowed. It is implemented by a hash table. The initial size of the hash table is default value or determined by the argument specified when the table is generated, and it is automatically increased when the number of elements are so increased that the hash table can contain all the elements. A keyed set process is generated by calling the predicate described below and the keyed set is manipulated by sending messages to the process.

pool:keyed_set(S*)

Creates a new keyed set, and acquires a stream S to that data structure. The standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319) is used for element hashing. The initial value for the size of the hash table is 1.

pool:keyed_set(S*, Size)

Creates a new keyed set, and acquires a stream S to that data structure. The standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319) is used for element hashing. The value of Size (a non-negative integer) is used as the initial value for the size of the hash table.

pool:keyed_set(Hasher, S*, Size)

Creates a new keyed set, and acquires a stream S to that data structure. The standard hashing function Hasher (code) is used for element hashing. However, Hasher must have the same interface as the standard hashing function hasher:hash/3 (see section 8.2, "Hashing" on page 319). The value of Size (a non-negative integer) is used as the initial value for the size of the hash table.

The messages that can be sent to the keyed set data type are as follows.

empty(^YorN) (See page 324)

empty(Key, ^YorN) (See page 324)

put(Key, ?X, ^OldX)

Adds an element of which the key is Key and the value is X to a keyed set. If there already exists an element of which the key is Key, updates it. The old element is returned in OldX in the format {?<value>} as an element vector. If there is no elements of which the key is Key, an empty vector {} is returned in OldX.

get(Key, ?X) (See page 324)

get_if_any(Key, ?X) (See page 324)

get_all(^O) (See page 324)

get_all(Key, ^O) (See page 324)

carbon_copy(^O) (See page 325)

carbon_copy(Key, ^O) (See page 325)

get_and_put(Key, ?X, ?Y) (See page 325)

get_if_any_and_put(Key, ^X, ?Y) (See page 325)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (See page 325)

get_load_factor(^A) (See page 325)

get_average_chaining_length(^L) (See page 325)

8.3.7 Keyed sorted bag

Key pool which is basically the same as keyed bag. However, keyed sorted bags differ from keyed bags in that element fetching occurs in the smallest first order defined by the specified comparison routine.

【Caution】 It is possible that the search algorithm used by the keyed sorted bag data type will be changed to an algorithm with a high degree of parallelism. It is possible that associated with that change the MRB of stored elements will be set to black. A keyed sorted bag process is generated by calling the predicates described below and the keyed sorted bag is manipulated by sending messages to the process.

pool:keyed_sorted_bag(S*)

Creates a new keyed sorted bag, and acquires a stream S to that data structure. The standard ascending order comparison routine comparator:sort/5 (See section 8.1.1, "Standard comparison routines" on page 317.) is used for element comparisons.

pool:keyed_sorted_bag(Comparator, S*)

Creates a new keyed sorted bag, and acquires a stream S to that data structure. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comparator.

The messages that can be sent to the keyed sorted bag data type are as follows. However, when elements are fetched, they are fetched starting from the element with the smallest key. When the comparison is stable and there are multiple elements with the same key, elements are fetched in order starting with the earliest added. When all the elements in a keyed sorted bag are fetched or referenced, they are returned starting from the smallest element.

empty(^YorN) (See page 324)

empty(Key, ^YorN) (See page 324)

put(Key, ?X) (See page 324)

get(Key, ?X) (See page 324)

get_if_any(Key, ?X) (See page 324)

get_all(^O) (See page 324)

get_all(Key, ^O) (See page 324)

carbon_copy(^O) (See page 325)

carbon_copy(Key, ^O) (See page 325)

get_and_put(Key, ?X, ?Y) (See page 325)

get_if_any_and_put(Key, ^X, ?Y) (See page 325)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (See page 325)

get_max_if_any(^Element)

If there are any elements in the keyed sorted bag, fetches the element with the largest key and returns it in Element in the format { ¡key¿, ?¡value¿ }. If the comparison is stable and there are multiple elements with the largest key, they are fetched in the order they were inserted. When an element is fetched, that element no longer appears in the keyed sorted bag. If the keyed sorted bag is empty, {} is returned in Element.

get_min_if_any(^Element)

If there are any elements in the keyed sorted bag, fetches the element with the smallest key and returns it in Element in the format { ¡key¿, ?¡value¿ }. If the comparison is stable and there are multiple elements with the smallest key, they are fetched in the order they were inserted. When an element is fetched, that element no longer appears in the keyed sorted bag. If the keyed sorted bag is empty, {} is returned in Element.

8.3.8 Keyed sorted set

Keyed pool which is basically the same as keyed sorted bag except that duplicate keys are not allowed. When an element with the same key is added, the original element with that key is automatically deleted.

【Caution】 It is possible that the search algorithm used by the keyed sorted set data type will be changed to an algorithm with a high degree of parallelism. It is possible that associated with that change the MRB of stored elements will be set to black. A keyed sorted set process is generated by calling the predicates described below and the keyed sorted set is manipulated by sending messages to the process.

pool:keyed_sorted_set(S*)

Creates a new keyed sorted set, and acquires a stream S to that data structure. The standard ascending order comparison routine comparator:sort/5 (See section 8.1.1, “Standard comparison routines” on page 317.) is used for element comparisons.

pool:keyed_sorted_set(Comparator, S*)

Creates a new keyed sorted set, and acquires a stream S to that data structure. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comparator.

The messages that can be sent to the keyed sorted set data type are as follows. However, when elements are fetched, they are fetched starting from the element with the smallest key. When all the elements in a keyed sorted set are fetched or referenced, they are returned starting from the smallest element.

empty(^YorN) (See page 324)

empty(Key, ^YorN) (See page 324)

put(Key, ?X, ^OldX) (See page 327)

get(Key, ?X) (See page 324)

get_if_any(Key, ?X) (See page 324)

get_all(^O) (See page 324)

get_all(Key, ^O) (See page 324)

carbon_copy(^O) (See page 325)

carbon_copy(Key, ^O) (See page 325)

get_and_put(Key, ?X, ?Y) (See page 325)

get_if_any_and_put(Key, ^X, ?Y) (See page 325)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (See page 325)

get_max_if_any(^Element) (See page 328)

get_min_if_any(^Element) (See page 328)

8.3.9 Error messages displayed by unkeyed and single keyed pools

Unkeyed and single keyed pools report errors when, for example, they receive undefined messages or an attempt is made to fetch a nonexistent element. This section describes the format and content of these reports. See section 2.1.5.2, (shell) "Error messages", (pages 21 and 22) for details on the display format when errors are reported in the shell, and section 4.1.8.3, (listener) "Error messages", (page 162) for details on the display format when errors are reported in the listener.

Reception of an Undefined Message by a Pool:

Error ID: Name of the pool being used

Error Info: illegal_message_protocol

Error Data: Message that caused the error

【Example1】 Reporting example when an error was detected during execution in the listener

```
[1] pool:keyed_bag([unknown(X)|In]).
  Error>> keyed_bag
   Info > illegal_message_protocol
   Data > unknown(X)
   Node > 0
 !! Exception (h for help):
```

【Example2】Reporting example when an error was detected during execution in the shell

```
Shell>> del:go
 Error reported !!
  Job-number : 2
  Node-number: 0
  Error ID   : keyed_bag
  Error Info : illegal_message_protocol
  Error Data : unknown(A)
 Job 2 was aborted .
Shell>>
```

Attempt to Fetch a Nonexistent Element from a Pool

Error ID: Name of the pool being used

Error Info: tried_to_get_nonexistent_element

Error Data: Message that caused the error

【Example】Reporting example when an error was detected during execution in the listener

```
[1] pool:keyed_bag([get(key,X)]).
  Error>> keyed_bag
   Info > tried_to_get_nonexistent_element
   Data > get(key,X)
   Node > 0
 !! Exception (h for help):
```

8.4 Multiply Keyed Pools

The multiply keyed pools provide a mechanism for storing arbitrary KL1 data with multiples keys, and searching based on those keys as indices. These pools are called the "multiply keyed bag [2] data type.

8.4.1 Overview

Multiply keyed bag creation: A process is activated by calling the specified predicate, and the data is manipulated by message passing. Elements are all stored as vectors of the form {attribute(0), attribute(1), $\cdots$, attribute(n-1)}. Arbitrary KL1 data can be assigned for each attribute. Except for the attribute(s) that are used as indices, data in the undefined state can be used. The user specifies the format and what kind of indices are attached to which attributes at creation time.

[2] Multiply is the adverbial form of the adjective multiple.

Functions: The functions provided by this data type include operations on the whole container with the same specifications as those for the more primitive data types, such as fetching all elements, as well as operations such as fetching elements according to user-specified conditions. These conditions are written using and, or, and not to specify what key under which indices. When fetching or referencing elements, it is possible to specify the required attributes in an element and only acquire those attributes. When fetching the element whose attribute is the smallest or largest of the stored elements, after specifying the attribute and requesting the largest or smallest value, fetch or reference of the element itself is then performed using that attribute value. Range search specification identical to that of more primitive pools is not possible. Also, we have not yet considered executing these operations on multiple nodes.

Indices: Each index is a process. Two types of index can be produced: hash table and tree structured indices. The element itself is not stored in an index, but rather what is stored is an unique identifier assigned to the element. Elements are stored indexed by this identifier in a data file managed by the main process. Of the elements, the MRB of attributes specified to be indices are set to black.

Data file: The data file is implemented as a vector. The element identifier expresses the position of the element in the vector. Therefore, a vector of size n can store a total of n elements. When the vector becomes full, the system automatically stores all the elements in a vector twice the size and creates a new index.

Element removal: When an element is deleted, the element itself is discarded and the place it was stored is temporarily set to 0. However, this empty area cannot be reused as is it is. This is because it is possible that it could be referenced by another index. Index updating is not performed at element deletion.

Packing: A manipulation called packing is performed to reclaim the empty space due to element removal. The packing operation consists of collapsing the empty space in the data file and remaking the index. By default, packing is performed when the empty space due to element deletion occupies half of the size of the data file. It is possible to perform the packing operation explicitly by sending a message. The packing ratio (the timing for automatic packing, i.e., the ratio x of the data file occupied by empty regions due to element removal) can be specified at pool creation time.

Element exchange: The element exchange operation is important since it can be used to avoid excessive creation of empty regions due to element removal. When the two operations get and put are used a single empty region is created. Also, all indices are recreated in the packing operation. Element update can be performed with minimum index updating and without creating empty regions by using the get_and_put manipulation.

Indices that take element subsets as their object: Although it may be desirable to fetch all elements that meet a certain condition, and to later perform conditional searches limited to those elements, this functionality is not supported in the current implementation.

8.4.2 Usage

A process is created by calling the following predicate, and it is manipulated by sending messages.

pool:multiply_keyed_bag(DataFormat, S*)

Creates a new multiply keyed bag and acquires a stream S to that object. DataFormat is a vector that expresses the element storage format. The initial value of the data file size is 1, and

of the packing ratio, 0.5.

pool:multiply_keyed_bag(DataFormat, Init, S*)

Creates a new multiply keyed bag and acquires a stream S to that object. DataFormat is a vector that expresses the element storage format. The initial value data for the data file is specified in Init.

Specification of the element storage format DataFormat **DataFormat** is a vector that expresses the element storage format. Each elements in this vector corresponds to each attribute. Each element must have one of the following formats.

{<attribute name(atom)>, <index specification parameter(atom)>}
{<attribute name(atom)>, <index specification parameter(atom)>,
<index initialization data(list)>}
<attribute name(atom)>

The <attribute name(atom)¿ is the name given to that attribute, and is used when specifying that index. All attributes must be specified. Note that the same name must not be used to specify more than one attribute.

The <index specification parameter(atom)¿ specifies one of ordered, hashed, or none. The following index types are provided for each of these parameter values.

ordered: An index implemented as a tree structure (for binary search)

hashed: An index implemented as a hash table.

none: No index is generated.

Normally, there is no need to specify the above parameter when an index is not created. A linear search is used in conditional searches over attributes with no index. The parameter "none" can be used by users who desire to specify explicitly the comparison routine for linear searches.

The <index initialization data¿ is specified as a list. The following can be used as the elements of this list. The default value is used when a parameter is omitted from this list. If a parameter unrelated to the index type specified with **DataFormat** is specified or if an illegal parameter is specified the pool creation will fail.

Parameters that can be set for binary search indices **comparator(Comp):** Specifies Comp as the comparison routine. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comp.

Parameters that can be set for hash search indices

index_size(Size): Sets the initial value for the hash table size to Size (a positive integer).

hasher(Hasher): Specifies the hash function to be Hasher (code).

comparator(Comp): Specifies Comp as the comparison routine. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comp.

Parameters that can be set when no index is generated:

comparator(Comp): Specifies Comp as the comparison routine. See section 8.1.3, "Comparison routine specification", (page 319) for details on specifying Comp.

Default values

Comparison routine: The standard ascending order comparison routine supplied by the PIMOS utilities. (See section 8.1.1, "Standard comparison routines" on page 317.)

Hashing function: The standard hashing function supplied by the PIMOS utilities. (See section 8.2, "Hashing" on page 319.)

Initial value for the hash table size: The data file size initial value.

【Example】 The following example is for the case where elements of the form {<name>, <age>, <address>} are stored. The name entry is indexed with a hash table and the age entry is indexed by searching a binary tree. The initial size of the hash table is set to 64. No index is created for the address entry.

```
DataFormat = {name(hashed,IdxInit), age(ordered), address},
IdxInit = [index_size(64)].
```

Specification of the data file initialization data Init: The data file initialization data is specified as a list or as a positive integer. If specified as a list, the following can be included as elements. When any of these are omitted, the default values are used. If an illegal parameter is specified the pool creation will fail.

size(Size): Sets the initial value for the data file size to Size (a positive integer).

packing_ration(P): Sets the packing ration to P (a real number such that 0<P<1).

If the initialization data is given as a positive integer, that value will be used as the initial value for the data file size.

8.4.2.1 Element search

Specifying the search conditions

The conditions are specified in the following format.

and(<condition1>, <condition2>) Sets elements that meet both <condition1> and <condition2> to be the object of search.
Can also be written as: **<condition1><condition2>**

or(<condition1>, <condition2>) Sets elements that meet either <condition1> or <condition2> to be the object of search.
Can also be written as: **<condition1><condition2>**

not(<condition>) Sets elements that do not meet <condition> as the object of search.
Can also be written as: **setminus <condition>**

equal(<attribute name>, <key>) Sets elements whose attribute <attribute name> is <key> to be the object of search.
Can also be written as: **<attribute name>=<key>**

not_equal(<attribute name>, <key>) Sets elements whose attribute <attribute name> is not <key> to be the object of search.
Can also be written as: **<attribute name> = <key>**

Element search by multiple conditions

Conditional search on multiple attributes is implemented by performing the respective searches in parallel with all elements as the object, and after candidate elements which fulfill the individual conditions have been acquired, the final result is returned by set operations.

【Hint】 In the future, it is possible that searches that involve the and of a conditional search for an unindexed attribute and a conditional search for an indexed attribute will be implemented by performing a linear search over candidates acquired as the result of the indexed attribute search. [3]

8.4.2.2 Specification of partial fetch and partial reference for elements

When fetching or referencing elements, it is possible to return not the whole element but only the desired attributes. Although this results in a slight loss of memory efficiency, it allows unnecessary blackening of attribute MRBs to be avoided. [4]

The required attributes in partial element fetch/reference are specified according to the following format. The format is a vector of $m(0¡m)$ elements. An attribute name is entered in each element in the vector. The attribute name can be an atom or the storage position, and the order is arbitrary. The same element may not be included more than once.

【Caution】 Correct operation is not guaranteed for the second and following fetches of a given attribute. Also, errors are not detected for this situation.

8.4.2.3 Attribute specification

While the attribute names (atoms) specified at pool creation are normally used for attribute specification, when the name is used, processing to derive the association between the attribute position in the element and the attribute name (atom) is required. For users particularly concerned with efficiency KL1 supports direct specification of attribute by the position (as a positive integer) of the attribute in the element. While this lowers the level of the user interface, it somewhat reduces the amount of processing required at execution time.

ˆL

8.4.2.4 Message protocol

The messages that can be sent to a multiply keyed bag are listed below. They are roughly divided into two groups. The basic messages are described in the section "Basic messages" (page 334), and the messages concerned with partial fetch/partial reference are described in the section "Specification of partial fetch/partial reference for elements" (page ??).

Basic messages

empty(ˆYorN)

Queries whether the container is empty or not. If the container is empty, it sets YorN to yes; otherwise no.

[3] In conditional search there are both cases where performing the search in parallel for each condition is more efficient, as well as cases where searching sequentially while restricting the space is more efficient. While this will depend on the size of the index and the severity of the conditions, cases where linear search is performed without an index most likely fall in the latter category.

[4] The MRB for an attribute specified as an index is black from the point it is created.

empty(C, ^YorN)

Queries whether an element that fulfills the condition C exists in the container. If there is no elements that fulfills the condition, it sets YorN to yes; otherwise no.

put(X)

Adds an element X to the container. Fails if an attempt is made to add an element of an illegal format.

get(C, ^X)

Fetches one element that fulfills the condition C from the container and returns it in X. If there are multiple elements that fulfill the condition, the element returned depends on the index used. When fetched that element is removed from the container. Fails if there is no element that fulfills the condition.

get_if_any(C, ^X)

Fetches one element that fulfills the condition C from the container and returns it in X. If there are multiple elements that fulfill the condition, the element returned depends on the index used. When fetched that element is removed from the container. If there is no element that fulfills the condition, this function returns {} in X.

max_if_any(Attr_name, ^Key)

Selects the element with the largest key with attribute name Attr_name from the elements in the container, and returns its key in Key with the format { <key> }. If the container is empty, it returns {} in Key.

【Caution】 If there is no secondary search index for the specified attribute, a linear search of all elements will be performed. If no comparison routine was particularly specified, the default one is used.

min_if_any(Attr_name, ^Key)

Selects the element with the smallest key with attribute name Attr_name from the elements in the container, and returns its key in Key with the format { <key>}. If the container is empty, it returns {} in Key.

【Caution】 If there is no secondary search index for the specified attribute, a linear search of all elements will be performed. If no comparison routine was particularly specified, the default one is used.

get_all(^A)

Fetches all the elements from the container, and returns them in A. The format of A is a list of elements. If the container is empty, it returns `[]` in A.

get_all(C, ^A)

Fetches all the elements which fulfill the condition C from the container and returns them in A. The format of A is a list of elements. If there is no element which meets the condition, it returns [] in A. In addition to the general condition notation, an attribute name (atom) can also be used as the condition C. In this case this means to fetch all elements in the index order provided for the specified attribute. (If there is no index specified, then elements are fetched in sorted order on the specified attribute. If a special comparison routine has been specified, that routine is used, otherwise the standard PIMOS supplied routine is used.)

carbon_copy(^A)

References all the elements in the container and returns them in A. The format of A is a list of elements. If the container is empty, returns [] in A.

【Caution】 The MRB of the element referenced by the execution of this message turns black.

carbon_copy(C, ^A)

References all the elements which fulfill the condition C in the container and returns them in A. The format of A is a list of elements. If the container is empty, it returns [] in A. In addition to the general condition notation, an attribute name (atom) can also be used as the condition C. In this case this means to reference all elements in the index order provided for the specified attribute. (If there is no index specified, then elements are fetched in sorted order on the specified attribute. If a special comparison routine has been specified, that routine is used, otherwise the standard PIMOS supplied routine is used.)

【Caution】 The MRB of the element referenced by the execution of this message turn black.

get_and_put(C, ^X, X1)

Fetches an element which fulfills the condition C and returns it in X. Then stores a new element X1 to the same location. If there is no element that fulfills the condition, the function fails. Index update is performed as necessary when replacing elements using this message.

get_if_any_and_put(C, ^X, X1)

Fetches an element which fulfills the condition C and returns it in X. Then stores a new element X1 to the same location. If there is no element that fulfills the condition, returns {} in X and stores a new element X1. Index update is performed as necessary when replacing elements using this message.

get_and_put_if_any(C, ^X, X1, ^X2)

Fetches an element which fulfills the condition C and returns it in X. Then stores a new element X1 to the same location, and returns {} in X2. If there is no element that fulfills the condition, returns {} in X, and returns X1 in X2, though nothing is performed for the container. Index update is performed as necessary when replacing elements using this message.

do(Stream*)

Processes consecutively messages in Stream when a list of messages has been passed to Stream.

【Hint】 This is useful when multiple processes are accessing a single pool and the user desires a single process to perform a series of manipulations consecutively.

pack

Rebuilds the index to close up empty areas created by element removal.

Messages related to partial fetch and partial reference for elements

getp(C, P, ^X)

Fetches from the container one element that meets the condition C and returns the contents of the attribute specified by P from that element in X. If there are multiple elements that fulfill the condition, the element returned depends on the index used. When fetched that element is removed from the container. Fails if there is no element that fulfills the condition.

getp_if_any(C, P, ^X)

Fetches from the container one element that meets the condition C and returns the contents of the attribute specified by P from that element in X. If there are multiple elements that fulfill the condition, the element returned depends on the index used. When fetched that element is removed from the container. If there is no element that fulfills the condition, it returns {} in X.

getp_all(P, ^A)

Fetches all the elements from the container, and returns the contents of the attribute of each element specified by P in A. The format of A is a list of them. If the container is empty, it returns [] to A.

getp_all(C, P, ^A)

Fetches all the elements which fulfill the condition C from the container, and returns the contents of the attribute of each element specified by P in A. The format of A is a list of them. If there is no element which meets the condition, it returns [] in A. In addition to the general condition notation, an attribute name (atom) can also be used as the condition C. In this case this means to fetch all elements in the index order provided for the specified attribute. (If there is no index specified, then elements are fetched in sorted order on the specified attribute. If a special comparison routine has been specified, that routine is used, otherwise the standard PIMOS supplied routine is used.)

carbon_copyp(P, ^A)

References all the elements in the container and returns the contents of the attributes of each element specified by P in A. The format of A is a list of them. If the container is empty, returns [] in A.

【Caution】 The MRB of the attribute referenced by the execution of this message turns black.

carbon_copyp(C, P, ^A)

References all the elements which fulfill the condition C in the container and returns the contents of the attributes specified by P in A. The format of A is a list of elements. If there is no element which meets the condition, it returns `[]` in A. In addition to the general condition notation, an attribute name (atom) can also be used as the condition C. In this case this means to reference all elements in the index order provided for the specified attribute. (If there is no index specified, then elements are fetched in sorted order on the specified attribute. If a special comparison routine has been specified, that routine is used, otherwise the standard PIMOS supplied routine is used.)

【Caution】 The MRB of the attribute referenced by the execution of this message turns black.

getp_and_put(C, P, ^X, X1)

Selects from the container one element that fulfills condition C and returns in X the contents of the attribute specified by P from that element. Furthermore, it replaces that attribute with the contents specified by X1. Fails if there is no element that meets the condition or if a value with an illegal format is specified in X1. Index update is performed as necessary when replacing elements using this message.

getp_and_put_if_any(C, P, ^X, X1, ^X2)

Selects from the container one element that fulfills condition C, if any, and returns the contents of the attribute specified by P in X. Furthermore, it replaces that attribute with the contents specified by X1, and returns {} in X2. If there is no element which meets the condition, returns {} in X, and returns X1 in X2, though nothing is performed for the container.

Index update is performed as necessary when replacing elements using this message.

^L

8.4.3 Examples

【Example1】 Creation of a multiply keyed bag for fish.

The elements in this example have the format {<English name>, <Japanese name>, <length>, <eye>, <ventral fin>, <pectoral fin>, <adipose fin>, <dorsal fin>, <salt/fresh water>}, a hash table index is produced for both the English and Japanese names, and a binary search index for the length.

```
initialize(In) :- true |
        pool:multiply_keyed_bag(Fmt,128,In),
        Fmt={name(hashed),jname(hashed),length(ordered),
                eye,ventral_fin,pectoral_fin,abura_fin,dorsal_fin,habitat}.
```

【Example2】 Elements Stored[5]

[5] The attributes have the following meanings.

Eye **0:** On the left and right sides of the head, **1:** On one side of the head only.

Ventral fin: **0:** none, **1:** To the rear of the pectoral fins, {**2**,**M**,**N**}: Below and in essentially the same position as the pectoral fins, and has M spines and N soft lines, **3:** In front of the pectoral fins, **4:** Modified to be a sucker.

Pectoral fin: **0:** At the lower part of the sides of the body, **1:** At the center or upper part of the sides of the body.

```
put_data(In,In1) :- true |
        In=[put({ell,        unagi,      60, 0,0,1,0,1,f}),
            put({cutlass_fish,tachiuo,  150, 0,0,1,0,1,s}),
            put({rubripes,   torafugu,   70, 0,0,1,0,1,s}),
            put({filefish,   kawahagi,   30, 0,0,1,0,1,s}),
                        ......
            put({sardine,    maiwashi,   36, 0,1,0,0,1,s}),
            put({herring,    nishin,     30, 0,1,0,0,1,s}),
            put({salmon,     sake,       70, 0,1,0,1,1,s}),
            put({rainbow_trout,nijimasu, 40, 0,1,0,1,1,f}),
            put({hemiramph,  sayori,     40, 0,1,1,0,1,s}),
            put({rabbitfish, aigo,       40, 0,{2,2,3},0,0,1,s}),
            put({red_sea_bream,madai,    60, 0,{2,1,5},0,0,1,s}),
            put({sea_bass,   suzuki,     90, 0,{2,1,5},0,0,2,s}),
            put({oceanic_bonito,katsuo, 100, 0,{2,1,5},1,0,2,s}),
                        ......
            put({flatfish,   hirame,     70, 1,3,1,0,1,s})|In1].
```

【Example3】 Fetching the data for barracuda

```
        In=[get(name=barracuda,X)|In1].
```

`X=barracuda(akakamasu,50,0,1,1,0,2,s)` is returned as the result.

This query can also be written as:

```
        In=[get(0=barracuda,X)|In1].
```

【Example4】 This query fetches one salt water fish whose eyes are on one side of the head only, and acquires only the English name, the Japanese name, and the length.

```
        In=[getp(C,P,X)|In1],
        C=(eye=1)/\(habitat=s),
        P={name,jname,length}.
```

`X=right_eyed_flounder(magarei,55)` is returned as the result.

【Example5】 This query references the English name and the length of fishes that have eyes on the left and right sides of the head, have ventral fins to the rear of the pectoral fins, and do not have adipose fins.

```
        In=[carbon_copyp(C,P,A)|In1],
        C=(eye=0)/\(ventral_fin=1)/\(abura_fin=0),
        P={name,length}.
```

`A=[sardine(36),herring(30),carp(60),loach(18),hemiramph(40), ... ]` is returned as the result.

【Example6】 This query fetches fish with multiple dorsal fins and returns their English names and the number of dorsal fins.

```
        In=[getp_all(C,P,A)|In1],
        C=(dorsal_fin\=0)/\(dorsal_fin\=1),
        P={name,dorsal_fin}.
```

Adipose fin: **0:** None, **1:** Has adipose fins.

Dorsal fin: The number of dorsal fins.

Salt/fresh water: **s:** Salt water, **f:** Fresh water

A=[barracuda(2),gray_mullet(2),sea_bass(2),horse_
mackerel(2),yellowtail(2),codfish(3),goby(2), ...] is returned as the result.

If it is known that the number of dorsal fins will always be either 2 or 3, the condition can be written as follows:

```
C=(dorsal_fin=2)\/(dorsal_fin=3)
```

【Example7】 This query fetches fish that have ventral fins at about the same position as the pectoral fins, have one spine, and have their pectoral fins at the lower part of the sides of the body, and returns the English name, the ventral fin data, and the number of dorsal fins.

```
In=[getp_all(C,P,A)|In1],
C=((ventral_fin={2,1,10})\/(ventral_fin={2,1,5}))/\(pectoral_fin=0),
P={name,ventral_fin,dorsal_fin}.
```

A=[alfonsino({2,1,10},1),red_sea_bream({2,1,5},1),sea_bass({2,1,5},2),
horse_mackerel({2,1,5},2),yellowtail({2,1,5},2), ...] is returned as the result.

【Example8】 This query fetches the longest fish.

```
In=[max_if_any(length,{Key}),get(length=Key,X)|In1].
```

X=bluefin_tuna(maguro,200,0,2,1,5,1,0,2,s) is returned as the result.

【Example9】 This query fetches all the data sorted by length.

```
In=[get_all(length,A)|In1].
```

A=[ayu(ayu,15,0,1,0,1,1,f),loach(dojo,18,0,1,0,0,1,f), ...] is returned as the result.

【Example10】 This example corrects the data for char (which are actually fresh water fish).

```
In=[getp_and_put(jname=iwana,{habitat},OldAttr,NewAttr),
        carbon_copy(jname=iwana,A)|In1],
NewAttr={f}.
```

OldAttr={s}, A=[char(iwana,25,0,1,0,1,1,f)] is returned as the result.

8.4.4 Error messages displayed by multiply keyed bags

Multiply keyed bags report errors when, for example, they receive undefined messages or an attempt is made to fetch a nonexistent element. This section describes the error messages displayed by multiply keyed bags and presents a few examples. See section 2.1.5.2, (shell) "Error messages", (pages 21 and 22) for details on the display format when errors are reported in the shell, and section 4.1.8.3, (listener) "Error messages", (page 162) for details on the display format when errors are reported in the listener.

【Hint】 Except for a very few, most of these error messages output data stored at the time the error was detected, and any data not required for error message output is not stored. While this is done to avoid loss of efficiency, it is possible that it can hamper program debugging. Therefore, we are considering supplying in the future a version that reports up to the minutest details of the error.

illegal_initiation_parameter	: illegal_initiation_parameter	(Examples 1 and 2)
illegal_data_format	: illegal_data_format	(Example 3)
illegal_attribute_name	: illegal_attribute_name	(Examples 4 and 11)
doubly_defined_attribute_name	: doubly_defined_attribute_name	(Example 5)
illegal_index_type	: illegal_index_type	(Example 6)
illegal_initiation_parameter_for_ordered_index	: illegal_initiation_parameter_for_ordered_index	(Example 7)
illegal_initiation_parameter_for_hashed_index	: illegal_initiation_parameter_for_hashed_index	(Examples 8 and 9)
illegal_condition	: illegal_condition	(Example 10)
illegal_message_protocol	: illegal_message_protocol	(Example 12)
tried_to_get_nonexistent_element	: tried_to_get_nonexistent_element	(Example 13)
illegal_format_for_partial_reference	: illegal_format_for_partial_reference	(Example 14)
illegal_data_format in Message	: illegal_data_format in Message	(Examples 15 and 16)

【Example1】 Illegal initiation parameter

```
[1] pool:multiply_keyed_bag({a,b,c},-1,[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter
   Data > -1
   Node > 0
 !! Exception (h for help):
 no.
```

【Example2】 Illegal initiation parameter

```
[2] pool:multiply_keyed_bag({a,b},[packing_ratio(0.0)],[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter
   Data > packing_ratio( +0.0 )
   Node > 0
 !! Exception (h for help):
 no.
```

【Example3】 Illegal data format

```
[3] pool:multiply_keyed_bag(a,[]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format
   Data > a
   Node > 0
 !! Exception (h for help):
 no.
```

【Example4】 Illegal attribute name

```
[4] pool:multiply_keyed_bag({1},[]).
  Error>> multiply_keyed_bag
   Info > illegal_attribute_name
   Data > 1
   Node > 0
 !! Exception (h for help):
 no.
```

【Example5】 Doubly defined attribute name

```
[5] pool:multiply_keyed_bag({a,a},[]).
 Error>> multiply_keyed_bag
  Info > doubly_defined_attribute_name
  Data > a
  Node > 0
!! Exception (h for help):
no.
```

【Example6】 Illegal index type

```
[6] pool:multiply_keyed_bag({a(wan)},[]).
 Error>> multiply_keyed_bag
  Info > illegal_index_type
  Data > wan
  Node > 0
!! Exception (h for help):
no.
```

【Example7】 Illegal initiation parameter for ordered index

```
[7] pool:multiply_keyed_bag({a(ordered,{a})},[]).
 Error>> multiply_keyed_bag
  Info > illegal_initiation_parameter_for_ordered_index
  Data > {a}
  Node > 0
!! Exception (h for help):
no.
```

【Example8】 Illegal initiation parameter for hashed index

```
[8] pool:multiply_keyed_bag({a(hashed,[a])},[]).
 Error>> multiply_keyed_bag
  Info > illegal_initiation_parameter_for_hashed_index
  Data > a
  Node > 0
!! Exception (h for help):
no.
```

【Example9】 Illegal initiation parameter for hashed index

```
[9] pool:multiply_keyed_bag({a(hashed,[hasher(c)])},[]).
 Error>> multiply_keyed_bag
  Info > illegal_initiation_parameter_for_hashed_index
  Data > hasher(c)
  Node > 0
!! Exception (h for help):
no.
```

【Example10】 Illegal condition

```
[10] pool:multiply_keyed_bag({a},[empty(a,YorN)]).
 Error>> multiply_keyed_bag
  Info > illegal_condition
  Data > a
  Node > 0
!! Exception (h for help):
no.
```

【Example11】 Illegal attribute name

```
[11] pool:multiply_keyed_bag({a},[empty((a=1)/\(b=1),YorN)]).
  Error>> multiply_keyed_bag
   Info > illegal_attribute_name
   Data > b
   Node > 0
 !! Exception (h for help):
 no.
```

【Example12】 Illegal message protocol

```
[12] pool:multiply_keyed_bag({a},[unknown(_,_)]).
  Error>> multiply_keyed_bag
   Info > illegal_message_protocol
   Data > unknown(A,B)
   Node > 0
 !! Exception (h for help):
 no.
```

【Example13】 Tried to get nonexistent element

```
[13] pool:multiply_keyed_bag({a},[get(a=1,X)]).
  Error>> multiply_keyed_bag
   Info > tried_to_get_nonexistent_element
   Data > get(a=1,X)
   Node > 0
 !! Exception (h for help):
 no.
```

【Example14】 Illegal format for partial reference

```
[14] pool:multiply_keyed_bag({a,b},[getp(a=1,[b],X)]).
  Error>> multiply_keyed_bag
   Info > illegal_format_for_partial_reference
   Data > [b]
   Node > 0
 !! Exception (h for help):
 no.
```

【Example15】 Illegal data format for the message put

```
[15] pool:multiply_keyed_bag({a,b,c},[put({1})]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format in put
   Data > {1}
   Node > 0
 !! Exception (h for help):
 no.
```

【Example16】 Illegal data format for the message getp_and_put

```
[16] pool:multiply_keyed_bag({a,b,c},[put({1,1,1}),getp_and_put(a=1,{b},X,{2,2})]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format in getp_and_put
   Data > {2,2}
   Node > 0
 !! Exception (h for help):
 no.
```

8.5 Messages for the Pool

Table 8.1: Table of messages

(1)bag, (2)stack, (3)queue, (4)sorted_bag, (5)keyed_bag, (6)keyed_set,
(7)keyed_sorted_bag, (8)keyed_sorted_set, (9)multiply_keyed_bag

Massage name	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
empty(^YorN)	○	○	○	○	○	○	○	○	○
empty(K,^YorN)	×	×	×	×	○	○	○	○	○
put(X)	○	○	○	○	×	×	×	×	○
put(K,^X)	×	×	×	×	○	×	○	×	×
put(K,X,^OldX)	×	×	×	×	×	○	×	○	×
get(^X)	○	○	○	○	×	×	×	×	×
get(K,^X)	×	×	×	×	○	○	○	○	○
getp(C,P,^X)	×	×	×	×	×	×	×	×	○
get_if_any(^X)	○	○	○	○	×	×	×	×	×
get_if_any(K,^X)	×	×	×	×	○	○	○	○	○
getp_if_any(C,P,^X)	×	×	×	×	×	×	×	×	○
get_all(^O)	○	○	○	○	○	○	○	○	○
get_all(K,^O)	×	×	×	×	○	○	○	○	○
getp_all(P,^O)	×	×	×	×	×	×	×	×	○
getp_all(C,P,^O)	×	×	×	×	×	×	×	×	○
carbon_copy(^O)	○	○	○	○	○	○	○	○	○
carbon_copy(K,^O)	×	×	×	×	○	○	○	○	○
carbon_copyp(P,^O)	×	×	×	×	×	×	×	×	○
carbon_copyp(C,P,^O)	×	×	×	×	×	×	×	×	○
get_and_put(K,X,Y)	×	×	×	×	○	○	○	○	○
getp_and_put(C,P,^X,Y)	×	×	×	×	×	×	×	×	○
get_if_any_and_put(K,^X,Y)	×	×	×	×	○	○	○	○	○
get_and_put_if_any(K,^X,Y,^Y1)	×	×	×	×	○	○	○	○	○
getp_and_put_if_any(C,P,^X,Y,^Z)	×	×	×	×	×	×	×	×	○
get_max_if_any(^X)	×	×	×	○	×	×	○	○	×
get_min_if_any(^X)	×	×	×	○	×	×	○	○	×
max_if_any(A,^K)	×	×	×	×	×	×	×	×	○
min_if_any(A,^K)	×	×	×	×	×	×	×	×	○
get_load_factor(^A)	×	×	×	×	○	○	×	×	×
get_average_chaining_length(^L)	×	×	×	×	○	○	×	×	×
pack	×	×	×	×	×	×	×	×	○
do(Stream*)	○	○	○	○	○	○	○	○	○

【Hint】 If the expression

usage(<pool name atom>)

is executed from the shell or listener, the system will display a table of messages for that pool. Note that if **pool** is specified for the pool name, the system will display a table of pool names.

【Example】 The following example displays a table of messages for the stack pool type in the listener.

```
[1] usage(stack).
Usage :
  pool:stack(S*)
Messages :
  empty(^YorN)
  put(Data)
  get(^Data)
  get_if_any(^X)  , X={Data};{}
  get_all(^O)     , O=[Data, .. ]
  carbon_copy(^O) , O=[Data, .. ]
  do(S*)
[2]
```

8.6 Sorting Filter

PIMOS provides a means for sorting any KL1 data set.

A sorter is an object that sorts the order of data sets according to a given comparison mechanism. In PIMOS, sorters are provided in a filter format. That is, a sorted data sequence is obtained from the output stream by giving the sorter KL1 data sequences as an input stream. When the input stream is closed output to the output stream will be started for the first time. If the comparison is stable, then the sort will also be stable.

【Hint】 A merge sort is used in the sorter implementation.

【Caution】 In the following message protocol notations, the symbols (e.g., ^ and ?) attached to the variables in the protocols follow the mode display method described in section 7.1. In general, except for those items used for comparisons, data is allowed to be in the undefined state in filters used for sorting and merging. Therefore, a ? will be used as the mode display for data input and result output. For details, see section 7.1, "Interpreting the I/O Mode Display", (page 231).

8.6.1 Unkeyed sorting filter

A normal sorter which generates a filter by calling the predicates described below.

sorter:sort(In*, ^Out)

Generates an unkeyed sorting filter. The standard ascending order comparison routine comparator:sort/5 (see section 8.1.1, "Standard comparison routines" on page 317) is used for element comparisons.

sorter:sort(In*, Comparator, ^Out)

Generates an unkeyed sorting filter. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for the specifying method of Comparator.

8.6.2 Keyed sorting filter

Takes the KL1 data sequences given in a {`Key, ?Data`} format as an input stream, sorts them by the key part, and returns the result in an output stream. The output stream is also in a {`Key, ?Data`} format. Generates a filter by calling the predicates described below.

sorter:key_sort(In*, ^Out)

Generates a keyed sorting filter. The standard ascending order comparison routine comparator: sort/5 (see section 8.1.1, "Standard comparison routines" on page 317) is used for element comparisons.

sorter:key_sort(In*, Comparator, ^Out)

Generates a keyed sorting filter. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for the specifying method of Comparator.

8.7 Merging Filter

PIMOS provides a means for merging any KL1 data set.

A merger is an object that generates a single sorted data sequence from two sorted data sequences. In PIMOS, mergers are provided in a filter format. That is, a single sorted data sequence is acquired as an output stream by passing two sorted KL1 data sequences as input streams. If the comparison is stable, then the sort will also be stable.

【Caution】 The merger described in this section is different from the merger built-in predicate (which merges multiple streams regardless of their content). See section 7.4.19, "Stream support", (page 297) for details.

8.7.1 Unkeyed merging filter

A normal merger which generates an unkeyed merging filter by calling the predicate described below.

sorter:merge(In1, In2, ^Out)

Generates an unkeyed merging filter. The standard ascending order comparison routine comparator:sort/5 (see section 8.1.1, "Standard comparison routines" on page 317) is used for element comparisons.

sorter:merge(In1, In2, Comparator, ^Out)

Generates an unkeyed merging filter. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for the specifying method of Comparator.

8.7.2 Keyed merging filter

Merges only by the key part, and returns the result in the output stream. Both the input and output streams are given in a {`Key`, `?Data`} format. Generates a filter by calling the predicates described below.

sorter:key_merge(In1, In2, ^Out)

Generates a keyed merging filter. The standard ascending order comparison routine comparator:sort/5 (see section 8.1.1, "Standard comparison routines" on page 317) is used for element comparisons.

sorter:key_merge(In1, In2, Comparator, ^Out)

Generates a keyed merging filter. The comparison routine specified by Comparator is used for element comparison. See section 8.1.3, "Comparison routine specification", (page 319) for the specifying method of Comparator.

8.8 Error Messages Displayed by Sorting and Merging Filters

These filters report errors in cases where, for example, their input data is in an illegal format. The content and format of these messages are described below. See section 2.1.5.2, (shell) "Error Messages", (pages 21 and 22 for details on the display format when errors are reported in the shell, and section 4.1.8.3, (listener) "Error messages" [Example 39], (page 162) for details on the display format when errors are reported in the listener.

Cases where illegal format input data is passed to an input stream to a keyed sorting filter or merging filter:

Error ID: The name of the predicate used (either key_sort or key_merge)

Error Info: illegal_data_protocol

Error Data: Data that caused the error

【Example1】 Reporting example when an error was detected during execution in the listener.

```
[1] sorter:key_sort([a],Out).
  Error>> key_sort
   Info > illegal_data_protocol
   Data > a
   Node > 0
 !! Exception (h for help):
```

【Caution】 These error reports are only reported in relation to errors that were detected due to attempts to perform a comparison, and will not always be detected for the whole data sequence. For example, errors will not be detected in data for which there is no need to perform any comparisons, such as in the following example. This is to prevent unnecessary losses in efficiency when these utilities are used correctly.

【Example2】

```
[2] sorter:key_merge([],[key],Out).
 Out = [key]
 yes.

[3]
```

Cases where the arguments used with a keyed merging filter input stream are illegal (e.g., a data type other than list)

Error ID: key_merge

Error Info: illegal_protocol

Error Data: Message that caused the error

【Example1】

```
[1] sorter:key_merge(a,[{key,data}],Out).
  Error>> key_merge
   Info > illegal_protocol
   Data > a
   Node > 0
 !! Exception (h for help):
```

【Caution】 These error reports are only reported in relation to errors that were detected due to attempts to perform a comparison, and will not always be detected for the whole data sequence. For example, errors will not be detected in data for which there is no need to perform any comparisons, such as in the following example. This is to prevent unnecessary losses in efficiency when these utilities are used correctly.

[Example 2]

```
[2] sorter:key_merge(a,[],Out).
Out = a
yes.

[3]
```

【Caution】 Since the built-in predicate merger is inserted in the input streams to the utilities other than key_merge, a unification failure will occur if illegal arguments are used.

Chapter 9

I/O Management

This chapter describes the use of I/O.

What I/O resources are provided by PIMOS?

PIMOS provides four types of I/O resources: window, file, timer, and string I/O devices. Each of these has the following characteristics.

Window: Windows support character I/O. Input can be aborted. Window size, position, font, title, and other aspects can also be changed.

File: Files support character I/O. Directory operations are also supported.

Timer: Timers report the current date and time. Timers support alarm clock and stop watch functions. It is also possible to abort these functions.

String I/O devices: These are devices provided to manipulate devices defined by the user using ESP on the FEP. See section 9.3.8, "String I/O devices", (page 390).

Using the PIMOS I/O Resources

All of the PIMOS provided resources are implemented as processes. The format for using I/O resources consists of first using predetermined procedures for acquiring streams to those I/O resources, and then sending messages to those streams. Users must carry out these procedures correctly.

Section 9.1, "I/O Resource Acquisition", (page 350) gives a simple description of the basic sequence of operations. Note that section 9.1 focuses on providing a simple description of the overall flow of operations for acquiring I/O resources. Since the individual operations in the sequence of operations performed are described in detail in the manual sections corresponding to those operations, we recommend also referring to those manual sections.

I/O Utilities

The I/O resources themselves are nothing more than raw devices without elaboration. They can only perform device dependent processing, such as limited length 16-bit character string I/O. They do not have functions for syntactic analysis of input strings using an appropriate syntax or outputting arbitrary KL1 data as character strings.

Therefore, PIMOS provides utilities for simply and efficiently implementing higher level I/O as a filter process. There are several types of filter corresponding to the various uses. These filters have internal operator tables and, based on those tables, they can generate character strings from

KL1 data and perform syntactic analysis of character strings according to an operator precedence syntax identical to that of Edinburgh Prolog.

These filters have a blocking/deblocking function to reduce the communications overhead generated by I/O. That is, when input is required, they input a certain number of characters, store them internally, and return them a few at a time, and on output, these filters collect a certain number of characters without actually performing the output, and perform all the output at one time when that number of characters has been collected.

These functions can be used by simply inserting these filters at the front of the output stream to the raw device acquired by the user. Section 9.2, "I/O Utilities", (page 355) describes the use of these I/O utilities.

I/O Device Control

In addition to processing the usual I/O messages, device also require a mechanism for suspending the processing of those messages, due to requests from PIMOS or interrupts from the user. All devices supplied by PIMOS have two control lines in addition to the stream, which is for normal I/O control. One is an abort line for receiving suspend requests from PIMOS, and the other is an attention line for informing PIMOS of interrupts from the user. Section 9.3, "I/O Control", (page 372) describes control methods for this type of device and the message protocol for manipulating raw devices.

Standard I/O Devices

When programming, it is convenient to have I/O devices that can be manipulated with a standard message protocol that does not depend on the type of the actual I/O device. PIMOS provides abstract I/O devices, called standard I/O resources, in task units. As is the case with normal I/O devices, first a stream to the standard I/O resource is opened, and then I/O is performed by sending standard messages to that stream. Section 9.4, "Standard I/O", (page 394) describes the standard I/O resources and section 9.4.2, "Techniques for input and output using standard I/O", (page 395) describes the techniques for using those resources.

What kind of I/O device is actually allocated as a standard I/O resource is assigned at task activation time. For example, when programs are executed from the shell or listener, the shell or listener will set the standard I/O devices corresponding to the various cases according to the commands input by the user. See section 2.1, "User Goal Activation", (page 10) for details on basic shell manipulations, and section 4.1.4, "Standard I/O", (page 78) for details on the listener.

On the other hand, allocation and control methods for the situation where a user program itself allocates and uses standard I/O resources for a task are described in section 9.4.3, "Assigning standard I/O to a task and managing that task" (page 399).

9.1 I/O Resource Acquisition

To use an I/O device provided by PIMOS, that I/O device itself must first be acquired. This section describes the required basic sequence of manipulations for acquiring an I/O device.

Note that this section focuses on providing a simple description of the overall flow of operations for acquiring I/O resources. Since the individual operations in the sequence of operations used are described in detail in their corresponding manual sections, we recommend also referring to those manual sections.

9.1.1 First, acquisition of a general request device

General request devices are devices that perform a variety of services and that are supplied to the user by PIMOS. Even when the user uses PIMOS I/O devices, the user must first acquire a general request device and make requests to that device. The shoen exception raising predicate "shoen: raise" to generate an exceptional event in the shoen is used to acquire a general request device as shown below.

```
shoen:raise(pimos_tag#task, general_request, GeneralRequestDevice)
```

A stream to the general request device held in "GeneralRequestDevice" by the executed task can be acquired by executing this predicate.

【Caution】 The following declaration must appear at the start of a module in which this predicate call is described.

```
:- with_macro pimos.
```

【Hint】 See section 11.1.9, "General request devices", (page 423) for details on general request devices, and section 7.5.5, "Occurrence of exception", (page 314) for details on the shoen exception raising predicate "shoen: raise".

9.1.2 Next, request device acquisition

PIMOS devices form a hierarchical structure based on their role and function. To use a device, that device must be acquired in the hierarchy, in order starting with the upper level devices. See section 9.3.1, "Device hierarchy", (page 372) for details on the device hierarchy.

Devices for creating windows, opening files, generating timers, and generating string I/O devices can be acquired by sending the following messages to the general request device acquired by the preceding procedure. The acquired device will be **a window request device**, **a file request device**, **a timer request device**, or **a string I/O request device**, respectively.

Note that these message descriptions are written according to the mode notation of section 7.1 For details, see section 7.1, "Interpreting the I/O mode" (page 231).

window(^Result)

This message creates a window request device on the FEP specified by the value of the **task: io_node** environment variable set in the task that acquired the stream to the general request device, and establishes the abort and attention lines.

If the creation succeeded, **normal(^Request, ^Abort, Attention)** is returned in Result. Request is a stream to the created window request device, and Abort and Attention are the abort and attention lines to the window request device. The term **abnormal** is returned in Result if the creation failed.

file(^Result)

This message creates a file request device on the FEP specified by the value of the **task:io_node** environment variable set in the task that acquired the stream to the general request device, and establishes the abort and attention lines.

If the create succeeded, **normal(^Request, ^Abort, Attention)** is returned in Result. Request is a stream to the created file request device, and Abort and Attention are the abort and attention lines to the file request device. The term **abnormal** is returned in Result if the create failed.

timer(^Result)

This message creates a timer request device on the FEP specified by the value of the **task: io_node** environment variable set in the task that acquired the stream to the general request device, and establishes the abort and attention lines.

If the create succeeded, **normal(^Request, ^Abort, Attention)** is returned in Result. Request is a stream to the created timer request device, and Abort and Attention are the abort and attention lines to the timer request device. The term **abnormal** is returned in Result if the create failed.

string_io(^Result)

This message creates a string I/O request device on the FEP specified by the value of the **task:io_node** environment variable set in the task that acquired the stream to the general request device, and establishes the abort and attention lines.

If the create succeeded, **normal(^Request, ^Abort, Attention)** is returned in Result. Request is a stream to the created string I/O request device, and Abort and Attention are the abort and attention lines to the string I/O request device. The term **abnormal** is returned in Result if the create failed.

【Hint1】 See section 9.3.3, "Device abort/attention processing of device", (page 374) for details on the abort and attention lines.

【Hint2】 It is also possible to use an FEP other than the default FEP, or I/O resources on a machine at a remote site connected over the network through the FEP. See section 11.1.9, "General Request Devices", (page 423), for details on the general request devices used for acquiring this sort of resource.

9.1.3 Finally, device acquisition

It is possible, now, to acquire a raw device, such as a window or file, by sending messages to the request device acquired as described above. The remainder of this section presents a few programming examples.

【Hint】 The messages that can be sent to these request devices are described in the following sections:

9.3.5.1, "Window request device" (page 377)
9.3.6.1, "File request device" (page 382)
9.3.7.1, "Timer request device" (page 387)
9.3.8.1, "String I/O request device" (page 390)

【Example1】 Creating a window

```
get_window_device(Device) :- true |
        shoen:raise(pimos_tag#task,general_request,GRS),
        GRS=[window(Result)],
        wait_window_request_device(Device,Result).

wait_window_request_device(Device,normal(WRS,_,_)) :- true |
        WRS=[create(Result)],
        wait_window_device(Device,Result).
otherwise.
wait_window_request_device(Device,Status) :- true | Device=Status.

wait_window_device(Device,normal(WS,Abort,Att)) :- true |
        Device=normal(WS1,Abort,Att),
        initiate(WS,WS1).
otherwise.
wait_window_device(Device,Status) :- true | Device=Status.

% Actually, the program ought to check Status.
initiate(WS,WS1) :- true |
        WS=[activate(_),reshape(mouse,mouse,_)|WS1].
```

【Example2】 Opening a file

```
get_file_device(Pathname,Mode,Device) :- true |
        shoen:raise(pimos_tag#task,general_request,GRS),
        GRS=[file(Result)],
        wait_file_request_device(Pathname,Mode,Device,Result).

wait_file_request_device(Pathname,Mode,Device,normal(FRS,_,_)) :- true |
        FRS=[open(Pathname,{Mode,Device})].
otherwise.
wait_file_request_device(_,_,Device,Result) :- true | Device=Result.
```

9.1.4 Performing I/O

When an I/O device has been acquired in the above manner, I/O can be performed by sending messages to the stream to the device.

【Hint】 The messages that can be sent to these devices are described in the following sections:
9.3.5.2 “Window” (378),
9.3.6.3 “file” (384),
9.3.6.4 “directory” (385),
9.3.7.2 “timer” (388),
9.3.8.2 “string I/O devices” (390).

【Example】 Inputting 256 characters from a window or an input file

```
input(DS,DS1,String) :- true |                  % DS and DS1 are streams to a device

        DS=[getb(256,Result)|DS1],
        wait_input(Result,String).

wait_input(normal(String),String1) :- true | String=String1.
otherwise.
wait_input(_,String) :- true | String=‘‘’’.
```

Execution example from the listener (inputting 256 characters from an input file)

```
[10] test:get_file_device(``login.com'',read,normal(DS,_,_)),test:input(DS,[],String) .
 DS = [getb(256,normal(``   % login?? ''))]
 String = ``   % login?? ''
```

9.1.5 Performing I/O with an I/O utility filter inserted

Devices can be successfully manipulated using the above procedure. However, this results in manipulating a raw device.

Thus it is normally desirable for general users to perform I/O with an I/O utility filter inserted at the front of the output stream to the device to perform I/O easily and efficiently. There are three types of I/O filter: input, output, and I/O. We recommend that these be used according to the requirements of the application. See section 9.2, "I/O Utilities", (page 355) for details on the I/O utilities. Filters insertion is described below.

【Example1】 Attaching an I/O utility filter to a window device.

```
get_user_window_device(Window) :- true |
        get_window_device(Device),
        wait_device(Window,Device).

wait_device(Window,normal(WS,_,_)) :- true |
        buffer:interaction_filter(Window,WS).
otherwise.
wait_device(_,Device) :- true |         % An error will be flagged if device generation fails.

        shoen:raise(pimos_tag#error,
            '$PIMOS$'(user_window,``failed_in_window_creation''),Device).
```

【Hint】 If an error report is coded as in the example above, the listener or shell will display an error in a standard form if an error occurs during execution. See section 11.5, "Standardly Supplied Error Reporting Methods", (page 450). Also, see sections ??, "Error messages", (pages 21 and 22) and section 4.1.8.3, "Error messages", [Example 39] (page 162) for the actual display format of the error messages in the shell and listener, respectively.

【Example2】 Input of 1 line from a window with an I/O utility filter attached

```
input_line(Win,Win1,Line) :- true |
        Win=[prompt(``input_line> ''),getl(Line)|Win1].
```

Execution example from the listener:

```
[10] test:get_user_window_device(Win),test:input_line(Win,[],Line).
 Line = ``abc''
 Win = [prompt(``input_line>''),getl(``abc'')]
```

【Example3】 Input of 1 term from a window with an I/O utility filter attached

```
input_term(Win,Win1,Term) :- true |
        Win=[prompt(``input_term>''),gett(Term)|Win1].
```

Execution example from the listener:

```
[11] test: get_user_window_device(Win),test:input_term(Win,[],Term).
 Term = term
 Win = [prompt(``input_term>''),gett(term)]
```

【Example4】 Input of 1 wrapped term from a window with an I/O utility filter attached

```
input_wrapped_term(Win,Win1,Result) :- true |
        Win=[prompt(``input_wrapped_term>''),getwt(Result)|Win1].
```

Execution example from the listener:

```
[12] test: get_user_window_device(Win),test:input_wrapped_term(Win,[],Result).
 Result = normal(atom(wrapped_term))
 Win = [prompt(``input_wrapped_term>''),getwt(normal(atom(wrapped_term)))]
```

【Hint1】 To handle variables as variables, terms must be handled in the **wrapped** state. A **wrapped term** is a normal term covered with a layer that follows the data type. The messages getwt/1 and putwt/1 are used for I/O of wrapped terms. See section 9.2.2, "Normal term and wrapped term".

【Hint2】 The actual character I/O messages to PIMOS windows and files have the same format. As a result, the I/O utility filters are able to perform I/O without knowing the actual type of the device.

Therefore, there is no need for the user to be concerned with the device type when inserting an I/O utility filter. Actually, the 4 sample programs presented above are valid even if filters are used.

9.2 I/O Utilities

This section describes the I/O utilities provided by PIMOS.

Note that the streams acquired by calls to the predicates described in this section all have a merger (see section 7.4.19) installed in advance.

9.2.1 Buffer

PIMOS provides a buffering filter (also called simply filter) which blocks and deblocks data to reduce communication overhead, and an infinite length buffering function to store any quantity of character data.

There are three types of buffering filters: input, output, and I/O, which are inserted between the user and the I/O device. An input filter stores a given quantity of input characters (when you need to input), and returns them little by little as required. An output filter receives and stores a certain number of characters, and outputs them as a whole. An I/O filter has both input and output features. These filters are automatically inserted into a request stream when a standard I/O stream is obtained. (See section 9.4.2.2.)

The infinite length buffer receives both input and output messages from the user. It has an extensible FIFO buffer, where output data is stored. When the system receives an input request, data is returned from the top of this buffer.

The buffering filter and the infinite length buffer have basically the common interface which can be used by redirecting the input and output of one single program. They also have an operator table (pool). By referring to this operator pool, they can analyze the structure of data character string expression (for input) and generate string expressions (for output) based on the same operator precedence grammar as the Edinburgh Prolog.

PIMOS generally uses the JIS 16-bit code system for character I/O and represents a new line as character code 10 (decimal representation).

9.2.1.1 Common protocol

The protocol used in common to filters and infinite length buffers is listed below in details.

【Caution】 The following description about the end of input or the abort conditions relates to filters only. Infinite length buffers are free from end of input or abort condition. If an input request such as getc is sent to an empty infinite length buffer, the input request processing is suspended until any data is stored in the buffer by another message.

Input message protocol

getc(^C)

Unifies the next one-character code and C. If it is at the end of input, unifies −1. If the device aborts, unifies −2.

getl(^String)

Unifies the next one line with String as a 16-bit string. If it is at the end of input, unifies −1. If the device aborts, unifies −2.

getb(Size, ^Block)

Unifies the next Size character with Block as a 16-bit string. If input terminated before Size characters has not been read, converts the characters so far received into a 16-bit string. If the input characters have come to the end, unifies −1, and if the device aborts, unifies −2.

getwt(^Result)

Reads characters until the full-stop (a period (.) followed by a blank, tab, or LF), and parses the string according to the operator pool (see page 364). If parsing is successful, Result is unified into `normal(WT)`, where WT is a wrapped term.

If the input has come to the end, Result is unified into `normal(end_of_file)`. If the input is only a full-stop, it is unified into `normal(empty)`.

In case of an error in the input term, Result is unified into `abnormal(ErrorInfo)`. `ErrorInfo` contains the following:

error({TL,Ps}) An error in the Ps'th location of the input tokens TL.

error({TL,ErrorReason,Ps}) An error in the Ps'th location of the input tokens TL. ErrorReason is an atom indicating the cause of the error and includes the following:

newline_occurred_in_string An LF is present in the character string.

newline_occurred_in_quoted_atom An LF is present in atoms enclosed with quotation marks.

integer_over_flow An integer operation overflow occurred while reading. It occurs while entering an integer constant which cannot be expressed with a 32-bit integer.

atom_device_abnormal Conversion of an atom name to an atom failed (probably due to PIMOS bug).

ambiguous({TL,Ps}) The Ps'th token of the input tokens TL is interpreted in several ways.

【Hint1】 See section ??, "Normal term and wrapped term", (page ??) for details of wrapped term.

【Hint2】 By passing ErrorInfo to `print_error(ErrorInfo)` (see page 360), the contents of the error can be displayed.

【Hint3】 gett described later in this chapter is implemented by using getwt.

【Caution】 The way in which errors are indicated will be revised in the future.

gett(^T)

gett(^T,NVP)

gett(^T,^VP,NVP)

Reads until full-stop (a period followed by a blank, tab, or LF), and parses the character string according to the operator pool. VP is a stream to the variable pool (see page ??) used in parsing, and NVP is a stream to the variable changed as a result of the parse.

If no VP is specified, an empty variable pool is created and used. If no NVP is specified, the variable pool terminates.

If parsing is successful, T is unified with the obtained (normal) term. If the input has come to the end, atom `end_of_file` is unified.

【Caution】 Input with full-stop alone is ignored.

If an error exists in the input terms, T is unified into one of the following of which each is corresponding to ErrorInfo of getwt/1.

$error({TL,Ps}) .

$error({TL,ErrorReason,Ps}) .

$ambiguous({TL,Ps}) .

skip(C)

Ignores the input until the character code C appears or the end of the input is reached. For the getc(C) just behind it, the character code next to C is unified. When the input comes to the end while ignoring, C and -1 are unified for the getc(C) just behind it.

ungetc(C)

Sends back character code C to the internal buffer. This code can be read at the getc(C) just behind it. Ungetc can be repeated as many times as needed, and the characters can be sequentially read beginning with the character specified in the last ungetc.

prompt(String)

It is same as putb(String) if it is connected to an interactive I/O device. Otherwise, it is ignored.

【Hint】 Prompt is a message used to display a message prompting for user input, and is intended to write programs independent of the type of originating I/O devices. When prompt/1 is issued to the window, it works as similar to putb/1, but if it is issued to a file, no operation takes place.

Output message protocol

putc(C)

Writes the character indicated with the character code C.

putl(String)

Writes the string indicated by String and enters an LF.

putb(String)

Writes the string indicated by String.

putwt(WT, PDepth,PLength, PSLength,PALength)

putwt(WT, PDepth,PLength, PSLength)

putwt(WT, PDepth,PLength)

putwt(WT)

Based on the operator pool, writes the wrapped term WT using the operator precedence grammar. For the part in which the depth of the structure exceeds PDepth "`&`" is written out, and "..." is written out for the part in which the length exceeds PLength, PSLength, and PALength. (If any one of PDepth, PLength, PSLength and PALength is not specified, the default value is used) (see page 361).

【Caution1】 For this message and putwtq, putt, and puttq messages listed below, the full-stop at the end of the term is not output.

【Caution2】 If WT is not wrapped, the output is suspended (when WT includes an undefined variable) or fails (when not wrapped correctly).

【Hint】 A module type is output in the format of "`module`$<package name>::<module name>$<module size>", a code type is output in the format of "`code`$<package name>::<module name>:<predicate name>/arity$<module size>", and the atom without any output image (not registered in the atom table) is output in the format of "`atom`$<atom No.>"

putwtq(WT, PDepth,PLength, PSLength,PALength)

putwtq(WT, PDepth,PLength, PSLength)

putwtq(WT, PDepth,PLength)

putwtq(WT)

Writes the wrapped WT with appropriate quotation marks. If a full-stop follows, the preceding term output by putwtq can be read in by getwt, gett, etc.

【Caution】 Note that putwtq or puttq must be used when the user desires to read in previously output terms at a later point with gett or getwt. These messages attach quotes to atoms and insert spaces between tokens appropriately so that they can be read in without error by gett and getwt.

putt(T, PDepth,PLength, PSLength,PALength)

putt(T, PDepth,PLength, PSLength)

putt(T, PDepth,PLength)

putt(T)

Writes the term T using the operator precedence grammar based on the operator pool.

【Caution】 If the T contains a variable which has not been instantiated, output is suspended. This rule also applies to the following puttq messages.

puttq(T, PDepth,PLength, PSLength,PALength)

puttq(T, PDepth,PLength, PSLength)

puttq(T, PDepth,PLength)

puttq(T)

Writes the term T with appropriate quotation marks. If a full-stop follows, the preceding term output by puttq can be read by getwt, gett, etc.

nl

Writes a line feed (LF).

tab(N)

Writes N blank codes.

putf(FormatString, ArgumentList)

Outputs ArgumentList according to the format FormatString. ArgumentList is a list of arguments, and arguments are passed to the formatting items which need an argument consecutively beginning from the top of the list.

The format is given by the string, and the characters except for the formatting items which begin with ~ are output without any modification. The general format of a formatting item is ~Nc, where c indicates the type of the formatting item and N is an optional positive integer.

~N**c** (Character) Interprets an argument as a character code and outputs that character N times. N defaults to 1.

~N**s** (String) Outputs N characters from the beginning of the argument string. N defaults to the length of the argument string.

~N**b** (Blank) Outputs N blanks. N defaults to 1.

~N**t** (Term) Outputs the term of the argument N times in the same format as putt. N defaults to 1.

~N**T** (Term) Outputs the term of the argument N times in the same format as puttq. N defaults to 1.

~N**w** (Term) Outputs the term of the argument N times in the same format as putwt. N defaults to 1.

~N**W** (Term) Outputs the term of the argument N times in the same format as putwtq. N defaults to 1.

~*N***n** (Newline) Outputs *N* newlines. *N* defaults to 1.

~*N*~ Outputs *N* "~"s. *N* defaults to 1.

【Caution】 The putf message will fail if the length of Argument is insufficient for the number of formatting items specified by FormatString. Excess items in Argument are ignored.

print_error(ErrorInfo)

Error information is displayed according to ErrorInfo. ErrorInfo must be the argument of `abnormal(ErrorInfo)` returned upon abnormal termination of getwt/1.

flush(^Status)

Writes the data stored in the buffer to the I/O device. Upon completion of output (if connected to the device on FEP, the contents of the buffer is sent to FEP, and if FEP informs of the completion of output operation), the atom normal is unified with Status. In the case of an infinite length buffer, the atom normal is returned in Status after all the output messages issued prior to this message have been processed.

In the filter, I/O data blocking is performed to reduce traffic of communication with I/O devices. Therefore, simply sending the output message merely causes the data to be stored in the filter buffer, and unlikely sent to the I/O device. Data in the buffer is sent to the I/O device if one of the following conditions is true.

- The buffer is full.
- The flush/1 or putl/1 message is received.
- When the stream is closed.

Operator message protocol

Protocol which operates the operators used for gett, putt, etc. (See section 9.2.1.6 on page 364 for details.)

add_op(Prec, Type, OpName)

Adds an operator OpName (atom) of which the priority is Prec and the type is Type to the operator pool.

remove_op(Prec, Type, OpName)

Removes the operator OpName (atom) of which the priority is Prec and the type is Type from the operator pool. If it is not registered, no operation takes place.

operator(OpName, ^Def)

Unifies the list of a pair of priority and type of the operator OpName with Def. The format of the pair is "`{Priority, Type}`". If OpName is not registered, Def is unified with `[]`.

Control message protocol

do(Stream*)

Stream is a list comprising the messages of the same protocol as that of the parent stream as the element (a stream which must always be closed). Messages in the Stream are continuously processed.

When a number of streams are merged in the message stream, the sequence of the messages within each stream is guaranteed, but which message in which stream in the merged stream group is processed first is not guaranteed. To ensure continuous processing of these messages (for example, to suppress intervention of other output until a series of strings is output and an LF is entered), it is necessary to guarantee continuous processing by using the do message.

【Hint】 `S = [do(S1)|(S2)]` has the same meaning as `append(S1, S2, S)`.

【Hint】 do may be contained in the argument list of message do.

replace_op_pool(OldOpPool, ^NewOpPool)

Changes the operator pool from OldOpPool to NewOpPool.

change_op_pool(^NewOpPool)

Same as `replace_op_pool([],` NewOpPool).

print_depth(PDepth)

print_length(PLength)

print_string_length(PSLength)

print_atom_length(PALength)

Changes the default values to limit the depth and length of a structure, the string length, and the atom name length to PDepth, PLength, PSLength and PALength respectively. The initial limit values are as follows: the depth of the structure: 4, the length of the structure: 10, the string length: 20, and the atom name length: 30.

print_depth(^OPDepth,NPDepth)

print_length(^OPLength,NPLength)

print_string_length(^OPSLength,NPSLength)

print_atom_length(^OPALength,NPALength)

Changes the default values to limit the depth and length of a structure, the string length, and the atom name length upon structure output from OPDepth, OPLength, OPSLength and OPALength to NPDepth, NPLength, NPSLength, and NPALength respectively.

Abort/attention message protocol

This protocol is the one described in section 9.3.3, “Device abort/attention processing”, (page 374).

reset(^Result)

resend(ID, ^Result)

cancel(ID, ^Result)

next_attention(^Result)

【Caution】 If a buffer enters the abort state, it remains in the abort state until termination.

9.2.1.2 Input filter

Creation

buffer:input_filter(Init, In*, ^Out)

Creates an input filter with the default setting of Init. Init is a list of the following elements. The filter reads the command protocol from In (the user side) and sends the command protocol to Out (the device side).

size(Size) Sets the size of the internal buffer to Size. The default is 490.

atom_table(^AtomDev) Sets the atom data base (see page 404) to AtomDev. AtomDev is the stream to the atom data base, and the default is `atom_table:cache_table(AtomDev)`.

operator_pool(^OpPool) Sets the operator pool (see section 9.2.1.6) to OpPool. OpPool is the stream to the operator pool, and the default is operator:pool(OpPool). `operator:pool(OpPool)`.

buffer:input_filter(In*, ^Out)

`Same as buffer:input_filter([], In, Out)\verb`.

User (In) protocol

Accepts the messages for the input (see page ??), operator pool (see page ??), control and abort/attention message protocol (see page ??).

【Caution】 A message with unknown protocol is passed to the device as is.

Device (Out) protocol

getb(Size, ^Status)

prompt(String, ^Status)

9.2.1.3 Output filter

Creation

buffer:output_filter(Init, In*, ^Out)

Creates an output filter with the default setting of Init. The specifications for Init are the same as those for the Init used with input filter. The filter reads the command message from In (the user side) and sends the command message to Out (the device side).

buffer:output_filter(In*, ^Out)

Same as buffer:output_filter([], In, Out).

User (In) protocol

Accepts the messages for the output (see page ??), operator pool (see page ??), control and abort/attention message protocol (see page ??).

【Caution】 A message with unknown protocol is passed to the device as is.

Device (Out) protocol

putb(String, ^Status)

flush(^Status)

9.2.1.4 Input/output filter

Creation

buffer:interaction_filter(Init, In*, ^Out)

Creates an input/output filter with the default setting of Init. The specifications for Init are the same as those for the Init used with input filter. The filter reads the command message from In (the user side) and sends the command message to Out (the device side).

buffer:interaction_filter(In*, ^Out)

Same as buffer:iteraction_filter([], In, Out).

User (In) protocol

Accepts the messages for the input (see page ??), output (see page ??), operator pool (see page ??), control and abort/attention message protocol (see page ??).

【Caution】 A message with unknown protocol is passed to the device as is.

Device (Out) protocol

getb(Size, ^Status)

prompt(String, ^Status)

putb(String, ^Status)

flush(^Status)

9.2.1.5 Infinite length buffer

Creation

buffer:character_pool(Init, In*)

Creates an infinite length buffer with the default setting of Init. The specifications for Init are the same as those for the Init used with input filter. Writing to and reading from the buffer are performed using the command message from In.

buffer:character_pool(In*)

`Same as buffer:character_pool ([], In).`

Protocol

Accepts messages for the input (see page ??), output (see page ??) and operator pool message protocol (see page ??).

【Caution】 A message with unknown protocol will fail.

9.2.1.6 Operator pool

The operator pool is a table of operators used for parse and unparse according to the operator precedence grammar. Each operator has its type and priority. Type identifies either of prefix, infix or postfix and also indicates the way of connection for the same priority expressed by fx, fy, xfx, xfy, xf, and yf. Priority is represented by an integer in a range between 1 and 1200 indicating the strength of connection (the smaller the number, the stronger the strength of connection).

【Caution】 One single atom can be registered as multiple types of operator.

Creation

operator:pool(In*)

Creates an operator pool where the standard operators have been registered.

Table 9.1 lists the available standard operators.

operator:empty_pool(In*)

Creates an empty operator pool where no operator is registered.

Protocol

Accepts the messages for the operator pool (see page ??).

9.2.1.7 Variable pool

A variable pool is a correspondence table among the variable names (string), variable addresses and the variables.

Creation

variable:pool2(In*)

Creates an empty variable table.

Protocol

count(^N)

Unifies the number of currently registered variable names with N.

Table 9.1: standard-operators

Priority	Type	Operator
1200	xfx, fx	`:-`
	xfx	`-->`
1150	fx	`public`, `implicit`, `local_implicit`, `with_macro`
1100	xfy	`;`, `|`
1090	xfx	`=>`
1050	xfy	`->`
1000	xfy	`,`
800	xfx	`:`
700	xfx	`=`, `\=`, `==`, `=:=`, `=\=`, `<`, `>`, `=<`, `>=`, `:=`, `<=`, `<<=`, `$=:=`, `$=\=`, `$<`, `$>`, `$=<`, `$>=`, `$:=`, `$<=`
	xfy	`@`
500	yfx,fx	`+`, `-`
	yfx	`/\`, `\/`, `xor`
400	yfx	`*`, `/`, `<<`, `>>`
300	xfy	`**`
	xfx	`mod`
200	fx	`&`
150	xf	`++`, `--`
100	xfx,fx	`#`
90	xfx	`::`
80	fx	`module`

unify

Unifies the variables registered with the same name.

reset

Unifies the variables registered with the same name and then initializes the variable table.

【Hint】 The variable pool sends reset to itself as the termination process.

put_var(Name, ?V)

Makes an entry of a variable V with a name Name.

get_name({PE,ADR}, ^Result)

Unifies Result into {`N, Name`}. N deretterenced and Name are the variable number and variable name of the variable addressed to processor number PE and address ADR. The variable number is a unique integer assigned in accordance with the variable table.

get_value(Name, ^Val)

Unifies Val with {`Value`} if the value of the variable registered under the variable name Name is Value. Unifies Val with {} if the variable is not registered.

【Caution】 The processor and address with a variable can vary according to unification and GC. Since the variable pool cannot determine whether the address is changed by unification or GC, it sometimes assumes different variables to be the same.

9.2.2 Normal term and wrapped term

Different from sequential type Prolog, in parallel inference language KL1, recognizing a variable still remaining as variable is primarily meaningless. This is because the variable might have been changed to anything else (due to concurrently running processes) shortly after it was identified as variable. Therefore, some provision is required when writing meta level programs (such as debugger) which need to identify a variable as it is. A solution to this includes normal term and wrapped term. This approach is an extended idea of freeze which assumes the conventional variables to be the ground term.

A normal term is one usually handled in KL1. A wrapped term is the result of a normal term wrapped with others according to its data type. For example, "X+1" is expressed in { `+, X, 1`} in terms of normal term, and as a wrapped term, it is internally expressed in `vector({atom(+),variable(''X''),integer(1)})`. A wrapped term may be taken to have the data tagged on a software level. The following rules apply when wrapping terms.

Atom: atom(Atom)

Integer: integer(Integer)

Floating point number: floating_point(FloatingPoint)

Variable variable(¡variable name¿): The variable name is a (normal) 16-bit string. The variable name is the variable name string if the variable was input as text, and is a string produced at the discretion of the system for variables created during execution. (See section 9.2.2.1.)

String: string(String)

List: list (WrappedList)

Vector: vector(WrappedVector)

Module: module(Module)

Code: code(Code)

Others: unknown(Unknown)

The predicates used to convert normal terms and wrapped terms are wrap and unwrap described in the following section.

9.2.2.1 Wrap

Wrap is a predicate which converts a normal term into a wrapped term.

variable:wrap(T, ^Result, D,L)

variable:wrap(T, ^Result, D,L, NVP)

variable:wrap(T, ^Result, D,L, ^VP,NVP)

Wraps a normal term within a range of depth D and length L. Result is unified with {`NewT, WT`}. NewT is unified with T and WT is unified with the term which has wrapped T.

If a variable pool VP is specified, it is used. Otherwise, an empty variable pool is used. A variable which is not registered in the variable pool is assigned an appropriate variable name. The variable pool updated by wrap is NVP. If no NVP is specified, the used variable pool terminates.

variable:wrap_address(T, ^Result, D,L)

It is the same as variable:wrap/4 except for handling of a variable. The variable is wrapped in the form of variable`variable({PE, ADR})`, where PE and ADR are the processor number and address containing the variable.

【Hint】 This is useful when address display of a variable is required for debugging.

【Example】 Address display of a variable can be obtained by performing putwt(WT) for the wrapped term WT obtained by wrap_address.

【Caution】 Wrap is intended to replace an undefined variable with a term at a certain time. So that the result of the wrap for one single term is not always the same. This function should be selectively used for particular programs requiring meta level processing and is not suitable for normal programs.

9.2.2.2 Unwrap

Unwrap is a predicate used to convert wrapped terms into normal terms.

variable:unwrap(WT, ^Result)

variable:unwrap(WT, ^Result, NVP)

variable:unwrap(WT, ^Result, ^VP,NVP)

Unwraps a wrapped term. Result is unified with {`NewWT, T`}. NewWT is unified with WT and T is unified with the unwrapped term.

If a variable pool VP is specified, it is used. Otherwise, an empty variable pool is used. The variable pool updated by wrap is NVP. If no NVP is specified, the used variable pool terminates.

【Caution】 The wrapped term obtained by wrap_address cannot be unwrapped.

9.2.3 Timer utilities

The PIMOS timers provide functions for acquiring the date and time. (See section 9.3.7.2, "Timers" on page 388.) These values are the day count from January 1, 1901 to the current day as a positive integer, and the time unit count (the default time unit is the millisecond, with values expressed as positive integers) since the previous midnight (i.e., 00:00:00 the current day). This section describe utilities for calculating the date and time from these values and for unit conversions of a given time unit count.

【Caution1】 The term "day count" (or "number of days") used in this section refers to the day count of days since January 1, 1901.

【Caution2】 The time unit count values passed to the utility for calculating the time from a time unit count must be expressed in millisecond units. Values expressed in other units must first be converted to milliseconds using the time unit conversion utility (page 371).

【Example】 Acquiring the day count and time unit count from the timers.

```
\begin{samplebox}\footnotesize\tt\rm\begin{program}
% go(Date,Time) :- true |
%         shoen:raise(pimos_tag#task,general_request,GRS),
%         GRS=[timer(Result)],
%         wait_timer_request_device(Date,Time,Result).
%
% wait_timer_request_device(Date,Time,normal(TRS,_,_)) :- true |
%         TRS=[create(Result)],
%         wait_timer_device(Date,Time,Result).
% otherwise.
% wait_timer_request_device(Date,Time,_) :- true | Date=0,Time=0.
%
% wait_timer_device(Date,Time,normal(TS,_,_)) :- true |
%         TS=[get_date_time(Result)],
%         wait_date_time(Date,Time,Result).
% otherwise.
% wait_timer_device(Date,Time,_) :- true | Date=0,Time=0.
%
% wait_date_time(Date,Time,normal(D,T)) :- true | D=Date,T=Time.
% otherwise.
% wait_date_time(Date,Time,_) :- true | Date=0,Time=0.

[1] timer_test:go(Today,Now).       % When the above goal is executed:
^Now = 53800039
^Today = 33166

[2]
```

9.2.3.1 Date utilities

timer_utility:get_date_string(Date_count, ^String)

Generates a character string that expresses the date from the day count in Date_count (a

positive integer).

【Example】

```
[10] v.
^Today = 33166

[11] timer_utility:get_date_string(Today,String).
 String = ''22-Oct-1991''
```

timer_utility:get_date_value(Date_count, ^Year, ^Month, ^Day, ^Week)

Calculates the date from the day count in Date_count (a positive integer), and returns the year, month, day, and day of the week in Year, Month, Day, and Week, respectively. Year is a positive integer that expresses the year, Month is a positive integer between 1 and 12, Day is a positive integer between 1 and 31, and Week is a positive integer between 0 (Sunday) and 6 (Saturday).

【Example】

```
[12] v.
^Today = 33166

[13] timer_utility:get_date_value(Today,Year,Month,Day,Week).
 Day = 22
 Month = 10
 Week = 2
 Year = 1991
```

timer_utility:get_date_count(Year, Month, Day, ^Date_count)

Calculates the day count since January 1, 1901 for the date expressed by Year, Month, and Day, and returns it in Date_count as a positive integer.

【Example】

```
[14] timer_utility:get_date_count(1992,6,1,^FGCS).
^FGCS = 33389
```

timer_utility:month(Mon_index, ^Name, ^AbbrName, ^Days)

Given a positive integer between 1 and 12 that expresses a month in Mon_index, this utility returns the name of the month (as a character string) in Name, the abbreviation for the month AbbrName (as a character string), and the day count in that month (as a positive integer) in Days.

【Example】

```
[15] timer_utility:month(10,Name,AbbrName,Days).
 AbbrName = ''Oct''
 Days = 31
 Name = ''October''
```

timer_utility:day(Day_index, ^Name, ^AbbrName, ^JapaneseName)

Given a positive integer between 0 and 6 that expresses a day of the week in Day_index, this utility returns the name of the day in Name (as a character string), the abbreviation for the

day in AbbrName (as a character string), and the Japanese name for the day in JapaneseName (as a character string).

【Example】

```
[16] timer_utility:day(2,Name,AbbrName,JapaneseName).
 AbbrName = ‘‘Tue’’
 JapaneseName = ‘‘火’’
 Name = ‘‘Tuesday’’
```

@@@

9.2.3.2 Time of day utilities

timer_utility:get_time_string(Time_count, ^String)

Generates a character string that expresses the time of day from the millisecond count Time_count (a positive integer) and returns it in String.

【Example】

```
[17] v.
^Now = 53800039

[18] timer_utility:get_time_string(Now,String).
 String = ‘‘14:56:40’’
```

timer_utility:get_time_value(Time_count, ^Hour, ^Min, ^Sec, ^Millisec)

Calculates the time of day from the millisecond count Time_count (a positive integer) and returns the hours, minutes, seconds, and milliseconds in Hour, Min, Sec, and Millisec respectively. Hour is a positive integer between 0 and 23, Min and Sec are positive integers between 0 and 59, and Millisec is a positive integer between 0 and 999.

【Example】

```
[19] v.
^Now = 53800039

[20] timer_utility:get_time_value(Now,Hour,Min,Sec,Millisec).
 Hour = 14
 Millisec = 39
 Min = 56
 Sec = 40
```

timer_utility:get_time_count(Hour, Min, Sec, Millisec, ^Time_count)

Calculates the milliseconds count since 00:00:00 for the time of day expressed by Hour, Min, Sec, and Millisec, and returns that value as a positive integer in Time_count.

【Example】

```
[21] timer_utility:get_time_count(12,0,0,0,Noon).
 Noon = 43200000
```

9.2.3.3 Utilities that handle both date and time

timer_utility:add_date_time(Date1, Time1, Date2, Time2, ^Date, ^Time)

Adds a time point and a time period (or a pair of time periods) expressed as day count and millisecond count pairs.

Adds the time point (or time period) expressed as the pair Date1 (a positive integer) and Time1 (a positive integer) and the time period expressed as the pair Date2 (a positive integer) and Time2 (a positive integer), and returns the result as the pair Date (a positive integer) and Time (a positive integer).

【Example】 Deriving the date of the day one week from today.

```
[22] v.
^Today = 33166

[23] timer_utility:add_date_time(Today,0,7,0,_D,_),
timer_utility:get_date_string(_D,NextWeek).
 NextWeek = ``29-Oct-1991''
```

timer_utility:subtract_date_time(Date1, Time1, Date2, Time2, ^Date, ^Time)

Subtracts a time period from a time point (or a time period) expressed as day count and millisecond count pairs.

Subtracts the time period expressed as the pair Date2 (a positive integer) and Time2 (a positive integer) from the time point (or time period) expressed as the pair Date1 (a positive integer) and Time1 (a positive integer), and returns the result as the pair Date (a positive integer) and Time (a positive integer).

【Example】 It is 222 days 9 hours and 3 minutes until FGCS.

```
[24] v.
^FGCS = 33389
^Now = 53800039
^Today = 33166

[25] timer_utility:subtract_date_time(FGCS,0,Today,Now,Days,_T),
timer_utility:get_time_string(_T,Time).
 Days = 222
 Time = ``09:03:19''
```

9.2.3.4 Time unit conversion utilities

time_unit_translator:trans({Unit, Count}, {NUnit, ^NCount})

Converts the time units Count (a positive integer) expressed in time units Unit (an atom) to time units Unit (an atom), and returns the result in NCount. Time units can be specified to be one of the following:

sec: Seconds

msec: Milliseconds

msec16: Sixteenths of a millisecond

【Hint】 Further time unit types may be added in the future.

【Caution】 A -1 will be returned if the result is an number of a size that cannot be expressed in an integer. This is expected to be corrected when the multiple precision integer operation module (bignum) is provided at some point in the future.

【Example】

```
[1] time_unit_translator:trans(msec(1000),sec(X)).
 X = 1
```

9.3 I/O Control

In addition to the stream used for normal I/O processing, I/O devices that are supplied by PIMOS also have two control lines. One of these is the abort line, which is used to receive abort requests from PIMOS, and the other is the attention line, which is used to inform PIMOS of interrupt from the user. This section describes methods for device control and the message protocols for manipulating raw devices.

9.3.1 Device hierarchy

The I/O devices supplied by PIMOS form a hierarchical structure according to their function. The highest point in the hierarchy is the general request device. The general request device is a device that performs a variety of services provided to the user by PIMOS. A device that has the role of generating and supplying I/O devices can be acquired by requesting an I/O resource from the general request device. A device that generates and supplies I/O devices is called a **request device**. Request devices form the layer directly below the general request device in the device hierarchy. Devices themselves, which have the actual I/O resources such as windows and files and which perform I/O processing, form the layer directly below the request **device** layer.

Request Devices

The request devices form the layer directly below the general request device. Request devices have the function of generating and providing to the user actual I/O devices such as windows and files. There are four types of request device corresponding to the device types: the window request device, the file request device, the timer request device, and the string I/O request device.

Actual devices are generated and streams to them acquired by sending messages over a stream to the corresponding request device. See section 9.1, I/O Resource Acquisition", (page 350) for details on acquisition of each request device type. Also note that request devices have abort and attention lines and support abort and attention processing. This is described in detail in section 9.3.3, "Device abort/attention processing", (page 374).

【Hint】 Aborting a request device means to abort the processing corresponding to messages received by the request device. This corresponds to aborting a window creation or a file open, or discarding those messages without executing them.

Devices

Devices form the layer directly below request devices. These are generated and supplied by the request device for the corresponding types. In PIMOS there are four types of device: windows, files, timers, and string I/O devices. Devices own their corresponding actual I/O resources, and perform I/O and other processing according to message communication over a stream. Also note that each device is provided with an abort and an attention line, and can perform abort and attention

processing. This is described in detail in section 9.3.3, "Device abort/attention processing", (page 374).

9.3.2 Methods for device communication

All device interfaces are implemented as message communication over a stream. Normal I/O is performed by sending messages to a device's I/O stream. The format of I/O messages is determined by a set of rules. Devices also have two lines for interrupting I/O based on request from the OS and interrupt from the user.

I/O Message Formats

I/O is performed by sending I/O messages to each device.

Each message consists of a message name, zero or more input arguments, and 1 output argument. Devices execute I/O according to the message name and input argument of the messages they receive, and return the result in the output argument. Value are returned in output arguments according to the following rules.

When the processing corresponding to a message has completed normally:

When only whether that processing succeeded or failed is to be returned:

The atom **normal** is returned in the output argument.

When the value acquired as the result of that processing is to be returned:

A vector which has the atom **normal** as its zeroth element is returned in the output argument. The values acquired as a result of the processing are stored starting in element 1.

When the processing corresponding to a message does not complete normally:

The reason is returned in the output argument. One of the following values is taken:

The atom "abnormal": The processing failed.

The atom "aborted": The processing was aborted.

The atom "disconnected": A network disconnection was detected on an attempt to perform processing during use of an I/O resource on a remote site over a network.

【Caution】 Currently it is not possible to restart I/O processing on a remote device for which a network disconnection has been detected. It is necessary to re-acquire a new device after the network has been established.

Messages that can be sent to PIMOS I/O resources are described according to the mode display described in section 7.1. For details, see section 7.1, "Interpreting the I/O Mode Display", (page 231). Following variable names are used as message output arguments:

^Status: Expresses a message output argument used when only the success or failure of the processing is to be returned for cases where the processing corresponding to the message terminated normally.

^Result: Expresses a message output argument used when the result of the processing is to be returned together with the values for cases where the processing corresponding to the message terminated normally.

Abort and Attention Lines

In addition to the normal message processings, devices require the mechanism which interrupts the operation during the message processing based on request from the OS and the interrupt from the user.

Devices normally processes the messages received over the stream in the order of receiving them. It is required to inform the device, prior to the messages waiting for being processed in the stream, of interrupts the messages being processed, so this stream cannot be used for notifying the interruption.

Thus, all devices which are supplied by PIMOS are provided with the communication path called abort line and attention line, in addition to the stream for receiving the normal messages. An abort line is used to direct the interruption of the processing to the device from the user or the OS, while an attention line is used to inform the user or the OS of the device being interrupted. See next section, "Device abort/attention processing" for details.

9.3.3 Device abort/attention processing

In addition to the stream for sending the normal I/O messages, every device has an abort line which is used to receive abort request from the OS, and an attention line which is used to inform the OS of interrupt from the user. These stream and lines are established at the time of the device creation.

These two lines are implemented as the variables. The OS or the user can interrupt the messages being processed by the device, by instantiating a variable of the abort line to the atom **abort**. Also when the user enters **^C or c+l-c** from the keyboard, a variable of the attention line is instantiated to the atom **attention** by the device, and informs the OS or the user of the interruption (currently only window device is available).

Devices have the following three states:

- Aborted state
- Normal state waiting for the messages
- Waiting state for the completion of a message processing

Because abort and attention lines are implemented as variables, it is not possible to reuse the line which has been used once. So the device, after the atom "abort" has been sent, becomes the state of not being provided with an abort line. This state is called the device state of being aborted. All the messages interrupted by this state and the normal I/O messages received in this state are queued in the device without being executed. It is necessary to re-establish a new abort line and attention line to reset the device to the normal state of waiting for the messages. Re-establishing lines is performed by sending the following messages over the stream.

reset(^Result)

Establishes a new abort line and an attention line.

Establishes new lines Abort and Attention when the message terminates normally, and returns the identification number ID of the interrupted message sequence in Result in the format of **normal(^Abort, Attention, ID)**. The identification number ID expresses the interrupted message sequence which are queued in the device without being executed due to the device state of being aborted just before the message has arrived after the device was aborted.

When a reset message is received during the line being established, the device assumes that an interrupt request has arrived. The device instantiates the existing abort line to the atom "abort", and establishes a new line. After waiting for message success, a new attention line

is established and the old attention line is instantiated to the atom `[]`. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

When the device becomes the state of being aborted, the subsequent message sequence, before the above reset message is received, are queuad in the device without being executed. After the device becomes the normal state of waiting for a message, the user can direct to resend or cancel the interrupted message sequence by the following messages. Resend or cancel can be directed by the time period unit of the device being aborted.

If no messages are sent, the messages being queued are canceled at the time of the device termination.

resend(ID, ^Status)

Resends the interrupted message sequence of the identification number ID. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. english nothing

cancel(ID, ^Status)

Cancels the processing of the interrupted message sequence with the identification number ID. In this case, the output argument of the canceled messages is instantiated to the atom **aborted**. english nothing

If an interruption is informed by the user, but there is no necessity to interrupt the processing, only an attention line can be newly established with the following message.

next_attention(^Result)

Establishes a new attention line.

If the message terminates normally, **normal(Attention)** is returned in Result and a new attention line Attention is established.

If the next_attention message is sent, when the attention line has already been established, an attention line is newly established after the success of the message. The old line is instantiated to the atom `[]`. If the message fails, the old line is available. english nothing

Note that the established attention line is instantiated to the atom `[]` on device termination.

9.3.3.1 Window device abort processing

An window device controls a string (a input remainder string) which has been entered from the window, but has not been returned to the user. See the getb from the window for details of input remainder string ("I/O Messages" in section 378 "Window" on page 378). Therefore, the messages about the abort to the window devices perform the following processings to the input remainder string in addition to the above actions.

reset(^Result)

Initializes an input remainder string to null string. The identification number ID returned as the result of this processing expresses both of the interrupted message sequence and the input remainder string under the control of the window device at the time of being aborted. If this ID is specified to the resend or cancel message, not only the interraptel message sequence but also the input remainder string are recovered or released. If no messages are sent, the input remainder string being stored are released at the time of the window device termination.

resend(ID, ^Status)

Recovers the input remainder string of the identification number ID. If an input remainder string already exists, this string is aborted and the input remainder string of the identification number ID is applied. If an incorrect identification number is specified, the existing input remainder string becomes available.

cancel(ID, ^Status)

Releases the input remainder string of the identification number ID.

9.3.4 Device names for I/O devices

I/O devices in use by the user are managed by the PIMOS resource management mechanism. Users can query PIMOS for I/O device resource state information. The following **device names** are used in the resource state display. Note that the comments displayed with the device name express the information indicated below.

General Request Device

Device name: general_request

Comment: "general_request"

Window Request Device

Device name: window_request

Comment: "window_request"

Window Device

Device name: window

Comment: window, or the buffer name character string specified at window creation.

File Request Device

Device name: file_request

Comment: "file_request"

Input File Device

Device name: read_file

Comment: File name character string

Output File Device

Device name: write_file

Comment: File name character string

Additional File Device

Device name: append_file

Comment: File name character string

Directory Device

Device name: directory

Comment: Directory name character string

Timer Request Device

Device name: timer_request

Comment: "timer_request"

Timer device, when the timer in the FEP is used

Device name: fep_timer

Comment: fep

Timer device, when the system timer is used

Device name: system_timer

Comment: {absolute_id(ABS_ID), relative_id(REL_ID)}
ABS_ID is the absolute node number for the system timer's node, and REL_ID is the relative node number.

String Input/Output Device

Device name: string_io_request

Comment: "string_io_request"

String Input/Output Request Device

Device name: string_io

Comment: Character string for the ¡package name¿##¡class name¿ of the ESP program that defined the device.

【Hint】 These device names and comments can be displayed by either executing the **rs** shell utility from the shell or listener, or by querying the resource states for the tasks for the goals being executed by the listener. See section 2.2, "Shell Utility Programs", (page 28) for details on the **rs** shell utility, and see section 4.1.2.7, "Execution Interruption", (page 69) for details on querying the resource states for the tasks for executing goals.
^L

9.3.5 Windows

The windows provided by PIMOS support the following functions.

- Character I/O
- Allocation of a SIMPOS pmacs buffer on window creation
- Setting the window size based either on the number of characters and lines for a given font, or on mouse commands.
- Setting the window position based either on the position of the upper left corner of the window in screen pixel coordinates, or on mouse commands.
- Aborting processing during I/O either by request from the OS or by interrupt from the user.

To perform I/O using a window, the window must be acquired through the device hierarchy starting by creating a window request device from the general request device, and then by creating the window device.

9.3.5.1 Window request device

A window request device is a device that generates and provides a window device. Also, it provides with the additional information about the window creation.

A window request device, like any other device, has both an abort line and an attention line. Aborting a window request device corresponds to either aborting the window creation processing or aborting without performing some processing corresponding to a message, such as a window

creation request message. The window request device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374 for a description of the control methols and the messages related to abort and attention processing.

The following I/O related messages can be sent to a window request device.

create(Buffer_name, ^Result)

Generates a window device and establishes an abort line and an attention line. Allocates a text buffer which is specified by the name of the text buffer Buffer_name (a character string) at the time of window creation.

When the message terminates normally, returns **normal(^Window, ^Abort, Attention)** in Result. Window expresses a stream to the generated window device, and Abort and Attention express the abort line and attention line, respectively.

If the specified buffer does not exist, a new buffer with this name is generated and allocated to the window. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

create(^Result)

Generates a window device and establishes an abort line and an attention line. A new text buffer with the default name is generated and allocated at the time of window creation.

If the message is terminated normally, **normal(^Window, ^Abort, Attention)** is returned in Result. english nothing

get_max_size(X, Y, Font_pathname, ^Result)

When specifying a font with the path name Font_pathname(string) and coordinates X and Y on the screen in screen pixels, the maximum size of the window, which can be generated assuming that the upper left corner of the window is placed at this coordinates, is returned in the number of characters and lines for the given font.

If the message is terminated normally, **normal(Characters, Lines)** is returned in Result. Characters and Lines express a number of characters and lines for the given font, respectively. english nothing

9.3.5.2 Window

A window device is a device that manages an actual window and performs the I/O processing for that window. A window device has an abort and an attention line and allows abort and attention processing during I/O processing. Control methals and the messages related to abort and attention processing are described in detail in section 9.3.3, "Device abort/attention processing", (page 374).

The I/O related messages that can be sent to a window device can be divided into three types: I/O messages, window attribute setting messages, and window attribute query messages. These messages are described below.

I/O messages

getb(Size, ^Result)

Inputs the strings as far as the carriage return code. The maximum length of the string to be input, Size, must be specified. When the length of the input string exceeds Size, the string with Size length whose trailing part is cut off is returned. The remaining string (called the input

remainder string) after cut off is kept in the device. If an input remainder string already exists when executing getb, returns as a result the partial string of specified size length in the input remainder string, without sending the input request to the window.

If the message terminates normally, **normal(String)** is returned in Result. String expresses the string which is returned complying with the above specifications. english nothing

【Caution】 **Abort and input remainder string:** The input remainder string is treated same as the interrupted message sequence during the abort processing of the window device. That is, the IDs in the messages of reset, resend, and cancel to the window device is the identification number indicating both the interrupted message sequence and the input remainder string. For details, see section 9.3.3.1, "Window device abort processing", on page 375 and section 9.3.3, "Device abort/attention processing", on page 374.

【Hint】 When the user inputs one line from the window, it is originally desirable to return the string as far as the carriage return code however long that line is. PIMOS uses this format to standardize input message protocols with the files and so on.

putb(String, ^Status)

Outputs a character string String on the window. There are no limitations on the length of strings. english nothing

prompt(String, ^Status)

Outputs a character string String on the window. There are no limitations on the length of strings. english nothing

【Hint】 prompt/2 is used to display the prompt for prompting input from the user. Although putb/2 is sufficient for displaying the prompt on the window, another one is provided with to describe a program regardless of the type of I/O devices. Prompt/2 operates in the same way as putb/2 if it is issued to a window, while prompt/2 does nothing if it is issued to a file.

ungetb(String, ^Status)

Outputs a character string String without moving the window input start position. There are no limitations on the length of strings. english nothing

【Hint】 ungetb/2 is used for displaying the guideline for input performed afterwards by the user. If it is issued to a file, all the guideline is regarded as having been deleted, and ungetb/2 does nothing.

flush(^Status)

Immediately outputs the contents of the most recent buffer on the window. english nothing

beep(^Status)

Beeps out. english nothing

Messages for Setting the Window Attributes

set_size(Size, ^Status)

Sets the size of the window.

If Size is specified in the format of char(Characters, Lines), sets the number of characters to Characters and the number of lines to Lines. If atom "mouse" is specified for Size, the size is set using a mouse. english nothing

【Caution】 Specify a window size bigger than the minimum size of a window which PIMOS can generate.

set_position(Position, ^Status)

Sets the location of the window on the screen.

If Position is specified in the format of at(X, Y), the upper left corner of the window is set in screen pixel coordinates (X, Y). If atom "mouse" is specified for Position, the location is set using a mouse. english nothing

reshape(Position, Size, ^Status)

Sets the location and size of the window on the screen.

If Position is specified in the format of at(X, Y), the location of the upper left corner of the window is set in the screen pixel coordinates (X, Y). If atom "mouse" is specified for Position, the location is set using a mouse.

If Size is specified in the format of char(Characters, Lines), sets the number of characters to Characters and the number of lines to Lines. If atom "mouse" is specified for Size, the size is set using a mouse. english nothing

set_title(String, ^Status)

Sets a title to String (string). english nothing

set_font(Pathname, ^Status)

Sets a font with the path name Pathname (string).

【Caution】 The specified font had to keep the size of the window bigger than the minimum size of that which PIMOS can generate even after it has been set.

select_buffer(Buffer_name, ^Status)

Sets a text buffer with the name Buffer_name (string). If the specified buffer does not exist, a new buffer is created with the same name and is allocated to the window. english nothing

activate(^Status)

Displays a window on the screen. english nothing

deactivate(^Status)

Stops displaying a window on the screen. The screen image of the window is maintained and can be displayed again on the screen by sending the message activate(^Status). english nothing

show(^Status)

Makes a window not hidden by other windows. However, if no window is displayed on the screen (in the state of deactivate), it operates in the same way as the message activate(^Status). english nothing

hide(^Status)

Moves a window to the lowest layer of the windows. english nothing

clear(^Status)

Clears the contents of the text buffer of the window. english nothing

Messages for Querying the Window Attributes

get_char_size(^Result)

Returns a window size in the number of characters and lines.

If the message terminates normally, normal(Characters, Lines) is returned in Result. Characters and Lines express the number of characters and lines, respectively. english nothing

get_position(^Result)

Returns the location of the upper left corner of the window in screen pixel coordinates.

If the message terminates normally, normal(X, Y) is returned in Result. X and Y express the screen pixel coordinates (X, Y). english nothing

get_title(^Result)

Returns a title.

If the message terminates normally, normal(String) is returned in Result. String expresses the title (string). english nothing

get_font(^Result)

Returns a font indicated by the path name.

If the message terminates normally, normal(Pathname) is returned in Result. Pathname expresses the path name (string) of the font. english nothing

get_buffer_name(^Result)

Returns the name of a text buffer specified in the window.

If the message terminates normally, normal(Buffer_name) is returned in Result. Buffer_name expresses the buffer name (string). english nothing

9.3.6 Files and Directories

The files provided by PIMOS support the following functions.

- Files have three modes: **read**, **write**, and **append**. Files are opened with one of these modes specified. **Read** is an input mode, **write** is a mode that outputs to a new file, and **append** is a mode that appends to the specified file if it exists.

- Character I/O
- Certain operations on directories

When I/O is performed to a file, the file must be acquired through the device hierarchy starting by creating a file request device from the general request device, and then by creating the file device with one of the modes described above.

9.3.6.1 File request device

A file request device is a device which generates and provides a file device and a directory device. A path must be specified when creating one of these devices. The file request device has the concept of **current directory**, and the path can be expressed as a relative specification from the current directory. See section 9.3.6.2, "Path name interpretation", (page 383) for details on path name interpretation and path specification.

A file request device, like any other device, has both an abort line and an attention line. Aborting a file request device corresponds to either aborting the file open processing or aborting without performing some processing corresponding to a message, such as a file open request message. The file request device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The following I/O related messages can be sent to a file request device.

open(Pathname, read(^Result))

Generates a file device specified in Pathname (string) in the input mode, and establishes an abort line and an attention line. If the message terminates normally, **normal(^File, ^Abort, Attention)** is returned in Result. File expresses the stream to the generated file device, and Abort and Attention express the abort line and the attention line, respectively.

If the specified file does not exist, **abnormal** is returned in Result. english nothing

open(Pathname, write(^Result))

Generates a file device specified in Pathname (a character string) in the output mode, and establishes an abort line and an attention line.

If the message terminates normally, **normal(^File, ^Abort, Attention)** is returned in Result.

If the specified file does not exist, a new file is created and opened. If the specified file exists, an updated file with the same name is created and opened. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

open(Pathname, append(^Result))

Generates a file device specified in Pathname (string) in the append mode, and establishes an abort line and an attention line.

If the message terminates normally, **normal(^File, ^Abort, Attention)** is returned in Result.

If the specified file does not exist, a new file is created and opened. If the specified file exists, the file is opened. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

directory(Pathname, ^Result)

Generates the directory device specified in Pathname (string), and establishes an abort line and an attention line.

If the message terminates normally, **normal(^Directory, ^Abort, Attention)** is returned in Result.

If the specified file does not exist, abnormal is returned in Result. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above. english nothing

set_current_directory(Pathname, ^Status)

Sets the directory specified in Pathname (string) to the current directory. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. english nothing

current_directory(^Result)

Returns the value (string) of the current directory.

If the message terminates normally, normal(Pathname) is returned in Result. Pathname expresses the current directory (string). See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

9.3.6.2 Path name interpretation

It is required to have the path names for identifying files and directories in order to obtain file devices and directory devices. Two types of path names for this specification are available:

- Absolute path name
- Relative path name

PIMOS, similar to SIMPOS, has a tree structure directory. The absolute path name specification is a method of specifying the entire path name from the root directory on the FEP specified when obtaining a file request device, while the relative path name specification is a method of specifying the path name from the directory containing the FEP specified when obtaining a file request device. The directory used for the relative path name specification is called the current directory.

The relationship among the file device, the directory device, and the current directory is as follows:

- The current directory is initially set to the value of the environment variable task:directory (string) at the time of obtaining a file request device.
- After a file request device is obtained, the current directory is kept by the device itself. The current directory can be changed for every file request device.

In PIMOS, the absolute path name and the relative path name are treated as follows:

- Path names beginning with '>' are absolute path names.
- Path names including a "::" are absolute path names.
- Otherwise, path names which are obtained by appending the specified path name string to the value of the current directory are used. When the current directory does not end with '¿', the '¿' is supplied before appending.

9.3.6.3 File

A file device is a device that manages an actual file and performs the I/O processing for that file.

A file device, like any other device, has both an abort line and an attention line. Aborting a file device corresponds to either aborting the I/O processing or aborting without performing some processing corresponding to an I/O message for that file. The file device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The I/O related messages that can be sent to a file device differ with the mode. The messages that can be sent to files open in any mode, the messages that can only be sent to read mode files, and the messages that can only be sent to write mode or append mode files are described below, in that order.

Messages Common to All Files

pathname(^Result)

Returns the full path name of the file.

If the message terminates normally, **normal(Full_pathname)** is returned in Result. Full_pathname expresses the full path name (string). See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

attributes(^Result)

Returns the attributes of the file.

If the message terminates normally, **normal(Owner, UpdateUser, CDate, Ctime, MDate, MTime, LinkCnt, FSize, RSize)** is returned in Result. These express the owner (as a character string), the updating user (as a character string), the creation date (as a character string), the creation time (as a character string), the modification date (as a character string), the modification time (as a character string), the link count (as a integer), the file size (as an integer), and the record size (as an integer), respectively. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

Messages to Input Files

getb(Size, ^Result)

Inputs a character string with the size length of Size from the file.

If the message terminates normally, **normal(String)** is returned in Result. String expresses the input string.

Returns only the part having the effective length, if the size of data which can be derived from the file is less than the size length (if the file ends on the way). In the next input, **end_of_file** is returned in Result, and in the subsequent inputs, **abnormal** is returned. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination cased by a reason other than that describad above. english nothing

【Caution】 The file that is the object of input must be adjusted to be a 16-bit width character string.

prompt(String, ^Status)

Nothing is done.

ungetb(String, ^Status)

Nothing is done.

end_of_file(^Result)

Queries whether the input is ended or not.

If the message terminates normally, **normal(Ans)** is returned in Result. Ans is instantiated with atom "yes" if the input is ended. Otherwise, it is instantiated with atom "no". See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

Messages to Output Files and Append Files

putb(String, ^Status)

Outputs the string String to the file. There are no limitations on the string length.

【Caution】 The element bit length of String had to be 16. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. english nothing

flush(^Status)

Immediately outputs the contents of the recent buffer to the file. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. english nothing

9.3.6.4 Directory

A directory device is a device which performs directory operation for the specified directory.

A directory device, like any other device, has both an abort line and an attention line. Aborting a directory device corresponds to either aborting the directory operation or aborting without performing some processing corresponding to a directory operation message. The directory device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The following directory operation related messages can be sent to a directory device.

Messages to Directories

pathname(^Result)

Returns the full path name of the directory.

If the message terminates normally, **normal(Full_pathname)** is returned in Result. Full_pathname expresses the full path name (string). See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

listing(Wildcard, ^Result)

Returns the names of the files which conform to the wildcard Wildcard (string).

If the message terminates normally, **normal(Filename)** is returned in Result. Filename expresses the list of file names.

If such files do not exist, or if the wildcard is not specified correctly, **abnormal** is returned in Result. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above. english nothing

【Caution】 Wildcard specification including the directory name is not allowed. (This rule is also applied to the following messages.)

delete(Wildcard, ^Status)

Deletes the files with the names conforming to the wildcard Wildcard (string).

If such files do not exist, or if the wildcard is not specified correctly, **abnormal** is returned in Status. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above. english nothing

undelete(Wildcard, ^Status)

Recovers the deleted files whose names conform to the wildcard Wildcard (string).

Even if such files do not exist, **normal** is also returned in Status. If the wildcard is not specified correctly, **abnormal** is returned. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status on abnormal termination caused by a reason other than that described above. english nothing

deleted(Wildcard, ^Result)

Returns the names of the deleted files which conform to the wildcard Wildcard (string).

If the message terminates normally, **normal(Filename)** is returned in Result. Filename expresses the list of file names.

If such files do not exist, or if the wildcard is not specified correctly, abnormal is returned in Result. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above. english nothing

purge(Wildcard, ^Status)

Purges all the files whose names conform to the wildcard Wildcard (string) except those with the maximum version number. The version number cannot be specified in Wildcard. The files which have been deleted when this message is received, are purged.

Even if such files do not exist, **normal** is also returned in Status. If the wildcard is not specified correctly, **abnormal** is returned. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status on abnormal termination caused by a reason other than that described above. english nothing

expunge(^Status)

Expunges all the files which have been deleted. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. english nothing

9.3.7 Timers

The timers provided by PIMOS support the following functions.

- Reporting the current date and time.
- An alarm clock function that informs a user process at a specified time.
- A stop watch function that informs a user process when a specified period has elapsed.
- Functions for aborting alarm clock and stop watch timers.

In using a timer, the timer must be acquired through the device hierarchy starting by creating a timer request device from the general request device, and then by creating the timer device.

9.3.7.1 Timer request device

A timer request device generates and supplies timer devices. Timer devices implemented by the system timer can be acquired on machines that have system clocks built-in, such as PIM and Multi-PSI. Other machine types use the timer in the FEP, as was the case in earlier implementations.

A timer request device, like any other device, has both an abort line and an attention line. Aborting a timer request device corresponds to either aborting the timer device creation processing or aborting without performing some processing corresponding to a timer device creation request message. The timer request device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The following I/O related messages can be sent to a timer request device.

【Caution】 Specifications for the time units and precision of the built-in system timers any more differ between machine types. To avoid user programs becoming machine dependent due to these differences, PIMOS standardizes the time units. Although the default value for time units is the millisecond, the units can be changed by sending a message to the timer device. See section 9.3.7.2, "Timers", (page 388) for details. Also, see section 9.2.3.4, "Timer utilities", (page 371) for details of a unit conversion utility for numeric values that express time counts.

create(^Result)

Creates a timer device and establishes an abort line and an attention line.

When the message terminates normally, **normal(^Timer, ^Abort, Attention)** is returned in Result. Timer is a stream to the generated timer device, and Abort and Attention are the abort and attention lines, respectively.

When executed on a machine that has system timers, the system timer on the node that acquired the general request device is used. When executed on a machine that does not have system timers, the FEP timer on the FEP (or I/O node) held by the general request device is used.

The default value for the time unit of a timer device generated by this message is the millisecond. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

create(Node, ^Result)

Creates a timer device using the timer specified in Node, and establishes an abort line and an attention line.

When executed on a machine that has system timers, this message generates a timer device using the system timer on the specified node if a node number (a non-negative integer) is

specified in Node. If the atom **fep** is specified, this message generates a timer device using the FEP timer on the FEP (or I/O node) held by the general request device.

When executed on a machine that does not have system timers, only the atom **fep** is valid for Node.

If the message terminates normally, **normal(^Timer, ^Abort, Attention)** is returned in Result. Timer is a stream to the generated timer device, and Abort and Attention are the abort and attention lines, respectively. The default value for the time unit of a timer device generated by this message is the millisecond. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

9.3.7.2 Timers

Timer devices have an actual internal clock or stopwatch and provide timer functions to the user. When the built-in system timer is allocated as the actual clock, the timer functions are implemented by sending requests to the PIMOS system timer management process.

A timer device, like any other device, has both an abort line and an attention line. Aborting a timer device corresponds to either aborting the alarm clock or stop watch timer functions or aborting without performing some processing corresponding to a timer operation message. The timer device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The following timer operation messages can be sent to a timer device.

get_date_time(^Result)

Returns the day count from January 1, 1901 to the current day and the time unit count since from previous midnight (i.e., 00:00:00 the current day) to the current time.

If the message terminates normally, **normal(^Date_count, ^Time_count)** is returned in Result. Date_count is a positive integer that expresses a day count and Time_count is a positive integer that expresses a time unit count. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

【Hint1】 The phrase "time unit count" means a length of time measured in specified time units. (Note that time units can be set independently for each timer device.) For example, if the time units for a particular timer device are milliseconds, then time unit counts are values expressed in milliseconds.

【Hint2】 See section 9.2.3, "Timer Utilities", (page 368) for details on utilities that convert the returned time unit counts to date and time expressions.

get_count(^Result)

Returns the time unit counts from 00:00 to the current time.

When the message completes normally, **normal(Count)** is returned in Result. Count is a positive integer that expresses a time unit count. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

on_at(Count, ^Result)

Informs a user process at a specified time by setting the time Count (a positive integer) in preset time unit count.

If the message terminates normally, **normal(Now)** is returned in Result. Now is instantiated with **wake_up** at the specified time. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

on_after(Count, ^Result)

Informs a user process when a specified period has elapsed by setting the time period Count (a positive integer) in preset time unit count.

If the message terminates normally, **normal(Now)** is returned in Result. Now is instantiated with wake_up when a specified period has elapsed. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

where(^Result)

Returns the node number for the system timer used by the current device.

If the message terminates normally, **normal(Node)** is returned in Result. Node is a non-negative integer that expresses a node number. If an FEP timer is used, the atom "**fep**" is returned. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. english nothing

get_time_unit(^Result)

Returns the time unit.

If the message terminates normally **normal(Unit) is returned in Result. Unit is an atom that expresses a time unit, and will be one of the following types:**

sec: Seconds

msec: Milliseconds

msec16: Sixteenths of a millisecond

english nothing

【Hint】 See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result when processing terminates abnormally. Other time unit types may be added in the future.

set_time_unit(Unit, ^Status)

Sets a time unit to the Unit.

Unit is an atom that expresses a time unit, and will be one of the following types:

sec: Seconds

msec: Milliseconds

msec16: Sixteenths of a millisecond

english nothing

【Hint】 See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Status. Other time unit types may be added in the future.

9.3.8 String I/O devices

String I/O devices are devices for manipulating under PIMOS devices defined by the user using ESP on an FEP. String I/O devices are created by specifying the package and class names defined in ESP as the device name.

There is only one I/O processing message provided for string I/O devices: **io(Command, ToString, ^Result)**. When a message is sent specifying a command name in **Command** and the data required to perform the processing in **ToString**, the string I/O device calls the corresponding method in the user defined ESP program. As long as **ToString** is a 16-bit character string, any data can be stored in that string.

If the processing corresponding to the message terminates normally, **normal(FromString)** is returned in Result. The data that is acquired as a result of the processing is stored in a 16-bit character string in FromString. The user can define what is to be stored in arbitrary ESP programs.

See chapter C, "Creating String I/O Devices", (page 452) for a description of device definition techniques in ESP.

In using a string I/O device, the string I/O device must be acquired through the device hierarchy starting by creating a string I/O request device from the general request device, and then by creating the string I/O device.

9.3.8.1 String I/O request device

A string I/O request device generates and provides the user with the string I/O device.

A string I/O request device, like any other device, has both an abort line and an attention line. Aborting a string I/O request device corresponds to either aborting the string I/O device creation processing or aborting without performing some processing corresponding to a string I/O creation request message. The string I/O request device an attention line of has no special meaning. See section 9.3.3, "Device abort/attention processing", (page 374) for a description of the control methods and the messages related to abort and attention processing.

The following I/O related messages can be sent to a string I/O request device.

create(DeviceName, ^Result)

Generates a string I/O device specified by DeviceName (character string) and establishes an abort line and an attention line.

If the message terminates normally, it returns **normal(^StringIO, ^Abort, Attention)** in Result. StringIO expresses a stream to the generated string I/O device, and Abort and Attention expresses an abort line and an attention line, respectively.

If the specified ESP device (class) does not exist, **abnormal** is returned in Result. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above.

【Hint】 The device name is normally specified as "SIMPOS ¡package name¿##¡class name¿".

9.3.8.2 String I/O devices

A string I/O device is a device that performs I/O operations on an ESP device defined by the user. String I/O devices have abort and attention lines, and if the user specifies abort and attention processing in the ESP program, then the device can perform abort and attention processing during I/O processing. See section 9.3.3, "Device abort/attention processing", (page 374 for a description of the messages related to abort and attention processing for string I/O devices under PIMOS.

The following I/O related messages can be sent to a string I/O device.

io(Command, ToFepString, ^Result)

Performs the I/O instructions specified in Command (integer) based on the data specified in ToFepString (string).

If the message terminates normally, **normal(FromFepString)** is returned in Result. FromFepString (string) expresses the result.

If the specified I/O instruction does not exist, or if it does not terminate normally as the result of execution, **abnormal** is returned in Result. See "I/O Message Formats" in section 9.3.2 on page ??? for the value of Result on abnormal termination caused by a reason other than that described above. english nothing

9.3.9 Table of device messages

Table 9.2: Table of PIMOS Device Messages

General Request Device I/O Messages:

Message	Meaning
window(^Result)	Generates a window request device, Result=normal(^WinReq,^Abort,Att)
file(^Result)	Generates a file request device, Result = normal(^FilReq, ^Abort, Att)
timer(^Result)	Generates a timer request device, Result = normal(^TimReq, ^Abort, Att)
string_io(^Result)	Generates a string I/O request device, Result=normal(^StrioReq,^Abort,Att)

Window Request Device Messages:

reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)
resend(ID, ^Status)	Retries the message sequence for the id number ID.
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.
next_attention(^Result)	Connects an attention line, Result = normal(Att)
create(Buffername, ^Result)	Creates a window, Result = normal(^Window, ^Abort, Att)
create(^Result)	Creates a window, Result = normal(^Window, ^Abort, Att)
get_max_size(X,Y,Pathname,^Result)	Acquires the maximum size for a window, Result=normal(Chars,Lines)

Window Device Messages:

reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)
resend(ID, ^Status)	Retries the message sequence for the id number ID.
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.
next_attention(^Result)	Connects an attention line, Result = normal(Att)
getb(Size, ^Result)	Inputs a character string as top as carriage return code, Result = normal(String)
putb(String, ^Status)	Outputs the character string String.
prompt(String, ^Status)	Outputs the character string String.
ungetb(String, ^Status)	Outputs the character string String without moving the input start position.
flush(^Status)	Immediately outputs the most recent contents of the buffer to the window.
beep(^Status)	Rings the bell.
set_size(Size, ^Status)	Sets the size of the window.
set_position(Position, ^Status)	Sets the position of the window.
reshape(Position, Size, ^Status)	Sets the size and position of the window.
set_title(String, ^Status)	Sets the window title to the character string String.
set_font(Pathname, ^Status)	Sets the window font to the character string Pathname.
select_buffer(Buffername, ^Status)	Sets the name of the text buffer to the character string Buffername.
activate(^Status)	Displays the window on the screen.
deactivate(^Status)	Stops display of the window on the screen.
show(^Status)	Makes the window visible (i.e., not hidden by any other window) on the screen.
hide(^Status)	Moves the window to the lowest layer of windows on the screen.
clear(^Status)	Deletes the contents of the text buffer.
get_char_size(^Result)	Returns the size of the window as the number of characters and the number of lines, Result = normal(Chars, Lines)
get_position(^Result)	Returns the position of the upper left corner of the window in screen pixel coordinates, Result = normal(X, Y)
get_title(^Result)	Returns the window title, Result = normal(String)
get_font(^Result)	Returns the window font, with path name Result = normal(Pathname)
get_buffer_name(^Result)	Returns the text buffer name, Result = normal(Buffername)

File Request Device Messages:

Message	Meaning	P
reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	
resend(ID, ^Status)	Retries the message sequence for the id number ID.	
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	
next_attention(^Result)	Connects an attention line, Result = normal(Att)	
open(Pathname, read(^Result))	Opens an input file Result = normal(^File, ^Abort, Att)	
open(Pathname, write(^Result))	Opens and output file Result = normal(^File, ^Abort, Att)	
open(Pathname, append(^Result))	Opens a file for append Result = normal(^File, ^Abort, Att)	
directory(Pathname, ^Result)	Creates a directory device Result = normal(^Dir, ^Abort, Att)	
set_current_directory(Pathname,^Status)	Sets the current directory.	
current_directory(^Result)	Returns the current directory, Result = normal(Pathname)	

File Device Messages:

reset(^Result)	Connects abort and attention lines Result = normal(^Abort, Att, ID)	
resend(ID, ^Status)	Retries the message sequence for the id number ID.	
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	
next_attention(^Result)	Connects an attention line, Result = normal(Att)	
pathname(^Result)	Returns a full pathname, Result = normal(Pathname)	
attributes(^Result)	Returns the file attributes, Result = normal(Owner, UpdateUser, Date, Time, Date1, Time1, LinkCnt, FileSize, RecSize)	
getb(Size, ^Result)	Inputs a character string of size Size, Result = normal(String)	
prompt(String, ^Status)	Does nothing.	
ungetb(String, ^Status)	Does nothing.	
end_of_file(^Result)	Queries the end of file state, Result = normal(Ans)	
putb(String, ^Status)	Outputs the character string String.	
flush(^Status)	Immediately outputs the most recent contents of the buffer.	

Directory Device Messages:

reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	
resend(ID, ^Status)	Retries the message sequence for the id number ID.	
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	
next_attention(^Result)	Connects an attention line, Result = normal(Att)	
pathname(^Result)	Returns a full pathname, Result = normal(Pathname)	
listing(Wildcard, ^Result)	Returns file names, Result = normal([Pathname, ...])	
delete(Wildcard, ^Status)	Deletes files.	
undelete(Wildcard, ^Status)	Undeleted files.	
deleted(Wildcard, ^Result)	Returns the names of deleted files, Result = normal([Pathname, ...])	
purge(Wildcard, ^Status)	Permanently removes all copies except the latest version for the specified files in one operation.	
expunge(^Status)	Permanently removes deleted files.	

Timer Request Device Messages:

Message	Meaning	Page reference
reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	374
resend(ID, ^Status)	Retries the message sequence for the id number ID.	375
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	375
next_attention(^Result)	Connects an attention line, Result = normal(Att)	375
create(^Result)	Generates a timer, Result = normal(^Timer, ^Abort, Att)	387
create(Node, ^Result)	Generates a timer on Node, Result = normal(^Timer, ^Abort, Att)	387

Timer Device Messages:

Message	Meaning	Page reference
reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	374
resend(ID, ^Status)	Retries the message sequence for the id number ID.	375
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	375
next_attention(^Result)	Connects an attention line, Result = normal(Att)	375
get_date_time(^Result)	Returns date and time unit count values, Result = normal(Date, Time)	388
get_count(^Result)	Returns a time unit count, Result = normal(Time)	388
on_at(Count, ^Result)	Informs the user at the specified time, Result = normal(WakeUp)	388
on_after(Count, ^Result)	Informs the user when a specified period has elapsed, Result = normal(WakeUp)	389
where(^Result)	Returns the node number for the timer in use, Result = normal(Node)	389
get_time_unit(^Result)	Returns the time unit, Result = normal(Unit)	389
set_time_unit(Unit, ^Status)	Sets the time unit.	389

String I/O Request Device Messages:

Message	Meaning	Page reference
reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	374
resend(ID, ^Status)	Retries the message sequence for the id number ID.	375
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	375
next_attention(^Result)	Connects an attention line, Result = normal(Att)	375
create(DeviceName, ^Result)	Generates a string I/O device, Result = normal(^StringIO, ^Abort, Att)	390

String I/O Device Messages:

Message	Meaning	Page reference
reset(^Result)	Connects abort and attention lines, Result = normal(^Abort, Att, ID)	374
resend(ID, ^Status)	Retries the message sequence for the id number ID.	375
cancel(ID, ^Status)	Cancels the message sequence for the id number ID.	375
next_attention(^Result)	Connects an attention line, Result = normal(Att)	375
io(Command, String, ^Result)	Executes the specified I/O instruction based on String, Result = normal(FromFepString)	391

9.4 Standard I/O

9.4.1 Standard I/O function

9.4.1.1 Relationship with the resource tree

A mechanism for registering the three I/O streams "standard input", "standard output", and "standard interaction" for all tasks is built into the resource tree. The mechanism that provides in advance streams that perform standard I/O to general user tasks using this registry mechanism, and performs such services as allowing those streams to be shared by tasks or switching the tasks that use the streams, is called the "standard I/O function".

Since the resource tree only provides functions that maintain standard I/O for tasks, it is necessary to provide mechanisms for sharing these streams between tasks and for exclusivity control for these streams separately. These functions are provided by the standard I/O function, which is

also referred to as the standard I/O server.

9.4.1.2 I/O stream sharing and exclusivity control

Stream sharing between tasks refers to accessing and performing I/O operations on a single resource stream from multiple tasks at the same time. Sharing occurs when a standard I/O stream used by a parent task is inherited by a child task and then used by that child task.

If pairs of tasks that are sharing a stream arbitrarily performed I/O operations on that stream at the same time, results (i.e., I/O operations) that were unexpected by the user programs that were using that stream could occur.

For example, if multiple tasks output messages to a single window at the same time, a user observing that window would not be able to tell which task output which message. Inversely, if multiple input requests were sent, the user would not know to which process the results of the window input would be sent.

To avoid such situations, it is necessary to perform both traffic control of messages flowing on shared streams as well as exclusivity control so that a single shared stream only receives I/O messages from at most one process during a predetermined time period. It is the standard I/O function that provides this exclusivity control mechanism, i.e., the mechanism that switches tasks using a shared stream.

9.4.1.3 Allocation and switching of I/O streams

It is necessary to specify at task activation time what sort of streams are actually allocated as standard I/O to each task. It is also necessary to provide exclusivity control for the allocated streams, i.e., to control when the switching operation is performed. The program that performs the management of these control types is the program that controls and manages general user programs, i.e., the shell or the listener. When activating a task that will execute a user program, standard I/O streams are allocated for that task, and if those streams are shared, it will be necessary to switch between tasks according to user requests, such as attention requests.

9.4.1.4 I/O operations using I/O streams

General user programs that are activated by the shell or listener can easily access resources such as windows and files by using a predetermined protocol, the standard I/O protocol.

9.4.2 Techniques for input and output using standard I/O

There are two techniques for a general user program to acquire the standard I/O streams that were allocated to that task.

The first is a general method for the case where the user program does not perform attention control over the standard I/O streams, but rather leaves attention control to the shell or listener.

The other is the case where the user program does perform attention control over the standard I/O streams. This is a special case, and in this case the shell or listener will use this function to handle the task control for the child task.

All standard I/O streams are acquired using the predicate shoen:raise/3. The following points require attention when acquiring standard I/O streams.

- The same standard I/O stream may be acquired multiple times. However, those streams are simply merged, and exclusivity control is not performed. Therefore, the merged result and the order of messages sent to the actual resource are not guaranteed.

- If either a standard input stream, a standard output stream, or a standard interaction stream is used, a standard I/O stream may not be used. (Note that the inverse is also true.) If this combination of streams is used, the operation during attention processing is not guaranteed.
- When an I/O stream is acquired by a program, "with_macro pimos." must appear in that program.

9.4.2.1 When attention control is not performed (general programs)

Stream Access

When a standard I/O stream is for the technique that does not perform attention control, a buffering filter (I/O buffer) that has functions for blocking and deblocking for reducing communications overhead is automatically inserted in each standard I/O stream. Therefore, the resource can be accessed using a message protocol that supplies an I/O buffer.

1. Access to a standard input stream.

   shoen:raise(pimos_tag#task, get_std(in), STD_IN)

2. Access to a standard output stream.

   shoen:raise(pimos_tag#task, get_std(out), STD_OUT)

3. Access to a standard interaction stream.

   shoen:raise(pimos_tag#task, get_std(inter), STD_INTER)

Communications Protocols

1. Standard input (STD_IN)
   The message protocol provided by `buffer:input_filter`.
2. Standard output (STD_OUT)
   The message protocol provided by `buffer:output_filter`.
3. Standard interaction (STD_INTER)
   The message protocol provided by `buffer:interaction_filter`.

See section 9.2.1, "Buffer" (page 355) for details of the message protocol.

9.4.2.2 When attention control is performed (special programs)

Stream Acquisition

Instead of acquiring `std_in`, `std_out`, and `std_inter` individually, the stream `std_io`, which provides all of those functions in one stream, must be used.

When acquired for methods that perform attention control, a standard I/O stream in the original state in which an I/O buffer has not been inserted is acquired. Special messages such as `reset/1` can be sent to this stream.

1. Access to a standard I/O stream

shoen:raise(pimos_tag#task, get_std(io), STD_IO)

Attention control is left to the parent task in the state immediately following stream acquisition. To actually control attention processing in the user task (the task that acquired `STD_IO`) the user task must acquire the attention variable by sending a `reset/1` message to `STD_IO`. This allows attention control (or processing) rights to be acquired.

When an attention occurs in the state in which the attention variable has been acquired, the atom `attention` is unified with the acquired attention variable. Also note that if I/O completes in this state, i.e., if the I/O stream is closed, then `[]` will be unified with the attention variable.

Once attention control rights have been acquired they can never be released. However, it is possible to inform the parent task that an attention has occurred. The parent task will be informed of the result when a `switch(Status)` message is passed to the I/O stream. The manner in which the parent task is informed is described in section ?? (page ??).

Communications protocols

1. Messages sent to a standard input stream

   getb(Size, ^StringWithStatus)

   prompt(String, ^Status)

   ungetb(String, ^Status)

2. Messages sent to a standard output stream

   putb(String, ^Status)

   flush(^Status)

3. Messages sent to a standard interaction stream

   igetb(Size, ^StringWithStatus)

   iprompt(String, ^Status)

   iungetb(String, ^Status)

   iputb(String, ^Status)

   iflush(^Status)

   Please refer to the description of I/O control in section 9.3 (page 372) for a detailed description of each of these messages.

   【Caution】 The functions of the igetb/2, iprompt/2, iputb/2, and iflush/1 messages are the same as those for the getb/2, prompt/2, putb/2, and flush/1 messages if the destination is anything other than standard interaction.

4. Abort/attention control messages

   reset(^AbortAttentionAndIdWithStatus)

   Even in cases when the user has acquired std_io with the intention of performing attention processing himself, any attentions that occur prior to the point that this message is sent

will be processed by the shell. That is, the user first acquires attention control rights by sending a reset/1 message. Thereafter, the user can perform the desired attention processing using reset and the following 4 messages.

【Caution】 Once attention control rights have been acquired they can never be released.

switch(^Status)

This message is provided so that a user who has detected an attention can forcibly switch his own job into the background.

next_attention(^AttentionWithStatus)

resend(Id, ^Status)

cancel(Id, ^Status)

Please refer to the description of I/O control in section 9.3 (page 372) for a detailed description of the reset/1, next_attention/1, resend/2, and cancel/2 messages.

9.4.2.3 Compatibility with earlier versions

Other than the format of the raise message, the techniques described above are upwardly compatible. All format differences between the new and old raise messages are described below.

- access to Standard input stream acquisition

  shoen:raise(pimos_tag#task, get_std(in), STD_IN) (new)

  shoen:raise(pimos_tag#shell, get_std_in, STD_IN) (old)

- access to Standard output stream acquisition

  shoen:raise(pimos_tag#task, get_std(out), STD_OUT) (new)

  shoen:raise(pimos_tag#shell, get_std_out, STD_OUT) (old)

- access to Standard interaction stream acquisition

  shoen:raise(pimos_tag#task, get_std(inter), STD_INTER) (new)

  shoen:raise(pimos_tag#shell, get_std_inter, STD_INTER) (old)

- access to Standard I/O stream acquisition

  shoen:raise(pimos_tag#task, get_std(io), STD_IO) (new)

  shoen:raise(pimos_tag#shell, get_std_io, STD_IO) (old)

It is also possible to acquire streams with the old format of the raise message. However, this is implemented by inserting a goal that re-sends the message in the new format and thus re-executes the raise. As a result, there is a significant delay in the acquisition. We recommend performing the raise in the new format if at all possible.

9.4.3 Assigning standard I/O to a task and managing that task

For a task to actually use standard I/O, it is necessary to specify standard I/O in advance when that task is started. Therefore, programs (such as the shell or the listener) that start general user programs as tasks must assign standard I/O to the child task when that child task is started.

9.4.3.1 Assigning standard I/O to a task

Assignment of standard I/O is performed when a task is started. In particular, this assignment is specified in the task activation message `execute` sent to the `general_request` stream. One of the arguments to the `execute` message (see section 11.1.9.2 on page 424) is the `stdio` parameter, and it is this parameter that is used to specify input and standard output.

stdio([{Type, RSC}, {Type, RSC}, ...], ^Result)

Here `Type` expresses the type of standard I/O to be specified, and "type" must be one of the following atoms.

in : Specifies a resource to be used as standard input.

out : Specifies a resource to be used as standard output.

inter : Specifies a resource to be used as standard interaction.

in_out : Specifies a resource to be used as standard I/O.

in_inter : Specifies a resource to be used as standard input and standard interaction.

out_inter : Specifies a resource to be used as standard output and standard interaction.

in_out_inter : Specifies a resource to be used as standard input, standard output, and standard interaction.

Also, `RSC` is a vector of the form `{Server, InInfo, OutInfo}`. Server is an atom (such as window_request or file_request) that expresses a server name registered in PIMOS. `InInfo` holds a variety of resource initialization parameters, and is sent to the corresponding server. OutInfo is instantiated after the server assigns the standard I/O resource.

In the standard I/O functions provided in the current PIMOS, `InInfo` must have one of the following formats.

1. For window_request:
   - ¡window title character string¿
   - {¡window title character string¿ }
   - {¡window title character string¿, AT, Char }
   - {¡window title character string¿, AT, Char, ¡font name character string¿ }

     Where `At` and `Char` are window position and size specifications.

     (a) Specification of the At parameter:

     at(X, Y) : Specifies the coordinate position of the window's left upper corner.

     mouse : Specification using the mouse.

     (b) Specification of the Char parameter:

     char(X, Y) : Specifies the size of the window in number of characters.

     mouse : Specification using the mouse.
2. For file_request:

- ¡file name character string¿
- { ^ ¡file name character string¿ }

【Caution】 Extensions to these specification formats will be added in the future.

When the specification completes normally, a value of the format "normal(ID, Info)" will be unified with `Result`. ID is an identifier to be used when using the standard I/O switching function, and Info is information concerning the inheritance state of the parent task's standard I/O.

Please see section 11.1.9.2 (page 425) for details of the specification method and returned values.

【Example】 This example shows the actual code used for specification.

```
.....
shoen:raise(pimos_tag#task, general_request, GR),
GR = [execute(CODE, ArgVector, [stdio(S_PARAM, S_Result)], Result),
    S_PARAM = [in({window_request,
                   {"Test1-input",0,300,20,5,"font:test_11"}, OUT1}),
               out({window_request,
                   {"Test1-output",0,500,20,5,"font:test_11"}, OUT2}),
               inter({window_request,
                   {"Test1-inter",0,700,20,5,"font:test_11"}, OUT3})],
.....
```

When standard I/O is specified with `execute`, a number is assigned to standard I/O and returned in an `execute` argument. In the example above, this is the `ID` returned as `normal(ID, Info)` in `S_Result`. When multiple child tasks are started at the same time, this number is used to specify which child task is allowed to access the shared resource.

However, resource control using this number cannot be performed until the standard I/O information returned along with this number becomes a ground value containing no undefined variables whatsoever. If an attempt to do so is made before that point, then it is possible that those messages will be ignored.

9.4.3.2 Controlling standard I/O that was assigned to a task

Control of (shared) standard I/O given to a child task is performed by acquiring a control stream as shown below and sending control messages to that stream.

- Access to a standard I/O control stream

shoen:raise(pimos_tag#task, get_std(switch), STD_SW)

A child task cannot use the parent task's standard I/O stream simply by assigning (i.e., sharing) that stream to the child when the child task is created. Although the child task is free to send messages to that stream, until the parent task has given authorization to use that stream to the child task, messages sent by the child task will not be processed, and the child task will wait on that processing.

【Caution1】 This limitation does not apply prior to the point a standard I/O control stream has been acquired, and up until that point all child tasks can perform I/O operations using the parent task's standard I/O streams. The only situation in which a child task's usage authorization is limited is when the parent task has acquired a standard I/O control stream.

【Caution2】 Child tasks created before the parent task acquired the standard I/O control stream fall under that control at the point the standard I/O control stream was acquired. As a result, from that point onward use of standard I/O is limited until the parent task gives usage authorization in advance of that use.

9.4.3.3 Control of standard I/O

Even if the parent task has acquired STD_SW, if the parent task takes no actions in that state, no changes in conditions occur. A message must be sent to allow a child task to use a parent task's stream. The messages used for that purpose are `attach/2` and `attach/3`.

By sending an `attach` message it is possible to assign exclusive use of a standard I/O stream to the child task with the ID specified in the attach, for the period from the point the `Status` argument in that message is bound until the point the `Detect` argument is bound.

The fact that the standard I/O usage authorization assigned to a child task has been returned is reported to the parent task by unifying a value with `Detect`. For example, if the atom `detected` is unified with the argument `Detect`, that indicates that the use of standard I/O has been suspended and returned to the parent task due to the detection of an attention by the specified child task. Also, a similar result can be acquired by a child process that has acquired control rights by sending a `reset` message, has sent a switch message. In this state, the standard I/O stream will not be connected to any process.

A similar result can be acquired without use of means such as attention processing if the parent task sends a `detach/1` message to the control stream. However, in this case the atom `disconnected` will be returned in the `Detect`.

There are cases where a single standard I/O stream is actually attached to multiple resources. For example, when a parent task's standard input is attached to file A, its standard output to file B, and its standard interaction to window C, if a child task inherits all the parent task's standard I/O, then an `attach` message will be sent to each of these 3 resources. (Strictly speaking, a message will be sent to the process managing the exclusivity control of the resource.) It is possible that each of the `attach` messages sent to the three resources returns a different result. In such a case, those results will be adopted in a certain priority order, and returned as the result of the attach message sent by the user. The numbers attached to the `Status` and `Detect` variables in the following are these priorities.

1. Attach message

   Standard I/O will be in a floating state (not connected anywhere) from creation to the point this attach is performed.

   attach(ID, ^Status, ^Detect)

   Enables the standard I/O specified by ID. All others are disabled.

   attach(^Status, ^Detect)

   Enables the most recently created standard I/O. All others are disabled.

   - ID = ¡non-negative integer¿
   - Status =
     - abnormal: Switching failed. (device dependent)
     - non_existent: No standard I/O corresponding to the specified ID exists.
     - normal: Switching succeed.

- Detect =
  - error: Standard I/O error
  - detected: attention detected | switch detected
  - non_existent: No standard I/O corresponding to the specified ID exists.
  - deserted: Standard I/O stream discarded.
  - disconnected: detach detected
  - closed: The standard I/O stream was closed. Hereafter "closed" will be returned no matter what is sent.

2. Detach messages

detach(^Status)

Disables all standard I/O.

- Status = normal (| abnormal)

detach(ID, ^Status)

Disables the standard I/O specified by ID, if enabled.

- ID = ¡non-negative integer¿
- Status =
  - abnormal: Switching failed.
  - non_existent: No terminal corresponding to the specified ID exists.
  - non_execute: The terminal corresponding to the specified ID was already disabled.
  - normal: Switching succeed.

9.4.4 Standard message output

In addition to the standard input stream, standard output stream, and standard interaction stream provided by the resource tree and the standard I/O functions, programs started from under shell management can use an I/O stream called standard message output. Please see the discussion of the shell in section 2.1 (page 7) for a description of the operation of standard message output.

Stream Access

A buffering filter that has functions for blocking and deblocking to reduce communications overhead is automatically inserted in the standard message output stream when the stream is acquired. (See section 9.2, "I/O Utilities", on page 355.)

1. Access to a standard message output stream

shoen:raise(pimos_tag#shell, get_std_mes, MesStream)

Communications Protocols

1. Standard message output (STD_MES)
   The message protocol provided by `buffer:output_filter`

9.4.5 Programming examples using standard I/O

This section presents several simple programming examples using standard I/O. Note that programs that use standard message output must be started under shell management.

Tee Program

This program copies the contents of standard input to both standard output and standard message output.

```
:- with_macro pimos.
:- module tee.
:- public go/0.

go :- true |
    shoen:raise(pimos_tag#task, get_std(in), In),
    shoen:raise(pimos_tag#task, get_std(out), Out),
    shoen:raise(pimos_tag#shell, get_std_mes, Mes),
    In = [prompt(string#"TEE>"), getl(Line) | NIn],
    loop(Line, NIn, Out, Mes).

loop(Line, In, Out, Mes) :- true |
    ( string(Line, L, _), L > 0 ->
        In = [prompt(string#"TEE>"), getl(NLine) | NIn],
        Out = [putl(Line), flush(_) | NOut],
        Mes = [putl(Line), flush(_) | NMes],
        loop(NLine, NIn, NOut, NMes) ;
    otherwise ;
      true -> In = [],
        Out = [flush(Stat1) | NOut], waitStatus(Stat1, NOut, []),
        Mes = [flush(Stat2) | NMes], waitStatus(Stat2, NMes, []) ).

waitStatus(Stat, In, Out) :- wait(Stat) | In = Out.
```

Echo Program

This program writes the terms in its argument to standard output.

```
:- with_macro pimos.
:- module echo.
:- public go/1.

go(Arg) :- true |
    shoen:raise(pimos_tag#task, get_std(out), Out),
    Out = [putt(Arg), nl, flush(Stat) | NOut],
    waitStatus(Stat, NOut, []).

waitStatus(Stat, In, Out) :- wait(Stat) | In = Out.
```

Chapter 10

Program Management

PIMOS manages program-related information, such as atoms and modules in its databases. When accessing one of the databases, you must make request to PIMOS, just the same way to access an I/O device. Each access path to these databases are managed as the resources defined by PIMOS.

On the other hand, predicates that can be called from user programs are provided for the module generation and management support programs, such as the compilers, provided by PIMOS.

10.1 Atom Management

10.1.1 Atom cache database

The atom cache database in a utility program to access the database which stores the associations between atoms and their print names.

Since PIMOS manages these associations in a unique database, it is basically necessary to always access one single database. However, this leads to congestion in access, causing a bottleneck of efficiency. If there are many accesses to the database, it is better to access a database cache for each node (processor or cluster). This program creates such kind of database cache.

The remaining part of this section uses the following meanings for the values returned in the argument named "**Status**" in predicates, unless otherwise noted:

normal(Object): Normal end. The extracted information is unified with Object.

abnormal: Abnormal end.

non_existent(Info): No corresponding atom exists. The information on what was to be extracted is returned in Info.

10.1.1.1 Creation

atom_table:cache_table(AtomDev)

Acquires a stream to the atom device and returns the stream to that object in AtomDev.

10.1.1.2 Access by predicate

The following predicates are available for a single access try:

atom_table:get_atom(AtomName, ^Status)

Returns the atom corresponding to the atom name specified in AtomName (a string) in the Object of Status.

atom_table:get_atom_name(Atom, ^Status)

Returns the print name (a string) associated with Atom (an atom) in the Object of Status.

atom_table:intern(AtomName, ^Status)

Makes an atom that has the print name specified in AtomName (a string), and returns it in the Object of Status. If this atom has already been registered, the associated atom is returned.

【Caution】 If access is necessary many times, the use of these predicates makes results in a very low efficiency.

10.1.2 Atom devices

These are databases that store the associations between atoms and their print names. At the point that PIMOS is initialized, a cache process is created on each node (processor or cluster), and cache process streams are returned for nodes for which there was a stream acquisition request.

The remaining part of this section uses the following meanings for the values returned in the argument named "**Status**" in messages, unless otherwise noted:

normal(Object): Normal end. The extracted information is unified with Object.

abnormal: Abnormal end.

non_existent(Info): No corresponding atom exists. The information on what was to be extracted is returned in Info.

10.1.2.1 Creation

An atom device is created by sending the following message to an atom request device.

atom(^Status)

Crates an atom device. If the create succeeded, "**normal(AtomDev, ^Abort, Attention)**" is returned in Status, and if it failed, "**abnormal**" is returned. Just as with other devices, Abort is the abort line, and Attention is the attention line. AtomDev is a stream to the created atom device.

10.1.2.2 Protocol

get_atom(AtomName, ^Status)

Returns the atom corresponding to the atom name specified in AtomName (a string) in the Object of Status.

get_atom_name(Atom, ^Status)

Returns the print name (a string) associated with Atom (an atom) in the Object of Status.

intern(AtomName, ^Status)

Makes an atom that has the print name specified in AtomName (a string), and returns it in the Object of Status. If this atom has already been registered, the associated atom is returned.

10.2 Module Management

10.2.1 Module cache database

The module cache database is a utility program to access the database which stores the associations between modules and their names (atoms). A module is a KL1 object and the entity of executable codes.

The package function is provided in PIMOS to support multiple module name spaces. Since this function is provided, the package name specified in the "task:package" environment variable is used as the default package when the package is not explicitly specified in the database access specifications shown below. See the description of packages later in this section for details on packages as used in PIMOS.

Package inheritance is performed for the messages and predicates in the protocols described below. The package inheritance function is not used for module update or deletion operations.

【Hint】 Although modules such as the pimos::buffer module inherited from pimos in a user package, e.g., "me", can be specified as me::buffer when acquiring code, they must be specified explicitly by, e.g., "pimos::buffer", when deleting. (However, note that in this example only the super user "super_pimos" can delete this module due to the protection mechanism.)

get_module/2, get_module/3, get_code/4, get_code/5 (message)

dynamic_link/3 (predicate)

The remaining part of this section uses the following meanings for the values returned to the argument named "**Status**" in predicates or messages, unless otherwise noted:

normal: Normal end.

normal(Object): Normal end. The extracted information is unified with Object.

abnormal: Abnormal end

non_existent: No corresponding module exists.

protection_violation: An attempt was made to update a protected module, e.g., a PIMOS module.

Also note that the ModuleSpec input argument to these predicates and messages is either a three-element vector (a compound term) of the type **Package::ModuleName** ({**(::), Package, ModuleName**} in vector notation) or simply the atom ModuleName. Here Package is a package name atom, and ModuleName is a module name atom.

【Caution】 Although the word "cache" is used with these module names, no actual caching function is included. This is because it is difficult to maintain consistency since modules are converted generally. We are considering introducing a caching function in the future.

10.2.1.1 Creation

module_table:cache_table(Stream)

Creates a cache for the module database and returns a stream to that object in Stream.

10.2.1.2 Protocol

set_default_package(Package, ^Status)

Sets the default package. Following the execution of this message, the atom specified in Package

is used as the default package when the package is not specified explicitly. See page 406 for a description of **Status**.

【Caution】 No confirmation of the existence of the package specified with this message is performed at the message processing stage. Therefore, errors will occur during the actual package access if a non-existent package is specified.

get_default_package(^Status)

Returns the previously specified default package. See page 406 for a description of Status.

define_package(Package, Super, Protection, ^Status)

Generates a new package with the name specified by Package (an atom). Super (an atom) is the name of the parent package, and must be the name of an existing package. Protection (an integer) is the protection mode value. The name of the parent package and the name of the new package must be different. See page 406 for a description of Status. (See section refresource, "Resource Acquisition and Management", (page 419) for a description of protection modes.)

【Caution】 Only the super user "super_pimos" can create new packages. See the section on packages below for a descriptions of the method for creating new packages.

remove_package(Package, ^Status)

Deletes the existent package specified by Package (an atom). See page 406 for a description of Status.

【Caution】 Only the super user "super_pimos" can delete packages.

protect_package(Package, Protection, ^Status)

Changes the protection of the existent package specified by Package (an atom). Protection is the new protection mode (an integer) to be set. (See section 11.1, "Resource Acquisition and Management", (page 419) for a description of protection modes.) See page 406 for a description of Status.

get_super_package(Package, ^Status)

Retrieves the name (an atom) of the parent package of the package specified by Package (an atom). See page 406 for a description of Status.

get_package_info(Package, ^Status)

Retrieves the protection information for the package specified by Package (an atom). (See section 11.1, "Resource Acquisition and Management", (page 419) for a description of protection modes.) A compound term of the form {{**OwnerID, GroupID**}, **Protection**} is returned in the Object of Status. Here, OwnerID is the owner ID of the package (an integer), GroupID is the group ID of the package (an integer), and Protection is the protection mode for the package (an integer). See page 406 for a description of Status.

get_all_packages(^Status)

Retrieves information on all packages currently registered. A three-element vector of the form {Package, Status1, Status2} is returned in the Object of Status. Information on the parent package is returned in Status1, and Information concerning the protection is returned in Status2.

Package: The package name (an atom)

Status1: The same as the Status returned by get_super_package/2.

Status2: The same as the Status returned by get_package_info/2.

get_module(Package, ModuleName, ^Status)

Returns the module with the name specified by Package (an atom) and ModuleName (an atom) in the Object of Status. See page 406 for a description of Status.

get_module(ModuleSpec, ^Status)

Returns the module specified by ModuleSpec in the Object of Status. See page 406 for a description of Status. See page 406 for a description of ModuleSpec.

get_module_in_the_package(Package, ModuleName, ^Status)

Returns the module with the name specified by Package (an atom) and ModuleName (an atom) in the Object of Status. See page 406 for a description of Status. Does not search parent packages, i.e., no package inheritance is performed.

get_module_in_the_package(ModuleSpec, ^Status)

Returns the module specified by ModuleSpec in the Object of Status. Functions identically to get_module_in_the_package/3. See page 406 for a description of Status and page 406 for a description of ModuleSpec.

get_code(Package, ModuleName, Predicate, Arity, ^Status)

Returns the code specified by the package Package (an atom), the module name ModuleName (an atom), the predicate Predicate (an atom), and the number of arguments Arity (a non-negative integer) in the Object of Status. If the module exists but the predicate corresponding to that module does not exist, a vector of the form **non_existent(code(^Module, ^Predicate, ^Arity))** is returned in Status. See page 406 for a description of Status.

【Hint】 The reason a vector of this form is returned is so that information such as the module name can be passed in the exception information when the returned result is passed as an argument to the built-in predicate apply.

get_code(ModuleSpec, Predicate, Arity, ^Status)

Returns the code for the module specified by ModuleSpec. Functions identically to get_code/5. See page 406 for a description of Status and page 406 for a description of ModuleSpec.

update_module(Package, ModuleName, Module, ^Status)

Updates the module specified by Package (an atom) and ModuleName (an atom) to be the new module specified by Module. If there is no module corresponding to ModuleName the module is simply added. See page 406 for a description of Status.

【Caution1】 If a module with the same name as the module to be updated exists in a higher level package, then the new module will simply be registered in the specified package.

【Caution2】 Updating the database does not imply update of the module itself. Modules already accessed from the database remain unchanged even if the database is updated, and, as long as calls do not use the dynamic link mechanism described below, the call destinations of goals being executed will not be updated.

update_module(ModuleSpec, Module, ^Status)

Updates the module specified by ModuleSpec to be the new module specified by Module. Functions identically to update_module/4. See page 406 for a description of Status and page 406 for a description of ModuleSpec.

remove_module(Package, ModuleName, ^Status)

Deletes the module specified by Package (an atom) and ModuleName (an atom) from the database. See page 406 for a description of Status.

【Caution】 This does not mean the erasure of the deleted module itself. Even if removed from the database, processes that already have a pointer to the deleted module can continue to use that module. Furthermore, goals that are already executing will never lose the code that they must execute as the result of the removal operation.

remove_module(ModuleSpec, ^Status)

Deletes the module specified by ModuleSpec from the database. Functions identically to remove_module/3. See page 406 for a description of Status and page 406 for a description of ModuleSpec.

get_all_modules(Package, ^Status)

Returns all modules under the current package specified by Package (an atom). Returns in the Object of Status a list of 3-element vectors that have as their elements a module name atom, a module, and the size of that module. See page 406 for a description of Status.

get_all_modules(^Status)

Returns all modules currently under management. Returns in the Object of Status a list of 2-element vectors that have as their elements a package name atom and the result that would be returned by sending that package in the get_all_modules/2 message. See page 406 for a description of Status.

10.2.1.3 Dynamic linkage

To reduce the overhead in the call of external module predicate, executable codes are linked statically. For this reason, if the called module is updated later, the calling module still refers to

the old module. To reflect the module update, the inter-module linkage should be relinked.

However, when a module is frequently updated (for example, under program development) and some overhead in execution is incurred, if there is no need of relinkage, it may cause greater advantage. In such a case, it is recommended to use the following predicate and apply the dynamic linkage where modules are dynamically linked by their names at the point of execution, instead of static linkage:

module_table:dynamic_link(ModuleSpec, Predicate, Args)

This predicate calls the argument vector Args (by using the built-in predicate apply) to the predicate of module Predicate (an atom) specified by ModuleSpec. Package inheritance is performed in this module search. See page 406 for a description of ModuleSpec.

A raise is generated if the specified module does not exist.

10.2.2 Module devices

Module devices are devices that associate code modules, which are representations of executable code, with their names (atoms). While these devices provide functions that are basically identical to those of the module cache databases described earlier, general users should use the module cache databases described above since the module devices do not perform default package processing, and have a high probability of specification changes due to functional extensions in the future.

Package inheritance is performed for the **get_module/2, get_module/3, get_code/4, and get_code/5** messages in the protocol. The package inheritance function is not used for module update and deletion operations.

The remaining part of this section uses the following meanings for the values returned to the argument named "**Status**" in predicates or messages, unless otherwise noted:

normal: Normal end.

normal(Object): Normal end. The extracted information is unified with Object.

abnormal: Abnormal end

non_existent: No corresponding module exists.

protection_violation: An attempt was made to update a protected module, e.g., a PIMOS module.

10.2.2.1 Creation

Module devices are created by sending the following message to a module request device.

module(^Status)

Creates a module device. Returns **normal(ModuleDev, ^Abort, Attention)** in Status if it succeeds, and **abnormal** if it fails. Here, as is the case for other devices, Abort is the abort line, and Attention is the Attention line. ModuleDev is a stream to the created module device.

10.2.2.2 Protocol

define_package(Package, Super, Protection, ^Status)

Generates a new package with the name Package (an atom). Super (an atom) is the name of the parent package, and must be the name of an existing package. Protection (an integer) sets

the protection mode. Status is described on page 410. (See section 11.1, "Resource Acquisition and Management", (page 419) for a description of protection modes.)

【Caution】 Only the super user "super_pimos" can create new packages. See the section, "Packages", below for a description of the new package creation procedure.

remove_package(Package, ^Status)

Deletes the existing package Package (an atom). Status is described on page 410.

【Caution】 Only the super user "super_pimos" can delete packages.

protect_package(Package, Protection, ^Status)

Changes the protection mode of the existing package Package (an atom). Protection (an integer) is the new protection mode to be set. Status is described on page 410. (See section 11.1, "Resource Acquisition and Management", (page 431) for a description of protection modes.)

get_super_package(Package, ^Status)

Retrieves the name of the parent package (an atom) of the package Package (an atom). Status is described on page 410.

get_package_info(Package, ^Status)

Returns protection information for the package Package (an atom). A compound term of the form {{OwnerID, GroupID}, Protection} is returned in the Object of Status. Here, OwnerID is the owner ID (an integer) of the package, GroupID is the group ID (an integer) of the package, and Protection is the protection mode (an integer) of the package. Status is described on page 410. (See section 11.1, "Resource Acquisition and Management", (page 419) for a description of protection modes.)

get_all_packages(^Status)

Returns the package information for all currently registered packages. A list of 3-element vectors of the form {Package, Status1, Status2} is returned in the Object of Status. Here information on the parent package is returned in Status1, and information related to the protection mode is returned in Status2. (See section 11.1, "Resource Acquisition and Management", on page 419.)

Package: The package name.

Status1: The same as the Status returned by get_super_package/2.

Status2: The same as the Status returned by get_package_info/2.

get_module(Package, ModuleName, ^Status)

Returns the module with the name specified by Package (an atom) and ModuleName (an atom) in the Object of Status. See page 410 for a description of Status.

get_module_in_the_package(Package, ModuleName, ^Status)

Returns the module with the name specified by Package (an atom) and ModuleName (an atom) in the Object of Status. Does not search parent packages, i.e., no package inheritance is performed. See page 410 for a description of Status.

get_code(Package, ModuleName, Predicate, Arity, ^Status)

Returns the code specified by the package Package (an atom), the module name ModuleName (an atom), the predicate Predicate (an atom), and the number of arguments Arity (a positive integer) in the Object of Status. If the module exists but the predicate corresponding to that module does not exist, a vector of the form **non_existent(code(^Module, ^Predicate, ^Arity))** is returned. See page 410 for a description of Status.

【Hint】 The reason a vector of this form is returned is so that information such as the module name can be passed in the exception information when the returned result is passed as an argument to the built-in predicate apply.

update_module(Package, ModuleName, Module, ^Status)

Updates the module specified by Package (an atom) and ModuleName (an atom) to be the new module specified by Module. If there is no module corresponding to ModuleName the module is simply added. See page 410 for a description of Status.

【Caution1】 If a module with the same name as the module to be updated exists in a higher level package, then the new module will simply be registered in the specified package.

【Caution2】 Updating the database does not imply update of the module itself. Modules already accessed from the database remain unchanged even if the database is updated, and, as long as calls do not use the dynamic link mechanism, the call destinations of goals being executed will not be updated.

remove_module(Package, ModuleName, ^Status)

Deletes the module specified by Package (an atom) and ModuleName (an atom) from the database. See page 410 for a description of Status.

【Caution】 This does not mean the erasure of the deleted module itself. Even if removed from the database, processes that already have a pointer to the deleted module can continue to use that module. Furthermore, goals that are already executing will never lose the code that they must execute as the result of the removal operation.

get_all_modules(Package, ^Status)

Returns all modules currently under the package Package (an atom). A list of 3-element vectors is returned in the Object of Status. These vectors consists of the following elements: the module name atom, the module itself, and the size of the module. See page 410 for a description of Status.

10.2.2.3 Utilities

These are module related utilities. The provided predicates are described below.

module:get_module_name(Module,^Name)

The name of the module Module and the package to which that module belongs are returned as a compound term of the form **Package::ModuleName**, ({**(::), Package, ModuleName**} as a vector notation).

module:get_module_name(Module,^Package,^ModuleName)

Returns the name (in ModuleName) of the module Module and the package (in Package) to which that module belongs.

module:get_attributes(Module,^Status)

Extracts the timestamp information assigned to the module Module, and returns that information in the argument Status. If timestamp information has been assigned, then a compound term of the form **normal({CDT, LDT, SourceFile, SDT, Depends})** is returned in Status. Here CDT is the module creation date and time, LDT is the module link date and time, SourceFile is the source file name (as a string), SDT is the source file update date and time, and Depends is a list of 2-element vectors, that, for each file upon which creation of the module depends, consists of the source file name (as a string) and the file update date and time for that file. Also, the date and time is a 2-element vector which consists of the number of the days and time in millisecond units (an integer). (See section 9.2, "I/O Utilities", on page 355 for details on the date and time representation.) If a timestamp has not been assigned, "non_existent" is returned in Status.

module:get_predicate_info(Info,Package,ModuleName,Predicate,Arity,SoftInfo)

Returns the following items extracted from the information (Info) reported from the firmware for an exception, for the location that generated the exception: the package name atom (in Package), the module name atom (in ModuleName), the predicate name atom (in Predicate), the number of arguments (in Arity), and finally, the information stored by the compiler (in SoftInfo).

【Caution】 In the current self-compiler, only the integer zero is stored in the section that corresponds to the argument SoftInfo above.

10.2.3 Packages

The package system in PIMOS is a function that provides multiple name spaces for modules, and packages are an environment that limits the user module name space. Packages in PIMOS are used for the following purposes.

- To divide the individual user name spaces.
- To increase programming efficiency by providing multiple programming environments.

10.2.3.1 Inheritance

Since the PIMOS package system has a unitary inheritance function, most packages are not completely independent, but rather inherit modules from the parent package. This inheritance is implemented by searching sequential parent modules if the specified module does not exist in the specified package. As a result, package specification in module reference is actually the specification of the starting point for this module search. When multiple modules with the same name exist in different packages, package inheritance results in the module in the closest package being referenced.

10.2.3.2 Built-in packages

This section describes the packages that already exist at the point PIMOS is booted.

root This is the root package for package inheritance. This package is a convenience, or a formality: i.e., the set of modules in this package is always the empty set. Modules cannot be registered in this package.

pimos Thus is the package that includes the modules that make up PIMOS. More accurately, the modules loaded by the IPL process are registered in this package. Its parent package is root.

user This is the package that is the direct parent of the individual user packages, i.e., the packages with names identical to user names. Its parent package is pimos.

work This is the package that is taken as the default package if the environment variable task:package is either undefined or has an illegal value. Its parent package is pimos.

privilege This is a package that can only be referenced by the super user "super_pimos". Only modules that require special protection are registered in this package. General users cannot access this package. Its parent package is pimos.

10.2.3.3 Utilities

The following released predicates are currently provided as utility modules concerned with packages.

package:define(Package, Super)

Creates a package with the package name given by the atom Package, and the parent given by the atom Super. The protection mode is set to 711 (octal).

A message is output to the shell standard (message) output if the creation fails.

package:define(Package, Super, Protect)

Creates a package with the package name given by the atom Package, and the parent given by the atom Super. Protect is the protection mode (an integer) of this new package. (See section 11.1, "Resource Acquisition and Management", (page 419) for a description of protection modes.)

A message is output to the shell standard (message) output if the creation fails.

package:define(Package, Super, Protect, Status)

Creates a package with the package name given by the atom Package, and the parent given by the atom Super. Protect is the protection mode (an integer) of this new package. (See section 11.1, "Resource Acquisition and Management", (page 431) for a description of protection modes.) A message is output to the shell standard (message) output if the creation fails.

The atom normal is returned in the argument **Status** if processing terminates **normally**, and the atom **abnormal** is returned in all other cases.

10.3 The Library

This section describes the predicates that can be called from the compiler, relinker, loader, and unloader programs. (See chapter 3, "Compiler, ReLinker, Loader, and Unloader" on page 36.)

In the descriptions that follow, module specifications are indicated either by a module name atom or by the compound term **Package::ModuleName** (or in a vector representation as: {**(::), Package, ModuleName**}). Also, the argument STDOUT is a stream to the shell's standard

output, and STDMES is a stream to the shell's standard message output. These streams must be connected their corresponding streams. If not, a deadlock will occur.

The atom normal is returned in the argument Status if processing terminates normally, and the atom abnormal is returned in all other cases.

10.3.1 Calling the compiler

The following predicate is provided so that the user can call the compiler from a program.

compile:compile(FileNames,^STDOUT,^STDMES,^Status)

Compiles the file specified in FileNames (a list of filename strings). The compiler output messages are output to the shell's standard output (STDOUT) and standard message (STDMES).

During this call, either the contents of the compiler parameter file "compile.param", or the values of environment variables will be used for the settings of the compiler parameters.

The atom **normal** will be returned in the argument **Status** if the compiler completes normally, and the atom **abnormal** will be returned otherwise.

10.3.2 Calling the relinker

The following predicates are provided so that the user can call the relinker from a program.

Note that all modules created from these relinker calls are registered in the default package (the value of the package environment variable task:package). However, if the value of the environment variable compile:original_package is the atom "on", the modules that are the object of relinking will be registered in the same package as the package in which they exist.

relink:do(ModuleNames,^STDOUT,^STDMES,^Status)

Relinks the modules specified in ModuleNames (a list of module specifications). The atom **normal** will be returned in the argument **Status** if the relinking completes normally, and the atom **abnormal** will be returned otherwise.

relink:do(TopModName,GoalModNames,^STDOUT,^STDMES,^Status)

Takes the module specified by TopModName as the top module and relinks all modules that are related to the reference functions up to the modules specified by GoalModuleNames. Only the packages specified by the environment variable compile:working_packages are included in the search range for this module search. If the environment variable is an atom ('*'), all packages will be searched. The atom **normal** will be returned in the argument **Status** if the relinking completes normally, and the atom **abnormal** will be returned otherwise.

relink:package(Package,^STDOUT,^STDMES,^Status)

Relinks all modules included in the package specified by Package (an atom). The atom **normal** will be returned in the argument **Status** if the relinking completes normally, and the atom **abnormal** will be returned otherwise.

10.3.3 Calling the loader

The following predicate is provided so that the user can call the loader from a program.

All modules created by the loader are registered in the default package (the value of the environment variable task:package). However, if the value of the environment variable compile:original_package is the atom "on", these modules will be registered in the same package as the package in which they existed at unload time.

load:do(FileNames,^STDOUT,^STDMES,^Status)

Loads all files specified by FileNames (a list of filename strings). The atom **normal** will be returned in the argument **Status** if the loading completes normally, and the atom **abnormal** will be returned otherwise.

Load distribution in file units can be specified with the setting of the environment variable **compile:distribution**. Also, file search is performed according to the setting of the environment variable **compile:path**.

10.3.4 Calling the unloader

The following predicates are provided so that the user can call the unloader from a program.

unload:do(ModuleNames,FileName,^STDOUT,^STDMES,^Status)

Unloads all the modules specified by ModuleNames (a list of module specifications) to the file specified by FileName (a filename string). The atom **normal(FullPathName)** will be returned in the argument **Status** if the unloading completes normally, and the atom **abnormal** will be returned otherwise. FullPathName will the name of the unloaded file as a string.

The default file extension is "sav".

10.4 Programming Examples

【Example】 The program in this example compiles, relinks, and unloads the specified file.

```
cru(FileNames,Status) :- list(FileNames) |
    null_device(Null0),      % Dummy standard output device
    merge({Null1,Null2,Null3},Null0),
    compile:compile(FileNames,Null1,Null2,CompStatus),  % Compile
    module_table:cache_table(ModTbl0),
    ModTbl0 = [get_default_package(PackageStatus)|ModTbl1],
    relink(CompStatus,PackageStatus,ModTbl1,Null3,Status).
otherwise.
cru(FileNameList,Status) :- true | Status = abnormal.

relink(normal,normal(Package),ModTbl0,Null0,Status) :- true |
    Null0 = {Null1,Null2,Null3},
    relink:package(Package,Null1,Null2,RelinkStatus),
    % Relinks all packages in the default package.
    get_all_modules(RelinkStatus,Package,ModTbl0,Null3,Status).
otherwise.
relink(_,_,ModTbl0,Null0,Status) :- true |
    ModTbl0 = [],
    Null0 = [],
    Status = abnormal.

% Be sure to remove modules only after confirming that relinking has completed.
get_all_modules(normal,Package,ModTbl0,Null0,Status) :- true |
```

```
    ModTbl0 = [get_all_modules(Package,AllModStatus)],
    convert(AllModStatus,Null0,Status).
otherwise.
get_all_modules(_,Package,ModTbl0,Null0,Status) :- true |
    ModTbl0 = [],
    Null0 = [],
    Status = abnormal.

convert(normal(AllMods),Null0,Status) :- list(AllMods) |
    Null0 = {Null1,Null2},
    convert_0(AllMods,Modules),
    unload:do(Modules,"allmodules",Null1,Null2,Status).
    % Unloads to the file allmodules.sav in the current directory.
otherwise.
convert(_,Null0,Status) :- true |
    Null0 = [],
    Status = abnormal.

% Converts to a list of module names.
convert_0([{Name,Module,Size}|Rest],Modules0) :- true |
    Modules0 = [Name|Modules1],
    convert_0(Rest,Modules1).
convert_0([],Modules0) :- true | Modules0 = [].

% Since this is output, only the flush/1 message needs to be attended to.
null_device([flush(Done)|Next]) :- true |
    Done = normal,
    null_device(Next).
null_device([]) :- true | true.
otherwise.
null_device([DontCareMessage|Next]) :- true |
    null_device(Next).
```

Chapter 11

Resource Management and Execution Management

Service units provided by PIMOS are called "OS defined resources". (On the other hand, resources such as computation time and memory area required for the execution of KL1 programs are called "programming language defined resources".) PIMOS has two types of OS defined resources. One type consists of the program execution unit called the task, which is the resource that is the base for the whole of PIMOS, the user service unit called the general request device, and the resource called standard I/O, a resource for which at least one instance is possessed by each task. The other type consists of resources that are dynamically defined by programs called servers, and these are OS defined resources that perform I/O and other user services. There are two types of server program that provide the latter type of resource to users: servers (also called **system defined servers**), which have functions corresponding to OS services such as windows and files, and **user defined servers**, which the user can define freely.

This chapter describes the programming methods related to the following items.

11.1 Resource Acquisition and Management

Resource acquisition methods and general techniques, focussing on the general request device (including task startup methods)

11.2 Tasks

Task control methods

11.3 Server Implementation Methods

Server definition techniques (including those for standard I/O servers)

11.4 User Data Management

Methods for using user managed resources

Note that the standard I/O resource is described in Section 9.4 (page 394) and that I/O resources are described in section 9.1 (page 350).

11.1 Resource Acquisition and Management

Services provided by the OS defined resources can be used by sending messages to the request stream acquired by the user for the corresponding resource. This section describes how to acquire such request streams.

11.1.1 Resource type

The OS defined resources include the following types:

General request device: Manages the messages to create devices for the use of tasks and other resources, and provides the request streams for the resources. Also, queries to PIMOS itself are made using this device.

Task: The unit of execution control for KL1 programs. Also used as the unit of management of OS defined resources. In PIMOS, tasks are handled as OS defined resources. Tasks, as OS resources, perform message processing directed at that task for such operations as execution interruption and reporting on the consumption states of programming language defined resources.

The standard I/O resource: This is the resource for performing the standard I/O that is defined for the task. This is a resource that is the virtualization of a general resource, and is specified by the parent task that created the task. It is acquired from the general request device.

Resources provided by system defined servers: These are resources that perform the user services that are registered in the PIMOS resource service table (see section 11.1.2, page 420). The following resources are registered in advance in PIMOS for user programs.

- window_request
- file_request
- timer_request
- string_io_request

  【Hint】See chapter 9 (page 349) for details.
- atom_request
- module_request

  【Hint】See chapter 10 (page 404) for details.
- user_request

  【Hint】See section 11.4 (page 446) for details.

The resources that can be further acquired from the above resources are as follows.

- window
- file
- directory
- timer
- string_io
- atom
- module

- user
- group
- login

Registering items in, and use of the server resource table are performed through the general request device.

11.1.2 Resource service table

In PIMOS, a management table called the resource service table is provided and used to perform resource usage services. A user program (referred to as a client) can acquire a communications path (in most cases, a stream) to a resource provided by the server by specifying one of the names (atoms) in this table.

The following elements are registered in the resource service table in individual server units.

Name (atom): The registration name.

Protection information: Information concerning the owner, group owner, and usage rights.

Control stream: Communication path between the resource service table and server tasks.

When PIMOS receives a resource acquisition request from a user it performs the following processing in the following order.

1. PIMOS confirms the usage rights of client task with respect to the resource corresponding to the name.
2. PIMOS acquires a communications path between the server task and the client over a control stream.
3. PIMOS provide a communications path with the server to the client side.

11.1.3 I/O nodes

Under PIMOS, it is possible to use resources on the FEP, or on a remote machine connected by the network through the FEP. Equipment units (including computers) that user programs can use via the general request device are referred to as I/O nodes. For users using an FEP that does not have a network, the FEP becomes the I/O node, and resources on that I/O node are used.

When a unit on an I/O node is used, a login name and password are always required on the FEP or remote site OS.

11.1.4 Tasks

A task, which is similar to the shoen, is the unit of execution control for KL1 programs and is, at the same time, the unit of management of OS defined resources.

A user ID is assigned to a task. This information is used in resource protection.

When a task is created it always has a standard I/O resource and an I/O node default value. Normally, the user is provided with this request stream by the OS, and then uses it with various devices. Note that the environment variable task:io_node (see section 11.1.7, page 421) is assigned the name of the I/O node to which this request stream is attached.

Tasks are implemented with the shoen mechanism so that any observation of control or status for the shoen can be similarly carried out for tasks. Differences between a task and a shoen are that: resources obtained within a task are automatically released after the task is completed, and no such mechanism is provided for the shoen; tasks are managed as resources so that an inquiry

can be made for all tasks for information such as their execution status through the PIMOS, and the PIMOS does not support such management for the shoen.

Also, a special task that creates profiling information (see section 7.5, page 302) using the shoen structure is provided in PIMOS. For details, see section 11.1.9.2 (page 423), and for details on tasks, see section 11.2 (page 431).

11.1.5 The standard I/O resource

It is convenient when programming to have preexisting I/O provided. PIMOS provides, in task units, an abstract I/O device called the standard I/O resource. The standard I/O resource is assigned as a task startup parameter at task startup by the parent process. The standard I/O resource cannot be changed by a task itself.

Actually, the standard I/O resource is a filter-like process that accesses resources registered in the resource service table, and when specifying the standard I/O resource, server information for the resource that will become the standard I/O resource is specified.

When a task creates a child task, the standard I/O resource is inherited by the child process. See section 9.4, "Standard I/O", (page 394) for details.

Note that the highest level task (the system defined server task) in the resource tree (see the following section) only has a pseudo-standard I/O resource.

11.1.6 Resource tree

The PIMOS manages all the OS defined resources including tasks under one tree structure.

When a task acquires OS defined resources, they are handled as the child resources of the task. As tasks created by a task are also the child resources, resources generally form a tree structure.

All the child resources of one task get different identifier (child resource identifier; positive integer). The same identifiers as those of the released resources are not reused for the child resources under one single task. Accordingly all the resources can be uniquely specified from the root of the resource tree using a list of child resource identifiers for each level of the tree.

When an inquiry is made for a specific resource, the relative position from its task is often useful to specify the resource. The following relative resource specifications (relative resource identifier) which is in a list format are usually used.

`[]`: Task itself.

`[N|relative resource identifier]`: A resource recursively specified by a relative resource identifier from the resource specified by N. N is defined as follows:

- **0:** The root of the resource tree, i.e., the top level task
- $n (n \geq 1)$ **:** Direct child resources with the identifier n
- -1**:** The resource itDself.
- $-n$ $(n \geq 1)$**:** Ancestor resources of $n-1$ generations
- **`*` :** All the direct child resources

`*` : All of its own child resources, that is, all of `[]`, `[*]`, `[*,*]`, `[*,*,*]`, ...

11.1.7 Environment variable

In the PIMOS, each task has an environment variable pool. An environment variable pool is an information reference table for executing programs.

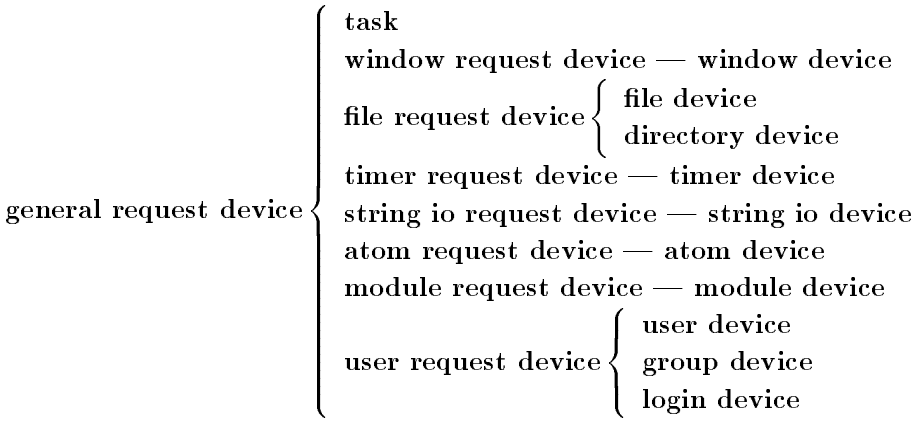

Figure 11.1: Resource Acquisition Hierarchy Provided by the System Defined Server

Users can set or refer to any ground value (a value including no undefined variables) for any environment variable (used as a "key") at any time. Environment variable values used when tasks are created are values of the parent task. Subsequently, environment values are changed independently for each tasks. By default, all tasks have the environment variable, task:node.

Various utilities supplied by PIMOS change their functions according to the values of the corresponding environment variables. Required functions of utilities invoked for each task can be obtained by properly setting values of environment variables.

【Hint】 The table of contents in this manual includes a table of the environment variables used by PIMOS.

11.1.8 Resource acquisition hierarchy provided by the system defined server

This section describes the resources provided by the system defined server as a resource acquisition hierarchy as seen from the hierarchy.

General request device should be first acquired and subsequent resource acquisition should be in a hierarchical way. For example, to use a window, acquire the stream to the window request server by sending a message to the general request device. Then acquire the necessary stream to an actual window by sending a message to the window request server (a window is actually created then). Input to and output from the window can be made by sending messages to this stream to the window.

The resource hierarchy supplied by the PIMOS is shown in Figure 11.1.

How to acquire request streams to the general request device and the message protocol for the resource acquisition made in a hierarchical way are described individually in this chapter and related sections.

11.1.9 General request device

11.1.9.1 How to acquire a request device

The shoen exception raising predicate "raise" is used for acquiring a stream to the general request device.

shoen:raise(pimos_tag#task, general_request, GeneralRequestDevice)

Acquires in the variable GeneralRequestDevice the request stream to the general request device attached to a task.

【Caution】 The following declaration should be issued at the top of the module which includes invocations of the above:

```
:- with_macro pimos.
```

shoen:raise(pimos_tag#task, general_request(IONode, User, Password), GeneralRequestDevice)

Acquires in the variable GeneralRequestDevice a stream to the general request device that has as its a priori value the I/O node IONode that is provided by a PIMOS device, with the given user name and password for the OS on that IONode. If an incorrect password is given, all messages sent to that stream will return abnormal. Note that the message connection described later can be used to confirm connection.

IONode: The I/O node name string for the network site for connection. The FEP number is used for connection to an FEP.

User: The user name string on the I/O node OS.

Password: The password string for the specified user name.

shoen:raise(pimos_tag#task, general_request(Fep, IONode, User, Password), GeneralRequestDevice)

Identical, except that it connects to the I/O node through the specified FEP. This specification is illegal if IONode is an FEP number.

Fep: The FEP number for the I/O node connection.

11.1.9.2 Protocol

request(Name, Cancel, ^Result)

Acquires the resource that is specified by the atom Name. That resource must be a resource registered in the resource service table (see page 420). If the acquisition succeeds, **normal(ServerVar)** is returned in **Result**. The ServerVar protocol is specified individually for each server. In cases where no specification is made, or the protection does not allow use, abnormal will be returned in Result. If the server side task is terminated during acquisition processing, aborted is returned. Also, an arbitrary value is unified with Cancel if processing was cancelled during request processing. In this case, aborted will be returned in Result.

request_anyway(Name, Cancel, ^Result)

In addition to the functions of the `request(Name, Cancel, ^Result)` form described above, this form also has the function of waiting until the specified name is registered, if that name was not registered on the server.

window(^Result)

file(^Result)

timer(^Result)

string_io(^Result)

atom(^Result)

module(^Result)

user(^Result)

Acquires a stream to the system defined server provided by PIMOS. This is a variation of the `request_anyway(Name, Cancel, ^Abort)` form. Corresponding request devices are acquired. If the acquisition is successfully completed, **normal(Request, Abort, Attention)** is returned in Result. If the acquisition is abnormally completed, **abnormal** is returned in Result. Request is the stream to the request device. See chapter 9 on page 349 for Abort and Attention.

register(Name, ProtectionMode, ^Result)

Registers a new resource in the resource service table. (This is called service registration.) For details, see section 11.3.3.1 (page 439).

【Caution】 This command can only be used by a privileged user.

reset(Name, ^Result)

Sends attention information to the server that is in charge of the service specified by **Name**. One of the following will be returned as the result **Result**.

normal(^NewAttention): The attention information was sent.

abnormal: The attention information transmission was denied.

aborted: The object server task is in termination processing.

【Caution】 Only users who are allowed to receive the respective service are allowed to use this command.

execute(Code, Argv, Param, ^Result)

Generates a task. Arguments are shown below. Many of those arguments are the same as those (see page 302) specified in generating a shoen.

Code: Predicate code for the top level goal of the task.

Argv: Argument vector for the top level goal.

Param: A list that holds a variety of parameters concerned with task creation. The elements of the list area as follows.

resource(HI, LOW): Initial amount of the resource. Taken to be 0 when omitted.

ionode(IONode): Either the I/O node name or the FEP number. Taken to be the same value as that for the parent task when omitted.

remote(RUser, RPass): The user name and password corresponding to the I/O node name. Taken to be the same as those for the parent task when omitted.

name(Name): The task name. Taken to be the empty character string when omitted.

prio(Min, Max): The priority range. Taken to be 0 (Min) and 4096 (Max) when omitted.

tag(Tag): The tag for exception processing. Taken to be pimos_tag#task when omitted.

【Caution】 Note that the system call tag bit (pimos_tag#task) is always raised regardless of the user specification for a shoen tag.

node(Start, Nodes): The start node and the number of nodes. Taken to be the same values as those for the parent task when omitted.

uid(Uid): Sets the user id of the task to be Uid. Can only be specified by the super user.

user(UName, Pass): Provides the user name and password for the task. If correct, the corresponding user ID is assigned to the task. If the password is incorrect, the task creation fails.

stdio(STDIO, ^STDResult): Specifies standard I/O. The parameters have the following meanings.

STDIO: A parameter list used to specify the device to be generated as the new standard I/O. The standard I/O streams not specified here are inherited from the parent's standard I/O (i.e., that of the invoking task). If all of the parent's standard I/O is to be inherited the null list ([]) is used for this parameter. Elements are of the form `{Type, RSC}`.

Type: The type of the standard I/O to be created.

in: Standard input

out: Standard output

inter: Standard interaction

in_out: Standard input and standard output

in_inter: Standard input and standard interaction

out_inter: Standard output and Standard interaction

in_out_inter: All standard I/O.

RSC: The corresponding server and its parameters in the form `{Server, InInfo, ^OutInfo}`.

Server: The server name as an atom.

InInfo: Parameters for standard I/O creation. Must consist of ground values with no undefined variables (for example, file names).

OutInfo: The result from the server. Consists of ground values with no undefined variables. If values are assigned here, that completes the setting of the standard I/O resource.

【Caution】 An empty list ([]) is returned in **OutInfo** if the server corresponding to **Server** is unregistered. In this case an empty standard I/O resource is assigned, and deadlock will occur when the user uses this resource, since no response is returned.

STDResult: Interpreted as follows:

normal(ID,Info): Normal termination.

ID is an identifier for the standard I/O of this task with respect to the standard I/O of the parent, and is a non-negative integer. It has no meaning when all standard I/O streams were created anew for this task. Note that the parent task acquires a switch for its own standard I/O, and sets the state where the child task can use this standard I/O by specifying this id.

The parameter **Info** is a list of information for confirming that the parent's standard I/O was inherited. It is a list of the following elements. This parameter will be the empty list ([]) if none of the parent's standard I/O was inherited. The elements of the list depend on the state of the parent's standard I/O. Note that **STDInfo** is a ground value with no undefined variables.

in(^STDInfo)

out(^STDInfo)

inter(^STDInfo)

in_out(^STDInfo)

in_inter(^STDInfo)

out_inter(^STDInfo)

in_out_inter(^STDInfo)

【Caution】 The timing with which **Info** is returned is the time that standard I/O is assigned. Therefore, it may not be possible to switch this task's standard I/O without waiting the variable **Info**.

abnormal:

aborted: Returned when this command was sent when the parent task terminates, or when overlapping parameters were specified.

Result: Task generating status. Either of the following values is returned:

normal(^Control, Report): A task has been successfully generated. The Control argument is the control stream of the task and the Report argument is the report stream of the task .

abnormal: Task generation fails.

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, ^Result)

Generates a task. Arguments are shown below. Many of those arguments are the same as those specified in generating a shoen.

Code: Predicate code for the top level goal of the task.

Argv: Argument vector for the top level goal.

Name: Task name string. This is returned on status inquiry and can be used as a comment.

MinPrio: Minimum priority of the task as a shoen.

MaxPrio: Maximum priority of the task as a shoen.

Tag: Tag for exception (An integer. See section 7.5.6 on page 314)

Result: Task generating status. Either of the following values is returned:

normal(^Control, Report): A task has been successfully generated. The Control argument is the control stream of the task and the Report argument is the report stream of the task.

abnormal: Task generation fails.

【Caution】 The initial execution status of the task is resource_low (computation resource is 0).

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, InitialResoc, ^Result)

Generates a task. The arguments are the same as those of the above execute/7 except InitialResoc:

InitialResoc: InitialResoc: Initial computation resource during task starting in a form of 2-element vector {**HighResoc, LowResoc**}. HighResoc is the upper 32 bits of the initial language defined resource of the shoen and LowResoc is its lower 32 bits (see section 7.5 on page 302 for language defined resources)

profile(Code, Argv, Param, ^Result)

profile(Code, Argv, Name, MinPrio, MaxPrio, Tag, ^Result)

profile(Code, Argv, Name, MinPrio, MaxPrio, Tag, InitialResoc, ^Result)

Creates a task with a profile structure attached. The arguments are the same as those for execute. See sections 7.5 (page 302) and 11.2 (page 431) for details on techniques for collecting profiling information.

connection(^Result)

Confirms whether a device can be created on the I/O node. Either "**normal(Node)**" (the state is normal, and the name of the connected I/O node is **Node**) or "**abnormal**" (for an abnormal state) is returned.

suicide(^Result)

Forces termination of a task; the task is not the one created through this General request stream but the task itself which sends the message. Either of **normal** (successfully completed) or **abnormal** (abnormally completed) is returned to the Result argument.

【Hint】 When a task is successfully completed, the task usually ends before examining this Result.

abort(ResourceId, ^Result)

Forces termination of a task specified in ResourceId.

ResourceId: Resource identifier

Result: Processing result. **normal** (successfully completed) or **abnormal** (abnormally completed)

number_of_feps(^Result)

Gets the number of FEPs currently connected. As the processing result, either of the following is returned in Result:

normal(Feps): Successfully completed. The number of FEPs in returned in the argument.

abnormal: Abnormally completed.

setenv(Name, Value, ^Result)

Sets a value to an environment variable.

Name: Environment variable name. Must be a ground value with no undefined variables.

Value: The value. Must be a ground value with no undefined variables.

Result: The result of the processing. The result will either be "**normal**" or "**abnormal**", corresponding to normal completion and abnormal termination, respectively.

getenv(Name, ^Result)

Gets an environment variable value.

Name: Environment variable name. Must be a ground value with no undefined variables.

Result: Processing result

normal(Value): Successfully completed. This returns the value in the argument.

abnormal There are no environment variables specified.

unsetenv(Name, ^Result)

Deletes an environment variable value.

Name: Environment variable name. Must be a ground value with no undefined variables.

Result: Processing result

normal: Successfully completed. This result is also returned even if no environment variable exists.

abnormal Abnormally completed.

getenv(^Result)

Returns all the environment variables and their values. As the processing result, either of the following is returned in Result:

normal(NameValueList): Successfully completed. An empty list is returned to NameValueList if no environment variable exists. A list of vectors in a form of {**Name, Value**} is returned if any environment variables exist. Name is the name of an environment variable and Value is its value.

abnormal Abnormally completed.

id(^Result)

Inquires about the resource identifier of the task. As the processing result, either of the following is returned in Result:

normal(ResourceId): Successfully completed. The child resource identifier (a positive integer, the child resource identifier in the parent task) is returned in ResorceID.

abnormal: Abnormally completed.

absolute_id(^Result)

Inquires about the absolute resource identifier of the task. As the processing result, either of the following is returned in Result:

normal(AbsoluteResourceId): Successfully completed. A list of child resource identifiers in each generation from the top task along the resource hierarchy is returned in AbsoluteResourceId.

abnormal: Abnormally completed.

status(ResourceIdList, ResourceType, Verbosity, ^Result)

Inquires about the task status. Arguments are as follows:

ResourceIdList: Tasks to be inquired. Specified by the relative resource identifier.

ResourceType: Resource type to be inquired. Types which can be specified currently are * (all the resources) and task (tasks only).

Verbosity: Degree of detailed information. Detail degrees which can be specified depend on each object resource type to be inquired. Currently, only detail can be specified for a task, and only brief can be specified for *.

Result: Result of the inquiry. Either of the following:

normal(Info): Successfully completed. A list of vectors which represent the status of resource(s) is returned in the Info argument. (Details are described later.) However, there are cases where the information for only the specified path resource ID is also included due to specification of the resource identifier.

abnormal: Abnormally completed.

The form of the vectors which represent the status of resources depends on resource types.

- **When the resource type is task and the detail degree for information is detail:**

  {Id, Name, TaskStatus, Limit, Consume, Children}

  or

  {Id, Children}

  Each element is defined as follows:

  Id: Relative resource identifier.

  Name: Task name (the string specified at the time of creation)

  TaskStatus: TaskStatus: Status of the task. A vector in the form of **{Cond, Stops}** is returned. When the status is unknown (for example, when the parent shoen is halted), unknown is returned.

  Cond represents the execution status of the task and is one of the following:

  normal: Normal status (which is, not in resource shortage status).

  resource_low: Resource shortage status.

  terminating: During termination processing

  shut_terminating: During termination processing due to the forced termination of the parent task.

  Stops is a list of task stop reasons. This is `[]` when the task is not stopped. Each element is in the form of **stop(Reason)**, where **Reason** is an atom which represents the stop reason.

Limit: A 2-element vector that expresses the upper limit for the language defined resource consumption.

Consume: A 2-element vector whose elements are integers and that expresses the amount of the language defined resource consumption up to the present (-1 when unknown).

Children: A list of information vectors for all the objective child tasks when child task information is also acquired.

- **When the object resource type is * and the detail degree for information is brief:**

  {Id, Type, TypeInfo, Children}

  or

  {Id, Children}

  Each element is defined as follows:

  Id: Child resource identifier.

  Type: Resource type.

  Name: Information associated with the resource type.

  Children: A list of information vectors for all the objective resources when resource information is also acquired.

11.1.10 Standard I/O resource acquisition

When a task uses a standard I/O resource it must first acquire a stream to that standard I/O resource. The remainder of this section describes the procedure for that acquisition. Note that the technique for assigning a standard I/O resource to a child task is described in section 11.1.9.2 (Page 425), and that the details of standard I/O and its protocols are described in section 9.4 (page 394).

11.1.10.1 How to acquire standard I/O resource

The shoen exception raising predicate "raise" is used for acquiring a stream to the standard I/O resource

shoen:raise(pimos_tag#task, stdio(Type), Stream)

Acquires a stream to a standard I/O resource. There are 5 types of streams to standard I/O, and that type is specified in the **Type** parameter to the predicate's second argument. The following types can be specified in **Type**.

in: A standard input stream

out: A standard output stream

inter: A standard interaction stream

io: A standard I/O stream

switch: A standard I/O control stream

11.1.11 Resource protection modes

PIMOS provides resource protection modes to support a multiple user environment. It is possible to limit the access range to a resource for each user by attaching these protection modes. Note that protection modes express their meaning as integers, and can be set by sending an appropriate protocol to the acquired resource.

【Caution】 Currently packages are the only resource for which protection modes have been introduced. (See section 10.2.1.2)

11.1.11.1 Resource owner

For resources for which protection modes have been introduced, there is a user that is the owner (called the owner) and there is a group that is the owner (called the "group owner"). When a PIMOS user uses one of these resources, the use of that resource is enabled by the rights of the user himself and the rights of the group to which the user belongs.

11.1.11.2 Protection modes

A protection mode is expressed as a 9-bit integer. If a particular bit is 1, then that mode is enabled. The descriptions of these bits are as follows.

0-bit: General user execution rights

1-bit: General user write rights

2-bit: General user read rights

3-bit: Execution rights for users associated with the group of the group owner.

4-bit: Write rights for users associated with the group of the group owner.

5-bit: Read rights for users associated with the group of the group owner.

6-bit: Owner execution rights

7-bit: Owner write rights

8-bit: Owner read rights

11.2 Tasks

A task is the management unit of program execution in PIMOS.

Tasks operate actively and require asynchronous communication, unlike other PIMOS defined resources. For this reason, they have two streams, one for control and the other for reporting. The control stream is the message stream from the user to the task; the report stream is one from the task to the user.

Note that the profiling information messages are only valid for profiling information creating tasks.

11.2.1 Messages to the task control stream

stop

Stops the task. The task being stopped can be reactivated by the variables (described later) in the corresponding report stream message.

allow_resource_report

Allows the resource shortage report. Once the resource shortage is reported, it will not reported again until this message is received.

statistics

Requests the report of resource consumption. The report is received as a message from the corresponding report stream.

abort(ResourceId, ^Result)

Aborts the task specified in ResourceId. The task is specified in the form of the relative resource identifier from the directly controlled task.

ResourceId: Resource identifier

Result: Processing results

normal: Successfully completed.

abnormal: Abnormally completed.

abort(^Result)

Aborts the directly controlled task. Same as `abort([], ^Result)`. The argument Result is set to one of the following values:

normal: Successfully completed.

abnormal: Abnormally completed.

add_resource(HighResoc, LowResoc)

Adds to the upper limits of the amount of language-defined resources to be consumed in the task. HighResoc represents the upper 32 bits of the initial computer resources in the shoen and LowResoc represents its lower 32 bits. (See section 7.5 on page 302 for the details on computer resources.)

id(^Result)

absolute_id(^Result)

status(ResourceIdList, ResourceType, Verbosity, ^Result)

Same as the processing of the messages to the general request device, except that the objective task is the controlled one (see section 11.1.9.2 on page 423).

profile

Starts profiling information collection. This is only valid for profiling information creating tasks.

11.2.2 Messages from the task report stream

statistics_started

resource_report_allowed

resource_added

resource_low

aborted

terminated

exception(Node, ExcCode, ExcInfo, ^NewCode, ^NewArgv)

return_resource(High_Resoc, Low_Resoc)

profiling_information(Pe, Type, Depth, Data)

profiling_started

profiling_completed

profiling_rejected

statistics(High_resoc, Low_resoc)

The specifications for all the streams above conform to those of the report stream of shoen (see section 7.5.4 on page 305).

stopped(^ReasonAtom, ^CancelVar)

Indicates that the task stop processing begins. The initial stop reason is none. The stop reason can be changed by unifying the stop reason atom with ReasonAtom. This is returned as the stop reason when querying the state of the task. The stop status can be released by instantiating CancelVar by any data.

11.3 Server Implementation Methods

This section describes PIMOS server implementation methods. Note that implementation methods for standard I/O processing functions are described in section 11.3.4 (page 446) and resource use is described in section 11.1 (page 419).

11.3.1 Overview of the client-server function

In earlier versions of PIMOS, inter-task communications functions were limited to parent and child task communications using the shoen raise structure. Thus it was difficult to realize inter-task communications in situations where the relationship in the task hierarchy was not a parent-child relationship. For example, in communications between sibling tasks, such as the pipes implemented in the shell, it was necessary for the parent task to manage the streams for communication.

Furthermore, it was necessary for the PIMOS top level task to manage all tasks across the multiple hierarchies to realize communications between unrelated tasks.

In the newly introduced client-server function, an inter-task communications function that does not depend on the task hierarchy is provided by registering the server name in the PIMOS resource manager.

This section describes the features of the client-server function.

11.3.1.1 Class-instance relation

It is easier to describe the PIMOS client-server function if we use a class-instance relationship.

- The server corresponds to the class.
- The only objects that PIMOS manages directly are server names that correspond to class names.
- Hierarchical device relationships, e.g., relationships such as that between a file server and a file, correspond to a class and an instance, respectively. This (hierarchical) relationship can be nested to an arbitrary depth, e.g., class -¿ instance (when seen as a class) → instance.
- The communications path between a client and a server does not pass through the OS. Therefore, requests to the OS from the client can be limited to the acquisition of the communications path to the server.

11.3.1.2 Protection functions

Protection mechanisms against unification errors and deadlocks caused by the client are implemented by the server while performing the inter-task communication. PIMOS does not protect the streams between the client and server.

However, PIMOS does provide a framework for server side filter program insertion for informing the server that the client task has terminated, and for communications protection.

This framework allows the server to terminate the devices it provided and handle remaining unprocessed variables.

11.3.2 Usage

This section describes the PIMOS provided client-server function by presenting the procedures involved in its use.

11.3.2.1 Service registration (Server)

Accessing PIMOS is performed, as with earlier functions, by acquiring a stream to the general request device (GRD), and issuing a register/3 protocol on the stream to the GRD. (See section 11.3.3.1) To use the client-server function, the server must first register its service name in the PIMOS resource service table. (See figure 11.2) This service table registration corresponds to registering the communications path to the server.

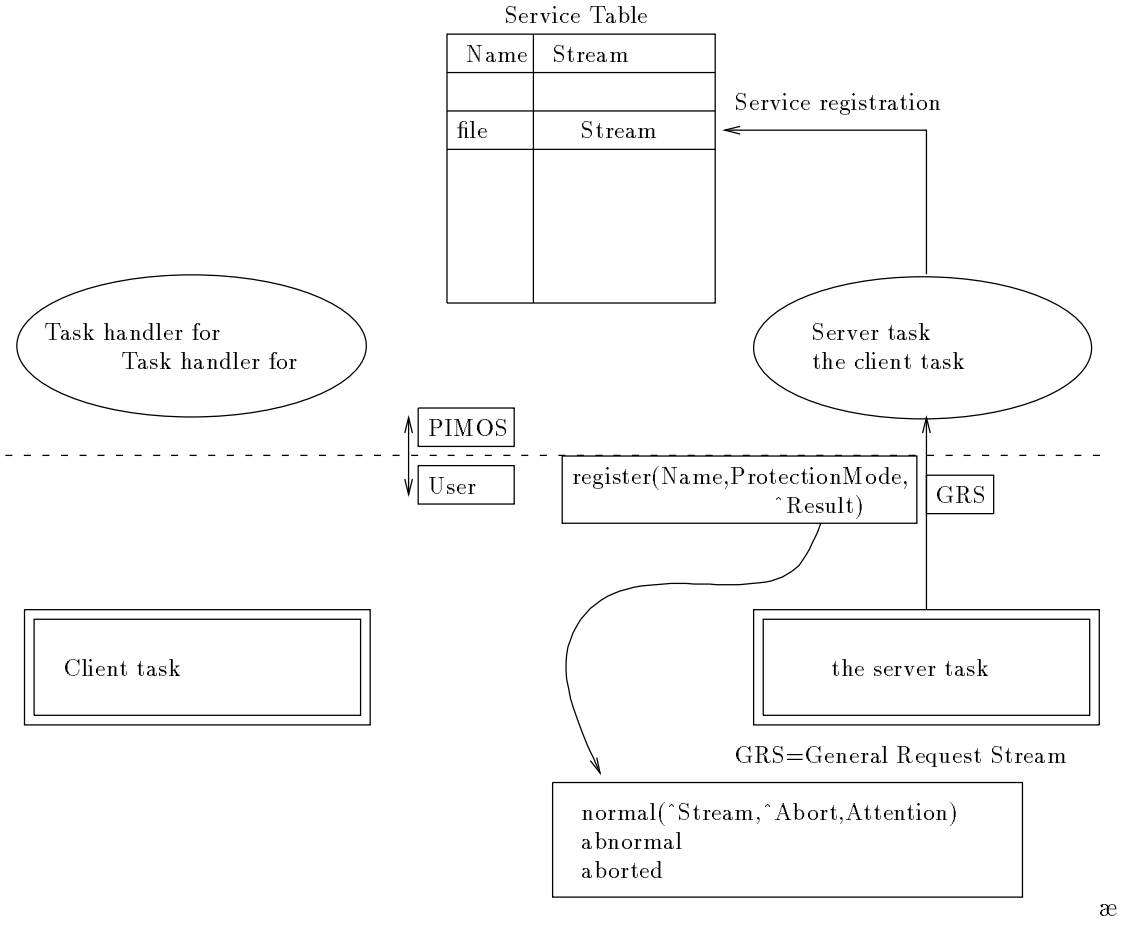

Figure 11.2: Service Registration

11.3.2.2 Resource acquisition (Client)

For the client to use the server, the server issues a request/3 protocol to the GRD to acquire a communications path to the server. (See figure 11.3.)

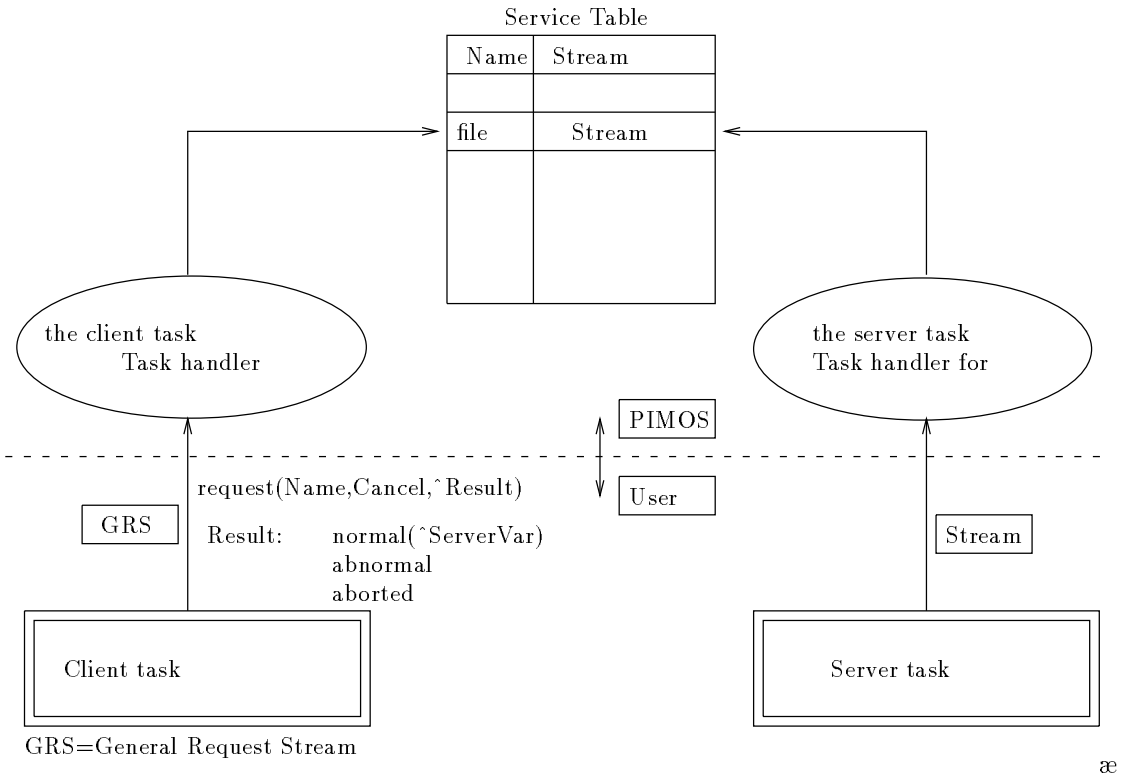

Figure 11.3: Resource Acquisition

11.3.2.3 Resource request reception start (Server)

To fulfill the request from the client, the server issues a get_request/3 reception protocol for the resource request. This protocol corresponds in a 1-to-1 relationship with the request/3 protocol issued by the client, and when both protocols have succeeded, a communications path between the client and server will have been established.

Another point here is that server must insert a protection filter in this communications path so that the server does not fail as a result of a failure on the part of the client. The code for the protection filter that prevents this problem can be assigned by get_request/3. (See figure 11.4.)

It is desirable that the devices created by the server at such times only be devices that simply create instances, such as the earlier PIMOS request devices. Providing a multiple acquisition hierarchy of this sort can reduce the concentration of accesses to the OS.

A structure for receiving announcements of the termination of client tasks is provided for management of client communication paths by the server. In particular, when the client task terminates, aborted is instantiated into the Abort variable that was acquired with the get_request message. Therefore, the server must manage this Abort variable and the communications path to the client.

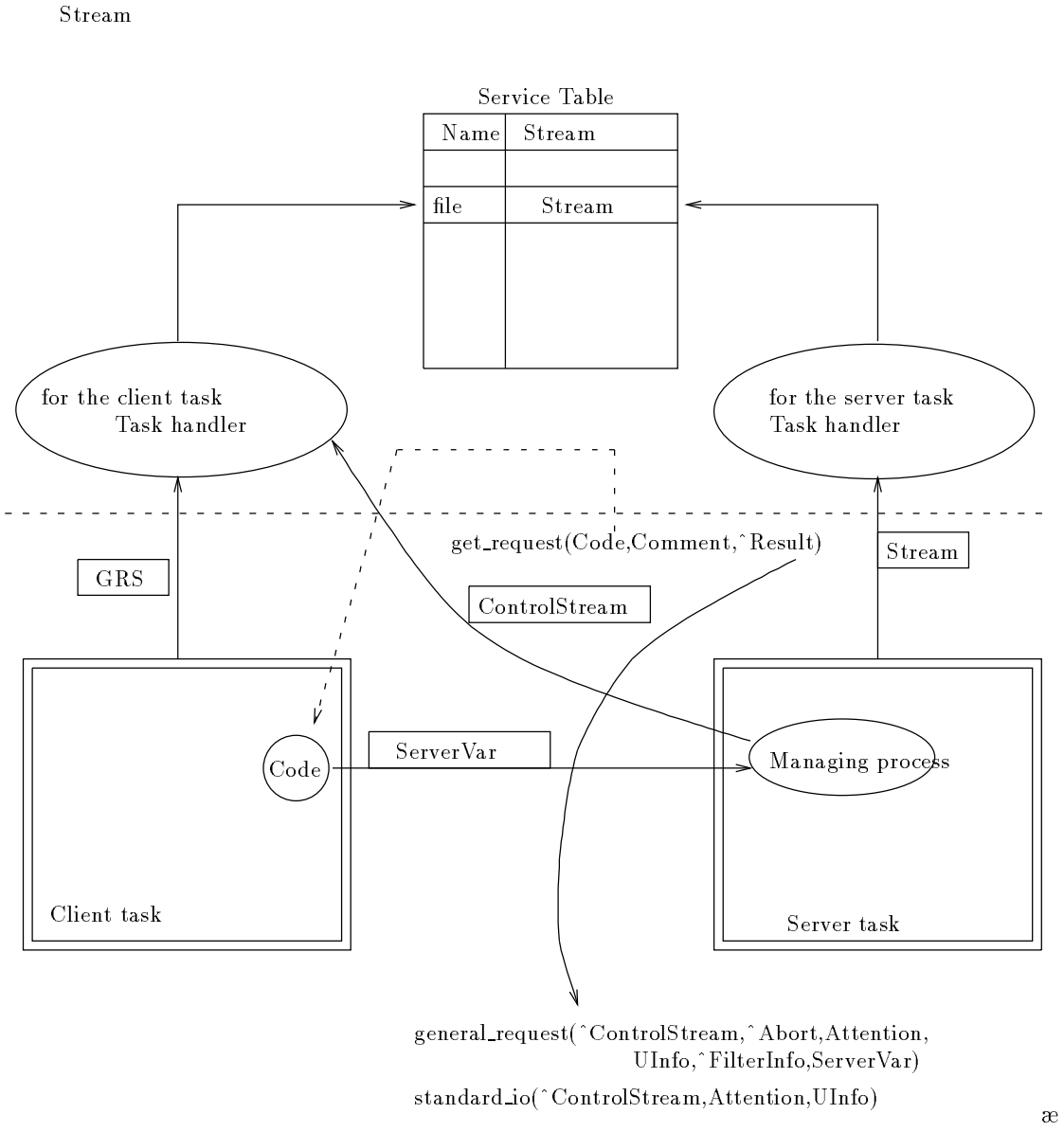

Figure 11.4: Resource Request Reception Start

11.3.2.4 Creation of child resources

When a server has created a lower level device as a child resource, the client-server communications path is increased by exactly that amount. The add_resource/3 protocol is provided on the ServerControl stream acquired by get_request/3 to manage this additional communication path. (See figure 11.5.) The Type and Name parameters provided by add_resource/3 are comment information, and the corresponding resource can be forcibly closed by instantiating aborted into the variable Abort.

11.3.3 Resource provision and acquisition with the client-server function

The only resources provided to users by the kernel are tasks, the general request device, and the standard I/O resource. Other resources, e.g., resources such as windows and filters, are implemented by tasks called servers, and user tasks receive use of services by using inter-task communications with these server tasks. Note that servers can be defined by user tasks.

In general then, the use of a resource involves communications between a task called a server, and a task called the client.

For a user to use a resource, the user must acquire by some method an access task to the server. To that end, the kernel provides an object called the resource service table, and resources can be used by the server registering its resource in this table, and the user acquiring an access path to the server from this table.

【Caution】 There is no requirement that the relationship between servers and tasks be `1:1`, but rather, the implementation is up to the implementer, with both a single task implementing multiple servers, or a single server being implemented by multiple tasks being possible.

11.3.3.1 Service registration methods

A service can be registered by the following general request device command. Note that only privileged users are allowed to use this command.

register(Name, ProtectionMode, ^Result)

Name: The service name atom (the registered name)

ProtectionMode: A 3-element vector specifying the following protection information.

{Uid, Gid, Mode}

Uid: The user id of the owner of the resource

Gid: The group if of the group owner of the resource

Mode: An integer. The following meanings are assigned to the lower 3 bits.

- Owner usage rights
- Group owner usage rights
- Usage rights of other users

A 1 in any of these bits enables those rights, and a 0 disables them.

Result: One of the following will be returned in Result.

abnormal: Registration failed.

aborted: The task is about to terminate.

normal(^Stream, ^Abort, Attention): Registration succeeded.

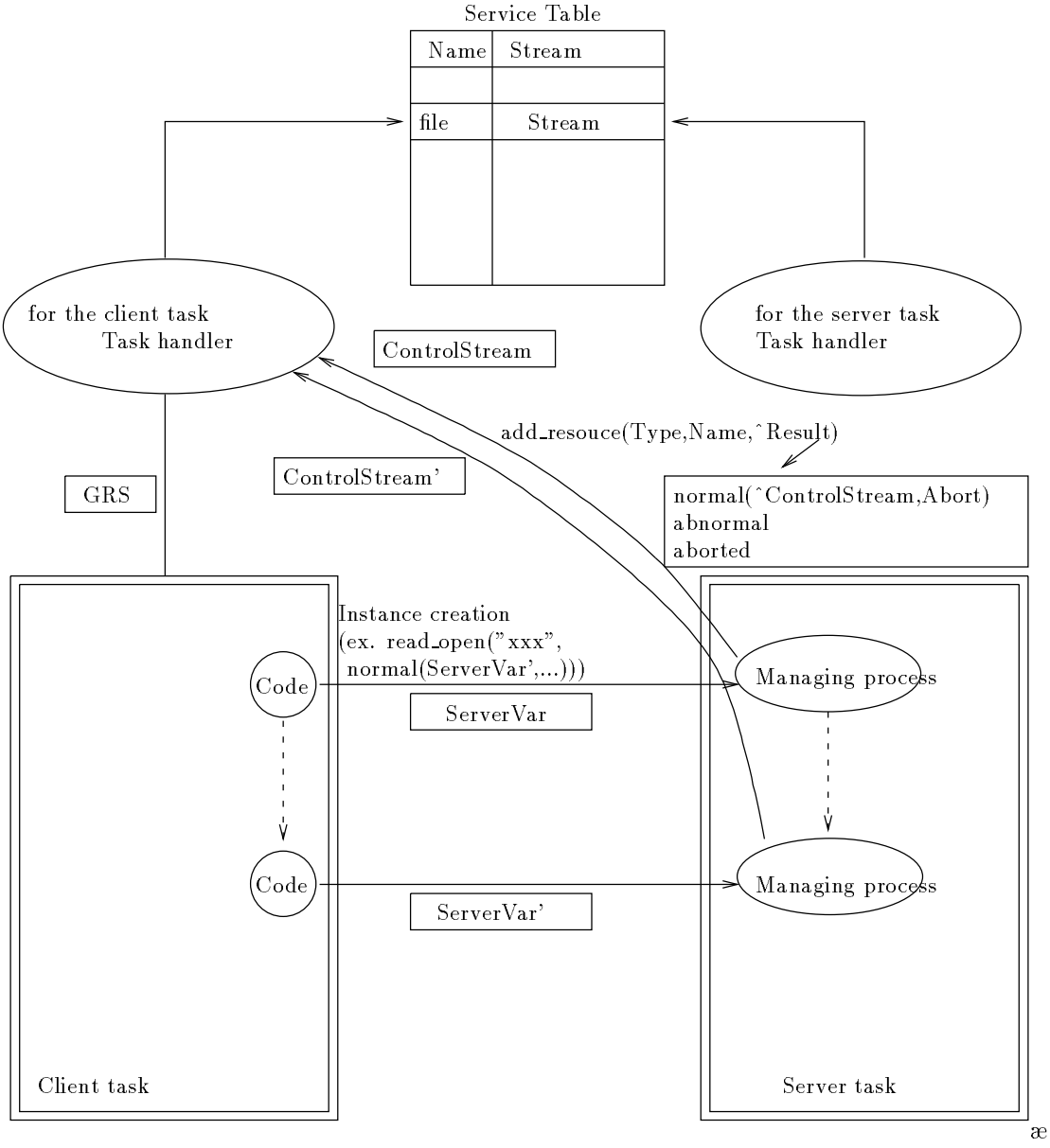

Figure 11.5: Child Resource Creation

Stream: A stream for service processing.

Abort: An abort variable for forcibly terminating use of the Stream.

Attention: A variable for informing the server of an attention from the user.

【Caution】 The result returned in **Result** is described in detail in the next section.

The service name in the service table is reserved by this command.

11.3.3.2 Service request reception

As was mentioned previously, the result from register/3 in the previous section, i.e., the result when the name is correctly registered in the service table is the following term:

normal(^Stream, ^Abort, Attention)

The server performs the service using the stream **Stream**. Inversely, service is terminated by closing this stream.

A service reception system for users can be acquired by sending the following message to the stream **Stream**.

get_request(Code, Comment, ^Request)

Code: The protection filter code for protecting the server from the user. The protection filter is a process for avoiding server deadlock that could occur because of a user protocol error (especially unification errors such as the user setting a value (such as `abnormal = normal`) without waiting for the server processing result) or because the user did not set a value.

This code is installed in the communications path between the service process started by the server and the user, and operates as a filter process in the user task. **Code** is a 3-argument predicate. PIMOS passes a special argument to **Code**, and Code is started in the user task as follows.

apply(Code,{FilterInfo, UserVar, ServerVar})

Code: The protection filter code

FilterInfo: Information specified by the server (described later)

UserVar: A variable for user communication

ServerVar: A variable for server communication

Comment: When a user acquires a server, that intent is registered in the user task resource tree (see section 11.1.6, page 421). At that time, this Comment is register along with the service name as data. While the data format can be any data that is a ground value containing no undefined variables, specifying character string data is common practice.

Request: A service request message from the user is sent from the service table. This message is a message of one of the following 2 types. They are described later in this section.

- **general_request(^ControlStream, ^Abort, Attention, UInfo, ^FilterInfo, ServerVar)**
  The normal resource acquisition request. It is described in the next section.
- **standard_io(ControlStream, Attention, UInfo)**
  The acquisition request for a standard I/O resource. Directly sent by PIMOS. It is described the second following section.

- **aborted**
  Server processing was forcibly terminated. Returned in cases where some value was assigned to the **Abort** argument to register/3, which was described in the previous section, and the server was forcibly removed from the resource service table.

At service termination time, the stream Stream that was acquired with the register message is closed. Also, after a get_request message has already been sent, that message can be cancelled by assigning some value to the **Abort** variable acquired with register. Once canceled, the stream cannot be reopened.

is returned when the user sends the `reset(Name, Status)` general request device command (page 423) as the report, and a variable for the next reset report reception is returned along with the attention report.

attention(NewAttention)

When service is terminated, the empty list ([]) is returned, and this becomes the termination confirmation result.

11.3.3.3 Requests from the general request device

The following message is returned as the response to a get_request message when the user requests resource acquisition using the general request device.

general_request(^ControlStream, ^Abort, Attention, UInfo, ^FilterInfo, ?Var)

The arguments are used as follows.

ControlStream: A process corresponding to the request, and that manages a device, is formed under the client task's task handler. This variable is the control stream for that process, and is also called a server control stream. Normally, this should be closed when the user is through using this device. The following messages can be sent to this stream.

add_resource(Type, Comment, Result)

Registers a new device usage in the client task's task handler. For example, this message is sent when a request device creates a device, and a new device is registered in the client. See section 11.3.3.6 (page 445) for details.

absolute_id(^Result)

Requests the resource absolute id for this device. One of the following values is returned in Result.

- **normal(AbsoluteID)**
- **abnormal**
- **aborted**

change_comment(Com, ^Result)

Changes the comment data for this device.

Com: The new comment data. While the data format can be any data that is a ground value containing no undefined variables, specifying character string data is common practice.

Result: One of the following.

- normal
- abnormal
- aborted

Abort: The abort variable for canceling the message sent to ControlStream.

【Caution】 ControlStream must be closed after this variable is instantiated.

Attention: A valve variable that is set by PIMOS when the client task which has acquired this task terminates.

UInfo: Client task data. See page 444 for details.

FilterInfo: Data passed in the first argument to the protection filter code. This must be a ground value containing no undefined variables.

Var: The variable for actually communicating with the client. Passed as the third argument to the protection filter code.

11.3.3.4 Standard I/O resource creation request

The following message, which sets standard I/O, can be sent as the reply to the get_request message.

standard_io(ControlStream, Attention, UInfo)

This is the acquisition request for a standard I/O resource. It is directly sent from the kernel at task creation time.

【Hint】 There is no need to use this message if a standard server is used. See section 11.3.4, page 446.

ControlStream: A standard I/O resource server control stream.

【Caution】 The I/O modes differ from those of the general request message of the previous section.

The message protocol of this stream is as follows:

open(STDInfo, switch(^ContSW, ^AbSW), io(^ContIO, ^AbIO))

Creates a standard I/O resource. The arguments are used as follows.

STDInfo: A vector of the following form.

{Type, IInfo, ^OInfo }

Type: The type of the acquisition method.

in, out, inter, in_out, in_inter, out_inter, in_out_inter

IInfo: An acquisition time parameter. A ground value containing no undefined variables.

OInfo: A parameter that returns the result of the acquisition. A ground value containing no undefined variables. An empty list ([]) is returned on an error.

ContSW: Creates a process that manages the switch for I/O stream corresponding to the request under the management process in the client task's OS. This is the control stream to that process.

AbSW: An abort variable that similarly corresponds to ContSW:

ContIO: Creates a process that manages the standard I/O I/O stream corresponding to the request under the management process in the client task's OS. This is the control stream to that process.

AbIO: An abort variable that similarly corresponds to ContIO:

request(Type, Node, Communication, FilterInfo)

This is a stream request for a standard I/O relationship from the user.

Type: The standard I/O acquisition type (`in, out, inter, io, switch`)

Node: The node number

Communication: The user I/O stream. The third argument to the protection filter code.

FilterInfo: Data for the protection filter code. Passed as the first argument to the protection filter code.

make_branch(SubControl, CShut,ID, UserInfo)

Divides a standard I/O resource and assigns it to a child task.

SubControl: Post-division standard I/O resource server control stream. Only the following message may be sent prior to division.

open(Type, Status, switch(^ContSW, ^AbSW), io(^ContIO, ^AbIO))

Type is the acquisition type and is one of the following.

in, out, inter, in_out, in_inter, out_inter, in_out_inter

The other parameters are identical the same command described earlier.

ID: Identifier for the branch. Used when switching with the switch command.

CShut: Variable for reporting the termination of a client task that was generated hierarchically. An appropriate value is assigned on termination.

UserInfo: Data concerning the task using standard I/O. Described in the next section.

Attention: A variable to which an appropriate value is assigned when the client task for the object of standard I/O resource assignment terminates.

UInfo: User data for the object of standard I/O resource assignment. Described in the next section.

11.3.3.5 User data acquired by the server

This section describes the client task data that is acquired at the same time that a server receives a resource acquisition request.

{ Uid, Gids, Remote, Node }

Uid: The user id.

Gids: The group id list corresponding to the Uid.

Remote: Data on the remote relationships held by the task. The content differs depending on the type of the server.

- For system servers:

  `{Fep, IONode, RUser, RPass, RInfo}`

  Fep: FEP number

  IONode: Either an I/O node name or an FEP number.

  RUser: The remote user name, or 0.

  RPass: The remote user password, or 0.

  RInfo: Remote data; currently consists of the directory on the I/O node.

- For general users:

  `{Fep, IONode}`

11.3.3.6 Client task resource usage registration

Under PIMOS, it is possible for a server to assign new resource to a new user using an existing control stream that a user and the server are communicating over. For example, a file_request server can confirm in general resources that are being used by the user, by reflecting the fact that a user is using a new file in the resource tree. (See section 11.1, page 419.)

This section describes the following command that implements the processing described above over control streams held by the server.

add_resource(Type, Comment, ^Result)

Type: The device name as an atom

Comment: A comment concerning the Type. Normally specified as a character string.

Result: One of the following values:

normal(^ControlStream, ^Abort): A process that corresponds to the device that was newly registered in the client task's managing process. The arguments are used as follows.

ControlStream: The control stream to the process that corresponds to the device. (See the description in section 11.3.3.4 on page 443.)

Abort: The abort variable for cancelling the message sent to this stream.

【Caution】 ControlStream must be closed after this variable is instantiated.

aborted: The server task is in termination processing.

abnormal: A server was not created.

The state in either of the latter cases is that the server task has terminated.

11.3.3.7 Resource acquisition

Resource acquisition methods are described in section 11.1 (page 419), the acquisition method for standard I/O resources is described in section 9.4 (page 394), and the method for assigning standard I/O to a child task is described in section 11.1 (page 419).

See the following section for a simple method for creating servers that have a standard I/O function.

11.3.4 Incorporating a standard I/O function in a general server

Resource servers must communicate with PIMOS according to a protocol stipulated by the resource tree. If it is desired to make a stream provided by that server into a stream that can also perform standard I/O, then it is necessary to add the function that performs standard I/O services to the function by calling the following predicate from the server program.

stdio_server:go(GR_IN, GR_OUT)

The following is an example of calling a standard I/O server that performs standard I/O services according to this procedure.

```
shoen:raise(pimos_tag#task,general_request,GR),
stdio_server:go(GR,GROut),                      <========
GROut = [register(window_request,{0,0,3},Res)],
```

There is no reason not to create one's own independent standard I/O server without using the existing standard I/O server (`stdio_server`). However, in that case it will be necessary to process the standard I/O messages that are stipulated by PIMOS.

In particular, the `standard_io/3` message must be processed correctly.

11.4 User Data Management

This section describes the user data managed by PIMOS and the access methods for that data.

11.4.1 User and group identifiers

The most important data items in the PIMOS managed user data are the user and group ids. They are described below.

User ID: Under PIMOS, each user is allocated a non-negative integer that is used as the user identifier. This identifier is called the user id.

In particular, the user with the user id 0 is called the **super user**, and is given system management privileges.

Group ID: PIMOS provides a framework that divides users into groups (with overlapping allowed) for use with protection modes of resources (see section 11.1.11 on page 431). A non-negative integer is allocated to each user group, and that number is used as the group identifier. This identifier is referred to as the group id.

In particular, the group with the group id 0 is called the **super group**, and the super user always belongs to this group.

11.4.2 User data databases

The user data managed by PIMOS includes the following data: user names, passwords, group names, and the currently logged in users. This user data is managed in databases. Accessing these databases is performed in the same manner as accessing the databases used for managing I/O equipment and program data, i.e., by sending requests to PIMOS. Note that database access is managed in the same manner as other resources, i.e., as PIMOS defined resources.

PIMOS provides the following devices as a resources for managing the user data database.

User device Manages the data for users that can log in to PIMOS. In concrete terms, this device manages user names, passwords, user ids, the top level goal on login (usually a shell), and current directories.

Group device Manages data related to the user groups defined by the user device. In concrete terms, this device manages group names, group ids, and the user ids associated with the groups. PIMOS checks the protection modes of supplied resources with the data from the user device described above. (Also see section 11.1.11 on page 431, and section 10.2.1.2 on page 407.)

Login device Manages data on the users who are currently logged in to PIMOS. In concrete terms, this device manages user names and login ids.

Note that the protocols used for accessing the user data databases can only be used by the super user, i.e., the user with user id 0.

11.4.2.1 User request device

```
shoen:raise(pimos_tag#task, general_request, GR)
```

A user request device can be acquired by first acquiring a stream to the general request device with the following command: and then sending commands similar to those for other resources to that stream.

The following commands can be sent to a user request device.

Command Protocol

create(user(^Result))

Creates a user device. If a device was created correctly, normal(UserDeviceStream, Abort, ^Attention) is returned in `Result`, otherwise abnormal is returned.

create(group(^Result))

Creates a group device. If a device was created correctly, normal(GroupDeviceStream, Abort, ^Attention) is returned in `Result`, otherwise abnormal is returned.

create(login(^Result))

Creates a login device. If a device was created correctly, normal(LoginDeviceStream, Abort, ^Attention) is returned in `Result`, otherwise abnormal is returned.

11.4.2.2 User devices

Command Protocol

set_up(UsersList, ^Result)

Registers the elements of the list `UsersList` as user data, and returns normal in `Result` on success, and abnormal on failure.

The format of `UsersList` must be as follows.

```
UsersList ::= [UserInfo...]
UserInfo ::= {UserName,Password,UID,Priority,Directory,Node,EntryPoint,CommentList}
EntryPoint ::= Package::Module:Predicate
```

```
CommentList ::= [Comment...]

UserName, Password, Directory : 16-bit strings (Password is the encrypted password)
Comment : A ground value containing no undefined variables
UID : An integer
Module, Predicate : Atoms
```

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, protection_violation will be returned in Result.

authenticate(Username, Password, ^Result)

Determines whether a user with the user name and password specified by `Username` and `Password` exists, and returns normal in `Result` on success, and abnormal on failure. There is no need to encrypt the password.

get_uid(UserName, ^Result)

Acquires the `UID` identifier for the user specified by `UserName`. Returns normal(UID) in `Result` on success, and abnormal on failure.

get_info(UID, ^Result)

Acquires the user data specified by `UID`. Returns normal({UserName, Priority, Directory, Node, EntryPoint, CommentList}) in `Result` on success, and abnormal on failure.

status(UID, ^Result)

Acquires the user reference data specified by `UID`. Returns normal(RefCount) in `Result` on success, and abnormal on failure. RefCount represents the number of executed get_info. After executing status, the object of the RefCount is cleared 0.

get_all_info(^Result)

Acquires all the user data registered in the user device. Returns normal(UserList) in `Result` on success, and abnormal on failure.

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, abnormal will be returned in Result.

change_password(UID, OldPassword, NewPassword,^Result)

Changes the password of the user with the user id specified in `UID` from `OldPassword` to `NewPassword`. Returns normal in Result on success, and abnormal on failure. Fails if the user id is not registered. Fails if OldPassword and the registered password do not agree.

11.4.2.3 Group device

Command Protocol

set_up(GroupsList, ^Result)

Registers the elements of the list `GroupList` as user data and returns normal in Result on success, and abnormal on failure.

The format of GroupList must be as follows.

```
GroupsList ::= [GroupInfo...]
GroupInfo ::= {GroupName,GID,[UID...],CommentList}
CommentList ::= [Comment...]

GroupName, UserName : 16-bit strings (Password is the encrypted password)
Comment : A ground value containing no undefined variables
GID : An integer
UID : An integer
```

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, protection_violation will be returned in Result.

get_gid(GroupName, ^Result)

Indentifies the GID identifier number for the group corresponding to the group name (16-bit string) specified in **GroupName**. Returns normal(GID) in `Result` on success, and abnormal on failure.

get_gids(UID, ^Result)

Identifies the GID identifier number for the group to which the user name belongs specified in `UID`. Returns normal([GID...]) in `Result` on success, and abnormal on failure.

GIDs are arranged in an ascending order in the resulting list of GIDs. In particular, the proceeding element of the GID list, i.e., minimum GID hold by the user is called **represent group ID**.

get_users(GID, ^Result)

Identifies the user to which the group belongs specified by `GID`. Returns normal([UID...]) in `Result` on success, and abnormal on failure.

get_all_info(^Result)

Acquires all the data for the user registered in the group device. Returns normal(GroupList) in `Result` on success, and abnormal on failure.

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, abnormal will be returned in Result.

11.4.2.4 Login device

Command Protocol

login(UserName, Info, ^Result)

Registers a user with the user name indicated by `UserName` along with the user data Info, and returns normal(LID) in Result on success, and abnormal on failure.

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, protection_violation will be returned in Result.

logout(LID, ^Result)

Deletes the user data indicated by `LID` and returns normal in Result on success, and abnormal on failure.

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, abnormal will be returned in Result.

get_info(LID, ^Result)

Acquires the user data indicated by `LID`. Returns normal({UserName, [Partition, Date, Time, IONode, IONodeUser, Network]}) in `Result` on success, and abnormal on failure.

get_login(UserName, ^Result)

Acquires the user data indicated by `UserName`. Returns normal([LID...]) in `Result` on success, and abnormal on failure.

get_all_info(^Result)

Acquires all the user data registered in the login device. Returns normal(LoginList) in `Result` on success, and abnormal on failure.

【Caution】 This protocol can only be used by the super user, i.e., the user with user id 0. If a general user attempts to use this protocol, abnormal will be returned in Result.

11.5 Standardly Supplied Error Reporting

PIMOS provides a standard framework for the error reporting format so that error reports from user programs can be handled in a unified manner. Use of this framework provides several advantages, including allowing the format of error messages displayed by the shell or listener to be unified and easy to understand, and reducing the efforts required in writing an error display routine. The remainder of this section describes the method for implementing standard error reporting in user programs.

11.5.1 Error reporting format

The following expression is used from user programs to enable standard format error reporting.

```
shoen:raise(pimos_tag#error, '$PIMOS$'(Error,Info), Data)
```

The arguments are interpreted as follows.[1]

Error : Must be a ground value. This is the value that will be output using the buffer putt message. Usually, the name of the device or utility that caused (or detected) the error is passed in this variable.

Info : Must be a character string. This value will be output using the buffer putb message. Normally a description of the error is passed in this variable.

Data : An arbitrary term can be specified. Variables may be included. The shell or the listener will output this term via the buffer putwt message after wrapping it. Normally, the message or message argument that caused the error is passed in this variable.

[1] See section 9.2.1, "Buffer", (page 355) for a description of the buffer messages.

【Hint】 Refer to the following sections for a description of the display format for the error reporting described here.

Shell: Section 2.1.5.3

Listener: Section 4.1.8.3

付録 C

HOW TO CREATE A STRING I/O DEVICE ON FEP

String I/O devices provide an interface for manipulating under PIMOS devices that are defined by the user on an FEP using ESP. This section describes the definition method for arbitrary string I/O devices in ESP. The package names mentioned below are all ESP packages used under SIMPOS.

A device name for string I/O operations is generally specified in the format of "package_name##class_name". The package name cannot be omitted. (Operation is not guaranteed if the package name is omitted, since the default package will be used when a device is generated.) Therefore, the ESP program must be defined with the package and class names corresponding to the device name by the user.

When creating this device program, the following rules must be followed. However, these rules are only valid for user-defined string I/O device programs (including classes that inherit those programs if inheritance is used), and users may use arbitrary definition types in other ESP programs that are called from string I/O device programs.

【Hint】 The classes are automatically loaded when the devices are created, once their templates are saved.

Class to be inherited: To obtain the function as a PIMOS device (data conversion between KL1 and KL0 or the like), the class, pimfep##as_pimos_string_io_device, is inherited.

Classes that cannot be inherited: Since inheritance has already been performed, simpos##as_program cannot be inherited.

Also, since simpos##as_resource is a class that inherits simpos##as_program due to SIMPOS rules, this class also cannot be inherited. Therefore, this class cannot be inherited by the classes that inherit pimfep##as_pimos_string_io_device.

Since classes such as windows and filters inherit simpos##as_resource, they also cannot be inherited.

Method or slot that cannot be defined: The following methods and slots have already been defined by the class, pimfep##as_pimos_string_io_device. Therefore, the user must not define them.

【Caution】 Since pimfep##as_pimos_string_io_device inherits simpos##as_program, methods and slots that are defined by classes that inherit this class must not be defined. (See the "Miniature PSI/PIMOS Programming Guide (I)" for details.)

Class method: :stream/2, :process/4

Instance method: :abort_loop/1, :loop_main/1, :reply_message/2, :set_device_id/2, :set_creation_status_stream/2, :set_output_stream/2, :set_input_stream/2,

:set_close_flag/2, :device_id/2, :creation_status_stream/2, :output_stream/2, :input_stream/2, :aborted/1, :attention/1, :disconnected/1, :is_closed/1, :close/3, :initiate/1, :goal/1, :trace_string_io/1, :abort_string_io/2, :do_io/6, :set_pimfep_trace_flag/2, :pimfep_trace_flag/2, :io/5, :exit/1

Instance attribute slot: none

Method to be defined: The following instance method is defined:

:io(Instance, Command, FromMulPString, ^ToMulPString)

The arguments are as follows:

Instance: The instance object of the device

Command: An integer to be used as an ID indicating the type of an I/O instruction

FromMulPString: A 16-bit string. Data sent from the parallel inference machine (KL1 side). Usually, data needed for execution of an I/O instruction is entered.

ToMulPString: A 16-bit string. Data sent to the parallel inference machine (KL1 side) after an I/O instruction is executed. Normally, the result of executing an I/O instruction is entered.

This method is defined so that an I/O instruction specified in Command is to be executed. When the execution of an actual I/O instruction fails for some reason, the method should be 'fail' so that the Status of I/O message on the KL1 side indicates "abnormal". No data is needed to be sent to the KL1 for some I/O instructions. In such a case, ToMulPString remains undefined.

Initialization: A device (instance) is usually initialized when it is created. However, the initialization of a string I/O device must be defined as one of the general I/O instructions.

【Hint】 This is due to the following reasons.

- Since the structure that manages the PIMOS device on the FEP does not function if initialization is performed at device creation time, error and termination processing cannot be guaranteed.
- If initialization is performed at device creation time, it is hard to determine whether a creation failure was due to a failure to create the specified device, or a failure to initialize that device.

Abort processing: As is the case with other PIMOS devices, string I/O devices can be aborted from KL1 programs. Although the currently executing :io/4 is aborted when a string I/O device is aborted, there is no process that is defined in particular to perform the abortion processing. Therefore the user can define :device_abort(Instance) when it is desirable to perform some processing under the string I/O device.

Termination processing: A process of a string I/O device terminates when the stream of KL1 is closed. If you want to process any data before termination, you may define an instance method :pimos_user_exit(Instance) for that purpose.

Debugging: When debugging a program for a defined string I/O device, it is possible to trace the program starting at a particular point using the SIMPOS debugger, by calling :trace_string_io(Instance) from that point in the program.

【Caution1】 This function does not result in tracing only the user-defined methods, but rather traces the whole device program. Therefore, care is required to avoid completely

blocking the device's processing. (We recommend using the SIMPOS spy function to trace only the user-defined methods.)

【Caution2】 This function can only be used in SIMPOS version 7.1 and later. (There are cases in versions of SIMPOS prior to version 7.1 in which this function does not work correctly.)

【Caution3】 This function cannot be used when PIMOS is used from a remote I/O node.

Example of Programming

The following is an example of a device program. In this program, commands are entered in the menu window and echoed back to another window.

ESP program on FEP

```
use_package(my_device_package).
class string_io_example has
nature
        pimfep##as_pimos_string_io_device ;
instance
attribute
        echo_window, read_menu ;

% Receives the device initialization command.
:io(Obj, 0, FromMulPString, ToMulPString) :- !,

    % Starts program tracing.
    :trace_string_io(Obj),
    %

    :init(Obj, FromMulPString, ToMulPString) ;
%%
:io(Obj, 1, FromMulPString, ToMulPString) :- !,
    :read_from_menu(Obj, FromMulPString, ToMulPString) ;
:io(Obj, 2, FromMulPString, ToMulPString) :- !,
    :write_to_window(Obj, FromMulPString, ToMulPString) ;

% Actual initialization processing
:init(Obj, Input, "") :-
    ItemsList = [
        {"test", common,0,_},      {"analysis",common,1,_},
        {"manipulate",common,2,_}, {"check",common,3,_},
        {"exit",common,4,_}],
    :create(#menu, [title("MENU"),items_list(ItemsList)], Menu),
    :activate(Menu), Obj!read_menu := Menu,
    string_element(Input, 0, Sizex),
    string_element(Input, 1, Sizey),
    string_element(Input, 2, Positionx),
    string_element(Input, 3, Positiony),
```

```
    :create(#pmacs_window,
        [size(Sizex, Sizey),position(Positionx, Positiony),
         title("ECHO WINDOW")], W),
    :activate(W), Obj!echo_window := W ;
%%

:read_from_menu(Obj, Input, String) :-
    W = Obj!echo_window,
    :putl(W, Input), :flash(W),
    :read(Obj!read_menu, Type),
    new_string(String, 1, 16),
    set_string_element(String, 0, Type) ;

:write_to_window(Obj, Input, "") :- !,
    W = Obj!echo_window,
    :putl(W, Input), :flash(W) ;

% Termination processing
:pimos_user_exit(Obj) :- !,
    W = Obj!echo_window,
    :putl(W, "Terminate Device"), :flash(W) ;
%%

end.
```

KL1 program on Unit

```
:- module string_io_example.
:- public go/0.
:- with_macro pimos.

go :- true |
    shoen:raise(pimos_tag#task, general_request, GRD),
    GRD = [string_io(STGIO)],
    go1(STGIO).

go1(normal(STGIO,_,_)) :- true |
    STGIO = [create(string#"my_device_package##string_io_example",STGIOR)],
    go2(STGIOR).

go2(normal(STGIO,_,_)) :- true |
    new_string(S, 4, 16),
    set_string_element(S, 0, 400, S1),
    set_string_element(S1, 1, 190, S2),
    set_string_element(S2, 2, 180, S3),
    set_string_element(S3, 3, 0, S4),
    STGIO = [io(0,S4,FromFEP)|STGIOR],
    read_menu(FromFEP, STGIOR).
```

```
read_menu(normal(_), STGIO) :- true |
    STGIO = [io(1,string#"PLEASE SELECT MENU",FromFEP)|STGIOR],
    execute(FromFEP, STGIOR).

execute(normal(FromFEP), STGIO) :- true |
    string_element(FromFEP, 0, Type, _),
    processing(Type, STGIO).

processing(0, STGIO) :- true |          % "test"
    STGIO = [io(2, string#"test start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(1, STGIO) :- true |          % "analysis"
    STGIO = [io(2, string#"analysis start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(2, STGIO) :- true |          % "manipulate"
    STGIO = [io(2, string#"manipulate start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(3, STGIO) :- true |          % "check"
    STGIO = [io(2, string#"check start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(4, STGIO) :- true |          % "exit"
    STGIO = [io(2, string#"SESSION END", _)].
```

付録 D

PREPROCESSOR CREATION

D.1 Overview

A preprocessor is a self-compiler structure for easing the development of higher level languages based on KL1. The user can add a preprocessor simply by creating modules according to the rules described below, and informing the self-compiler of those modules. (The method for performing this addition is described in chapter 3, "COMPILERS", page 44.)

【Caution1】 While many of the specification described below differ from version 2.5, the version 3.0 preprocessor function can actually be used in version 2.5. However, the source file position display functions introduced in version 3.0 are invalid.

【Caution2】 If a bug or abnormality occurs in a user-defined preprocessor, an exception will be raised, and as a result it is possible that the self-compiler may crash.

【Hint】 A verification function was added in version 3.0. This corresponds to the UNIX C compiler lint function, and is provided to perform static analysis and checking of user-defined programs. In the PIMOS self-compiler, this function is generally referred to as the verifier, and it is possible for the user to install a similar user-defined verifier in the self-compiler.

D.2 Structure of the Preprocessor

- Preprocessors are allocated in compilation in the self-compiler by the following procedure.

  1. Parser allocation

     If a parser corresponding to the extension of the opened file can be found, that parser is connected to the file stream. If no corresponding parser can be found, the standard parser, kl1_standard_preprocessor, is allocated.

  2. Preprocessor allocation

     A preprocessor is allocated by searching for a parser corresponding to the extension of the opened file. If a corresponding preprocessor cannot be found, the system assumes that the language is KL1 and searches for a corresponding preprocessor.

  3. Preprocessor execution

     The parser output is connected to the preprocessor, and the preprocessor is executed. At the same time the preprocessor output language is extracted from the preprocessor settings, and if that output language is not "pkl1", another preprocessor is allocated. (This is called preprocessor chaining.) If the output is "pkl1" and the preprocessors have terminated correctly, the KL1 compiler is invoked.

4. Display message sorting

   After confirming that execution has completed normally, the messages output by the parser and preprocessor are sorted. If execution terminated abnormally, the results of preprocessor processing following that termination are discarded. If a comparison predicate is defined in the parser, then the sorting is performed using that comparison predicate. If there is no such predicate defined, then the comparison predicate defined in the kl1_standard_preprocessor is used.

   【Hint】 The reason that the display messages are sorted by the positional information is that it is desirable to output the displayed messages in their order in the source file, even for messages from later stages of preprocessing.

5. Display message output

   If a predicate called message/4 is defined in the parser, then how the information concerning position in the sorted display messages is displayed will be determined by that predicate. If no such predicate is defined, then this sort of filter processing will not be performed.

- Also, the `!translate` command is used with the following sequence.

1. Parser allocation

   Identical to that for the compiler.

2. Preprocessor allocation

   Identical to that for the compiler.

3. Preprocessor execution

   While the sequence through preprocessor chaining is identical to that for the compiler, at the point that either chaining runs out or the output language is "pkl1", an output file is opened, and, if an unparser for the corresponding language is found, the output from the parser and preprocessor is connected, and it is executed. If no such unparser is found, the standard unparser, the unparser for the kl1_standard_preprocessor is allocated.

4. Display message sorting Identical to that for the compiler.

5. Display message output

   Identical to that for the compiler.

- On the other hand, the procedure for the `!verify` command is as follows.

1. Parser allocation

   Identical to that for the compiler.

2. Verifier allocation

   If a verifier is assigned for the corresponding language, the parser output is connected to that verifier. If no verifier is assigned, a corresponding preprocessor is assigned in the same manner as for the compiler.

3. Preprocessor execution

   While the sequence through preprocessor chaining is identical to that for the compiler, at the point that either chaining runs out or the output language is "pkl1", if no verifier has been assigned processing is terminated with an error.

4. Verification across files When all verifier allocation for the specified files has completed, if the predicate inter_file_verify/3 is defined in the verifier, then inter-file verification is executed for that predicate.

5. Display message sorting
   Identical to that for the compiler.
6. Display message output
   Identical to that for the compiler.

This section provides a graphic representation of the compilation structure in the compile, translate, and verify commands for cases where the following sort of preprocessor is assigned.

```
COMPILE> !preprocessor lm2 kl1 my_preproc
COMPILE> !parser lm2 my_parser
COMPILE> !checker lm2 me_verifier
```

- **KL1** For KL1 language compilation

```
[kl1 source file] → kl1_standard_preprocessor (parser)
                          ↓ [kl1 language]
                    kl1_macro_expander (macro expansion preprocessor)
                          ↓ [pkl1 language]
                    kl1 self-compiler → [module]
```

- For **LM2** language compilation

```
[lm2 source file] → my_parser
                          ↓ [lm2 language]
                    my_preproc
                          ↓ [kl1 language]
                    kl1_macro_expander (macro expansion preprocessor)
                          ↓ [pkl1 language]
                    kl1 self-compiler → [module]
```

- For **KL1** language translation

```
[kl1 source file] → kl1_standard_preprocessor
                          ↓ [kl1 language]
                    kl1_macro_expander (macro expansion preprocessor)
                          ↓ [pkl1 language]
                    kl1_standard_preprocessor → [pkl1 source file]
                     (unparser)
```

- For **LM2** language translation

```
[lm2 source file] → my_parser
                          ↓ [lm2 language]
                    my_preproc
                          ↓ [kl1 language]
                    kl1_macro_expander (macro expansion preprocessor)
                          ↓ [pkl1 language]
                    kl1_standard_preprocessor → [pkl1 source file]
                     (unparser)
```

- For **KL1** language verification

```
[kl1 source file] → kl1_standard_preprocessor (parser)
                          ↓ [kl1 language]
                    kl1_macro_expander (macro expansion preprocessor)
                          ↓ [pkl1 language]
                    kl1_standard_preprocessor (verifier)
```

- For **LM2** language verification

```
[lm2 source file] → my_parser
                              ↓ [lm2 language]
                          my_verifier
```

To avoid repetition in the sections that follow, here we provide description of variables whose usage is common to the predicates described.

- DevIn, DevOut

  DevIn is a stream to the input device (file). No buffer is inserted.

  DevOut is a stream to the output device (file). As is the case with DevIn, no buffer is inserted.

- In, Out

  These are lists passed between preprocessors. In is a list that is passed to a preprocessor, and Out is a list that is output from a preprocessor. In is the output from the parser or preprocessor at the previous stage, and Out is the input to the preprocessor or unparser at the next stage.

  The elements in these lists are vectors of the form {**(-),WrappedTerm,PositionInfo**}. Here WrappedTerm is a wrapped term to which getwt was applied from a buffer. PositionInfo is 2-element vector of the form {**Position,Info**}, where Position is a term that indicates the position in the source file, and Info is a term that the parser can embed arbitrarily. Position is a list that consists of the source file name string and an integer that indicates the position in the file, in that order. If the source file is included in another file by the include mechanism, then two elements are appended to this list: the file name string for the include file and an integer indicating the position. Thereafter, if the include depth increases, this list is increased accordingly.

  【Hint】 The units term, line, and character can be considered for use as the units for the integers that indicate position in the source file.

  【Caution】 In the specifications for this list in version 2.5, the list elements were simply of type WrappedTerm.

- VP

  A stream to the compiler option storage pool. See the section "Preprocessor Creation Tools" (page 464) for details on the protocol for this stream. Note that a merger is inserted.

- MESG

  A stream for message output. See the section "Preprocessor Creation Tools" (page 465) for details on the protocol for this stream.

- MESGnew, MESGold

  These are differential streams for message output. See the section "Preprocessor Creation Tools" (page 465) for details on the protocol for this stream.

- status

  This is a variable for reporting normal or abnormal termination of parser or preprocessor processing. A return value of normal indicates normal termination, and one of abnormal indicates abnormal termination.

  【Caution】 If an error is output in a display message (messages error/2 or error/3 in the protocol) then this variable will always be set to abnormal.

D.3 Predicates Common to Parsers, Preprocessors, Unparsers, and Verifiers

Preprocessor:describe(^VP, ^MESGold, MESGnew)

A predicate provided to display the option settings valid for this module. It is possible to confirm the actual option settings using this predicate. Called from the self-compiler's `!describe` command.

D.4 Parser Module Creation

- Required public predicates

  Parser:parse(DevIn, ^Out, ^VP, ^MESG, ^Status)

  Specifies the syntactic analysis.

  【Caution】 The parse predicate in version 2.5 was a 6-argument predicate, but it has been changed to be a 5-argument predicate starting with version 3.0. However the version 2.5 parse/6 can still be used.

- Optional public predicates

  Parser:sort(MessageX, MessageY, Small, Large Swapped)

  A public predicate for sorting display messages based on their positional data. (See chapter 8, "UTILITIES", (page 317) for the specifications of this predicate.) MessageX and MessageY are messages on the MESG stream, and are sorted on the basis of the positional information they hold. Caution is required regarding messages that include no positional information. Messages with no positional information are taken to be message that proceed all messages that include positional information.

  The predicate **kl1_library_message:position(Message, ^Position)** can be used to extract the positional information from MessageX and MessageY. See the section titled "Library message output" (page 467) later in this appendix for details on messages.

  Parser:message(Mode, MESGin, ^VP*, ^MESGout)

  A filter for determining how much of a display message to display. It mainly specifies processing that determines how to display the arbitrary terms that the parser adds to the positional information. The variable Mode is instantiated to either the atom silent or the atom verbose specified with the self-compiler `!message` command. (See chapter 3, "COMPILERS", page 46.) MESGin is a display message stream, and MESGout is the output after passing through the display filter.

【Hint】 The parser definition framework:

```
:- module my_parser.
:- public parse/5.
:- public sort/5, message/4.    % Optional public predicate

parse(In,Out,VP,MESG,Status) :- true |
```

```
    ....

sort(X,Y,Small,Large,Swapped) :- true |
    ....

message(Mode,MESGin,VP,MESGout) :- true |
    ....
```

D.5 Preprocessor Module Creation

A preprocessor module must have the following public predicates.

Preprocessor:translate(In, ^Out, ^VP, ^MESG, ^Status)

Specifies a language translation.

【Caution】The translate predicate in version 2.5 was a 6-argument predicate, but it has been changed to be a 5-argument predicate starting with version 3.0. However the version 2.5 translate/6 can still be used.

【Caution】In version 2.5 it was necessary to define a translate/2 predicate, but that is no longer necessary in version 3.0.

【Hint】The preprocessor definition framework:

```
:- module my_preproc.
:- public translate/5.

%% translate(In,Out) :- true | In = "abc", Out = "kl1". %% No longer required as of version

translate(In,Out,VP,MESG,Status) :- true |
    ....
```

D.6 Unparser Module Creation

An unparser module must define the following public predicates.

Unparser:unparse(In, ^DevOut, ^VP, ^MESG, ^Status)

In is a list of wrapped terms that were converted thus far by preprocessors, and Out is a stream to the output device. No buffer is inserted in that stream.

【Caution】The unparse predicate in version 2.5 was a 6-argument predicate, but it has been changed to be a 5-argument predicate starting with version 3.0. However the version 2.5 unparse/6 can still be used.

【Hint】The unparser definition framework:

```
:- module my_unparser.
:- public unparse/5.
```

```
unparse(In,Out,VP,MESG,Status) :- true |
    ....
```

D.7 Verifier Module Creation

- Required public predicates

  Verifier:verify(In, ^IFC, ^VP, ^MESG)

  Performs a static analysis on the list of terms input from In, and outputs the result to MESG. Here IFC is a stream for performing inter-file checks, and is intended to provide adequate information to the predicate inter_file_verify/3 (described later) for performing inter-file verification. The protocol specifications for this stream are arbitrary. However, the empty list [] must be bound to this variable if inter-file verification is not performed. If this condition is not met, a deadlock will occur.

- Optional public predicates

  Verifier:inter_file_verify(IFCs, ^VP, ^MESG)

  A predicate for performing static verification that spans files. IFCs is a structure of the form [{**Messages, Sorter**}, **...**], Messages is the second argument to the predicate verify/4 described earlier, and Sorter is the code for a comparison predicate (sort/5) for use in sorting display messages.

  This sort is for sorting display messages when positional information is included in the argument Messages. Whether the messages are sorted or not, and if sorted, in what order they are sorted is left up to this predicate. Also, whether the supplied comparison predicate is used or not is left up to this predicate.

  【Caution】 More than the language that the verifier corresponds, it is the possibility of the existence of an upper level preprocessor that requires caution, since the format of the positional information and the specifications of the comparison predicate may differ.

【Hint】 The verifier definition framework

```
:- module my_verifier.
:- public verify/4.
:- public inter_file_verify/3.    % Optional public predicate

verify(In,IFC,VP,MESG) :- true |
    ....

inter_file_verify(IFCs,VP,MESG) :- true |
    ....
```

D.8 Preprocessor Creation Tools

D.8.1 The KL1 standard preprocessor

The following predicates from the kl1_standard_preprocessor are provided to users to aid user preprocessor development.

kl1_standard_preprocessor:parse_with_op_pool(DevIn, ^Out, ^OpPool, ^VP, ^MESG, ^Status)

A predicate that can be used when creating a parser with operators that differ from the KL1 standard operators. The argument OpPool is a stream to an operator pool that uses the buffer that is used in reading terms from the input device.

【Example】 Definition of a parser with operators differing from the KL1 standard.

```
parse(In,Out,VP,MESG,Status) :- true |
    kl1_standard_preprocessor:
        parse_with_op_pool(In,Out,MyOpPool,VP,MESG,Status),
    operator:empty_pool(MyOpPool0),    % Generates an empty operator pool.
    MyOpPool0 = [add_op(...), add_op(...), ... |MyOpPool].
    % Defines the desired operators.
```

D.8.2 Compiler option storage pool

The compiler option storage pool accepts the following messages. A merger is inserted in the stream to this pool.

get_atom_table(AtomTable*)

Returns a stream to the atom table. Note that a merger is inserted in this stream. (See chapter 10, "Program Management", on page 405 for a description of the protocol for this stream.)

get_directory(Directory)

Returns the currently assigned current directory as a string.

get_file_request(FileRequest*)

Returns a stream to a file request device. Since the default directory setting is performed to this file request device, this stream should be used when opening files other than the source file in a preprocessor. (See section 9.1, "I/O Resource Acquisition", on page 382 for a description of the protocol for this stream.)

get_module_table(ModuleTable*)

Returns a stream to a module table. Note that a merger is inserted in this stream. (See chapter 10, "Program Management", on page 406 for a description of the protocol for this stream.)

get_package(DefaultPackage)

Returns the currently assigned default package as an atom.

input_language(InputLanguage)

Returns the language of the file currently being compiled as a string.

source_filename(FileName)

Returns the name of the file currently being compiled as a string.

add_dependency(FileName,DateTime)

Sets the file name and file update date and time (as a 2-element vector consisting of the date as an integer and the time on that day as a millisecond interval) for a file that the module currently being compiled depends on. When this assignment is performed, the self-compiler `!make` command will check the update date and time for this file for modules generated by this compilation.

do(Stream*)

Causes continuous processing to be performed when messages are sent to Stream. A merger is inserted in this stream.

get(Key,Status)

An access protocol for the preprocessor variable pool. Key is an arbitrary KL1 term (variables are excluded). For Status, either the value assigned to the environment variable compile:language_options or the value set with the interactive compiler `set` command is returned in the Values variable of the normal(Values) term. If the corresponding key does not exist, non_existent is returned.

【Caution】 In version 2.5 Key was required to be a 16-bit string, but in version 3.0 arbitrary base values are allowed.

D.8.3 Library message output

Library message output is a process that is used by the PIMOS libraries, such as the compiler. Messages are automatically distributed either to the shell's standard output (see section 9.4, "Standard I/O", on page ?? for a description of the protocol for this stream) or to standard message output (see page 402 in the same section). This process also maintains a count of errors and warnings and can output those counts.

The remainder of this section describes the messages accepted by this process. Note that the variable FormatString that appears in these messages is the same as that in the format for the formatted output putf predicate from the buffer utilities, and that Args is the argument list for that format.

info(FormatString,Args)

Outputs a message in the specified format to the shell's standard message output.

info(FormatString,Args,PosInfo)

After outputting the positional information to the shell's standard message output, functions identically to info/2.

message(FormatString,Args)

Outputs a message in the specified format to the shell's standard output.

message(FormatString,Args,PosInfgo)

After outputting the positional information to the shell's standard output, functions identically to message/2.

warning(FormatString,Args)

Outputs the message in a format with the character string "!WARNING!" prefixed to standard output. Note that this message is counted as a warning.

warning(FormatString,Args,PosInfo)

After outputting the positional information directly after the character string "!WARNING!", functions identically to warning/2.

error(FormatString,Args)

Outputs the message in a format with the character string "!ERROR!" prefixed to standard output. Note that this message is counted as a error.

error(FormatString,Args,PosInfo)

After outputting the positional information directly after the character string "!ERROR!", functions identically to error/2.

syntax_error(ErrorInfo)

A message provided to report syntax errors. Outputs to standard output.

syntax_error(ErrorInfo,PosInfo)

After outputting the positional information, functions identically to syntax_error/1.

stdout(Messages)

Identical to executing a do(Messages) for the shell's standard output.

stdout(Messages,PosInfo)

After outputting the positional information, functions identically to stdout/1.

stdmsg(Messages)

Identical to executing a do(Messages) for the shell's standard message.

stdmsg(Messages,PosInfo)

After outputting the positional information, functions identically to stdmsg/1.

flush

Flushes both standard output and standard message output. Essentially equivalent to the flush/1 message sent to a buffer.

【Caution】 Since display messages are now sorted in versions starting with PIMOS version 3.0, a significant effect cannot be expected from this message.

Also, The following predicate is provided for use in the definition of a sort/5 predicate for use in a parser module.

kl1_library_message:position(Message,^Position)

A predicate for extracting positional information from messages corresponding to library message output. The argument Message is a message, and Position is the position (an integer) included in that message. The atom [] is returned for a message that has no positional information.

D.9 Programming Examples

【Example】 Test stub for the preprocessor

```
% InFile : A stream to the input file.
% OutFile : A stream to the output file.
test(InFile,OutFile) :- true |
    kl1_library_message:message(MESG0),
    kl1_library_variable_pool:compile(VP0,MESG0,MESG1),
    merge({VP1,VP2,VP3},VP0),
    module_table:cache_table([get_code(my_parser,sort,5,SortCode)]),
    sorter:sort(MESG2,SortCode,MESG1),
    merge({MESG3,MESG4,MESG5},MESG2),
    my_parser:parse(InFile,List0,VP1,MESG3,ParseStatus),
    my_preproc:translate(List0,List1,VP2,MESG4,PreproStatus),
    my_unparser:unparse(List1,OutFile,VP3,MESG5,UnparseStatus).
```

【Caution】 Since, strictly speaking the kl1_library_variable_pool module is not a compiler option storage pool as used in a preprocessor, it will not accept the do/1, add_dependency/2, source_filename/1, and input_language/1 messages.

付録 E

PROTOCOL DEFINITIONS

E.1 Outline

To issue a request to PIMOS, a user program sends a message via the stream. In PIMOS, a specification language (hereafter, called the protocol definition language) is used to define protocols for messages. A part of PIMOS is automatically created from protocol definitions written in this language.

The protocol definition language is outlined and the practical protocol definitions are presented in the following sections.

E.2 Protocol Definition Language

E.2.1 Grammar

The grammar of the protocol definition language is as follows:

```
 definition ::= name[(formals)] = [inheritance] body.

inheritance ::= {super &}
      super ::= invocation
       body ::= [sign] (invocation | {enumeration}) [error]
 invocation ::= name [(actuals)] | parameter
enumeration ::= pattern [error] {; pattern [error]}
    pattern ::= atom [(actuals)]
    formals ::= parameter {, parameter}
    actuals ::= body {, body}
  parameter ::= name
       sign ::= + | -
      error ::= # [atom] | ## [atom]

  |     Select either of the parameters.
{ }     Repeat the parameter zero or more times.
[ ]     Omissible
```

E.2.2 System defined protocol

The system defined protocol includes atom, integer, string, code, module, ground, any, list(p), vector(p), vector([p1,p2,$\cdots$,pN]), and stream(p).

atom, integer, string, code, and module specify the type of data to be handled. ground indicates that the data is fully instantiated. any indicates that any data including variables can be specified.

In stream(p) or list(p), the protocol designates a stream or list and the element is determined by the protocol specified in p. The vector can be described in either of the two formats: vector(p) or vector ([p1,p2,$\cdots$,pN]). vector(p) is defined that the size of the vector is not fixed and the protocol of each element follows the p. vector ([p1,p2,$\cdots$,pN]) is defined that the size of the vector N and that p1, p2, $\cdots$, pN represent the protocols of elements.

E.2.3 Description of grammar

This section makes brief explanation about the grammar mentioned in the previous section.

E.2.3.1 Protocol definition

Protocol definition: A protocol can be defined in the following format:

Protocol name = Protocol definition

In a protocol definition, a message pattern or a protocol name to be referred to can be described. A message pattern is encircled with { } and partitioned with ';'.

E.2.3.2 Inheritance

Multiple inheritance of protocols is available when the protocol names are specified.

E.2.3.3 Sign

A sign is used to identify a person who sets the protocol. The default value of the sign is '+'. If '−' is specified, the protocol is assumed to have been set by a person different from one who has set the protocol referencing it.

E.2.3.4 Protocol definition with a parameter

name(formals) = [inheritance] body.

A user can define a protocol with a parameter in the above format. The protocol receives an actual value corresponding to the parameter when it is referred to. The value is passed to the right hand side and expanded to be an actual definition.

E.2.3.5 Error data specification

Following a protocol definition, it is possible to specify the error data that the protection filter outputs when there is a mistake in the use of that protocol in a user program. This specification has two levels, and these consist of error data (#[atom]) that is added to previously specified data and error data and (##[atom]) that replaces the previously specified data, which is discarded.

If the error data is omitted and only '#' or '##' is specified, either the protocol name for that definition or the "atom/arity" for the pattern is assigned as a default value.

E.3 How to Read a Protocol Definition

This section explains how to read a protocol definition using the first portion of a PIMOS-supported message protocol.

```
genreq_filter = stream(genreq_command).                    (1)

genreq_command = ask_task_command & {                      (2)
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
        execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#, fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_super(modulename#,predname#,arity#,argv#,taskname#,
                minprio#,maxprio#,tag#,initrsc#,-task)#;
        execute_super(modulename#,predname#,arity#, argv#, taskname#,
                minprio#, maxprio#, tag#, -task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#,userid#,-task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, user#, -task)#;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#,user#,fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, user#, fepnodeno#,ruser#,rpassword#,-task)# ;

        suicide(-status)# ;
        abort(rscpath# , -status)# ;
        number_of_feps(-status(integer))# ;

        setenv(envname# , value#, -status)# ;
        getenv(envname# , -status(value))# ;
        getenv(-status(envs))# ;
        unsetenv(envname, -status)# ;

        window(-rscstream(winreq_command)##window_request)# ;
        file(-rscstream(filereq_command)##file_request)# ;
        timer(-rscstream(timerreq_command)##timer_request)# ;
        string_io(-rscstream(strioreq_command)##string_io)# ;
        module(-rscstream(modulereq_command)##module_request)# ;
        atom(-rscstream(atomreq_command)##atom_request)#;
        user(-rscstream(userreq_command)##user_request)#

}.

ask_task_command = {                                       (3)
```

```
        id(-status(integer))# ;
        absolute_id(-status(rscpath))# ;
        status(rscpath# , rsckind# , verbosity# , -status(rscinfo))#
}.
```

(1) indicates the definition of the top-level protocol. genreq_filter defines the stream for general_request_device which is the top-level device of the resource tree. Messages can be sent to the general_request_device via this stream. stream(genreq_command) on the right hand side indicates that only the message defined with a protocol name of genreq_command can be sent to the stream.

The definition of a protocol name of ask_task_command is inherited as the definition of genreq_command. The structure encircled with { } and three structures defined in ask_task_command constitute an actual message. The message is defined with the message name (described in atom) and the definitions of protocols for the arguments. A message may be an atom without any argument.

Take the following two examples to discuss messages:

- setenv(envname# , value# , -status)#
- window(-rscstream(winreq_command)##window_request)#

setenv(envname# , value# , -status)#

The above message represents a message for setting a value in an environment variable, and has three arguments. When values are set in envname and value, and then the message is sent, a status value will be returned. This is defined with the sign -. For envname, value, and status, the following definitions are provided in the last part:

```
envname = ground.
value = ground.

status = {
        normal ;
        abnormal ;
        aborted
}.
```

Both envname and value are defined as ground. Therefore, the set values are the ground-type data. status is defined as any atom of normal, abnormal, and aborted.

setenv(envname# , value# , -status)#

The single pound marks of this form specify the addition of error data. Since the error data is not specified, 'envname', 'value', and 'setenv/3' are assigned as the default values.

window(-rscstream(winreq_command)##window_request)#

The above message is used to acquire window_request_stream. When this message is sent, an argument value is returned. The value is defined as rscstream(winreq_command). This is a user-defined protocol with a parameter. rscstream( ) is defined as follows:

```
rscstream(p) = {
        normal(-stream(p), -abort, attention) ;
        abnormal ;
        aborted
}.
```

Table E.1: Message Protocols Corresponding to Target Devices

Message Destination	Protocol Name
general_request_device	genreq_command
window_request_device	winreq_command
file_request_device	filereq_command
timer_request_device	timerreq_command
string_io_request_device	strioreq_command
module_request_device	modulereq_command
atom_request_device	atomreq_command
window_device	window_command
file_device	rfile_command/wfile_command/dir_command
timer_device	timer_command
string_io_device	strio_command
module_device	module_command
atom_device	atom_command
task	control_command/report_command

p in rscstream(p) is a parameter. When the protocol is referred to, a value is passed to the parameter. In this example, the value represents winreq_command. Then, any one of the following values is returned in window(-rscstream(winreq_command)):

- normal(-stream(winreq_command), -abort, attention)
- abnormal
- aborted

The first argument in normal( ) is window_request_stream, the second argument is an abort line, and the third argument is an attention line. The message which is defined with the protocol name of winreq_command can be sent on window_request_stream. The '−' signs added to the first and second arguments indicate that the values the user issues to PIMOS are set in the arguments of window_request_stream and the abort line, while normal( ) is returned from PIMOS to the user.

window(-rscstream(winreq_command)##window_request)#

In the above form, all the previously assigned error data is cleared from the acquired window_request_stream, and 'window_request' is assigned as the error data. These types of specifications allow error data to be given for each level in the resource tree. These specifications allow this error data to be raised and displayed when a protection filter discovers a protocol error.

Thus, reading protocol definitions leads to understand message protocols correctly.

To help check the definitions of messages, Table E.1 lists the destinations of messages and the protocol names used in the protocol definitions.

E.4 Definition of Protocol

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the general request device
%
```

```
% There are three types of message as follows.
%        (1) inquiring the status
%        (2) inquiring and setting of environment variables
%        (3) acquiring request stream to each device
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

genreq_filter = stream(genreq_command).

genreq_command = ask_task_command & {
        execute(execode# , argv# , initrsc_or_list# ,-task)# ;
        execute_super(modulename# , predname#,arity#,argv# ,
                      initrsc_or_list# ,-task)# ;
        profile(execode# , argv# , initrsc_or_list# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc_or_list# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
        profile(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc# ,-task)# ;
        profile(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
  execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                 tag#,initrsc#,
                 fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                 tag#, fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_super(modulename#,predname#,arity#,argv#,taskname#,
                minprio#,maxprio#,tag#,initrsc#,-task)#;
  execute_super(modulename#,predname#,arity#, argv#, taskname#,
                minprio#, maxprio#, tag#, -task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#,initrsc#,userid#,-task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#, user#, -task)#;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#,initrsc#,user#,
               fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#, user#,
               fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_user_with_proc(execode#, argv#, taskname#, minprio#, maxprio#,
               integer,integer,tag#,initrsc#,userid#,-task)# ;

        suicide(-status)# ;
        abort(rscpath# , -status)# ;
        number_of_feps(-status(integer))# ;

        setenv(envname# , value#, -status)# ;
        getenv(envname# , -status(value))# ;
```

```
        getenv(-status(envs))# ;
        unsetenv(envname, -status)# ;

        get_puid(-status(uid))# ;
        connection(-status(node_name))#;

        register(atom#, vector([uid,gid,integer]),
                 -register_status##register)# ;
        request(atom#, any, -status_any0##request)# ;
        request_anyway(atom#, any, -status_any0##request_anyway)# ;

        window(any)# ;
        file(any)# ;
        timer(any)# ;
        string_io(any)# ;
        module(any)# ;
        atom(any)# ;
        user(any)#
}.

status_any0 = {abnormal#; aborted#; normal(any)#}.

initrsc_or_list = vector([integer,integer]) &
                  list(task_init_command).

task_init_command =
     { resource(integer,integer);
       stdio(list(stdio_param),-status(integer,list(stdio_result)));
       io_node(ground);
       remote(string,string);
       uid(integer);
       user(string,string);
       name(string);
       node(integer,integer);
       prio(integer,integer);
       tag(integer)}.

stdio_param = { in(stdio_param_body)#;
                out(stdio_param_body)#;
                inter(stdio_param_body)#;
                in_out(stdio_param_body)#;
                in_inter(stdio_param_body)#;
                out_inter(stdio_param_body)#;
                in_out_inter(stdio_param_body)# }.

stdio_param_body = vector([atom,ground,-ground]).

stdio_result = { in(status)#;
                 out(status)#;
```

```
                inter(status)#;
                in_out(status)#;
                in_inter(status)#;
                out_inter(status)#;
                in_out_inter(status)# }.

register_att = { []#;
                 attention(register_att)#}.

register_command =
     { get_request(code,ground,-request_out)# }.

request_out =
     {general_request(-stream(server_control)##server_control,
                      any, ground, ground,-ground,any);
      standard_io(stream(std_control)##std_control,atom,ground);
      abnormal;
      aborted}.

register_status = {abnormal;
                   aborted;
                   normal(-stream(register_command),-any,register_att)}.

server_control = {add_resource(atom,ground,-server_control_status)#;
                  change_comment(ground,-status);
                  absolute_id(-status(list(integer)))}.

server_control_status =
     {abnormal; aborted; normal(-stream(server_control),any)#}.

std_control =
    {open(vector([atom,ground,-ground]),
          {switch(-stream(server_control),any)#},
          {io(-stream(server_control),any)})#;
     request(any,integer,any,-ground)#;
     make_branch(stream(std_subcontrol),any,ground,ground)#}.

std_subcontrol =
    {open(vector([atom,-ground]),
          {switch(-stream(server_control),any)#},
          {io(-stream(server_control),any)})#;
     request(any,integer,any,-ground)#;
     make_branch(stream(std_subcontrol),any,ground,ground)#}.


ask_task_command = {
        id(-status(integer))# ;
        absolute_id(-status(rscpath))# ;
        status(rscpath# , rsckind# , verbosity# , -status(rscinfo))#
```

```
}.

task = {
        normal( -control##control_stream, -report##report_stream ) ;
        abnormal ;
        aborted ;
        disconnected
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the abort line and the attention line
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

aa_command = {
        reset(-status(-abort, attention# , msgid# ))# ;
        next_attention( -status(attention))# ;
        resend( msgid# , -status)# ;
        cancel( msgid# , -status)#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Control stream messages
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

control = stream(control_command).

control_command = ask_task_command & aa_command & {
        stop ;
        allow_resource_report ;
        statistics ;
        profile ;
        abort(rscpath# , -status)# ;
        abort(-status)# ;
        add_resource(hrsc# ,lrsc# )#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Report stream messages
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

report = -stream(report_command).
```

```
report_command = {
        statistics_started ;
        resource_report_allowed ;
        resource_added ;
        resource_low ;
        aborted ;
        terminated ;
        exception(exckind, excinfo, -ncode# , -nargv# )# ;
        statistics(hstat,lstat) ;
        return_resource(hrsc,lrsc) ;
        profiling_information(profile_node, profile_type,
                              profile_depth, profile_data) ;
        profiling_started ;
        profiling_rejected ;
        profiling_completed ;
        stopped( -reason_stream# , -cancel# )#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the window
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

winreq_filter = -rscstream(winreq_command).

winreq_command = aa_command & {
        create( -rscstream(window_command)##window)# ;
        create( buffername# , -rscstream(window_command)##window)# ;
        get_max_size( x# , y# , pathname# , -status(charnum, linenum))#
}.

window_command = input_command & output_command & aa_command & {
        beep( -status)# ;

        set_size( winsize# , -status)# ;
        set_position( winpos# , -status)# ;
        set_title( title# , -status)# ;
        reshape( winpos# , winsize# , -status)# ;
        set_font( pathname# , -status)# ;
        select_buffer( buffername# , -status)# ;

        activate( -status)# ;
        deactivate( -status)# ;
        show( -status)# ;
        hide( -status)# ;
        clear( -status)# ;
        get_char_size( -status(charnum, linenum))# ;
```

```
        get_position( -status(x,y))# ;
        get_title( -status(title))# ;
        get_font( -status(pathname))# ;
        get_buffer_name( -status(buffername))#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the file
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

filereq_filter = -rscstream(filereq_command).


filereq_command = aa_command & {
        open( pathname# ,
        {read(-rscstream(rfile_command)##read_file)# ;
                write(-rscstream(wfile_command)##write_file)# ;
                append(-rscstream(wfile_command)##append_file)# }#
         file_mode )# ;
        directory( pathname# , -rscstream(dir_command)##directory)# ;
        current_directory(-status(string))# ;
        set_current_directory(string#pathname, -status)#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Shared message sections for files
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cfile_command = {
        pathname(-status(pathname))# ;
        attributes(-file_attributes)#
}.

file_attributes = {
        normal(owner,updateuser,cdate,ctime,mdate,mtime,
               linkcnt,fsize,rsize) ;
        abnormal ;
        aborted ;
        disconnected
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the input file
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rfile_command = cfile_command & input_command & aa_command & {
        end_of_file( -status(eof))# ;       % for vpimos
        flush( -status)#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the output file
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

wfile_command = cfile_command & aa_command &
        output_command.

dir_command = aa_command & {
        pathname(-status(pathname))# ;
        listing( wildcard# , -status(list(filename)))# ;
        delete( wildcard# , -status)# ;
        undelete( wildcard# , -status)# ;
        deleted( wildcard# , -status(list(filename)))# ;
        purge( wildcard# , -status)# ;
        expunge( -status)#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the timer
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


timerreq_filter = -rscstream(timerreq_command).
timerreq_command = aa_command & {
        create( timer_id#, -rscstream(timer_command)##timer)# ;
        create(-rscstream(timer_command)##timer)#
}.

timer_id = integer & {fep}.

timer_command = aa_command & {
        get_count( -status(count))# ;
        on_at( count# , -status(now))# ;
        on_after( count# , -status(now))# ;
        get_date_time( -status(date_count, time_count))# ;
```

```
        where( -status(timer_id))# ;
        set_time_unit( time_unit#, -status)# ;
        get_time_unit( -status(time_unit)#)
}.

time_unit = { sec ; msec ; msec16 ; usec }.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the string_io
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


string_io_req_filter = -rscstream(strioreq_command).
stroreq_command = aa_command & {
        create( devicename# , -rscstream(strio_command)##string_io)#
}.


strio_command = aa_command & {
        io( io_command# , tofepstring# , -status(fromfepstring))#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the module
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


modulereq_filter = -rscstream(modulereq_command).
modulereq_command = aa_command & {
        create( -rscstream(module_command)##module_device)#
}.


module_command = aa_command & {
        define_package(packagename# , packagename# ,
                        protection# , -retrieval_status)# ;
        set_package_owner(packagename# , owner_info# , -retrieval_status)# ;
        protect_package(packagename# , protection# , -retrieval_status)# ;
        get_package_info(packagename# , -retrieval_status(protectionInfo))# ;
        get_super_package(packagename# , -retrieval_status(packagename))# ;
        get_all_packages(-packageInfoList)# ;
        remove_package(packagename# , -retrieval_status_1)# ;
        get_module_info(packagename# , modulename# , -module_status_1)# ;
        get_module(packagename# , modulename# , -module_status_2)# ;
        get_module_in_the_package(packagename#,
```

```
                                    modulename#, -module_status_2)# ;
        get_all_modules(packagename# , -module_status_3)# ;
        get_all_modules(-module_status_4)# ;
        get_code(packagename# , modulename# , predname# , arity# ,
                 -module_status_5 )# ;
        update_module(packagename# , modulename# , module#update_module,
                      -retrieval_status_1 )# ;
        remove_module(packagename# , modulename# , -retrieval_status_1)
}.

packageInfoList = {
        normal( list( packageInfo ) ) ;
        protection_violation ;
        abnormal }.

owner_info = vector([uid,gid]).

packageInfo = vector( [ packagename, packageInfo1, packageInfo2 ] ).
packageInfo1 = retrieval_status( packagename ).
packageInfo2 = retrieval_status( ground ).
packageInfo3 = vector( [ packagename, module_status_3 ] ).

module_status_1 = {
        normal( packagename, module, moduleSize ) ;
        non_existent ;
        protection_violation ;
        abnormal }.

module_status_2 = {
        normal( module ) ;
        non_existent ;
        protection_violation ;
        abnormal }.

module_status_3 = {
        normal( moduleInfoList ) ;
        non_existent ;
        protection_violation ;

        abnormal }.

module_status_4 = {
        normal( list( packageInfo3 ) ) ;
        protection_violation ;
        abnormal }.

module_status_5 = {
        normal( code ) ;
        non_existent ;
```

```
        non_existent( ground ) ;
        protection_violation ;
        abnormal }.

retrieval_status = {
        normal ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent }.

retrieval_status(x) = {
        normal(x) ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent }.

retrieval_status_1 = {
        normal ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent( ground ) }.

protectionInfo = vector( [ ground, protection ]) .
moduleInfoList = list( moduleInfo ).
moduleInfo = vector( [ modulename, module, size ] ).
moduleSize = integer.
userid = ground.
groupid = ground.
size = integer.
protection = integer.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the atom
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


atomreq_filter = -rscstream(atomreq_command).
atomreq_command = aa_command & {
        create( -rscstream(atom_command)##atom_device)#
}.


atom_command = aa_command & {
```

```
        get_atom(atomname# , -retrieval_status(atom,dkey))# ;
        get_atom_name(atom# , -retrieval_status(atomname,dkey))# ;
        intern(atomname# , -status(atom))# ;
        get_all_atoms(-status(atominfo))#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Messages relevant to the user device
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

userreq_filter = -rscstream(userreq_command).
userreq_command = aa_command & {
        create({user(-rscstream(user_command)##user_device);
                group(-rscstream(group_command)##group_device);
                login(-rscstream(login_command)##login_device)}#user_mode)# }.


user_command = aa_command & {
        set_up(list(user_all_info),-status)#;
        authenticate(user_name#,password#,-status)#;
        get_uid(user_name#,-status(uid))#;
        get_info(uid#,-status(user_info))#;
        status(uid#,-status(change_count))#;
        get_all_info(-status(list(user_all_info)))#;
        change_password(uid#,old_password#,new_password#,-status)#
}.

old_password = password.
new_password = password.

user_name = string.
password = string.
uid = integer.
priority = integer.
directory_name = string & {*}.
entry_point = {':'({'::'(packagename,modulename)},predicate)}.

predicate = atom & vector(ground).
comment_list = list(string).
pe_number = integer & {*}.

change_count = integer.

user_all_info = vector([user_name,password,uid,priority,
                    directory_name,pe_number,entry_point,comment_list]).
user_info = vector([user_name,priority,directory_name,
                    pe_number,entry_point,comment_list]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the group device
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

group_command = aa_command & {
        set_up(list(group_info),-status)#;
        get_gid(group_name#,-status(gid))#;
        get_gids(uid#,-status(list(gid)))#;
        get_users(gid#,-status(group_members))#;
        get_all_info(-status(list(group_info)))#
}.

gid = integer.
group_name = string.
group_members = list(uid).
group_info = vector([group_name,gid,group_members,comment_list]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Messages relevant to the login device
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

login_command = aa_command & {
        login(user_name#,login_info#,-status(lid))#;
        logout(lid#,-status)#;
       get_info(lid#,-status(login_info))#;
        get_login(user_name#,-status(list(lid)))#;
        get_all_info(-status(list(vector([user_name,lid,login_info]))))#
}.

lid = integer.
login_info = ground.
node_name = fepnodeno.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Shared messages relevant to the device
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

input_command = {
        getb( blocksize# , -eofstatus(string))# ;
        ungetb( string#ungetb , -status)# ;
```

```
        prompt( string#prompt, -status)#
}.

output_command = {
        putb( string#put_data, -status)# ;
        flush( -status)#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Shared sections that check message elements
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscpath = {[rscid | rscpath];
            *               ;
            []
         }.
rscid = integer & { * }.
rsckind = rscname & {
        * }.
rscname = atom.
verbosity = {
        brief ;
        detail
}.
reason_stream = stream(atom).
rscinfo = list(rscinfo2).
rscinfo2 = brief_info & other_brief_info & detail_info & other_detail_info.
brief_info = vector([rscid, rscname, ground, bchildren]).
other_brief_info = vector([rscid,bchildren]).
both_brief_info = brief_info & other_brief_info.
detail_info = vector([rscid, ground, task_status, limit, consume, dchildren]).
other_detail_info = vector([rscid,dchildren]).
both_detail_info = detail_info & other_detail_info.
task_status = vector([atom, stops]).
stops = list(stopreason).
stopreason = {stop(atom)}.
binfo2 = brief_info & other_brief_info.
dinfo2 = detail_info & other_detail_info.
bchildren = list(binfo2).
dchildren = list(dinfo2).
msgid = integer.
winpos = {
        mouse ;
        at(x,y)
}.
winsize = {
```

```
        mouse ;
        char(charnum, linenum)
}.
now = {wake_up; aborted}.
attention = {attention ; []}.

atominfo = list(atominfo2).
atominfo2 = vector([atom,string]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Basic protocols
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


excargv = any.
cancel = ground.
hstat = integer.
lstat = integer.
nargv = any.
ncode = any.
exccode = any.
excinfo = vector(any).
exckind = atom.
term1 = any.
term2 = any.
rtype = any.
info = any.
execode = code.
argv = vector(any).
taskname = string.
minprio = integer.
maxprio = integer.
tag = integer.
hrsc = integer.
lrsc = integer.

envname = ground.
value = ground.

abort = any.

buffername = string.
x = integer.
y = integer.
pathname = string.
devicename = string.
```

```
fepno = integer.
blocksize = integer.
charnum = integer.
linenum = integer.
title = string.
filename = string.
wildcard = string.
owner = string.
updateuser = string.
cdate = string.
ctime = string.
mdate = string.
mtime = string.
linkcnt = integer.
fsize = integer.
rsize = integer.
count = integer.
date_count = integer.
time_count = integer.
io_command = integer.
tofepstring = string.
fromfepstring = string.
modulename = atom.
packagename = atom.
predname = atom.
arity = integer.
dkey = atom & string.
atomname = string.

limit = vector([hrsc,lrsc]).
consume = vector([hrsc,lrsc]) & {unknown}.
initrsc = vector([hrsc,lrsc]).
name = string.
eof = {yes ;
       no
}.

envs = list(env).

env = vector([envname,value]).

user = integer.
ruser = string.
rpassword = string.
fepnodeno = integer & string.
nodeno = string.

profile_node = vector(integer).
profile_type = integer.
```

```
profile_depth = integer.
profile_data = ground.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Parameterized protocols
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscstream(p) = {
        normal(-stream(p), -abort, attention) ;
        abnormal ;
        aborted ;
        disconnected
}.

status = {
        normal ;
        abnormal ;
        aborted ;
        disconnected
}.

status(p) = {
        normal(p) ;
        abnormal ;
        aborted ;
        disconnected
}.

status(x,y) = {
        normal(x,y) ;
        abnormal ;
        aborted ;
        disconnected
}.
status(x,y,z) = {
        normal(x,y,z) ;
        abnormal ;
        aborted ;
        disconnected
}.

retrieval_status(x,y) = {
        normal(x) ;
        protection_violation ;
        abnormal ;
        aborted ;
```

```
        non_existent(y)
}.
eofstatus(p) = {
        normal(p) ;
        abnormal ;
        aborted ;
        end_of_file ;
        disconnected
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Standard I/O
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

stdwin_filter = stream(window_command).
stdrfile_filter = stream(rfile_command).
stdwfile_filter = stream(wfile_command).
stdafile_filter = stream(wfile_command).

stdstring_io_filter = stream(strio_command).

std_in_filter = stream( std_in_command ).
std_out_filter = stream( std_out_command ).
std_inter_filter = stream( std_inter_command ).
std_io_filter = stream( std_io_command ).
std_switch_filter = stream( std_switch_command ).

std_in_command = aa_command & input_command .
std_out_command = aa_command & output_command .
std_inter_command = aa_command & interaction_command
    & stdio_special_command .
std_io_command = aa_command & input_command & output_command
    & interaction_command & stdio_special_command .
std_switch_command = {
    attach( switch_no#, -attach_status#, -detect# )# ;
    attach( -attach_status#, -detect# )# ;
    detach( switch_no#, -detach_status# )# ;
    detach( -detach_status# )# ;
    kill( switch_no_with_all#, -kill_status# )# ;
    kill( -kill_status# )#
}.

switch_no = integer.
switch_no_with_all = integer & { all }.
attach_status = atom.
detect = atom.
```

```
detach_status = atom.
kill_status = atom.

interaction_command = {
    igetb( blocksize#, -eofstatus( string ) )# ;
    iungetb( string#ungetb , -status)# ;
    iprompt( string#prompt, -status )# ;
    iputb( string#put_data, -status )# ;
    iflush( -status )#
}.

stdio_special_command = {
    switch( -status )# ;
    am_I_there( -status)#
}.
```

付録 F

SERVER PROGRAM SOURCE CODE EXAMPLE

F.1 Overview

This section presents an example of the source code for a server program.

The server presented here is a server that provides a data pool service called a "whiteboard" that is available to users.

Whiteboards provide the following functions.

- Whiteboards exist in user and group units.
- The data that can be registered consists of character strings keyed by atoms.
- While a table of the keys for each whiteboard can be viewed, whiteboards basically only support data insertion and retrieval.

The following is an example of the use of a whiteboard.

```
shoen:raise(pimos_tag#task, general_request, GR),
GR = [request(whiteboard, Cancel, WRResult)],

%   The server name is 'whiteboard'.
%

Result = normal(WSStream),

%   Although aborted and abnormal could also be returned for Result,
%   this example omits those cases.
%   Thus a request device was acquired in WSStream.

WSStream = [user(UResult)],

%   Here, a whiteboard in user units is used.

UResult = normal(UStream),

%   Acquisition success.

UStream = [put(example, "Write a string here", status)],
```

```
%   One datum registered.

%   Here, we can expect that Status == normal.
```

F.2 Server Program Example

```
:- module whiteboard_server.
:- with_macro pimos.
:- public go/0, filter/3.

%   The calling port is whiteboard_server:go.
%   the predicate filter/3 is declared public for use
%   as the protection filter code

go:-
    true|
        shoen:raise(pimos_tag#task,general_request,GR),
%
%       Server name:     whiteboard
%       Owner:           The super user
%       Group owner:     The privileged group
%       Protection mode: Available for any user

%             Register the server.
%
        GR = [register(whiteboard,{0,0,2#"111"},Res)],
        wait_register(Res).

wait_register(normal(Control,Abort,Attention)):-
    true|
%
%    Since registration succeeded:
%        Control:   A control stream
%        Abort:     A variable to be used for informing the server that
%                   the registration should be deleted immediately.
%        Attention: A variable for announcing abnormalities. Not used here.
%%%   ignore reset Attention
%
%       A pool for holding whiteboard protocol groups.
%
        pool:keyed_set(Pool),
        merge(PoolIn,Pool),
%
%       Retrieve the protection filter code.
%
        predicate_to_code(module#whiteboard_server,filter,3,Code),
```

```
%
%       Start reception of user processing.
%       Register the task as "whiteboard" in the resource tree
%       during user use.
%
        Control = [get_request(Code,"whiteboard",Res)|Next],
        wait_get_request(Res,Next,Code,Abort,PoolIn).
wait_register(abnormal):- true | true.
wait_register(aborted):- true | true.

%
%       A single processing request arrived from a user.
%

wait_get_request(general_request(Control,Abort,Attention,
                                                UInfo,FilterInfo,Var),
                                Next,Code,ServerAbort,Pool):-
    true|
%
%       First enable the next processing reception.
%
        Next = [get_request(Code,"whiteboard",Res)|NextN],
        wait_get_request(Res,NextN,Code,ServerAbort,Pool1),
%
%       Acquire from the user data a sequence consisting of the
%       user id and the id of the associated group, and the node
% number where the user executed the processing request
% (an absolute node number valid for all shoens).
%
        set_vector_element(UInfo,0,Uid,0,UInfo1),
        set_vector_element(UInfo1,1,Gids,0,_),
%
%       Pass the group id of the group to which the user belongs
%       to the protection filter for checking the group ids that
%       each user can use from within the user task.
%
        FilterInfo = Gids,
%
        Pool = {Pool1,Pool2},
        merge(ControlIn,Control),
%
%       Invoke a processing routine as a request device.

        rhandler(Var,Pool2,Attention,
                    ControlIn,Abort,ServerAbort,Uid).
%
%       The whiteboard process is terminated if the server
%       itself interrupts the processing.
%
```

```
wait_get_request(aborted,Next,_,_,Pool):-
    true|
        Next = [],
        abort_pool(Pool).
wait_get_request(abnormal,Next,_,_,Pool):-
    true|
        Next = [],
        abort_pool(Pool).

%
%       Request device processing
%
rhandler(User,Pool,Attention,Control,Abort,ServerAbort,_):-
%
%       If a report of server processing interruption arrives:
%
    wait(ServerAbort)|
%
%       Return aborted in response to user service post-processing
%       that has already been performed, i.e., all requests.
%       This is a service provided so that the protection filter
%       operating in the user process will not be deadlocked.
        abort_rhandler(User,Attention),
        Pool = [],
        Abort = abort,
        Control = [].
%
rhandler(_,Pool,Attention,Control,Abort,_,_):-
%
%      For the case where the client task terminated.
&
    wait(Attention)|
        Pool = [],
        Abort = abort,
        Control = [].
alternatively.
%
%      Acquire a whiteboard in user units.
%
rhandler([user(Result)|Next],Pool,Attention,
                    Control,Abort,ServerAbort,Uid):-
    true|
        Pool = [put(uid(Uid),NewUID,UIDV)|Pool1],
%
%       Register the use of the user unit whiteboard as a user_board
%       name and Uid in the user task resource tree.
%
        Control = [add_resource(user_board,Uid,Status)|ControlN],
%
```

```
%       Return acquisition success to the user request.
%
        Result = normal(User),
%
%       Perform processing for the post-acquisition/generation
%       whiteboard pool corresponding to Uid.
%
        exist(UIDV,NewUID,Status,User,Attention,ServerAbort),
%
%       Perform the next processing.
%
        rhandler(Next,Pool1,Attention,ControlN,Abort,ServerAbort,Uid).
rhandler([group(Gid,Result)|Next],Pool,Attention,
                    Control,Abort,ServerAbort,Uid):-
%
%       The processing for group unit whiteboards.
%
    true|
        Pool = [put(gid(Gid),NewGID,GIDV)|Pool1],
        Control = [add_resource(group_board,Gid,Status)|ControlN],
        Result = normal(User),
        exist(GIDV,NewGID,Status,User,Attention,ServerAbort),
        rhandler(Next,Pool1,Attention,ControlN,Abort,ServerAbort,Uid).
rhandler([],Pool,_,Control,_,_,_):-
%
%       Terminate processing since the user closed the stream.
%
    true|
        Pool = [],
        Control = [].

exist(IDV,NewID,Status,User,Attention,ServerAbort):-
%
%       Retrieve the registered whiteboard.
%
    vector_element(IDV,0,ID)|
        ID = {IDPool,NewID},
%
%       Wait for the results of registration in the resource tree.
%
        wait_add_resource(Status,User,IDPool,Attention,ServerAbort).
otherwise.
exist(_,NewID,Status,User,Attention,ServerAbort):-
%
%       Since the whiteboard did not exist, create one.
%
    true|
        pool:keyed_bag(BAG),
        merge(BAGIN,BAG),
```

```
        BAGIN = {IDPool,NewID},
        wait_add_resource(Status,User,IDPool,Attention,ServerAbort).

wait_add_resource(normal(Control,Abort),User,
                    UID,Attention,ServerAbort):-
%
%       A control stream was acquired since the registration succeeded.
%
    true|
        handler(User,UID,Attention,Abort,ServerAbort,Control).
otherwise.
wait_add_resource(_,User,UID,Attention,_):-
%
%       Since the failure of registration in the resource tree was
%       due to the user task being about to terminate, user service
%       is terminated here also.
%
    true|
        UID = [],
        abort_handler(User,Attention).

%
% handler(User,UID,Attention,Abort,ServerAbort,Control)
%
%       The main section for user services.
%
handler(_,UID,Attention,_,_,Control):-
%
%       The server interrupted processing for this user.
%
    wait(Attention)|
        UID = [],
        Control = [].
handler(User,UID,Attention,Abort,_,Control):-
    wait(Abort)|
%
%       The user task is terminated.
%
        abort_handler(User,Attention),
        UID = [],
        Control = [].
handler(User,UID,Attention,_,ServerAbort,Control):-
    wait(ServerAbort)|
%
%       The server itself interrupted all processing.
%
        abort_handler(User,Attention),
        UID = [],
        Control = [].
```

```
alternatively.
handler([],UID,_,_,_,Control):-
%
%       Terminate processing since the user closed the stream.
%
    true|
        UID = [],
        Control = [].
handler([get(Key,Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       Retrieve the character string corresponding to the key.
%
    true|
        UID = [get_if_any(Key,X)|UIDN],
        wait_get_if_any(X,Result),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([put(Key,X,Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       Register data.
%
    true|
        UID = [put(Key,X)|UIDN],
        Result = normal,
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([get_keys(Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
    true|
%
%       Return a table of keys.
%
        UID = [do([get_all(ALL)|PUTALL])|UIDN],
        Result = normal(Keys),
        get_keys(ALL,Keys,PUTALL),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([get_all(Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       Retrieve all the data.
%
    true|
        UID = [get_all(ALL)|UIDN],
        Result = normal(ALL),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).

wait_get_if_any(V,Result):-
    vector_element(V,0,X)|
        Result = normal(X).
```

```
otherwise.
wait_get_if_any(_,Result):-
    true|
        Result = abnormal.

get_keys([],Keys,PUTALL):-
    true|
        Keys = [],
        PUTALL = [].
get_keys([{Key,Value}|Next],Keys,PUTALL):-
    true|
        Keys = [Key|KNext],
        PUTALL = [put(Key,Value)|PUTALLN],
        get_keys(Next,KNext,PUTALLN).

abort_pool(Pool):-
    true|
        Pool = [get_all(ALL)],
        abort_all_pool(ALL).

abort_all_pool([]):-
    true|
        true.
abort_all_pool([{_,IDP}|Next]):-
    true|
        IDP = [],
        abort_all_pool(Next).

abort_rhandler(_,Attention):-
    wait(Attention)|
        true.
otherwise.
abort_rhandler([],_):-
    true|
        true.
abort_rhandler([user(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_rhandler(Next,Attention).
abort_rhandler([group(_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_rhandler(Next,Attention).

abort_handler(_,Attention):-
    wait(Attention)|
        true.
otherwise.
abort_handler([],_):-
```

```
    true|
        true.
abort_handler([get(_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
abort_handler([put(_,_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
abort_handler([get_keys(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
abort_handler([get_all(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).

%
%  The following is the code for the protection filter.
%
%  The protection filter is coded with the following principles:
%       It does not send anything to the server until all user input is
%  assembled, and,
%       Anything that can be returned as result output to the user is
%  returned completely.
%
%   filter(Gids,User,Server)
%      Gids:   A sequence of user Gids, sent by the server described above.
%      User:   A user input variable.
%      Server: An output variable to the user.

filter(_,[],Server):-
    true|
        Server = [].
filter(Gids,[user(Result)|User],Server):-
    true|
        Server = [user(SResult)|ServerN],
        wait_cfilter(SResult,Result),
        filter(Gids,User,ServerN).
filter(Gids,[group(Gid,Result)|User],Server):-
    true|
        member(Gid,Gids,GidsN,Yes),
        wait_gid(Yes,Gid,Server,ServerN,Result),
        filter(GidsN,User,ServerN).

wait_gid(yes,Gid,Server,ServerN,Result):-
    true|
```

```
        Server = [group(Gid,SResult)|ServerN],
        wait_cfilter(SResult,Result).
otherwise.
wait_gid(_,_,Server,ServerN,Result):-
    true|
        Server = ServerN,
        Result = abnormal.

member(_,[],GidsN,Yes):-
    true|
        GidsN = [],
        Yes = no.
member(Gid,[Gid|Next],GidsN,Yes):-
    true|
        GidsN = [Gid|Next],
        Yes = yes.
otherwise.
member(Gid,[Gid2|Next],GidsN,Yes):-
    true|
        GidsN = [Gid2|GidsNN],
        member(Gid,Next,GidsNN,Yes).

wait_cfilter(normal(Server),Result):-
    true|
        Result = normal(User),
        cfilter(User,Server).
otherwise.
wait_cfilter(SResult,Result):-
    true|
        Result = SResult.

cfilter([],Server):-
    true|
        Server = [].
cfilter([get(Key,Result)|User],Server):-
    atom(Key)|
        Server = [get(Key,SResult)|ServerN],
        wait_get_cfilter(SResult,Result),
        cfilter(User,ServerN).
cfilter([put(Key,Value,Result)|User],Server):-
    atom(Key),
    string(Value,_,_)|
        Server = [put(Key,Value,SResult)|ServerN],
        wait_put_cfilter(SResult,Result),
        cfilter(User,ServerN).
cfilter([get_keys(Result)|User],Server):-
    true|
        Server = [get_keys(SResult)|ServerN],
        wait_get_keys(SResult,Result),
```

```
        cfilter(User,ServerN).
cfilter([get_all(Result)|User],Server):-
    true|
        Server = [get_all(SResult)|ServerN],
        wait_get_all(SResult,Result),
        cfilter(User,ServerN).

wait_get_cfilter(normal(Value),Result):-
    true|
        Result = normal(UValue),
        wait_value(Value,UValue).
otherwise.
wait_get_cfilter(SResult,Result):-
    true|
        Result = SResult.

wait_put_cfilter(SResult,Result):-
    wait(SResult)|
        Result = SResult.

wait_get_keys(normal(Keys),Result):-
    true|
        Result = normal(UKeys),
        wait_get_keys2(Keys,UKeys).
otherwise.
wait_get_keys(SResult,Result):-
    true|
        Result = SResult.

wait_get_keys2([],UALL):-
    true|
        UALL = [].
wait_get_keys2([E|ALL],UALL):-
    true|
        UALL = [UE|UALLN],
        wait_value(E,UE),
        wait_get_keys2(ALL,UALLN).

wait_get_all(normal(ALL),Result):-
    true|
        Result = normal(UALL),
        wait_get_all2(ALL,UALL).
otherwise.
wait_get_all(SResult,Result):-
    true|
        Result = SResult.

wait_get_all2([],UALL):-
    true|
```

```
        UALL = [].
wait_get_all2([E|ALL],UALL):-
    true|
        UALL = [UE|UALLN],
        wait_get_all3(E,UE),
        wait_get_all2(ALL,UALLN).

wait_get_all3(E,UE):-
    vector_element(E,0,Key),
    vector_element(E,1,Value)|
        new_vector(UE0,2),
        wait_value(Key,UKey),
        wait_value(Value,UValue),
        set_vector_element(UE0,0,_,UKey,UE1),
        set_vector_element(UE1,1,_,UValue,UE).

wait_value(Value,UValue):-
    wait(Value)|
        UValue = Value.
```