PIMOS Manual
(V 1.5)

ICOT PIMOS Developping Group

December 1989

© *Copyright* 1989 *Institute for New Generation Computer Technology, Japan*
All RIGHTS RESERVED

Contents

1 INTRODUCTION **9**

2 KL1 LANGUAGE SPECIFICATION **10**

2.1 Syntax . . . 10
2.1.1 Modular structure . . . 10
2.1.2 Treatment of asymmetric clauses . . . 11
2.1.3 Data notation . . . 12
2.1.4 Execution priority specification . . . 13
2.1.5 Processor specification . . . 14
2.1.6 Macro notation . . . 14
2.2 Built-in Predicate . . . 23
2.2.1 Type check . . . 24
2.2.2 Comparison of integer . . . 25
2.2.3 Comparison of floating point number . . . 26
2.2.4 diff . . . 26
2.2.5 Integer operation . . . 27
2.2.6 Floating point arithmetic . . . 28
2.2.7 Conversion of integer-floating point number . . . 29
2.2.8 Vector . . . 30
2.2.9 String . . . 30
2.2.10 Atom . . . 32
2.2.11 Code and module . . . 32
2.2.12 Stream support . . . 33
2.2.13 Special input/output . . . 33
2.2.14 Others . . . 34
2.3 Shoen Function . . . 35
2.3.1 Outline of Shoen function . . . 35
2.3.2 Creation of shoen . . . 36
2.3.3 Control stream . . . 36
2.3.4 Report stream . . . 36
2.3.5 Occurrence of exception . . . 39
2.3.6 Exception management utility . . . 40
2.3.7 Exception tag . . . 40

3 UTILITY **42**

3.1 Comparison . . . 42
3.2 Hashing . . . 43
3.3 Pool . . . 43
3.3.1 Pool without key . . . 43
3.3.2 Pool with keys . . . 45
3.4 Sorting/Merging Filter . . . 49
3.4.1 Sorting filter . . . 49
3.4.2 Merging filter . . . 50
3.5 Buffer . . . 50
3.5.1 Common Protocol . . . 51

3.5.2 Input filter . . . 56
3.5.3 Output filter . . . 56
3.5.4 Device protocol . . . 56
3.5.5 Input/Output filter . . . 56
3.5.6 Infinite length buffer . . . 57
3.5.7 Operator pool . . . 57
3.5.8 Variable pool . . . 57
3.6 Normal Term and Wrapped Term . . . 59
3.6.1 Unwrap . . . 60

4 Acquiring Resources **61**
4.1 Resource Type . . . 61
4.2 Task . . . 62
4.3 Resource Tree . . . 62
4.4 Environment Variable . . . 62
4.5 Hierarchy of Resource Acquisition . . . 63
4.6 General Request Device . . . 63
4.6.1 How to Get Request Device . . . 63
4.6.2 Protocol . . . 63

5 I/O Control **68**
5.1 Communication Method of I/O Control . . . 68
5.1.1 I/O Messages . . . 68
5.2 Device Abort/Attention Processing . . . 68
5.2.1 Window Device Abort Processing . . . 70
5.3 Path Name . . . 70
5.4 Request Devices . . . 71
5.4.1 Window Request Device . . . 71
5.4.2 File Request Device . . . 71
5.4.3 Timer Request Device . . . 72
5.4.4 String I/O Request Device . . . 73
5.5 Devices . . . 73
5.5.1 Window . . . 73
5.5.2 File . . . 75
5.5.3 Timer . . . 77
5.5.4 String I/O . . . 77

6 Controlling Tasks **79**
6.1 Task Control Stream . . . 79
6.2 Task Report Stream . . . 80

7 PROGRAM INFORMATION MANAGEMENT **81**
7.1 Program Information Databases . . . 81
7.1.1 Atom Cache Database . . . 81
7.1.2 Module Cache Database . . . 82
7.2 Self-compiler . . . 83
7.2.1 Compiler Directive Predicate . . . 83
7.2.2 Starting the compiler . . . 84
7.2.3 Interactive compile . . . 84
7.2.4 Batch compile . . . 85
7.3 Relinker . . . 86
7.3.1 Using the relinker . . . 86
7.4 Unloader . . . 86
7.4.1 Using the unloader . . . 87
7.5 Loader . . . 87

8 shell **88**
8.1 Outline . . . 88
8.2 Use . . . 89
8.2.1 Start . . . 89
8.2.2 Shell window commands . . . 90
8.2.3 Commands to the shell or jobs . . . 91
8.2.4 Messages displayed by the shell . . . 94
8.2.5 Error handling . . . 96
8.2.6 Error handling in resource shortage . . . 97
8.2.7 Specifying the processor executing the job . . . 97
8.2.8 Specifying the range of job execution priority . . . 98
8.2.9 Command files . . . 98
8.2.10 Manipulating the environment variables . . . 98
8.2.11 Initialization file . . . 99
8.3 Creating Shell Utility Programs . . . 99
8.3.1 Acquiring the standard I/O stream . . . 99
8.3.2 Standard I/O stream message protocols . . . 100
8.3.3 Shell utility program samples . . . 101
8.3.4 Tee program . . . 101

9 SHELL UTILITY PROGRAM **103**
9.1 GC Start . . . 103
9.2 Status Display . . . 104
9.2.1 Displaying the resource tree . . . 104
9.2.2 Displaying the task tree . . . 104
9.3 Terminates PIMOS . . . 104
9.4 Copy . . . 105
9.5 Module-related commands . . . 105
9.5.1 Displaying usage of the modules . . . 105
9.5.2 Library function . . . 105
9.6 Commands to Character Strings and Text Files . . . 106
9.6.1 Echo . . . 106
9.6.2 Retrieving a character string . . . 106
9.6.3 Counting the number of lines . . . 106
9.6.4 Displaying the directory . . . 106

10 Listener **108**
10.1 Function Summary . . . 108
10.2 Operating Method . . . 109
10.2.1 Starting . . . 109
10.2.2 Input of Goal Sequence . . . 109
10.2.3 Variable Pool . . . 111
10.2.4 Interruption of Execution . . . 111
10.2.5 Termination . . . 113
10.3 Standard Input/Output . . . 113
10.4 Deadlock Detection . . . 114
10.4.1 Summary . . . 114
10.4.2 Starting of Deadlock Detection . . . 115
10.4.3 Types of Deadlocked Goals . . . 115
10.4.4 Deadlock Processing . . . 116
10.5 Commands . . . 117
10.5.1 Basic Commands . . . 117
10.5.2 Environment Commands . . . 118
10.5.3 Variable Pool Operational Commands . . . 121
10.5.4 Debug Commands . . . 121
10.6 Tracer . . . 125
10.6.1 Summary . . . 125

10.6.2 Trace Output . . . 127
10.6.3 Trace Commands . . . 130
10.7 Error Handling . . . 140

11 Inspector **146**
11.1 Outline of Functions . . . 146
11.2 Data Display Format and Analysis/Display Position Specification Procedure . . . 146
11.2.1 Data display format . . . 146
11.2.2 Analysis/display position specification procedure . . . 147
11.3 Operating Procedure . . . 148
11.3.1 Invokation . . . 148
11.3.2 Command input . . . 148
11.3.3 Variable pool . . . 148
11.3.4 Setting the environment for output to the window . . . 149
11.3.5 Switching of the input/output destination . . . 149
11.3.6 Output interruption . . . 149
11.3.7 Termination . . . 150
11.4 Command . . . 150
11.4.1 Basic command . . . 150
11.4.2 Inspection command . . . 151
11.4.3 Environment command . . . 154
11.4.4 Variable pool manipulation command . . . 156

12 Administration **157**
12.1 Booting and Shutdown . . . 157
12.1.1 Booting procedure . . . 157
12.1.2 Shutdown procedure . . . 158
12.2 How to Set Up Booting Procedure . . . 159
12.2.1 How to set the PIMOS environment . . . 159
12.2.2 Contents of the environment set-up file for release . . . 160
12.3 User Environment Set-up Program . . . 160
12.3.1 Setting up the user environment . . . 160
12.3.2 User environment file . . . 160
12.3.3 Contents of the user environment file for release . . . 161

A How to create a string I/O device on FEP **162**

B Performance Meter **165**
B.1 Start up . . . 165
B.2 Setting up . . . 165
B.3 Display Start . . . 165

C PROTOCOL DEFINITIONS **166**
C.1 Outline . . . 166
C.2 Protocol Definition Language . . . 166
C.2.1 Grammar . . . 166
C.2.2 System Defined Protocol . . . 166
C.2.3 Description of grammar . . . 167
C.3 How to read the protocol definition . . . 167
C.4 Definition of Protocal . . . 169

Contents

List of Figures

2.1 Bit position of exception tag 40

4.1 Resource Hierarchy 63

List of Tables

2.1 KL1 data notation 13
2.2 Arithmetic Comparison Macros 16
2.3 Arithmetic operation macro 17
2.4 Exception tag for language definition exception 41

3.1 Standard Operators 58

C.1 Message protocals corresponding to target devices 169

Chapter 1

INTRODUCTION

This manual details how to use the Operating System PIMOS, version 1.5, for parallel inference machines. PIMOS is written in the concurrent logic language KL1, and it is assumed that user programs running under PIMOS are also coded in KL1. KL1 is based on Flat GHC and has various extensions. This manual contains the outlined specification of the extensions of KL1 to GHC. For programming techniques in KL1, see the KL1 programming textbook "KL1 Programming Introduction/For beginners/Intermediate" (the fourth laboratory). Version 1.5 described here has been created from Version 1 (July 1989) and has many new functions. The user interface specification remains basically the same.

Chapter 2

KL1 LANGUAGE SPECIFICATION

The computers on which PIMOS runs use the concurrent logic kernel language KL1 as the software and hardware interface. PIMOS itself and application programs are coded in KL1 (directly or through a higher level language). KL1 is based on Flat GHC (simply called FGHC below), a subset of the concurrent logic language GHC, and has a lot of new functions. The main extended functions include extended syntax for ease of coding, built-in data types and built-in predicates function for improving processing efficiency, and the mechanism for meta-level control such as exception handling. This section explains only the differences between KL1 and FGHC as a rule. For details of FGHC, see other documents and reference materials.

2.1 Syntax

KL1 has various extended functions of FGHC, and the syntax has also been extended. This section describes the major extensions. The major syntax extensions include:

- Modular structure
- Asymmetry of order of clauses
- Execution priority specification
- Processor specification
- Macro notation

2.1.1 Modular structure

KL1 programs are coded in **modules**.

Module definition

A module definition begins with:

```
:- module module-name.
```

which declares the name of the module. The predicate that is present in the module and may be called by other modules must be specified in the form:

```
:- public predicate-name/arity, predicate-name/arity, ... .
```

For a predicate to be executed with the built-in predicate "apply" (described later in this manual) or for a predicate to be specified for creation of shoen, "public" needs to be declared even if it is executed only within the same module. Multiple clauses for the same predicate must be contiguously coded. If these clauses are separately coded, individual predicates are treated as different predicate definitions even if they have the same predicate name and the same number of arguments, and will result in a multiple definition error at compile time.

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)
```

2.1.2 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 of clauses. Another example is the simplified coding of vagetion. The condition for the selection of clauses may be completely written in the "guard" part. In case in which the selection of "these and others" is needed, "otherwise" is a simplified notation of the negative condition provided by KL1.

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 available or not is determined only on the executing processor. In other words, if the data required for reduction is contained in another processor and if there is a clause which can be executed without suspension by accessing up to the other processor before "alternatively", sometimes a clause (group) executable with only data in the same processor after "alternatively" is executed. Even in such case, access to the other processor for the data attributable to the suspension starts. Consequently, waiting for the same data (if the data in other processor is ready) makes it possible to execute sooner or later the clause which uses that data.

In general, the priority of predicates specified with "alternatively" is not always observed, so that a correct program including "alternatively" must be the 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 officiency.

[Example]Priority merger

```
        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).
```

Simplified notation of negation (otherwise)

If it is determined that all clauses coded before "otherwise" cannot be selected (the guard part fails) 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 vegation 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).
             ...
```

2.1.3 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.

Variable: Same as Edinburgh Prolog.

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.

Floating point number: Floating point numbers are expressed in the form:

`[sign] number + decimal point number + [e or E [exponent sign] number +]`

where, [ ] contains an option, and number + denotes the digits of one or more characters. This notation cannot contain spaces. Similar to integer, sign is included in floating point numbers. Reference: In multi PSI, mental 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 mental representation. Chances are that unification will fail even if the two numbers look the same when displayed. In general, it is recommended to avoid unification of floating point numbers.

Atom same as Edinburgh Prolog

[Caution]At present, compilers treat "`$`" the same as symbol characters such as "`+`" and "`-`", and "`$:=`" is treated as a normal atom. In the future, "`$`" will be used for the notation of special constants (such as a floating point numbers denoting infinity). Therefore, it is recommended to avoid using "`$`" for normal atom.

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 the 0th element as an atom) 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.

List As in Edinburgh Prolog, it is enclosed with brackets (`[]`) with the elements delimited by commas. Note that it is also the save 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 uses JIS code. 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 (but it can be dynamically created at execution time for use).

Table 2.1: 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#"acb"`
Module	····· `module#foo, module#bar`

[Caution]In KL1, a string is an independent data type and does not serve as notation for a list which contains the character code as elements unlike in Edinburgh Prolog.

[Caution]Unification between strings is performed successfully if 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 that 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.
2.1 outlines the notation of data in KL1.

2.1.4 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 processor is possible, reduction normally begins with the goal assigned the highest priority.

[Caution]Priority is valid only within a processor and is not guaranteed outside individual processors. 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 processor.

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 in a processing system, 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".

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 :

$$\text{lower bound value} + (\text{upper bound value} - \text{lower bound value}) \times \frac{\text{Rate}}{4096} \qquad (0 \leq \text{Rate} \leq 4096)$$

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 @ priority($, Rate)

Assuming the logical priority of the called goal is Cp, the logical priority of the goal "Goal" is determined by

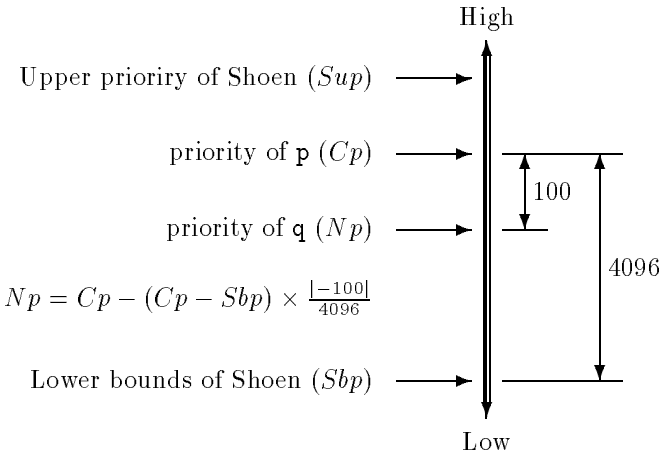
$$Cp + (\text{upper bound value} - Cp) \times \frac{|\text{Rate}|}{4096} \qquad (0 \leq \text{Rate} \leq 4096)$$

or

$$Cp - (Cp - \text{lower bound value}) \times \frac{|\text{Rate}|}{4096} \qquad (-4096 \leq \text{Rate} < 0)$$

[Example]Assuming the logical priority of "Goal" is Cp in the following program, logical priority Np of goal `q` is computed as follows:

```
Goal: ?- p.
Clause: p :- true | q @ priority($, -100).
```



2.1.5 Processor specification

In KL1, the processor to run each goal may be specified. The processor is specified with the processor number in the format:

Goal @ processor (processor-number)

In particular, a call of a goal which does not specify processor is executed with the same processor as that of the parent goal (which called that goal).

2.1.6 Macro notation

KL1 provides macro extensions for improved program coding. Macros may be defined by the user, but the macro with the following functions are provided as a standard feature.

- Constant description

- Arithmetic operation
- Conditional branch
- Implicit argument

In the following description, there are macros to be extended to built-in predicates. For details, see the section on built-in predicate.

Constant description

These macros are used to create constants.

`module#module-name`
It expands to module type data corresponding to module-name.

`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#'['`).

`key#lf`
ASCII code (10) which denotes line feed.

`key#cr`
ASCII code (13) which denotes carriage return.

Unification

`Left side = right side`
Expands to the unification of the left and the right side. 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]It is recommended to avoid using diff/2 because it is provided under tentative specification.

`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.

Table 2.2: Arithmetic Comparison Macros

Priority	Operators	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)

`Left side $:= right side`
: Expands to the unification of the left and right side. 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 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 2.2.

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.
- The right side of `<=`, `$<=` of the implicit argument macro. The result of the arithmetic operation is set to the argument specified in the left side. `:=` is for integer arithmetic and `$:=` 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 argument is a constant) when macro expansion takes place, computation is performed at compile time. Operators listed in 2.3 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 `( )`.

Table 2.3: Arithmetic operation macro

Priority	Operator	Pattern after expansion	Created 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)

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 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 colon may include "alternatively" or "otherwise" in addition to the above format. Their meaning is the same as that for 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 programs.

```
     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).
```

The preprocessor of the current compiler actually performs the above expansion. Predicate `'$foo/2/0'` is the predicate newly created by the preprocessor which performs macro expansion. The name of the predicate to be created always begins with `$` (it is generally recommended that the user should avoid using atom which begins with `$`).

[Caution]In the current compiler, conditional branch cannot be nested nor two or more conditional branches be written in the same clause.

Implicit argument

Since KL1 has no concept of global variable, any data to be accessed must be passed as an argument. While this rule 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 may have global declaration valid for an entire module or 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". The declaration of a global implicit arguments may appear only once at the beginning of the module definition (after public declaration). This declaration cannot be changed afterwords. 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 are 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 deelared implicit are added in the order of their declaration before the explicitly written arguments. The arguments are arranged in the order of:

- global implicit argument
- local implicit argument
- explicit argument

[Example]

```
:- module test.
:- public XXX.
:- implicit a:oldnew, b:shared.

p(X) --> true | q(X), r.
     %%  At this point, implicit argument is only "a" and "b".
     ...

:- local_implicit d:oldnew.
     %%  In what follows, implicit arguments include a, b, and d.
     ...

:- local_implicit d:shared, e:stream.
     %%  In what follows, implicit arguments include a, b, d, e.
     %%  Note that the type of argument named is changed to
     "shared".
     ...
```

```
:- local_implicit.
     %%  In what follows, 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 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 | &counter <= &counter + 1, q.`
  After expansion: `p(Cnt) :- true | add(Cnt,1,Cnt1), q(Cnt1).`
- Before expansion:

  ```
  p --> true | &counter <= &counter+1, &counter <= &counter*2,q.
  ```

  After expansion:

  ```
  p(Cnt) :- true |
         add(Cnt,1,Cnt1), multiply(Cnt1,2,Cnt2), q(Cnt2).
  ```

- Before expansion:

  ```
  p --> true | &counter <= &counter(2), q.
  ```

  After expansion:

  ```
  p(Cnt) :- true |
         set_vector_element(Cnt,2,Elem,Elem,_), q(Elem).
  ```

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 arguments of 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 | merge({In1,In2},Win), q(In1), r(In2).`
- Before expansion: `p --> true | &window <<= [putb("gazonk")],r.`
  After expansion: `p(Win) :- true | Win=[putb("gazonk")|Win1], r(Win1).`

oldnew type argument

This type of argument expresses a pair of arguments and is similar to DCG (definite clause grammars) function. With DCG, the two arguments indicate "difference list". In KL1, extended function allows allows vectors 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 for reasons of efficiency 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. 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 an integer, as similar to the shared type, `<=` 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

     &argument-name (position) (other than in the left side of <=)

Element update (2)

     &argument-name (position) <<= [element1, element2, ...]
```

Update (both 1 and 2) may be used only in the body part where built-in predicate set_vector_element/5 is used. For reference, vector_element3 is used in the guard part and set_vector_element/5 is used in the body part. 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), position's column of the vector is replaced by the right side of `<=` and `$<=`, 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 inserted as the elements of "difference list", and a new end is set to the position's column 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]

- Declaration: `:- implicit mutter:oldnew.`

- Before expansion: `p --> true | q, r.`
  After expansion: `p(Old,New) :- true | q(Old,Mid), r(Mid,New).`
- Before expansion: `p --> true | &mutter <<= [d_n], r.` After expansion: `p(Old,New) :- true | Old=[d_n|Mid], r(Mid,New).`
- Before expansion: `p --> true | &mutter(3) <= d_n, r.` After expansion:

```
p(Old,New) :- true |
        set_vector_element(Old,3,_,d_n,Mid), r(Mid,New).
```

- Before expansion: `p --> true | &mutter(1) <= &mutter(3), r.`
  After expansion:

```
p(Old,New) :- true |
        set_vector_element(Old,3,Elem,Elem,Midi),
        set_vector_element(Mid1,1,_,Elem,Mid2), r(Mid2,New).
```

- 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|Crd],Cdr,Mid], r(Mid,New).
```

- Before expansion: `p --> true | &mutter <= &mutter+1, r.`
  After expansion: `p(Old,New) :- true | add(Old,1,Mid), r(Mid,New).`

  Before expansion:

```
p(X) --> true |
        X = [&mutter(old)|XX], &mutter <= &mutter+1, p(XX).
```

  After expansion:

```
p(Old,New) :- 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 user-defined goal in the body part, the following processings are performed according to the declared argument

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
```

```
go :- true |
     merge(FILEout, FILEin),
     file:create(FILEin, "del.del", r),
     file:create(Answer, "/tmp/miyadel", w),
     loop(FILEout, Answer, [], 100 ,_).
loop(_) --> &counter =< 0 | true.
otherwise.
loop(A) --> true |
     &counter <= &counter - 1,
     &input <<= [getc(X)],
   {{ check(X, &output, &counter) }},
     loop(A).
check(ascii#a, Oh, Ot, Counter) :- true | Oh=[putt(Counter),
          nl|Ot].

otherwise.
check(_, Oh, Ot, _) :- true | Oh=Ot.
```

In the above example, 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 three implicit arguments 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)|Int1],
check(X, Oh,Om, Cnt1),
loop(In1, Om,Ot, Cnt1, A).
```

Note that predicate "check/4" enclosed with `{{` and `}}` does not have implicit arguments and is expanded as the predicate of four arguments. To pass the value of an implicit argument to such predicate, `&argument-name` as described above must be used.

[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).

User-defined macro

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 marco-definition-name.
macro-name1#macro-name2 => format after expansion.
```

In the current version, `#` must be used 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 this macro in the source program, it must be declared 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.

[Hint]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.

[Caution]In the self compiler of Version 1, The file for macro definition is temporarily fixed (with a file name of $> sys > csp > PIMOS.DIR > pimos.mac$"). Because this file usually contains the macro defined by PIMOS, a macro definition required by the user must be added to the existing macro. In the future, the user will have freedom of setting the macro definition file. The directory containing the macro definition file is the same as that defined by `pimos:` as standard directory. In the current version, which file be used as macro definition will not be affected by how the definition was set with `pimos:`.

[Caution]Because PIMOS uses the macro with Sharp sign (`#`) in the following format as standard, it is recommended for the user to avoid defining such macro format.

```
fep#???  shoen#???  builtin#???
pimos_tag#???  shell_tag#???  listener_tag#???
exception_code#???  exception_tag#???
```

where `???` is some terms.

[Caution]In the future versions, more generalized user-defined macro expansion mechanism will be introduced.

2.2 Built-in Predicate

This section describes the built-in predicate available on multi PSI Version 2. Each built-in predicate is shown in the following format:

vector(X, ^Size) :: **G**

↑ calling format ↑ describable location

Describable location: Built-in predicates have limited describable location due to its characteristics. If **G** is the describable location, it means the predicate can be described in the guard part alone. **GB** means that the predicate may be described in either the guard or body part.

Mode of argument: An argument with ^ indicates the argument (sometimes called "output argument" later) to be unified. Other arguments are input arguments. Where input argument has not been instantiated, if the built-in predicate appears in the guard part, the execution using that clause 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 in the guard part fails and the execution in the body part is treated as exception.

The macro coded in the following syntax is prepared for arithmetic operation (refer to the DEC-10 Prolog for meaning of operators).

<comparison expression>	::=	<expression> <comparison operator> <expression>
<integer operation expression>	::=	<variable> ":=" <expression>
<floating point arithmetic expression>	::=	<variable> "$:=" <expression>
<expression>	::=	<term> | <expression> <operator to add or subtract> <term>
<term>	::=	<factor> | <term> <operator to multiple or divide> <factor>
<factor>	::=	<integer> | <floating point number> | "(" <expression> ")"
<comparison operator>	::=	<integer comparison operator> | <floating point comparison>
<integer comparison operator>	::=	">" | "<" | ">=" | "=<'' \ "=:=" | "=\=" |
<floating point number comparison>	::=	"$>" \| "$<" | "$>=" \| "$<=" | "$=:=" \| "$=\=" |
<addition/subtraction operator>	::=	"+" | "-" | "/\" | "\/" | "xor"
<multiply/division operator>	::=	"*" | "/" | "<<" | ">>" | "mod"

2.2.1 Type check

wait(X) :: G

Suspends if X is undefined and successful otherwise.

atom(X) :: G

Suspends if undefined, successful if X is an atom, or fails otherwise.

integer(X) :: G

Suspends if X is undefined, successful if X is an integer, and fails otherwise.

floating_point(X) :: G

Suspends if X is undefined, successful if it is floating point number, or fails otherwise.

list(X) :: G

Suspends if X is undefined, successful if it is a list, or fails otherwise.

vector(X, ^Size) :: G

Suspends if X is undefined, if it is a vector, unifies the size with Size, or fails otherwise.

vector(X, ^Size, ^NewVector) :: B

Suspends if X is undefined. If it is a vector, unifies that size with Size and unifies X with NewVector. Raises an exception otherwise.

string(X, ^Size, ^ElementSize) :: G

Suspends if X is undefined. If it is a string, unifies that size with Size and unifies element size with ElementSize. Fails otherwise.

string(X, ^Size, ^ElementSize, ^NewString) :: B

Suspends if X is undefined. If it is a string, unifies that size with Size and unifies element size with ElementSize along with unifying X with NewString. Raises an exception otherwise.

module(X, ^Size) :: G

Suspends if X is undefined. If it is a module, unifies that size with Size. Fails otherwise.

code(X, ^Arity, ^Module, ^Position) :: G

Suspends if X is undefined. If it is a code, unifies that number of arguments with Arity, unifies the module to which the code belongs with Module and unifies the position from the beginning of the module with Position. Fails otherwise.

unbound(X, ^Result) :: B

If X is undefined variable at present (when unbound/2 is executed), vector {PE, Address, NewX} is unified with Result and the execution is successful. PE is processor number (integer) with variable, Address is the address of the variable, NewX is the result of unifying X. If X has been defined, {NewX} is unified with Result.

[Caution]In KL1 which is a concurrent language, recognizing a still undefined variable is not primarily meaningful as in a sequential language prolog. Such a variable may have been defined (by another concurrently running process) immediately after recognizing it as undefined. Therefore, this built-in predicate must be carefully used with proper care.

2.2.2 Comparison of integer

less_than(Integer1, Integer2) :: G

Suspends if Integer1 or Integer2 is undefined. Execution is successful if both are integer and Integer1 is less than Integer2. If the condition is anything else, execution fails. 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

Suspends if Integer1 or Integer2 is undefined. Successful if both are integer and Integer1¿=Integer2 is true. Fails if the condition is anything else. 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

Suspends if Integer1 or Integer2 is undefined. Successful if both are integer and Integer1 `=:=` Integer2 is true. Fails if the condition is anything else. Normally, the following macro notation is used.

```
X =:= Y => equal(X,Y).
```

not_equal(Integer1, Integer2) :: G

Suspends if Integer1 or Integer2 is undefined. Successful if both are integer and Integer1 `=\=` Integer2 is true. Fails if the condition is anything else.

Normally, the following macro notation is used.

```
X =\= Y => not_equal(X,Y).
```

2.2.3 Comparison of floating point number

floating_point_less_than(Float1, Float2) :: G

Suspends if Float1 or Float2 is undefined. Successful if both are floating point number and value of Float1 is less than that of Float2. Fails if the condition is anything else. 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

Suspends if Float1 or Float2 is undefined. Successful if both are floating point number and value of Float1 is not less than that of Float2. Fails if the condition is anything else. 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

Suspends if Float1 or Float2 is undefined. Successful if both are floating point number and value of Float1 is equal to that of Float2. Fails if the condition is anything else. Normally, the following macro notation is used.

```
X $=:= Y => floating_point_equal(X,Y).
```

floating_point_not_equal(Float1, Float2) :: G

Suspends if Float1 or Float2 is undefined. Successful if both are floating point number and value of Float1 is not equal to that of Float2. Fails if the condition is anything else. Normally, the following macro notation is used.

```
X $=n= Y => floating_point_not_equal(X,Y).
```

2.2.4 diff

diff(X, Y) :: G

If X or Y is not atomic term (data expressed in 1 word, that is, integer, atom, or empty vector) other than undefined variable no operation is guaranteed. Successful if "X is not equal to Y" is true. Fails if the condition is anything else. Normally, the following macro notation is used.

```
X = Y => diff(X,Y).
```

[Caution]Since this built-in predicate has temporary specification, use of "otherwise" is most recommendable. See the example below:

```
p(X, Y) :- 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).
```

2.2.5 Integer operation

add(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. The value obtained by adding Integer1 to Integer2 is unified with NewInteger. Program fails or exception is assumed if overflow occurs. Normally, the following macro notation is used.

```
Z := X + Y => add(X, Y, Z).
```

subtract(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. The value obtained by subtracting Integer2 from Integer1 is unified with NewInteger. Program fails or exception is assumed if 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

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. The value obtained by multiplying Integer1 by Integer2 is unified with NewInteger. Program fails or exception is assumed if overflow occurs. Normally, the following macro notation is used.

```
Z := X * Y => multiply(X, Y, Z).
```

divide(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. The quotient obtained by dividing Integer1 by Integer2 is unified with NewInteger. Program fails or exception is assumed if overflow occurs. [1] Normally, the following macro notation is used.

```
Z := X / Y => divide(X, Y, Z).
```

modulo(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. Program fails or exception assumed if Integer2 is 0. The remainder obtained by dividing Integer1 by Integer2 is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X mod Y => modulo(X, Y, Z).
```

shift_left(Integer, ShiftWidth, ^NewInteger) :: GB

Suspends if Integer is undefined variable. Program fails or exception is assumed if it is not integer. It suspends if ShiftWidth is undefined variable. It fails or exception assumed if it is an integer from 0 to 31. Integer is logically shifted left by ShiftWidth bits and the result is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X << Y => shift_left(X, Y, Z).
```

[1] An overflow occurs when the negative number with maximum absolute value is divided by -1.

shift_right(Integer, ShiftWidth, ^NewInteger) :: GB

Suspends if Integer is undefined variable. Program fails or exception is assumed if it is not integer. It suspends if ShiftWidth is undefined variable. It fails or exception assumed if it is other than an integer from 0 to 31. Integer is logically shifted right by ShiftWidth bits and the result is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X >> Y => shift_right(X, Y, Z).
```

and(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. Each bit of Integer1 and Integer2 is ANDed and the result is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X /\ Y => and(X, Y, Z).
```

or(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. Each bit of Integer1 and Integer2 is ORed and the result is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X \/ Y => or(X, Y, Z).
```

exclusive_or(Integer1, Integer2, ^NewInteger) :: GB

Suspends if Integer1 or Integer2 is undefined. Program fails or exception is assumed if they are not integer. Each bit of Integer1 and Integer2 is XORed and the result is unified with NewInteger. Normally, the following macro notation is used.

```
Z := X xor Y => exclusive_or(X, Y, Z).
```

complement(Integer, ^NewInteger) :: G

Suspends if Integer is undefined. Program fails if it is not an integer. The value obtained by inverting each bit of Integer is unified with NewInteger. Normally, the following macro notation is used.

```
Y := n(X) => complement(X, Y).
```

[Caution]At present, this predicate can only be written in the guard part. This will be revised in the future.

2.2.6 Floating point arithmetic

floating_point_add(Float1, Float2, ^NewFloat) :: GB

Suspends if Float1 or Float2 is undefined. Program fails or exception is assumed if it is not floating point number. The value of Float1 plus Float2 is unified with NewFloat. If overflow occurs, infinity is output to NewFloat. Normally, the following macro notation is used.

```
Z $:= X + Y => floating_point_add(X, Y, Z).
```

[Caution]The following operation also results in invalid operation, and program fails or exception assumed.

```
(+infinity) + (-infinity)
(-infinity) + (+infinity)
```

floating_point_subtract(Float1, Float2, ^NewFloat) :: GB

Suspends if Float1 or Float2 is undefined. Program fails or exception is assumed if it is not floating point number. The value obtained by subtracting Float2 from Float1 is unified with NewFloat. If overflow occurs, infinity is output to NewFloat. Normally, the following macro notation is used.

```
Z $:= X - Y => floating_point_subtract(X, Y, Z).
```

[Caution]The following operation also results in invalid operation, and program fails or exception assumed.

```
(+infinity) - (+infinity)
(-infinity) - (-infinity)
```

floating_point_multiply(Float1, Float2, ^NewFloat) :: GB

Suspends if Float1 or Float2 is undefined. Program fails or exception is assumed if it is not floating point number. The value obtained by multiplying Float1 by Float2 is unified with NewFloat. If overflow occurs, infinity is output to NewFloat. Normally, the following macro notation is used.

```
Z $:= X * Y => floating_point_multiply(X, Y, Z).
```

[Caution]The following operation also results in invalid operation and program fails or exception assumed.

$$(\pm 0.0) * (\pm infinity)$$

$$(\pm infinity) * (\pm 0.0)$$

floating_point_divide(Float1, Float2, ^NewFloat) :: GB

Suspends if Float1 or Float2 is undefined. Program fails or exception is assumed if it is not floating point number. If Float2 is 0.0, infinity is output to NewFloat. The quotient obtained by dividing Float1 by Float2 is unified with NewFloat. If overflow occurs, infinity is output to NewFloat. Normally, the following macro notation is used.

```
Z $:= X / Y => floating_point_divide(X, Y, Z).
```

[Caution]The following operation also results in invalid operation and program fails or exception occurs.

$$(\pm 0.0) \div (\pm 0.0)$$

$$(\pm infinity) \div (\pm infinity)$$

2.2.7 Conversion of integer-floating point number

floating_point_to_integer(Float, ^Integer) :: GB

Suspends if Float is undefined. Failure or exception occurs if it is not floating point number. Float is converted to integer (ignoring decimal) and unified with Integer. If overflow occurs, program fails or exception occurs. Normally, the following macro notation is used.

```
Y := int(X) => floating_point_to_integer(X, Y).
```

integer_to_floating_point(Integer, ^Float) :: GB

Suspends if Float is undefined. Failure or exception occurs if it is not integer. Integer is converted to floating point number and unified with Float. Normally, the following macro notation is used.

```
Y $:= float(X) => integer_to_floating_point(X, Y).
```

2.2.8 Vector

new_vector(^Vector, Size) :: B

Suspends if Size is undefined. Exception occurs if Size is other than non-negative integer. In other cases, a vector whose number of elements is Size is newly created and unified with Vector. All elements of the created vector is initialized with integer 0.

vector_element(Vector, Position, ^Element) :: G

Suspends if Vector is undefined. Failure occurs if it is other than vector. It suspends if Position is undefined. It fails if it is not non-negative integer, or it is more than the number of elements of Vector. In other cases, the Position'th element of Vector is unified with Element.

vector_element(Vector, Position, ^Element, ^NewVector) :: B

Suspends if Vector is undefined. Exception occurs if it is other than vector. It suspends if Position is undefined. Exception occurs if it is not non- negative integer or if it is more than the number of elements of Vector. In other cases, the Position'th element of Vector is unified with Element along with unifying Vector with NewVector.

set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B

Suspends if Vector is undefined. Exception occurs if it is other than vector. It suspends if Position is undefined. Exception occurs if it is not non- negative integer or if it is more than the number of elements of Vector. In other cases, the Position'th element of Vector is unified with OldElem, and a new vector is created by replacing Position'th element of Vector with NewElem and is unified with NewVect.

[Hint]NewElem is not necessarily defined and may be undefined variable when this built-in predicate is executed.

2.2.9 String

new_string(^String, Size, ElementSize) :: B

Suspends if Size is undefined. Exception occurs if it is other than non-negative integer. It suspends if ElementSize is undefined. Exception occurs unless it is an integer other than 1, 8, 16, 32. In other cases, A string whose number of elements Size and element bit length is ElementSize is created and unified with String. All elements of the created string are initialized with integer 0.

string_element(String, Position, ^Element) :: G

Suspends if String is undefined. Failure occurs if it is other than string. It suspends if Position is undefined. It fails if it is other than non-negative integer or if it is more than the number of elements of String. In other cases, Position'th element of String is unified with Element as an integer.

string_element(String, Position, ^Element, ^NewString) :: B

Suspends if String is undefined. Exception occurs if it is other than string. It suspends if Position is undefined. Exception occurs if it is other than non-negative integer or it is more than the number of element of String. In other cases, Position'th element of String is unified with Element along with unifying String with NewString.

set_string_element(String, Position, NewElement, ^NewString) :: B

Suspends if String is undefined. Exception occurs if it is other than string. It suspends if Position is undefined. Exception occurs if it is other than non-negative integer or more than the number of element of String. It suspends if NewElement is undefined. Exception occurs if it is other than positive integer in a range accommodated in the element. In other cases, a new string is created by replacing Position'th element of String with NewElement and unified with NewString.

[Caution]Exception taking place depending on the range of NewElement value is different from the specifications of PDSS.

At present, the following predicates are not taken as built-in predicates. However, string-related utilities provided by PIMOS are invoked by coding them in macro notation. Some of them will possibly become built-in predicates but the source program needs not be modified as long as these predicates are coded in macro notation. When using the macro notation that follows, the following declaration must be coded at the beginning of the module.

```
:- with_macro pimos.
```

builtin#substring(String, Position, Length, ^SubString, ^NewString) :: B

Suspends if String is undefined. Failure occurs if it is other than string. It suspends if Position is undefined. It fails if it is other than non-negative integer or if it is more than the number of elements of String. It suspends if Length is undefined. It fails if it is other than positive integer or if Position+Length exceeds the number of elements. In other cases, a new string is created by copying the length "Length" from the Position'th location of String and is unified with SubString. String and NewString are also unified.

builtin#set_substring(String, Position, SubString, ^NewString) :: B

Suspends if String is undefined. Failure occurs if it is other than string. It suspends if Position is undefined. Failure occurs if it is other than non-negative integer or if it is more than the number of elements of String. It suspends if SubString is undefined. Failure occurs if it is other than the string with the same element bit length as the String. Failure occurs if Position+(SubString length) exceeds the number of elements of String. In other cases, a string is created by replacing the string after Position'th location of String with the string indicated by SubString, and unified with NewString.

builtin#append_string(String1, String2, ^NewString) :: B

Suspends if Sting1 or String2 is undefined. Failure occurs if they are other than the string of the same element bit length. In other cases, a new string, with the contents of String2 appended to the contents of String1, is created and unified with NewString.

builtin#append_string([String1, String2, ..., StringN], ^NewString) :: B

Suspends if anything in a range from String1 to StringN is undefined. Failure occurs if they are other than the string of the same element bit length. In other cases, a new string, with the contents of StringN appended from String1, is created and unified with NewString.

2.2.10 Atom

new_atom(^Atom) :: B

A new atom is created and unified with Atom.

[Caution]PIMOS is responsible for providing correspondence between atom and the name string in PIMOS/KL1, 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 (7).

2.2.11 Code and module

module_element(Module, Position, ^Element) :: B

Suspends if Module is undefined. Exception occurs if it is other than module. It suspends if Position is undefined. Exception occurs if it is other than non-negative integer or if it is more than the number of elements of Module. In other cases, the Position'th element of Module is unified with Element.

predicate_to_code(Module, Name, Arity, ^Code) :: B

Suspends if Module is undefined. Exception occurs if it is other than module. It suspends if Name is undefined. Exception occurs if it is other than atom. It suspends if Arity is undefined. Exception occurs if it is other than non-negative number. In other cases, the code denoting module Module, predicate name Name, and argument count Arity is unified with Code. In the absence of the corresponding predicate in the module (undefined or not declared in public), 3-element vector that is `{Module, Name, Arity}` is unified with Code.

code_to_predicate(Code, ^Module, ^Name, ^Arity, ^Info) :: B

Suspends if Code is undefined. Exception occurs if it is other than code. In other cases, unifies the module to which Code belongs with Module, unifies predicate atom with Name, argument count with Arity, and 1-word information (¡contents of Code address −3¿) for software with Info.

apply(Code, Args) :: B

Suspends if Code is undefined. Exception occurs if it is other than code. Spy exception occurs if it is corresponds to spied predicate. It suspends if Args is undefined. Exception occurs if it is other than vector. In other cases, the predicate corresponding to Code is called with argument environment Args.

apply_tracing(Code, Args, ID) :: B

Suspends if Code is undefined. Exception occurs if it is other than code. Spy exception occurs if it is corresponds to spied predicate. It suspends if Args is undefined. Exception occurs if it is other than vector. In other cases, the predicate corresponding to Code is called with argument environment Args as the goal to be traced. ID is used to identify this goal (optional data type is acceptable), and is returned when trace exception occurs.

apply_spying(Code, Args, Spied_code_vector, ID) :: B

Suspends if Code is undefined. Exception occurs if it is other than code. Spy exception occurs if it is corresponds to spied predicate. It suspends if Args is undefined. Exception occurs if it is other than vector. In other cases, the predicate corresponding to Code is called with argument environment Args as the goal to be spied. Spied_code_vector is a vector intended to specify the predicate to be spied, and has iteration sequence of module_name_atom, predicate_name_atom, and arity as the element. The wildcard of module_name_atom and predicate_name_atom is integer 0, the wildcard for arity is `[]` (NIL). If the element is undefined, illegal input exception

occurs. ID is used to identify this goal (optional data type is acceptable) and is returned when spy exception occurs.

Spied_code_vector has the following format.

```
{ Mod1, Pred1, Arity1, Mod2, Pre2, Arity2, ... , ModN, PredN, ArityN }
```

[Caution]This built-in predicate is subject to temporary specification and will be revised in the future.

2.2.12 Stream support

merge(In, ^Out) :: B

Creates a merger and unifies the input stream for that merger with In, and unifies the output stream with Out. When receiving a vector, the merger adds all elements in the vector as the input stream. 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([], 0) :- true | 0=[]
merge([A|I], 0) :- true | 0=[A|NO], merge(I, NO).
merge({}, 0) :- true | 0=[].
merge({I}, 0) :- true | merge(I, 0).
merge({I1, I2}, 0) :- true | merge(I1, I2, 0).
merge({I1, I2, I3}, 0) :- true | merge(I1, I2, I3, 0).
                ...
merge([], I2, 0) :- true | merge(I2, 0).
merge(I1, [], 0) :- true | merge(I1, 0).
merge([A|I1], I2, 0) :- true | 0=[A|NO], merge(I1, I2, NO).
merge(I1, [A|I2], 0) :- true | 0=[A|NO], merge(I1, I2, NO).
merge({}, I2, 0) :- true | merge(I2, 0).
merge(I1, {}, 0) :- true | merge(I1, 0).
merge([I3, I4], I2, 0) :- true | merge(I3, I4, I2, 0).
merge({I3, I4, I5}, I2, 0) :- true | merge(I3, I4, I5, I2, 0).
                ...
                ...
```

2.2.13 Special input/output

[Caution]The built-in predicates for input and output described here are provided in low level in order to debug easily, and are not suitable for I/O use in terms of both function and performance. If PIMOS debugging function is enhanced in the future, there is a strong likelihood that the call from user programs is suppressed.

read_console(^X) :: G

Reads the data input from CSP window and unifies it with X. It is inputed as "tag part [blank] value part" each being entered with hexadecimal numbers. If the tag part is omitted, it is assumed to be an integer.

[Caution]All processors halt when waiting for input.

display_console(X) :: G

Suspends if X is undefined. The current value of X is written to the CSP window and LF is entered. If X is 8-bit string, it is assumed to be ASCII string, and if X is 16-bit string, it is assumed to be 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.

[Caution]All processors halt during output.

put_console(X) :: G

Suspends if X is undefined. If X is an integer, it is assumed to be ASCII code and one character is written to the CSP window. At this time, no LF is entered. If X is 8-bit string, it is assumed to be ASCII string, and the contents of that string is written to the CSP window with no LF entered. If X is other data type, no operation is guaranteed.

2.2.14 Others

current_priority(^Priority, ^Min, ^Max) :: B

The logical priority of this predicate is unified with Priority, the minimum priority acceptable in sho-en to which the predicate belongs is unified with Min, and the maximum priority is unified with Max.

current_processor(^ProcessorNumber, ^X, ^Y) :: B

The processor number to which this predicate belongs is unified with ProcessorNumber, the number of processors in X direction is unified with X, and the number of processors in Y direction is unified with Y. The total number of processors is indicated by X x Y, and processor numbers are assigned from 0 to $X \times Y - 1$.

[Caution]Form of physical connection of the processors corresponding to the processor number is determined by CSP. Standard correspondence will be documented in the future.

hash(X, ^Value, ^NewX) :: B

Suspends if X is undefined variable. In other cases, an integer (hash value) is created based on X and is unified with Value, X and NewX are also unified. X may be optional type.
The following integers are created 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

[Caution]The specification of hash function will be modified in the future. Therefore, there is a likelihood that the revised specification will possibly suspend execution if vector elements are undefined. 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). Hash value never changes even in cases where the pending hash is completed because of the instanciation of variable.

builtin#halt :: G

Halt all processors.

[Caution]This predicate is tentatively intended for debugging and will not be used from user programs in the future.

2.3 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.

[Caution]PIMOS provides the task function (4.2) which accommodates the management function for I/O devices in addition to the shoen function.

2.3.1 Outline of Shoen function

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 (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 including computing time or memory areas used in shoen execution. Specifically, as the consumption of resource comes close to the specified upper bound, it is reported. When resource consumption actually exceeds the upper bound, execution is suspended. The upper bound of resource consumption can be increased later(once the upper bound 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. Resource managed by KL1 language shoen function is called "Language Definition Resources" in contrast to the resource managed by PIMOS.

[Caution]The resource consumption currently managed only includes those corresponding to the number of reductions (1 for goal creation and 2 for termination of the reduction, thus 2 is taken for each reduction). In the future, computing time, memory consumption, and communication traffic between processors will also be managed.

Exception management: Manages the exceptions taking place during shoen execution. Failed execution (absence of the candidate for the goal, or failed unification) is also treated as exception. Each shoen has the mask pattern (integer value) used to determine what exception be managed. Each exception event also has tags with integer value according to its type, and exception is reported to the innermost shoen for which AND of the exception tag and the mask pattern is not 0. Information indicating what exception occurs in the event of exception 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 caused exception are passed to the exception manager. By instanciating these variables, the goal caused exception is replaced with the goal specified by the code and the argument vector. Since there are goals in the shoen awaiting the instanciation of these variables, shoen execution does not successfully terminate unless exception management has been performed.

2.3.2 Creation of shoen

shoen:execute(Code, Argv, MinPrio, MaxPrio, ExcepMask, Control, ^Report)

Creates a shoen.

The goal at the top level in the created shoen is specified by Code (code type data corresponding to top level predicate) and Argv (argument vector of top level goal).

Execution control of the created shoen is performed from stream Control, and report of the execution status is made from stream Report (specifically described later). The range of priority of the goals in the shoen is specified with MinPrio and MaxPrio. Priority is specified with integer value 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.

2.3.3 Control stream

The control stream of the shoen is a message stream in the direction from the user to the shoen. Shoen execution or resource consumption can be controlled according to the messages listed below:

start

Starts execution of the shoen. Immediately after creation, the shoen is dormant, and it is made ready for execution when this message is sent. This message is also used to restart the execution suspended by "stop" message shown below.

stop

Suspends the execution of the shoen. Execution can be restarted by "start" message.

abort

Aborts execution of the shoen. If execution is aborted by with this message, execution cannot be restarted later.

statistics

Indicates the report concerning the amount of resource consumed by the execution within the shoen. Report is sent from the report stream.

add_resource(Amount_h, Amount_l)

The value whose high order 32 bits are expressed in Amount_h and low order 32 bits are expressed in Amount_l is added to the upper bound value of resource consumption amount.

allow_resource_report

It allows the report to be sent as the upper bound value of the amount of consumed resource is nearly reached. Report is normally ready for sending, but if the report is made to the effect that the resource comes close to its upper bound, report is no longer sent. If this message is received, the report ready state enters again. This mechanism is intended to eliminate number of useless reports.

2.3.4 Report stream

The report stream of the shoen is a message stream in the direction from the shoen to the user. The following two kinds of message are routed through the report stream.

Event report message: This message is used to report the events occurring associated with shoen execution.

Receive ACK message: This message is put on the stream when the 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" and "resource_low" makes it possible to determine whether additional resource is needed.

terminated

Reports the termination of the shoen. Whether or not abort message is ahead of this message in the report stream makes it possible to determine whether normal termination upon termination of the reduction of all goals in the shoen or forced abnormal termination. After reporting this message, reduction of goals in the shoen or unification is no longer performed.

exception(Kind, Info, ^NewCode, ^NewArgv)

Reports the exception events taking place during shoen execution.

Kind is an atom indicating the kind of exception. Info is information vector concerning exceptions and varies in format according to the kind of exception. NewCode and NewArgv are used for specifing the goals to be executed in place of the goal causing exception, and each of them is the variable corresponding to code and argument vector. The format of exception information (Info) for each type of exception (kind) is shown below. In the explanation, Proc is the number of the processor containing the goal (built-in predicate) which caused exception, and all exceptions are assigned this Proc (number). Module is the module of that goal, Offset is the offset from the beginning of the module indicating goal position, Name is the name of that goal, and Argv is argument.

illegal_input: {Proc, Module, Offset, Name, Position, Argv}
Position'th argument is illegal ($1 \leq Position$).

range_overflow: {Proc, Module, Offset, Name, Argv}
Argument is not within the range of correct values.

out_of_bounds: {Proc, Module, Offset, Name, Argv}
Structure index is out of range.

integer_overflow: {Proc, Module, Offset, Name, Argv}
Integer operation overflows.

integer_zero_division: {Proc, Module, Offset, Name, Argv}
Division by zero.

invalid_floating_point_operation: {Proc, Mod, Offset, Name, Argv}
Invalid operation in real number operation.

[Hint]The following operation is invalid.

(+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: {Proc, Mod, Offset, Name, Argv}
Number of arguments of high order predicate mismatch.

illegal_merger_input: {Proc, Module, Offset, Name, MI, FMI}
Illegal merger input. MI is illegal merger input, FMI is the merger input used (sent correct data) to return from exception.

[Caution]This exception has not be implemented, and illegal merger input results in unification_failure.

reduction_failure: {Proc, Code, Argv}
Failure of guard for all candidate clauses. Code is the code of failed predicate.

unification_failure: {Proc, X, Y}
Failure in body unification. X and Y are two operands for failed unification.

[Hint]While the top level of X and Y must always be instanciated (otherwise, unification never fails), elements for structure is not always instanciated.

[Caution]Unification between structures is separately performed for each element. So that failure in unification of `[a, b, c]` and `[a, b, d]` is reported as the failure in the unification of c and d.

deadlock: {Proc, Type, Code, Argv}
Report of deadlock on user-defined predicate. Type is the type of deadlock (integer), Code is the code of the predicate suffering deadlock.

deadlock: {Proc, Type, Mod, Offset, Name, Argv}
Report of deadlock on built-in predicate. Type is the type of deadlock (integer).

merger_deadlock: {Proc, Type, Mod, Offset, Name, Abadonedv}
Report of deadlock on built-in predicate merge/2. Type is the type of deadlock, Abandanedv is the vector of the abandoned input argument of the merger.

raised: {Proc, Info, Data} Exception was intentionally generated by predicate shoen:raise/3 (explained later). Info and Data are assigned to the argument of raise.

trace: {Proc, Trace_id_list, Sub_goals_info_list} Trace exception. Trace_id_list is ID list used to identify the goal and is given by built-in predicate apply_tracing/3. Sub_goals_info_list is the list of subgoal information which include the following.

User goal: {enqueue, Code, Argv}

Priority specified user goal: {enqueue_with_prio, Prio, Code, Argv}

Processor specified user goal: {enqueue_to_proc, Proc, Code, Argv}

Module external user goal: {enqueue_external, {Module, Pred, Arity}, Argv}

Body built-in predicate: {Blt_name, Argv}

[Caution]This exception is temporary specification and will be modified in the future for protection of OS.

spy: {Proc, Spy_id, Mod, Offset, Name, Spy_info} Spy exception. Spy_id is ID used to identify the goal is given by built-in predicate apply_spying/4. Spy_info is the vector of the goal information detected with spy and includes the following.

Goal type User goal	{Code, Argv}	execute enqueue_goal
Module external	{Code, Argv}	execute_external enqueue_goal_external
Priority specified user goal	{Prio, Code, Argv}	enqueue_goal_with_priority
Processor specified user goal	{Proc,Code,Argv}	enqueue_goal_to_processor
Special built-in predicate	Argv	apply apply_tracing apply_spying create_shoen

[Caution]This exception is temporary specification and will be modified in the future for protection of OS.

etc: {Proc, Sub_code, Sub_info}
Report of the exceptions other than the above listed. Sub_code and Sub_info are auxiliary code and information of exception.

[Caution]This exception should not occur at present.

resource_low

Reports that the resource comes close to the upper bound.

[Caution]Report on how closer is dependent on the processing system.

statistics(Info_h, Info_l)

Statistics report on resource consumption. Statistics starts when control message "statistics" is received. Upon starting, statistics_started (described later) is returned, and this message is sent upon completion of statistics. Info_h is high order 32 bits and Info_l is lower 32 bits of the integer value which indicate the total amount of resource consumption in the shoen)including descendent shoens).

If statistics is sent upon completion of shoen execution, this message returns correct value. When the shoen is in operation, reported amount of resource consumption is incorrect, but it is guaranteed that the amount should not exceed the actual consumption, and several times of statistics should result in simple increase.

[Caution]Info_h and Info_l are likely modified in the future.

return_resource(Amount_h, Amount_l)

Reports that the upper bound of resource consumption was exceeded. Amount_h is high order 32 bits of the resource amount, Amount_l is low order 32 bits accordingly. The amount of resource which was not consumed at the time shoen execution terminated is reported with this message. If consumable amount of resource (the upper bound value of consumption minus consumed amount) exceeds 2_{63} during operation, the result is also reported.

started

stopped

aborted

resource_added

resource_report_allowed

statistics_started

These are Receive ACK messages each corresponding to control messages start, stop, abort, add_resource, allow_resource_report, and statistics.

2.3.5 Occurrence of exception

Exception events occur in built-in predicates and also can occur intentionally using the following predicates.

shoen:raise(Tag, Info, Data)

Raises an exception. Exception tag is Tag.

If Info is a structure, or if an element is a structure, raising of an exception is suspended until either element is completely instanciated. This restriction does not apply to Data. For

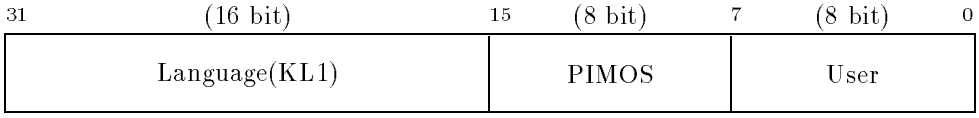


Figure 2.1: Bit position of exception tag

these reasons, as long as the exception management side is checking the contents of Info alone, exception management will not be suspended for reason of incomplete instanciation.

[Hint]In usual case, Data is not checked its contents by 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".

2.3.6 Exception management utility

Based on the Module and Offset of Info returned upon reporting an exception event, it is possible to determine which predicate raises that exception by using the following predicates.

exception_table:get_info(Module, Offset, ^Blt_name, ^Mod, ^Pred, ^Arity)

Blt_name, Mod, Pred, and Arity are obtained from Module and Offset returned upon report of the exception event. Each being the built-in predicate name atom raising the exception, the module name atom, predicate name atom, and arity of the predicate calling that built-in predicate.

2.3.7 Exception tag

Bit position of the exception tag is divided into three area as shown in 2.1, 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.

2.4 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 changed in the future, so that it is recommended to use the macro prepared by PIMOS when specifying the mask pattern of the shoen (or 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.

Table 2.4: Exception tag for language definition exception

Kind of exception	Bit position	Meaning
illegal_input	16	illegal input argument
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	mismatch number of arg
(reserved)	23	
illegal_merger_input	24	illegal merger input
reduction_failure	25	all candidate clause failure
unification_failure	26	unification failure
deadlock	27	deadlock detected
merger_deadlock	28	merger deadlock detected
trace	29	trace
spy	29	spy
(reserved)	30	
etc	31	others

Chapter 3

UTILITY

PIMOS has common utilities which are available for any application programs as well as PIMOS. These utilities provide the functions:

- Comparing the values of KL1 data
- Calculating hash function on KL1 data
- Database (pool) where data is stored and retrieved
- Filters for sorting and merging data sets
- Database where character string data is stored and retrieved (infinite-length buffer)
- Filters for blocking and deblocking character string data (buffering filter)

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 with such predicate called. Note that all the streams obtained by predicate call contain the built-in merger (see 2.2.12)

3.1 Comparison

In general, PIMOS has a mechanism that can compare any KL1 data uniquely:

comparator:sort(X, Y, ^S, ^L, ^Swapped)

In this predicate, first `X` and `Y` are compared. If these values are different, smaller one is unified with `S` and larger one with `L`. If these are the same, `X` is unfied with `S` and `Y` is unified with `L` (this is called "stable"). If `X` is larger than `Y`, `yes` is unified with `Swapped`, otherwise `no`.

Comparison of KL1 Data

KL1 Data is totally ordered according to the following rule. The order between different types is determined by their types, except of between integer and floating-point. The order on types is that integer, floating-point < atom < string < list < vector < module < code < others. order. The order between integer and floating-point number depend on their value. Their values are compared in a natural arithmetical sense. If the values are exactly the same, the integer type argument is assumed larger than the floating-point type. The order on the same type data is defined as follows:

Integer Signed integer

Floating-point Signed floating-point number

Atom Atom number

String For the strings of which bit length of each element different, the comparison is based on the bit length. For those of which bit length of each element the same, the comparison is based on the lexicographical order.

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.

[Caution]The definition of type preference may be changed in the future.

3.2 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 unified with `H`. Or, `X` is unified with `Y` as is. The value unified with `H` is always non-negative.

[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 2.2.14)

3.3 Pool

PIMOS has a mechanism (pool) that can generally store any KL1 data. Except those for search keys and data for comparison, any data not defined can be stored in the pool.

3.3.1 Pool without key

Bag

Simple pool which has only the basic functions such as input and output. You cannot retrieve an element from this pool with leaving it in the pool: after you have got the element, it does not exist in the pool.

Generating Objects .

pool:bag(S)

Generates a new object, and gets stream `S` in it.

Message Protocol .

empty(^YorN)

Inquires whether Bag is empty or not. If Bag is empty, it sets `YorN` to `yes`; otherwise `no`.

put(X)

Adds element `X` to Bag.

get(^X)

Fetches an element from Bag and unifies with it to `X`. Which element will be gotten is uncertain. When you get an element, it disappears from Bag. If Bag is empty, this function fails.

get_if_any(^X)

Fetches an element from Bag, if any, and unifies it with `X` in the {*Value*} format. Which element will be gotten is uncertain. When you get an element, it disappears from Bag. If Bag is empty, this function unifies `{}` with `X`.

get_all(^O)

Fetches all the elements from Bag, and unifies them with `O` in the list format. If Bag is empty, it unifies `[]` to `O`. The order of the elements in the list is undefined.

do(Stream)

Processes sequentially the messages in Stream when it gets a list containing messages in Stream.

Stack

Basically the same as Bag, except that `get` and `get_if_any` get the elements in the reverse order that `put` added them (LIFO).

Generating Objects .

pool:stack(S)

Generates a new stack, and gets stream S in it.

Message Protocol Same as Bag, except that the retrieving order is LIFO.

Queue

Basically the same as Bag, except that `get` and `get_if_any` get the elements in the same order that `put` added them (FIFO).

Generating Objects .

pool:queue(S)

Generates a new queue, and gets stream `S` to it.

Message Protocol Same as Bag, except that the retrieving order is FIFO.

Sorted Bag

Basically the same as Bag. Elements are fetched in an ascending order. Besides PIMOS uses by default the standard routine (`comparator:sort/5`) for quantitative comparison, you can specify a user-defined comparison routine. In such case, the user-defined routine must have the same interface as the standard 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.

Generating Objects .

pool:sorted_bag(S)

Generates objects of `sorted bag` with the standard comparison routine (`comparator:sort/5`).

pool:sorted_bag(Comparator, S)

Generates `sorted bag` with the comparison routine by specifying the comparison routine code as the first argument. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

Message Protocol Same as Bag. `get` and `get_if_any` returns the current *minimum* element, and `get_all` returns a ascending-order list.

3.3.2 Pool with keys

PIMOS provides a mechanism capable of storing any KL1 data with keys and retrieving them by key.

For a pool with hash table, the initial size of the hash table is default or determined by an argument on table generation, and it is automatically increased when the number of elements are so increased that the hash table can contain all the elements.

Keyed Bag

Basic keyed pool. It uses a hash table.

Generating Objects .

pool:keyed_bag(S)

Generates a keyed bag object with the standard hash function (`hasher:hash/3`). The initial size of the hash table is 1.

pool:keyed_bag(S, Size)

Generates a keyed bag object with the standard hash function (`hasher:hash/3`) and a specified initial table size, by specifying the initial size of the hash table as the second argument.

pool:keyed_bag(Hasher, S, Size)

Generates a keyed bag object with a specified hash function and the initial table size, by specifying a hash function code as the first argument and the initial size of a hash table as the second argument. Note that this hash function must be declared as *public* and have the same number of arguments and the same contents as those of the standard hash function `hasher:hash/3`.

Message Protocol .

empty(^YorN)

Inquires whether Keyed bag is empty. If empty, unifies `yes` with `YorN`, otherwise `no`.

empty(Key, ^YorN)

Inquires whether Keyed bag contains an element of which key corresponds to a specified one (`Key`). If there is no corresponding element, unifies `yes` with `YorN`, otherwise `no`.

put(Key, X)

Adds an element of which key is `Key` and value is `X`.

get(Key, ^X)

Fetches an element of which key within Keyed bag corresponds to `Key` (specified one), unify it with `X`. If there are two or more keys corresponding to the specified one, it is undefined that which key is fetched. The fetched key is deleted from the keyed bag. If there is no corresponding key, this function fails.

get_if_any(Key, ^X)

Fetches any element of which key within Keyed bag corresponds to `Key` (specified one), unifies it with `X` in a {*Value*} format. If there are two or more keys corresponding to the specified one, it is undefined that which key is fetched. The fetched key is deleted from the keyed bag. If there is no corresponding key, this function unifies `{}` with `X`.

get_all(^O)

Fetches all the elements from Keyed bag, and unifies `O` with a list of elements in the form of {`Key, Data`}. If Keyed bag is empty, unifies `[]` with `O`.

get_all(Key, ^O)

Fetches all the elements of which key corresponds to `Key` (specified one), and unifies it with `O` in a list format. (If Keyed bag is empty, unifies `[]` with `O`.)

get_and_put(Key, ^X, Y)

Fetches an element of which key corresponds to `Key` (specified one), unifies it with `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 key corresponds to `Key` (specified one), unifies it with `X` in a {*Value*} format, and stores a new element `Y` in the same location. If there is no corresponding element, unifies `{}` with `X` and adds `Y`.

do(Stream)

Processes sequentially messages in Stream when passing a list of messages into Stream.

Keyed Set

Keyed pool that does no allow duplicate keys. It uses a hash table.

Generating Objects .

pool:keyed_set(S)

Generates a keyed set object with the standard hash function `hasher:hash/3`. The initial size of the hash table is 1.

pool:keyed_set(S, Size)

Generates a keyed set object with the standard hash function `hasher:hash/3` and a specified initial table size, by specifying the initial size of the hash table as the second argument.

pool:keyed_set(Hasher, S, Size)

Generates a keyed set object with a specified hash function and initial table size, by specifying a hash function code as the first argument and the initial size of the hash table as the second argument. Note that this hash function must be declared as *public* and have the same number of arguments and the same contents as those of the standard hash function `hasher:hash/3`.

Message Protocol .

empty(^YorN)

Inquires whether the Keyed set is empty. If it is empty, unifies `yes` with `YorN`, otherwise `no`.

empty(Key, ^YorN)

Inquires whether Keyed bag contains an element of which key corresponds to a specified one (`Key`). If there is no corresponding element, unifies `yes` with `YorN`, otherwise `no`.

put(Key, X, ^OldX)

Adds an element of which key is `Key` and value is `X`, to Keyed set. If the Keyed set already contains a specified key, it is updated and the previous value is unified with `OldX` in an element vector {*Value*} format. If there is no corresponding key, null vector `{}` is unified with `OldX`.

get(Key, ^X)

Fetches an element of which key within Keyed bag corresponds to `Key` (specified one), unified it with `X`. The fetched element is deleted from the keyed bag. If there is no corresponding element, this function fails.

get_if_any(Key, ^X)

Fetches any element of which key within Keyed bag corresponds to `Key` (specified one), unifies it with `X` in a {*Value*} format. The fetched element is deleted from the keyed bag. If there is no corresponding key, this function unifies `{}` with `X`.

get_all(^O)

Fetches all the elements from Keyed bag, and unifies `O` with a list of elements in the form of {`Key, Data`}. If Keyed bag is empty, unifies `[]` with `O`.

get_all(Key, ^O)

Fetches all the elements of which key corresponds to `Key` (specified one), and unifies it with `O` in a list format. (If there is no corresponding key, unifies `[]` with `O`.)

get_and_put(Key, ^X, Y)

Fetches an element of which key corresponds to `Key` (specified one), unifies it with `X`, and stores a new element `Y` in 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 key corresponds to `Key` (specified one), unifies it with `X` in a {*Value*} format, and stores a new element `Y` in the same location. If there is no corresponding element, unifies `{}` with `X` and adds `Y`.

do(Stream)

Processes sequentially messages in `Stream` when passing a list of messages into Stream.

Keyed Sorted Bag

Same as Keyed bag, except that the elements are fetched in ascending order of the keys.

Generating Objects .

pool:keyed_sorted_bag(S)

Generates a keyed sorted bag object with the standard comparison routine (`comparator:sort/5`).

pool:keyed_sorted_bag(Comparator, S)

Generates a keyed sorted bag with a specified comparison routine specified as code type data in the first argument. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

Message Protocol Same as Keyed bag, except that the elements are retrieved in the ascending order of keys. If the comparison is stable and there are two or more same keys, the elements are retrieved in the order they were entered.

Keyed Sorted Set

Same as Keyed sorted bag, but it does not allow duplicate keys. If an element with the same key as the existing one is added, the existing element is automatically deleted.

Generating Objects .

pool:keyed_sorted_set(S)

Generates a keyed sorted set object with the standard comparison routine (`comparator:sort/5`).

pool:keyed_sorted_set(Comparator, S)

Generates a keyed sorted set with a specified comparison routine specified as code type data in the first argument. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine
`comparator:sort/5"`.

Message Protocol Same as Keyed set,except that get_all/1 returns the elements in the ascending order of keys.

3.4 Sorting/Merging Filter

PIMOS provides a means for sorting and merging any KL1 data set.

Sorter sorts the order of data sets according to a given comparison mechanism, and merger generates a sorted data string from two sorted data strings. In PIMOS, these programs are provided in the form of filter. These filters take a KL1 data string as input stream, and give the resultant data string as output stream.

[Caution]The merger shown here is different from the built-in predicate merger which merges multiple streams regardless of their contents (see 2.2.12)

3.4.1 Sorting filter

The sorting filter, sorter, takes a KL1 data string, and gives a sorted data string as output stream. Only after the input stream is closed, the sorter starts generating an output stream. If comparison is stable, sorting is also stable.

[Caution]We implement the sorter in a merge/sort fashion.

Sorting filter without key

Normal sorter.

Generating a filter .

sorter:sort(In, ^Out)

Generates a sort filter with the standard comparison routine (`comparator:sort/5`).

sorter:sort(In, Comparator, ^Out)

Generates a sort filter with the comparison routine, by specifying a comparison routine code as the second argument. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

Sorting filter with key

This filter takes KL1 data strings given in a {`Key, Data`} format as input stream, sorts them by a specified key, and returns the result into an output stream. The output stream is also in a {`Key, Data`} format.

Generating filter sorter:key_sort(In, ^Out)

Generates a key_sort filter with the standard comparison routine (`comparator:sort/5`).

sorter:key_sort(In, Comparator, ^Out)

Generates a key_sort filter with the comparison routine by specifying a comparison routine code as the second argument. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

3.4.2 Merging filter

The merging filter, merger, takes two sorted KL1 data strings as input stream, and generates one sorted data strings as output stream. If comparison is stable, merging is also stable.

Merging filter without key

Normal merger.

Generating filter .

sorter:merge(In1, In2, ^Out)

Generates a merge filter with the standard comparison routine (`comparator:sort/5`).

sorter:merge(In1, In2, Comparator, ^Out)

With the comparison routine code specified as the third argument, it generates a merge filter with the comparison routine. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

Merging filter with key

This filter merges the input stream by key, and returns the result to the output stream. The input stream is given in a {`Key, Data`} format.

Generating filter .

sorter:key_merge(In1, In2, ^Out)

Generates a key_merge filter with the standard comparison routine.

sorter:key_merger(In1, In2, Comparator, ^Out)

With the comparison routine code specified as the third argument, it generates a merging filter with the comparison routine. Note that this routine must be declared as *public* and have the same number of arguments and the same contents as those of the standard comparison routine `comparator:sort/5`.

3.5 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 filter: input, output, and I/O, which are placed between the user and the I/O device. The input filter stores a given quantity of input characters, and returns them as required. The output filter receives and stores a certain number of characters, and outputs them as a whole. The I/O filter has both input and output features. These filters are automatically inserted into a request stream when you get a standard I/O stream (see 8.3.2)

The infinite-length buffer receives both input and output messages from the user. It has a 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 the program. They also have an operator table (pool). By referring to this operator pool, they can analyze the syntax of data string expression (for input) and generate string expressions (for output) based on the same operator preference syntax 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).

3.5.1 Common Protocol

The protocol used in common to filter and infinite length buffer is listed below in details.

Description about the end of input or the abort conditions relates to files for which infinite length buffer is free from end of input or abort condition. If getc is sent to an empty infinite length buffer, no unification applies to C until the buffer is emptied 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, and if the device aborts, unifies -2.

getl(^String)

Unifies the next one line with String as 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 16-bit string. If input terminated before `Size` character has not been read, converts the characters so far received into 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 untill the full-stop (a period (.) followed by a blank, tab, or LF), and parses the string according to the operator's instruction. If parsing is successful, Result is unified into `normal(WT)`, where WT is the term wrapped. If the input has come to the end, Result is unified into `normal(end_of_file)`. If the input is only 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 in input token column `TL`.

error({TL,ErrorReason,Ps}) An error in the `Ps`'th location of input token column `TL`. `ErrorReason` is an atom indicating the cause of the error and includes the following:

newline_occurred_in_string LF is present in the string.

newline_occurred_in_quoted_atom LF is present in atoms enclosed with quotation marks.

integer_over_flow Integer operation overflow occurred during read. It occurs when entering integer value constant of the value which cannot be expressed with 32-bit string.

atom_device_abnormal Conversion of atom name and atom failed (probably due to PIMOS bug).

ambiguous({TL,Ps}) The `Ps`'th token in the input token column `TL` is interpreted in several ways.

[Hint1]See section "3.6 Normal term and wrapped term" for details of wrapped term.

[Hint2]By passing `ErrorInfo` to `print_error(ErrorInfo)`, 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) appears, and parses the string according to the operator's instruction. If variable pool `VP` is specified, this variable pool is used for parsing. If no `VP` is specified, empty variable pool is created and used. `NVP` is an updated variable pool. If no `NVP` is specified, 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 each corresponding to `ErrorInfo` of `getwt/1`.

$error({TL,Ps}) .

$error({TL,ErrorReason,Ps}) .

$ambiguous({TL,Ps}) .

skip(C)

Ignores the input until character code `C` appears ignores untill the end of the input. For direct `getc(C)`, the character code next to C is unified. When the input comes to the end while ignoring it, `C` and -1 are unified immediately after getc(C).

ungetc(C)

Rewrites character code `C` to the internal buffer. Immediately following `getc(C)` can be read by using this code. `Ungetc` can be repeated as many times as needed, and 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 (such as window). Otherwise (i.e., connected to a file), 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 character `C`.

putl(String)

Writes the string indicated by `String` and enters LF.

putb(Block)

Writes the character string indicated by `Block`(string).

putwt(WT, PDepth,PLength)

putwt(WT)

Based on the operator pool, writes the wrapped term `WT` using operator order 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`. (If neither `PDepth` nor `PLength` are specified, default value is used.)

[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 or fails.

[Hint]Module type is output in the format of "module*ModuleName*ModuleSize", code type is output in the format "code*ModuleName* : *PredicateName*/*Arity*ModuleSize".

putwtq(WT,PDepth,PLength)

putwtq(WT)

Writes the wrapped `WT` with appropriate quotation marks. By entering a full-stop, the term output with `putwtq` which precedes the full-stop can be read by using `getwt` or `gett`.

putt(T,PDepth,PLength)

putt(T)

Writes the term `T` using the operator order 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 puttq message.

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.

nl

Enters line feed(LF).

tab(N)

Write `N` pieces of blank code.

putf(FormatString, ArgumentList)

Outputs `ArgumentList` according to the format `FormatString`. `FormatString` is a string of formatting characters. `ArgumentList` is a list of items to be outputted. The character ~ introduces a control sequence. The general format of a control sequence is ~*N*`C`, where `C` determined the type of the control sequence and *N* is optional positive integer.

~*N***s** (Character) Interpret an argument as a character code and outputs that character *N* times. *N* defaults to 1.

~*N***t** (String) Outputs *N* characters from the beginning of argument String. *N* defaults to the length of argument String.

~*N***t** (Term) Outputs the term of the argument *N* times in the same format as putt. *N* defaults to 1.

~*N***b** (Blank) Outputs *N* blanks. *N* defaults to 1.

~*N***n** (Newline) Outputs *N* newlines. *N* defaults to 1.

~*N*~ (tilde) Outputs *N* "~" itself. *N* defaults to 1.

print_error(ErrorInfo)

Error information is displayed according to `ErrorInfo`. `ErrorInfo` must be argument of `abnormal(ErrorInfo)` returned upon abnormal termination of getwt/1.

flush(^Status)

Writes the data stored in buffer to 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), atom `normal` is unified with Status. If infinite length buffer is involved, no operation takes place, and promptly unifies atom `normal` with Status. In the filter, I/O data blocking is performed to reduce traffic of communication with I/O device. 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. Buffer is sent to the I/O device if the following condition is true.

- The buffer is full.
- `flush/1`, `putl/1` message is received.
- When the stream is closed.

Operator message protocol

Protocol which operates the operators used for `gett` and `putt`. (See 3.5.7 for details.)

add_op(Prec, Type, OpName)

Adds priority `Prec`, operator `OpName` (atom) of type `Type` to the operator pool.

remove_op(Prec, Type, OpName)

Removes priority `Prec`, operator `OpName` (atom) of type `Type` from the operator pool. If it is not installed, no operation takes place.

operator(OpName, ^Def)

Unifies `Def` a list of a pair of priority and type of the operator `OpName`. Format of the pair is (*Priority, Type*). If `OpName` is not registered, `Def` is unified with `[]`.

[Caution]The format of information unified with `Def` is likely modified in the future.

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 (the stream 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 word wrapped), it is necessary to guarantee continuous processing by using do message.

[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 depth and length of structure, string length, and default value of atom name length limit upon structure output to `PDepth`, `PLength`, `PSLength` and `PALength`. The initial value of the limit value is structure depth 4, length 10, string length 20, and atom length 30.

print_depth(^OPDepth,NPDepth)

print_length(^OPLength,NPLlength)

print_string_length(^OPSLength,NPSLength)

print_atom_length(^OPALength,NPALength)

Changes the depth and length of the structure, string length, the default value of atom name limit upon structure output from `OPDepth`, `OPLength`, `OPSLength` and `OPALength` to `NPDepth`, `NPLength`, `NPSLength`, and `NPALength`.

Abort/attention message protocol

This protocol is the one described in Device Abort/Attention Processing.

reset(^Result)

resend(ID, ^Result)

cancel(ID, ^Result)

next_attention(^Result)

3.5.2 Input filter

Creation

buffer:input_filter(Size, In, ^Out)

Creates the input filter with the buffer size of `Size`. The filter reads the command message from `In` (user side) and sends the command message to `Out` (device side).

buffer:input_filter(In, ^Out)

Creates the input filter with a buffer of default size.

User protocol

Accepts messages for the above input, operator pool, control and abort/attention.

[Caution]Messages with unknown protocol is passed to the device as is.

Device protocol

getb(Size, ^Status)

prompt(String, ^Status)

3.5.3 Output filter

Creation

buffer:output_filter(Size, In, ^Out)

Creates the output filter with a buffer size of `Size`. The filter reads the command message from `In` (user) and sends the command message to `Out` (device).

buffer:output_filter(In, ^Out)

Creates the output filter with a buffer in default size.

User protocol

Accepts messages for the above output, operator pool, control, and abort/attention.

[Caution]Messages with unknown protocol is passed to the device as is.

3.5.4 Device protocol

putb(String, ^Status)

flush(^Status)

3.5.5 Input/Output filter

Creation

buffer:interaction_filter(Size, In, ^Out)

Creates the input/output filter with a buffer in size `Size`. The filter reads the command message from `In` (user) and sends the command message to `Out` (device).

buffer:interaction_filter(In, ^Out)

Creates the input/output filter with a buffer in default size.

User protocol

Accepts messages for the above output and input, operator pool, control, and abort/attention.

[Caution]Messages with unknown protocol is passed to the device as is.

Device protocol

getb(Size, ^Status)

prompt(String, ^Status)

putb(String, ^Status)

flush(^Status)

3.5.6 Infinite length buffer

Creation

buffer:character_pool(In)

Creates an infinite length buffer. Write to and read from the buffer is performed using the command message from `In`.

Protocol

Accepts the messages for the above input, output, and control.

[Caution]Messages with unknown protocol will fail.

3.5.7 Operator pool

The operator pool 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 integer in a range between 1 and 1200 each indicating the strength of connection (the smaller the number, the stronger the strength of connection).

[Caution]The same atom can be registered as multiple types of operator.

Creation

operator:pool(In)

Creates the operator pool having standard registered operators. Table 3.1 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.

3.5.8 Variable pool

The variable pool lists variable names (string) and their addresses corresponding to each other.

Table 3.1: Standard Operators

Precedence	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	`#`
80	fx	`module`

Creation

variable:pool2(In)

Creates an empty variable table.

Protocol

count(^N)

Unified the number of currently registered variable names with `N`.

unify

Unifies the variables registered with the same name.

reset

Unifies the variables registered with the same name and initializes the variable table.

[Hint]The variable pool sends `reset` to itself as termination process.

put_var(Name, V)

Makes an entry of a variable `V` with its name `Name`.

get_name(PE,ADR,^Result)

Unifies `Result` into {`N`,`Name`}. `N` 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.

[Caution]The processor and address with 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 the different variables to be the same and the same variable to be the different.

3.6 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 base term. Normal terms are those usually handled in KL1. 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)})`. Wrapped terms 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(VarName)` : VarName is (normal) string.

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.

Wrap

Wrap is a predicate which converts a normal term into 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 a term wrappend term of `T`.

If variable pool `VP` is specified, it is used, and if not, 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 variable. The variable is wrapped in the form of `variable({PE,ADR})`, where `PE`, `ADR` is the processor number and address containing the variable.

[Hint]This is useful when address display of variable is required for debugging.

[Example]Address display of variables 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 ground term at a certain time. So that the result of the wrap for the same 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.

3.6.1 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 the wrapped terms. `Result` is unified with {`NewWT,T`}, where `NewWT` is implemented in `WT` and `T` has its unwrapped term unified. If variable pool `VP` is specified, it is used, and if not, 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.

Chapter 4

Acquiring Resources

Resources provided by which perform input, output or other services are called "OS defined resources". (On the other hand, resources such as computation time or memory area required for the execution of KL1 programs are called "language-defined resources".) Services provided by the OS definition resources can be used by sending messages to the request stream for the resource. This section describes how to get such request streams.

4.1 Resource Type

The types of the PIMOS definition resources are classified as follows:

General request device: This processes creation request messages for various request devices which, in turn, are devices fpr creating tasks or devices.

- general_request_device

Request devices: This processes creation request messages for devices and provides request streams for them.

- window_request_device
- file_request_device
- timer_request_device
- string_io_request_device
- atom_request_device
- module_request_device

Devices: This performs input, output, or other services on request messages from users.

- window_device
- file_device
- time_device
- string_io_device
- atom_device
- module_device

Task: This processes request messages for tasks.

- task

4.2 Task

A task, which is similar to the shoen, is a unit of execution control for KL1 programs and is a unit of management of OS-defined resources.

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.

See 6 for detailed information on tasks.

4.3 Resource Tree

The PIMOS manages all OS defined resources including tasks under one tree structure.

When a task acquires OS-defined resources, they are child resources of the task. As tasks created by a task are also child resources, resources generally form a tree structure.

All child resources of one task get different identifier (child resource identifier; positive integer). The same identifiers as those of released resources are not reused for child resources under the same task. Accordingly all 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:** root of resource tree, or top level task.
- **$n (n \geq 1)$:** direct child resource with identifier n.
- **-1:** resource itself.
- **$-n (n \geq 1)$:** ancestor resource of n-1 generations.
- *** :** all direct child resources.

*** :** all of its own child resources, that is, all of `[*]`,`[*,*]`,`[*,*,*]`,...

[Caution]Currently, tasks are the resources which can have child resources. This may be changed with future extensions.

4.4 Environment Variable

In the PIMOS, each task has an environment variable pool. An environment variable pool is a information reference table for executing programs.

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 environment variables, task:fep and task:node. (That is to say, the top task sets these two environment variables.)

Various utilities supplied by the PIMOS changes its functions according to the values of environment variables. Required functions of utilities invoked for each task can be obtained by properly setting values of environment variables.

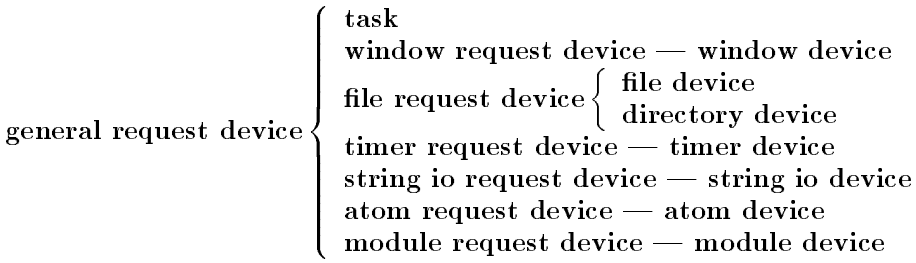


Figure 4.1: Resource Hierarchy

4.5 Hierarchy of Resource Acquisition

To use resources supplied by the PIMOS, general_request_device should be first acquired and subsequent resource acquisition should be in a hierarchical way. For example, to use a window, get window_request_ device by sending a message to general_request_device and then get necessary window_device by sending a message to window_request_device (a window is actually created then). Input to and from the window can be made by sending messages to this window_device.

The resource hierarchy supplied by the PIMOS is shown in 4.1 How to get request streams to 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.

4.6 General Request Device

4.6.1 How to Get Request Device

shoen:raise(pimos_tag#task, general_request, GeneralRequestDevice)

This obtains request streams for General request device into GeneralRequetsDevice.

[Caution]The following declaration should be issued at the top of the module which includes invocations of the above:

```
:- with_macro pimos
```

4.6.2 Protocol

window(^Status)

file(^Status)

timer(^Status)

string_io(^Status)

atom(^Status)

module(^Status)

Corresponding request devices are acquired. Corresponding FEP is determined by the default value (a value of the environment variable task:node in the task). If the acquisition is successfully completed, **normal(Request, Abort, Attention)** is returned to the Status. If the acquisition is abnormally completed, abnormal is returned to the Status. The Request is the request stream for the request device. See 5 for Abort and Attention.

window(FepNumber, ^Status)

file(FepNumber, ^Status)

timer(FepNumber, ^Status)

string_io(FepNumber, ^Status)

Various request devices are acquired by specifying the FEP number.

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, ^Status)

A task is produced. Arguments are shown below. Many of those arguments are the same as those specified in creating a shoen.

Code: Predicate code for the top level goal of 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 (Integer value, see 2.3.7)

Status: Task creating status. Either of the following values is returned:

normal(^Control,Report): Task was successfully created. The Control argument is the control stream of the task and the Report argument is the report stream of the task.

abnormal: Task was abnormally completed.

[Caution]The initial execution status of the task is resource_low (computation resource is 0).

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, InitialResoc, ^Status)

This creates tasks. The arguments are the same as those of the above execute/7 except Initial-Resoc:.

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 computation resources of the shoen and LowResoc is the lower 32 bits. (refshoen)

suicide(^Status)

This forces termination of a task which has this general request stream; 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 Status argument.

[Hint]If a task is successfully completed, the task usually ends before examining this Status.

abort(ResourceId, ^Status)

This forces termination of a task specified in ResourceId. Note that ancestral tasks cannot be terminated.

ResourceId: Resource identifier

Status: Processing results, normal(successfully completed) or abnormal(abnormally completed)

number_of_feps(^Status)

This gets the number of FEPs currently connected. As a processing result, either of the following is returned to the Status:

normal(Feps): Successfully completed. This returns the number of FEPs to the argument.

abnormal: Abnormally completed.

setenv(Name, Value, ^Status)

This sets value to an environment variable.

Name: Environment variable name. This should be a ground value.

Value: A value. This should be ground value.

Status: Processing result, normal(successfully completed) or abnormal(abnormally completed)

getenv(Name, ^Status)

This gets an environment variable value.

Name: Environment variable name. This should be a ground value.

Status: Processing result

normal(Value): Successfully completed. This returns the value to the argument.

abnormal: There are no environment variables specified.

unsetenv(Name, ^Status)

This deletes an environment variable value.

Name: Environment variable name. This should be a ground value.

Status: Processing result

normal(Value): Successfully completed. This result is also returned when no environment variable exists.

abnormal: Abnormally completed.

getenv(^Status)

This returns all environment variables and their values. As a processing result, either of the following is returned to the Status:

normal(NameValueList): Successfully completed. Null list is returned to NameValueList if no environment variable exists. A list of vectors in a form of {**Name, Value**} is returned if some environment variables exist. Name is the name of an environment variable and Value is its value.

abnormal: Abnormally completed.

id(^Status)

This inquires about the resource identifier of the task. As a processing result, either of the following is returned to the Status:

normal(ResourceId): Successfully completed. The child resource identifier (positive integer, the child resource identifier in the parent task) is returned to ResourceId.

abnormal: Abnormally completed.

absolute_id(^Status)

This inquires about the absolute resource identifier of the task. As a processing result, either of the following is returned to the Status:

normal(AbsoluteResourceId): Successfully completed. A list of child resource identifiers in each generation from the top task along the resource hierarchy is returned to AbsoluteResourceId.

abnormal: Abnormally completed.

status(ResourceIdList, ResourceType, Verbosity, ^Status)

This inquires about task status. Arguments are as follows:

ResourceIdList: The tasks to make the inquiry.. This is specified by a relative resource identifier.

ResourceType: Resource type to be inquired. Types which can be specified are *(all resources) or task (tasks only).

Verbosity: Degree of detailed information. Detail degrees which can be specified depend on each object resource type to be inquired. Currently, detail can be specified for a task and only brief can be specified for *.

Status: Result of inquiry, either of the following:

normal(Info): Successfully completed. A list of vectors which represent the status of resource(s) is returned to the Info argument.

abnormal: Abnormally completed.

The form of vectors which represent the status of resources depends on resource types.

- When the resource type is task and detail degree for information is detail:

  {Id, Name, TaskStatus, Limit, Consume, Children}

  Each elements is defined as follows:

  Id: Relative resource identifier.

  [Caution]A bug exists here in the current version.

  Name: Task name (string specified on creation)

  TaskStatus: Status of task. A vector in the form of **{Cond, Stops}** is returned. When status is unknown (for example, when the parent shoen is stopped), unknown is returned. Cond represents the execution of a task and is one of the following:

  normal: Normal status (which is, not in resource shortage status).

  resourcelow: Resource shortage status.

  terminating: During termination processing

  shut_terminating: During termination processing by 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 stop(Reason) form, where Reason is an atom which represents a stop reason.

  Limit: Upper limit value of language-defined resource consumption.

  Consume: Current consumption of language-defined resources. (-1 represents unknown status.)

  Children: A list of information vector for all child tasks when child task information is acquired.

- When the object resource type is * and detail degree for information is brief:

  {Id, Name, Children}

  Each elements is defined as follows:

Id: Child resource identifier.

[Caution]A bug exists here in the current version.

Name: Task name

Children: A list of information vector for all child resources when child resource information is acquired.

[Caution]Other messages for the second and third arguments will be provided in the future.

Chapter 5

I/O Control

PIMOS provides four types of I/O devices, including window, file, timer, string I/O. Interface with each device is composed of message communication through the stream. Users direct the I/O operation by sending the messages to the stream for each device.

5.1 Communication Method of I/O Control

5.1.1 I/O Messages

I/O operation is performed by sending the I/O messages to each device.

A message is composed of the massage name, zero or more arguments for input, one argument for output. Each device performs the I/O operation according to the name of the message received and the arguments for input, and returns the results to the argument for output.

If the operation corresponding to the message terminates normally, Atom normal, or the vector with more than one elements whose first element is atom normal is returned to the argument for output, according to the type of message. When the vector is returned, the subsequent elements are the results obtained by each massage.

If the message does not terminate normally, the reason for not terminating normally is returned to the argument for output. In the subsequent description which is otherwise stated, atom abnormal is returned to the argument for output if the message processing fails, or atom aborted is returned if the processing is aborted.

In the subsequent description, the argument for output is represented by ^**Result** if the operation terminates normally and the vector is returned, or otherwise by ^**Status**.

Abort/Attention Line

Devices require the mechanism which not only processes the normal messages, but also suspends the operation during the message processing due to the request from the OS or the interruption by the user.

Devices normally processes the messages received through the stream in the order of receiving them. It is required to notify the device, prior to the messages waiting for being processed in the stream, of suspending the messages being processed, so this stream cannot be used for notifying the interruption.

Thus, all devices 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 for the user or the OS to direct the suspension of the processing to the device, while an attention line is used to notify the user or the OS of the device being interrupted.

5.2 Device Abort/Attention Processing

Every device has the abort line for receiving the pending request from the OS, and the attention line for notifying the OS of the interruption from the user, in addition to the stream for sending

the normal I/O messages. These stream and lines are establishid at the time of the creation of devices.

These two lines are implemented by the variables. The OS or the user can suspend the messages being processed by the device, by instantiating the variable of the abort line to atom abort. The vaviable of the attention line is instantiated to atom attention by the device, when the user enters ^C with the keyboard, to notify the OS or the user of the interruption (only window device is available at this time).

Devices have the following three states:

- Being aborted
- Waiting for the normal messages
- Waiting for the completion of the processing in response to the messages

Because an abort line and an attention line is implemented by the variables, the line which is used once cannot be reused. So the device, after atom abort is sent, becomes the state of not being established with the abrot line. This state is called the device being aborted. All the messages suspended by this state and the normal messages received in this state are stored in the device without being executed. It is required to establish the new abort line and attention line to reset the device to the normal state of waiting for the messages. Reset the device and establishing the lines is performed by sending the following message via the stream.

reset(^Result)

Establishes the new abort line and attention line.

Establishes the line **Abort**, **Attention** when the message terminates normally, and returns the identificaton number **ID** of the pending message sequence to **Result** in the form of **normal(Abort, ^Attention, ^ID)**. The identification number **ID** represents the pending message sequence, after the device being aborted and immediately before the arrival of this message, which are stored in the device without being executed due to the device state of being aborted.

When reset message is received during the line being established, the device assumes that the pending request has arrived so that it instantiates the existing abort line to atom abort, and establishes the new line.

When the device becomes the state of being aborted, the subsequent message sequence, before the above reset is received, are stored in the device without being executed. After the device becomes the state of waiting the normal message, you can direct to resend or cancel to the pending message sequence by the following messages. Resend or cancel can be directed as a unit of the time period of the device being aborted.

If no messages are sent, the messages being stored are cancelled at the time of the termination of the device.

resend(ID, ^Status)

Resends the pending message sequence of the identification number **ID**.

cancel(ID, ^Status)

Cancels the processing of the identification number **ID** pending message sequence. In this time, the argument for output of the messages which are cancelled is instantiated with atom aborted.

If the interruption is informed by the user, but there is no necessity to suspend the processing, only the attention line can be newly established with the following message.

next_attention(^Result)

Establishes the new attention line.

If the message terminates normally, **normal(^Attention)** is returned to **Result** and the new attention line **Attention** is established.

When next_attention message is sent if the attention line has already been established, the attention line is newly established after the the success of the message. The old lins is instantiated to atom `[]`. If the message fails, the old line is available.

5.2.1 Window Device Abort Processing

An window device controls the string (the input remainder string) which has been entered from the window, but has not been returned to the user (see the getb from the window). Therefore, the messages about the abort to the window devices perform the following processings to the input remainder string, in addtion to the above actions.

reset(^Result)

Initializes the input remainder string to null string. The identification number **ID** returned as a result of this processing represents both of the pending message sequence and the input remainder string controlled by the window device at the time of being aborted. If this **ID** is spedified to resend, cancel message, not only the pending message sequence but also the input remainder string are recoverd and released. If no messages are sent, the input remainder string being stored are released at the time of the termination of the window device.

resend(ID, ^Status)

Recovers the input remainder string of the identification number **ID**. If the input remainder string already exists, this string is aborted and the input remainder string of the identification number **ID** is applied. If the incorrect identification number is specified, however, the existing input remainder string becomes available.

cancel(ID, ^Status)

Releases the input remainder string of the identification number **ID**.

5.3 Path Name

Is is required to have the pathnames for identifying files and directories in order to obtain the file devices and the directory devices. You can use two types of pathnames for this specification:

- Absolute pathname
- Relative pathname

PIMOS, similar to SIMPOS, uses the directories of the tree structure. The absolute pathname specification is the method of specifying the entire pathname from the root directory on the FEP specified when obtaining the file request devices, while the relative pathname specification is the method of specifying the pathname from the directory ahead specified when robtaining the file request devices. The directory used for the relative pathname 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 the file request device.
- After the 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 pathname and the relative pathname are treated as follows:

- If the specified pathname begins with '>', it is regarded as an absolute pathname.
- Otherwise, the specified pathname is regarded as a relative pathname. The pathname which is obtained by appending the specified pathname string to the value of the current directory, is used. When the current directory does not end with '>', the '>' is supplied before appending.

[Caution]SIMPOS logical names (me:, etc) and SIMPOS remote files are not supported in PIMOS so that the programs using these functions may not operate in the future.

5.4 Request Devices

Request devices provides the physical devices such as windows, files. PIMOS has the window request device, the file request device, the timer request device, and the string I/O request device by the type of device.

Each physical device can be generated and the stream to it can be obtained by sending the messages to each request device via the streams (see 4.6.1 for obtaining the stream to each request device). Each request device is provided with abort line and attention line, by which the abort/attention processing can be performed (see 5.2).

5.4.1 Window Request Device

Generates and provides the window device. Also, it provides with the additional information about the window generation. Window request devices have the following message protocols:

create(Buffer_name, ^Result)

Generates the window device and establishes the abort line, attention line. Specifies the name of the text buffer which is assigined when generating the window as **Buffer_name** (string).

When the message terminates normally, it returns **normal(Window, Abort, ^Attention)** to **Result**. **Window** represents the stream to the window device, and **Abort** and **Attention** represents the abort line and attention line, respectively.

If the specified buffer does not exist, the new buffer with this name is generated and assinged to the window.

create(^Result)

Generates the window device and establishes the abort line, attention line. The new text buffer with the default name is generated and assingned when generating the window.

If the message is executed correctly, **normal(Window, Abort, ^Attention)** is returned to **Result**.

get_max_size(X, Y, Font_pathname, ^Result)

When specifies the font with the pathname **Font_pathname** (string) and specifies the coordinates **X** and **Y** on the screen by the number of dots, the maximum size of the window, which can be generated assuming that the upper-left- most corner of the window is placed at this coordinates, is returned.

If the message is executed correctly, **normal(^Characters, ^Lines)** is returned to **Result**. **Characters** and **Lines** represents the number of characters and lines in the specified font, respectively.

5.4.2 File Request Device

Generates and provides the file device and the directory device. File request devices have the following message protocols:

open(Pathname, read(^Result))

Generates the file device specified in **Pathname** (string) in the input mode, and establishes the abort line, attention line. If the message terminates normally, **normal(File, Abort, ^Attention)** is returned to **Result**. **File** represents the stream to the file device generated, and **Abort** and **Attention** represent the abort line and the attention line, respectively.

If the specified file does not exist, abnormal is returned to **Result**.

open(Pathname, write(^Result))

Generates the file device specified in **Pathname** (string) in the output mode, and establishes the abort line, attention line.

If the message terminates normally, **normal(File, Abort, ^Attention)** is returned to **Result**.

If the specified file does not exist, the new file is created and opened. If the existing file is specified, the file with the same name is created and opened in higher version.

open(Pathname, append(^Result)

Generates the file device specified in **Pathname** (string) in the append mode, and establishes the abort line, attention line.

If the message terminates normally, **normal(File, Abort, ^Attention)** is returned to **Result**.

If the specified file does not exist, the new file is created and opened. If the existing file is specified, the file itself is opened.

directory(Pathname, ^Result)

Generates the directory device specifed in **Pathname** (string), and establishes the abort line, attention line.

If the message terminates normally, **normal(Directory, Abort, ^Attention)** is returned to **Result**.

If the specified file does not exist, abnormal is returned to **Result**.

set_current_directory(Pathname, ^Result)

Sets the pathname specified in **Pathname** (string) to the current directory. If the message terminates normally, **normal** is returned to **Result**.

If the message terminates abnormally, abnormal is returned to **Result**.

current_directory(^Result)

Returns the value (string) of the current directory. If the message terminates normally, **normal(Pathname)** is returned to **Result**.

If the message terminates abnormally, abnormal is returned to **Result**.

5.4.3 Timer Request Device

Generates and provides the timer device. Timer request devices have the following message protocols:

create(^Result)

Generates the timer device and establishes the abort line, attention line.

If the message terminates normally, **normal(Timer, Abort, ^Attention)** is returned to Result. **Timer** represents the stream to the timer device generated, and **Abort** and **Attention** represent the abort line and the attention line, respectively.

5.4.4 String I/O Request Device

Generates and provides the string I/O device. String I/O device is the device for operating the device on the PIMOS, which is defined on the FEP written in ESP by the user (see Appendix A for defining the device written in ESP). String I/O request devices have the following message protocols:

create(Device_Name, ^Result)

Generates the string I/O device specified in **Device_Name** (string) and establishes the abort line, attention line.

If the message terminates normally, **normal(StringIO, Abort, ^Attention)** is returned to **Result**. **StringIO** represents the stream to the string I/O device generated, and **Abort** and **Attention** represent the abort line and the attention line, respectively.

If the specified string I/O device does not exist, abnormal is returned to **Result**.

5.5 Devices

Each device is generated and provided by the request device corresponding to the type of device. PIMOS has the four types of devices, including window, file, timer, and string I/O.

Each device has physical I/O resources and performs the operations such as I/O through the message communication via the stream. Each device has an abort line, attention line, through which the abort/attention processing can be performed (see 5.2).

Message protocols for the devices are described in the following:

5.5.1 Window

I/O messages

getb(Size, ^Result)

Inputs the strings as far as the carriage return code. The maximum length of string to be input, **Size**, must be specified. When the length of 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, and returns as a result the partial string of specified size length from among the input remainder strings, without sending the input request to the window, if the input remainder strings already exist when executing getb.

If the message terminates normally, **normal(^String)** is returned to **Result**. **String** represents the string which is returned complying with the above specifications.

[Caution]Abort and input remainder string: The input remainder string is treated as the same as the pending 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 pending message sequence and the input remainder string (see 5.2).

[Hint]When you input the one line from the window, it is originally desirable to return the string as far as the carriage return code however that line is long. PIMOS uses this form to make the message protocols for input same as the files and so on.

putb(String, ^Status)

Outputs the string **String** on the window. There are no limitations on the length of strings.

prompt(String, ^Status)

Outputs the string **String** on the window. There are no limitations on the length of strings.

[Hint]prompt/2 is used for displaying the prompt for input by the user. Although putb/2 is, originally, sufficient for displaying the prompt on the window, another one is provided with to describe the program without depending to the types of I/O device. Prompt/2 operates as the same way as putb/2 if it is issued to the window, while prompt/2 does nothing if it is issued to the file.

flush(^Status)

Immediately outputs the contents of the recent buffer on the window.

beep(^Status)

Beeps out.

Messages for setting the window attributes

set_size(Size, ^Status)

Sets the size of the window.

If **Size** is specified in the form of **char(Characters, Lines)**, the number of characters **Characters** and the number of lines **Lines** are set. If **Size** is specified as **mouse**, the size is set with the mouse.

set_position(Position, ^Status)

Sets the location of the window on the screen.

If **Position** is set in the form of **at(X,Y)**, the location of the upper-left-most corner of the window is set with the dot coordinate **(X, Y)**. If Position is set to **mouse**, the location is set with the mouse.

reshape(Position, Size, ^Status)

Sets the location and size of the window on the screen.

If **Position** is set in the form of **at(X,Y)**, the location of the upper-left-most corner of the window is set to the dot coordinate **(X, Y)**. If **Position** is set to **mouse**, the location is set with the mouse.

If **Size** is set in the form of **char(Characters, Lines)**, the number of characters **Characters** and the number of lines **Lines** are set. If **Size** is set to **mouse**, the size is set with the mouse.

set_title(String, ^Status)

Sets the title to **String** (string).

set_font(Pahtname, ^Status)

Sets the font with the pathname **Pathname** (string).

select_buffer(Buffer_name, ^Status)

Sets the text buffer with the name **Buffer_name** (string). If the specified buffer does not exist, the new buffer is created with the same name and is assigned to the window.

activate(^Status)

Displays the window on the screeen.

deactivate(^Status)

Stops displaying the window on the screen. The screen image of the window is maintained and can be displayed again on the screeen by sending the message activate(^Status).

show(^Status)

Makes the window not hidden by other windows. If the window is not displayed on the screen (in the state of deactivate), however, it operates as the same way as the message activate(^Status).

hide(^Status)

Moves the window at the lowest layer of the other windows.

clear(^Status)

Clears the contents of the text buffer of the window.

Messages for quering the window attributes

get_char_size(^Result)

Returns the window size indicated by the number of characters and lines.

If the message terminates normally, **normal(^Characters, ^Lines)** is returned to **Result**. **Characters** and **Lines** represent the number of characters and lines, repectively.

get_position(^Result)

Returns the location of the upper-left-most corner of the window with the numbers of dots on the screen.

If the message terminates normally, **normal(^X, ^Y)** is returned to **Result**. **X** and **Y** represent the dot coordinate (X, Y) on the screen.

get_title(^Result)

Returns the title.

If the message terminates normally, **normal(^String)** is returned to **Result**. **String** represents the title (string).

get_font(^Result)

Returns the font indicated by the pathname.

If the message terminates normally, **normal(^Pathname)** is returned to **Result**. **Pathname** represents the pathname (string) of the font.

get_buffer_name(^Result)

Returns the name of the text buffer specified in the window.

If the message terminates normally, **normal(^Buffer_name)** is returned to **Result**. **Buffer_name** represents the buffer name (string).

5.5.2 File

Messages common to all files

pathname(^Result)

Returns the full pathname of the file.

If the message terminates normally, **normal(^Full_pathname)** is returned to **Result**. **Full_pathname** represents the full pathname (string).

Messages to input files

getb(Size, ^Result)

Inputs the string of size length **Size** from the file.

If the message terminates normally, **normal(^String)** is returned to **Result**. **String** represents the input string.

Returns the part of data 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 to **Result**, and in the subsequent inputs, abnormal is returned.

[Caution]The total length of the string, now, must be 1960 double bytes or less.

end_of_file(^Result)

Querys whether the input is completed or not.

If the message terminates normally, **normal(^Ans)** is returned to **Result**. **Ans** is instantiated with yes if the input is completed, or it is instantiated with no if the input is not completed.

Messages to output files and append files

putb(String, ^Status)

Outputs the string **String** to the file.

[Caution]The total length of the string, now, must be 1960 double bytes or less.

prompt(String, ^Status)

Nothing is done.

flush(^Status)

Immediately outputs the contents of the recent buffer to the file.

Messages to directories

pathname(^Result)

Returns the full pathname of the directory.

If the message terminates normally, **normal(^Full_pathname)** is returned to **Result**. **Full_pathname** represents the full pathname (string).

listing(Wildcard, ^Result)

Returnes the names of the files which conforms to the wildcard **Wildcard** (string).

If the message terminates normally, **normal(^Filename)** is returned to **Result**. **Filename** represents the list of file names.

If such files do not exist, or if the wildcard is not specified correctly, abnormal is returned to **Result**.

[Caution]You cannot specify the wildcard including the directory name (and so on).

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 to **Status**.

undelete(Wildcard, ^Status)

Recovers the files deleted whose names conform to the wildcard **Wildcard** (string).

If such files do not exist, normal is returned also to **Status**. If the wildcard is not specified correctly, abnormal is returned.

deleted(Wildcard, ^Result)

Returns the names of the files that has been deleted, conforming to the wildcard **Wildcard** (string).

If the message terminates normally, **normal(^Filename)** is returned to **Result**. **Filename** represents the list of file names.

If such files do not exist, or if the wildcard is not specified correctly, abnormal is returned to **Result**.

purge(Wildcard, ^Status)

Purges all the files except for the files with the maximum version number, whose names conform to the wildcard **Wildcard** (string). The part of version number cannot be specified in **Wildcard**. The files being deleted at the time of this message received are purged.

If such files do not exist, normal is also returned to **Status**. If the wildcard is not specified correctly, abnormal is returned.

expunge(^Status)

Expunges all the files being deleted.

5.5.3 Timer

get_count(^Result)

Returns the counts of miliseconds from 0:0 to the current time.

If the message terminates normally, **normal(^Count)** is returned. **Count** represents the integer number indicating the counts of miliseconds.

on_at(Count, ^Result)

Specifies the time **Count** (integer number) in miliseconds, and informs of it when it is the appropriate time.

If the message terminates normally, **normal(^Now)** is returned to **Result**. **Now** is instantiated with wake_up when it is the specified time.

on_after(Count, ^Result)

Specifies the time period **Count** (integer number) in miliseconds, and informs of it after the appropriate time period.

If the message terminates normally, **normal(^Now)** is returned to Result. **Now** is instantiated with wake_up after the specified time period.

5.5.4 String I/O

io(Command, ToFepString, ^Result)

Performs the I/O instructions specified in **Command** (integer number), based on the data specified in **ToFepString** (string).

If the message terminates normally, **normal(^FromFepString)** is returned to **Result**. **FromFepString** (string) represents the results.

If the specified I/O instruction does not exist, or if it does not terminate normally as a result of execution, abnormal is returned to **Result**.

Chapter 6

Controlling Tasks

A task is the unit management of program execution in PIMOS.

Because tasks operate actively and require asynchronous communication, unlike other PIMOS-defined resources, they have two streams of control/report. A control stream is the message stream from the user to the task, and a report stream is the one from the task to the user.

6.1 Task Control Stream

stop

Stops the task. The task being stopped can be reactivated by the variables (stated below) in the corresponding report stream message.

allow_resource_report

Allows the resource low report. Once resource low is reported it will not reported 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, ^Status)

Aborts the task specified in ResourceId. The task is specified in the form of the relative resource identifier from the direct controlling task. The higher tasks, however, cannot be aborted.

Path: resource identifier

Status: processing results

normal: normal termination

abnormal: abnormal termination

abort(^Status)

Aborts the direct controlling task. Same as `abort([],^Status)`. The argument Status is set to one of the following values:

normal: normal termination

abnormal: abnormal termination

add_resource(HighResoc, LowResoc)

Adds to the upper limits of the amount of language- defined resources being consumed in the task. HighResoc represents the upper 32 bits of the initial computer resources in the shoen and LowResoc represents the lower 32 bits (see 2.3 for details about the computer resources).

id(^Status)

absolute_id(^Status)

status(ResourceIdList, ResourceType, Verbosity, ^Status)

Same as the processing of the messages to the general request device, except that the target task becomes the controlled task (see 4.6.2).

6.2 Task Report Stream

statistics_started

resource_report_allowed

resource_added

resource_low

aborted

terminated

exception(Proc, ExcCode, ExcInfo, ^NewCode, ^NewArgv)

statistics(Resource)

The specifications for all the streams above conform to those of the report stream of shoen (see 2.3.4).

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 to ReasonAtom. This is returned as a stop reason when querying the state of the task. CancelVar can be instantiated by any data to release the stop reason.

Chapter 7

PROGRAM INFORMATION MANAGEMENT

This chapter explains how to access the program information (atoms, modules) managed by PIMOS and how to compile the user program.

7.1 Program Information Databases

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 I/O device, because these databases are managed as the resources in PIMOS. But there are some utility programs to access to the databases, you do not have to care how to get these resources. The remaining part of this chapter 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 to Object.

abnormal: Abnormal end

non_existent(Info): No corresponding object exists.

7.1.1 Atom Cache Database

The atom cache database is a utility program to access the database which contains the associations between atom and its print name.

Since PIMOS manages these associations in a unique database, this leads to congestion in access, causing a bottleneck of efficiency. If there are many accesses to the database, it is better to create a database cache. This program creates such kind of database cache.

Creation

atom_table:cache_table(AtomDev)

Create an atom database cache process, and returns the stream- to-cache to the argument AtomDev.

Protocol

get_atom(AtomName, ^Status)

Returns the atom for the atom name specified at the argument AtomName, which is a 16bit string, to the Object in the argument Status.

get_atom_name(Atom, ^Status)

Returns the print name (16bit string) associated with the atom specified at the argument Atom, to the Object in the argument Status.

intern(AtomName, ^Status)

Makes an atom that has the print name specified in AtomName (16bit string), and returns it to Object in Status. If this atom has already been registered then the associated atom returns.

Access by predicate

The following predicates are available for a single access try:

atom_table:get_atom(AtomName, ^Status)

Returns the atom that has the print name specified in AtomName (16bit string), to Object in Status.

atom_table:get_atom_name(Atom, ^Status)

Returns the print name (16bit string) associated with Atom (atom), to Object in Status.

atom_table:intern(AtomName, ^Status)

Makes an atom that has the print name specified in AtomName, and returns it to 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 big decrease of efficiency.

7.1.2 Module Cache Database

The module cache database is a utility program to access the database which contains the associations between module and its module name atom. Module is a KL1 object and the entity of executable codes.

Creation

module_table:cache_table(Stream)

Makes a module database cache process, then returns the stream- to-cache to the argument Stream.

Protocol

get_module(Name, ^Status)

Returns the module which has the name specified in the argument Name (atom), to Object in Status.

get_code(Name, Predicate, Arity, ^Status)

Returns the code specified as module name atom (argument Name), predicate name atom (argument Predicate), and the number of arguments (argument Arity), to Object in Status. If the module exists, but the corresponding predicates does not exists, a term code(Name, Predicate, Arity) returns.

update_module(Name, Module, ^Status)

Updates the module specified in Name (atom) to the new module specified in the argument Module. If the module specified in Name does not exist, the new module is only added to the database. If an attempt is made to update a protected module (for example, the module constitutes PIMOS), protection_violation returns to Status.

[Caution]Updating the module database does not always result in the update of the module itself. If a module would have already been extracted from the database, the execution of the module might not be changed. This is because the inter-module references are not affected by the update of the database (See also relinker). Also the module under execution is not changed (See also dynamic_link).

remove_module(Name, ^Status)

Removes the module specified by the argument Name (atom) from the database. If an attempt is made to remove a protected module, protection_violation returns to Status.

[Caution]The removal from the database means that the module is just out of the module management. Therefore, the removed module can not be accessed through the database, but the related inter- module references are in active (See also relinker). This removal does not affect the module under execution (See also dynamic_link).

get_all_modules(^Status)

Returns all modules that are currently under management. Information returning to the argument Status is a list of two-element vector containing a pair of module name atom and the associated module object.

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 (also see relinker).

When a module is frequently updated (for example, under program development) and some overhead in execution is incurred, dynamic linkage to the external module may cause greater advantage. In such a case, it is recommended to use the following predicate, instead of static linkage;

module_table:dynamic_link(Name, Predicate, Args)

Calls the predicate specified as module name atom (argument Name) and predicate name atom (argument Predicate) with the argument specified as the argument vector (argument Args) using built-in predicate apply.

7.2 Self-compiler

Self-compiler is a program that compiles, assembles, and links KL1 program. Since self-compiler uses standard I/O, it must be invoked from the PIMOS shell. The source program is read from the file specified. If the whole sequence of compilation succeeds, it automatically generates the module and register it into the module database (also see relinker).

7.2.1 Compiler Directive Predicate

The self-compiler supports the following compiler directive predicates:

:- module ¡mdoule_name¿. Indicates the start of module definition. The ¡module_name¿ is an atom.

:- with_macro ¡macro_name¿. Specifies the macro used in the source file. The ¡macro_name¿ is an atom.

[Caution]The current self-compiler ignores this macro specification. It always assumes the default macro to have been set. If this specification is originally necessary because of consistency with the cross system (CAL), it is desirable that the specification be included.

:- public ¡predicate_name¿/¡arity¿, ¡predicate_name¿/¡arity¿, ... Declares that the predicates specified with the ¡predicate_name¿ and airty (number of arguments) are public predicates.

[Hint]Multiple public declarations are valid.

:- op(¡precedence¿, ¡type¿, ¡operator¿). Defines the operator used for parsing. The ¡precedence¿ must be an integer from 1 to 1200. The ¡type¿ must be one of the fx, fy, yfx, xfx, xfy, xf, or yf. The ¡operator¿ should be an atom.

[Caution]The scope of operator definition is to the end of the file from the term next to the operator definition.

7.2.2 Starting the compiler

Self-compiler can be invoked from the shell in two ways: interactive compile and batch compile.
compile

compile

Starts the interactive compilation. The files to be compiled are specified through the standard input.

compile

compile(FileNameList)

Starts the batch compile. The argument FileNameList is a list of filenames (16bit strings) to be compiled.

[Example1]Starting the interactive compile

```
SHELL> compile
```

[Example2]Starting the batch compile - 1

```
shell> compile([''me:testprog1.kl1'', ''me:testprog2.kl1''])
```

[Example3]Starting the batch compile - 2

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

7.2.3 Interactive compile

How to specify the source files to be compiled

In the interactive compile mode, files specified through the standard input. Multiple files can be specified by separating by a comma (,) or a space. If the input line ends with a comma, the input continues to the next line. The filenames do not need to be enclosed with double quotes ("). The filename extension "kl1" or "KL1" is omissible.

Link sequence

To resolve the external references, the modules compiled so far are linked if possible and then linked with the modules in the module databases. If an undefined module is detected, the unresolved module(s) will be retained in the compiler process and regarded as outstanding module(s). If this unresolved external reference are resolved by subsequent compilation.

If the outstanding module become unnecessary and is desired to be discarded, input a period (.) and return.

[Caution]The modules specified in one input are linked together. Any modules referenced mutually should thus be specified in one input. If the file specification is split into two or more tries of input, the proper compilation results can be obtained only when the modules are compiled beginning with the lower hierarchy module in the reference network.

Terminating the interactive compile

The interactive compilation process is terminated by input "!exit". If one or more outstanding modules exist, they are discarded.

[Example1]Executing the interactive compile

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

[Example2]Executing the interactive compile

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

[Caution]The two above examples obtain the same results if (i) there is no reference relationship between the module in file demo1 and the one in file demo2, or (ii) the module in demo2 references the one in demo1. Otherwise, no compile results are guaranteed.

7.2.4 Batch compile

In the batch compile mode, only the files specified in the argument are compiled together. Each file name extension "kl1" or "KL1" is omissible. If an undefined external reference is detected during the link process, this causes error. In this case, the compile ends without register any module.

The batch compile is useful in the shell command files.

[Example]Executing the batch compile

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

[Hint]The above example is equivalent to execution of the following interactive compile:

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

7.3 Relinker

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

The compiler generates only the modules as specified. Module references are generally consists a network structure. If part of the reference network is updated by compilation, the module referencing to the compiled module(s) refers to the old one (before the compilation). Since the compiler only knows which modules to be linked, it is impossible to know the modules to be relinked that refers to the compiling modules. One way to obtain the proper compilation, all modules that consist the reference network should be compiled together.

In the case of developing a large program that consists of many modules, this is a time consuming way. This can be avoided using this relinker. Relinker generates the copies of specified modules updating the inter-module linkage(s) and registers to the module database. Two top level predicates relink:go/1 and relink:go/2 are described as followings.

relink

relink(ModuleAtomList)

relink(ModuleAtomList) Invoke a relink process. Argument ModuleAtomList is a list of module names (each module name is an atom). References from the module specified in ModuleAtomList are updated.

relink

relink(TopModuleAtom, ModuleAtomList)

Invoke a relink process. The top level module (atom) specified at the argument TopModuleAtom is searched for all modules related with the ones specified in the argument ModuleAtomList (list of module name atoms). They are then relinked.

7.3.1 Using the relinker

Invoke relink process in the following ways:

[Example1]Using relink:go/1

```
SHELL> relink([module_1, module_2, module_3])
Relinks modules module_1, module_2, and module_3.
```

[Example2]Using relink:go/2

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

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

7.4 Unloader

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

unload

unload(ModuleAtomList, FileName)

Invoke unload process. The argument ModuleAtomList is a list of the module names (atoms) that specify the target modules. The argument FileName is the filename to be saved.

[Caution]Since any unloaded file is in binary form, it cannot be edited by text-editor (for example, using PMACS of SIMPOS).

7.4.1 Using the unloader

Starts the unloader in the following way from the shell:

[Example]Using unload:go/2

```
SHELL> unload([module_1, module_2, module_3],''module123.unl'')
```

Unloads modules module_1, module_2, and module_3 into file "module123.unl".

[Hint]The total turn-around time required for unloading and loading modules is usually shorter than that for executing IPL again with CAL.

7.5 Loader

he loader is a shell program that reads the file saved by the unloader and generates the original modules.

load

load(FileName)

Starts the loader. The argument FileName specifies the name of the file generated by the unloader.

The links of the modules loaded by the loader (reference from the loaded modules) are established, from the beginning, during load processing. Since any link via which a loaded module is referenced is not influenced by loading, it must be relinked.

[Example]Using load:go/1

```
SHELL> unload(''module123.un1'')
```

Loads the module that has been unloaded under file name module123.unl.

Chapter 8

shell

8.1 Outline

The shell starts the user-specified goal. It also operates as the user interface that manages execution of this goal. This chapter outlines the shell.

Generating the task and executing the goal

After the user has designated the goal interactively via the shell window, the shell started it. At this time, the shell generates a new task, then inserts the goal in it for execution.

Standard I/O stream

When starting a user task with the shell, assign the task three 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.

More one stream called the **standard interaction** is also assigned to the task. It is linked to the following: the shell that has a shell window and is nearest to the user task in the user task's ancestor. That is, it is sure that the standard interaction stream is linked to the shell window of one of the shells. Of course the user can acquire and use other devices in the program. The shell has no relation with the devices.

Inter-task communication facility (pipe)

Output from a task may be desired to be used as input to another to implement task concatenation. To do this, concatenate standard input and output using the inter-task communication facility (pipe) provided by the shell. This enables the tasks to communicate. The pipe provides buffering. When the buffer in the pipe goes full, the task in the front stage of the pipe (producer) waits for a certain proper period until the buffer is emptied. If the buffer is full and the producer sends data expected not to be consumed, the pipe broken state is entered.

Job

The job consists of a series of tasks concatenated with the pipe. The shell manages user tasks started by itself, job by job, rather than task by task.

Management mentioned above includes (i) monitoring the job execution status, (ii) changing the job status such as the job start, stop, restart, or end, and (iii) inquiring about the job status.

Sharing the shell window

The shell usually has a shell window, used for interaction with the user. This window is used also as the standard I/O of the task started by the shell. It may thus be used competitively by several

tasks. For this reason, the shell executes shell window exclusive control between the shell itself and the started job.

Foreground and background

The shell offers two environments of foreground and background as job execution environments, which are used for shell window exclusive control. Several jobs can be entered in the background, whereas only one job can be in the foreground. The shell and the job entered in the foreground use the shell window exclusively. Access to the shell window from the jobs in the background is continuously suppressed. If this job sends an I/O message to the shell window, it cannot reach the window until the job is entered in the foreground. The above rule does not apply to output of standard messages. The background job can also display standard messages on the shell window.

Shell commands

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

8.2 Use

8.2.1 Start

The shell is started in the three ways below. When you start the shell, the window for shell appears, which is followed by prompt display. After this, the shell can accept various commands.

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

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

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

The meanings of the arguments in the goals are as follows:

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

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

W Number of characters per line in the window

H Number of lines of the window

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

[Caution1]If a specified argument includes a negative numeric in a position or size, or includes a type mismatch in character string or numeric data etc., the system regards this as a severe error. In this case, all specifications will be ignored; the position or size must be specified from the mouse, and the default font file is assumed. If the window from the specified position and size is out of the display screen, it will be created from 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 FEP specified in environment variable `task:node`.

8.2.2 Shell window commands

The shell window commands are classified into two groups: commands to start the user goal and those to the shell or jobs. According to the syntax rules mentioned later, each of the commands to start the user goal may appear more than once on one line. This is not permitted for the commands to the shell or jobs. No variable may be included in a command line. Each [...] below is the abbreviation for the command.

Starting the user goal

In command lines, several user goals are specifiable 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", `X, Y, W, H`)

  Window: `window`("window-name", `X, Y, W, H, Font`)

  [**Caution**]If a `^` is added to the file path name during the append mode after omitted from the specification, the file will be opened in the write mode. The window name is the window title, but not the text buffer name of the PMACS corresponding to the window. See "Starting the shell" for the meanings of the other arguments.

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

  [**Example1**]If the command line has no argument:

  ```
  > test:start
  ```

  [**Example2**]If the command line contains arguments:

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

  The colons and predicates are omissible. If they are omitted, go will be assumed as the predicate name.

- Only one standard message output device is specifiable in <task>. If it is omitted, the shell's standard message output will be shared for use.

  [**Example1**]Using the shell's standard message output

  ```
  > test:start
  ```

  [**Example2**]Specifying the file as the standard message output

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

- <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 -> file("mes1.doc")
                              | test2:start -> file("mes2.doc")
  ```

- Only one standard input device and only one standard output device are specifiable in <job>. If the specification is omitted, the shell window will be used.

  [Example1]Specifying all of the standard input, standard output, and standard message

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

  [Example2]Specify one standard input and one standard output for each job

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

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

  [Example1]Entering <job> in the foreground

  ```
  > test:start
  ```

  [Example2]Entering <job> in the background

  ```
  > test:start &
  ```

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

  [Example1]Only `test3:start` is entered in the foreground. The others are entered in the background.

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

  [Example2]All jobs are entered in the background.

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

8.2.3 Commands to the shell or jobs

The shell offers the following commands to give instructions to the started job or another shell:

Job management command

stop

stop JobNo

Stops job JobNo.

fore

fore JobNo

Restarts job JobNo in the foreground.

back

back JobNo

Restarts job JobNo in the background.

kill

kill JobNo

Forcedly stops job JobNo. If all is specified in JobNo, the currently active jobs will forcedly be stopped.

status

status

Displays information about all jobs started by the shell.

If JobNo is omitted from a format including JobNo specification, the JobNo value of the last started job will be assumed.

Attention keys

To move the active job in the foreground to the background, press the Attention key (usually Control-C).

Commands to the shell

exit

exit

Terminates the shell. At this time, all started jobs are forcedly terminated.

shell

shell

shell

shell(X, Y, W, H)

shell

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

Each of these creates a new shell window, then starts another shell. See "Starting the shell" for the meanings of the arguments.

sh

sh

Enables the shell to call another (child shell) in a recursive manner.

suspend

suspend

Stops the child shell (attention processing for the child shell).

set

set prompt = Prompt

Changes the prompt displayed on the command input line. Use a character string or atom for prompt specification.

get

get prompt

Displays the currently set prompt.

history

history

Displays the contents of history stack.

non-negative-integer

non-negative-integer

Executes the contents of history stack. If the value is 0, the preceding command will be executed.

get

get history

Displays the history stack length.

set

set history = Length

Sets the history stack length. Length must be a non- negative integer or infinite.

get

get

Lists the settings of all shell variables.

help

help

Lists the commands.

cd

cd Directory

Sets the value of environment task:directory. Directory must be a character string. If it is omitted, the value of task:directory will be displayed.

8.2.4 Messages displayed by the shell

`<< syntax error >>`
An error was found in the input command line.

`Job` JobNo `was aborted .`
Job whose identifier was JobNo terminated abnormally.

`Illegal filename for take .`
The specified command file was not found.

`Illegal directory name .`
The specified directory was not found.

`Default resource-increment is` Inc `.`
Displays the unit value of resource to increment when resource shortage happens.

`Default PE-number is` PE `.`
Displays the processor that executes (starts) the job.

`PE-number` PE `is illegal .`
The processor having the specified processor number does not exist.

`Default minimum-priority is` Min `.`
Displays the lower bound of the priority used in starting the job.

`Default maximum-priority is` Max `.`
Displays the upper bound of the priority used in starting the job.

`The value of the` Key `is` Value `.`
Displays the value of environment and shell variables.

`The value of the` Key `not existed .`
Displays the value of environment and shell variables. (The specified key does not exist.)

`No environment variable existed .`
Displays the value of environment and shell variables. (No variable exists.)

`Current prompt is` Prompt `.`
The currently used prompt is Prompt.

`Length of the history-stack is` Length `.`
The current history stack length is Length.

`Resource-low occured in job` JobNo `.`
Shortage of resource occurred.

`Resource-increment` Inc `added into job` JobNo `.`
The predefined unit value of resource was added because of shortage of resource.

JobNo `-->` Status JobName
Displays the job status. The name of job JobNo is JobName, and the current status is Status.

`Deadlock occured !!`
A deadlock occurred.

`Merger-deadlock occured !!`
A merger deadlock occurred.

`Reduction failure occured !!`
User-defined predicate Pred failed.

`Unification failure occured !!`
Failure in unification.

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

`Range overflow occured !!`
The input number of argument of the built-in predicate is too many.

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

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

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

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

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

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

`Illegal merger input occured !!`
An invalid data was found in the merger input argument.

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

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

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

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

`Something special exception occured !!`
A special exception occurred.

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

`Something illegal situation occured !!`
A failure occurred during job execution.

`Unknown exception detected !!`

`exception(jobNo(`JobNo`), type(`Type`))`
An exception that the shell does not know was raised.

The arguments displayed together with the messages are as follows:

JobNo: Job number

Inc: An unit value to increment resource

Min: Lower bound of job execution priority

Max: Upper bound of job execution priority

Key: Key of the environment or shell variable

Value: Value of the environment or shell variable

PE: Processor number

Pred: Predicate causing the exception

Bltn: Built-in predicate causing the exception

Pos: Position of the argument causing the exception

SubCode: Exception subcode

[Caution]Specification details for this are undetermined.

Term1, Term2: Terms that failed in unification

DType: Deadlock type

[Caution]Specification details for this are undetermined.

MDType: Merger deadlock type

[Caution]Specification details for this are undetermined.

Status: Job execution status

running In execution
stopped At a stop
rsc_low At a stop caused by resource shortage
terminate Terminate processing started.
zombi Terminate processing is in progress.
abnormal The job was put in a faulty state.

Type: Failure type

raised An exception that the shell does not know was raised.
undefinedCode The specified code is undefined.
illegalGoalName An error was found in the goal name.
illegalIoDevice An error was found in the I/O specification.
pipeBroken Pipe Broken was detected.
Others Errors detected by PIMOS etc.

[Caution]An error detected by PIMOS is displayed, for example, in format `fatal("xxxxx", []).`

8.2.5 Error handling

When the user goal develops a failure (for example, fails in processing, or causes an exception), the shell forcedly stops the entire job containing the task as well as the task itself.

8.2.6 Error handling in resource shortage

When the language definition resource runs short during job execution, resource_low is output. At this time, the shell stops the job, and inquires of the user whether to continue it. If the resource is added to continue the job, merely restart the job using the job management command (fore or back). In this case, add the predefined unit value regardless of whether fore or back is used, before restarting the job. The resource quantity to be added can be inquired using one of the following commands:

set

set rscinc = Inc

Sets to Inc, the unit value of resource to be added when resource shortage occurs in operation of subsequently started jobs. Inc is a non-negative integer equal to or less than 10^9, vector representation High,Low, or atom infinite. The ranges of the High and Low values are the same as Inc. The default is 10^8. If infinite is specified in Inc, the predefined quantity of resource (10^9) is added repeatedly and automatically. This means that apparently limitless resource quantity is assigned.

[Caution]The default value will be changed.

get

get rscinc

Inquires the predefined resource quantity to be added. The information is output to the shell's standard output destinations (usually window).

set

set rscinc(JobNo) = Inc

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

get

get rscinc(JobNo)

Inquires the incremental quantity of resource that has been set for the job. The information is output to the shell's standard output destination (usually window).

8.2.7 Specifying the processor executing the job

set

set pe = PE

Specifies the processor that will execute subsequently started jobs. The top level goal of every task under the job starts with the processor that has the number specified with non-negative integer PE. The default is the processor that started the shell.

get

get pe

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

[Caution]There is much possibility that the mechanism specifying the processor will be modified.

8.2.8 Specifying the range of job execution priority

set

set minprio = Min

set

set maxprio = Max

These specify the lower and upper bounds of the priority range of each task being under subsequently started 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: $0 \leq \text{Min} < \text{Max} \leq 4000$. The default values are 0 and 4000, respectively.

get

get minprio

get

get maxprio

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

[Caution]There is much possibility that the mechanism specifying the priority will be modified.

8.2.9 Command files

Shell command input is usually from the shell window. The input information can be temporarily changed into a file, then its internal command string can be executed.

take

take "command-file-name"

Inputs shell commands from the file specified in the command file name, then executes them.

[Caution1]In the command file, all jobs must be background jobs.

[Caution2]The Attention key is not accepted during command file read.

8.2.10 Manipulating the environment variables

setenv

setenv Key = Value

Sets value Value in the environment variable pool that the task of the shell itself has. Key is used as the key.

unsetenv

unsetenv Key

Deletes the environment variable(s) that use Key as the key, from the environment variable pool that the task of the shell itself has.

getenv

getenv Key

Read the value(s) that use 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

getenv

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

8.2.11 Initialization file

The initialization file contains commands executed automatically when the shell starts. Its file name is fixed to shell.com. When the shell starts, these file is searched according to the values of environment variables task:node, task:fep, and task:directory. If the file is not found, the message indicating this is displayed. In this case, the default values are set in the shell variables. See "Command files" for specifying the command file.

8.3 Creating Shell Utility Programs

This section explains how to create a program that runs under the shell.

8.3.1 Acquiring the standard I/O stream

Every standard I/O stream can be acquired using a raise. Note the following points in the acquisition:

- The same standard I/O stream may be acquired more than once. If this was done, the acquired streams are merely merged; the order of the messages flowing from different streams is not guaranteed.
- If std_in, std_out, or std_inter is used, std_io must not be used. If std_io is used, none of std_in, std_out, and std_inter must be used. If this rule is disobeyed, the operation during attention processing is not always correct.

    1. Requesting the shell to execute attention processing (in usual case)

shoen:raise(pimos_tag#shell, get_std_in, InStream)

shoen:raise(pimos_tag#shell, get_std_out, OutStream)

shoen:raise(pimos_tag#shell, get_std_mes, MesStream)

shoen:raise(pimos_tag#shell, get_std_inter, InterStream)

2. If the user executes attention processing:
   Instead of acquiring std_in, std_out, and std_inter, the user must use std_io, which executes the merged function of these three.

   shoen:raise(pimos_tag#shell, get_std_io, IoStream)

   shoen:raise(pimos_tag#shell, get_std_mes, MesStream)

   For normal attention processing, the pipe must not be used in the job during execution. That is, normal attention processing is enabled only when the job consists of one task.

8.3.2 Standard I/O stream message protocols

When the I/O stream is acquired, a buffering filter is automatically inserted into it. The filter is used, for example, for blocking/deblocking to reduce the communication overhead. (See Section 3.5)

Messages available for standard input (std_in)

Messages offered by buffer:input_filter

Messages available for standard output (std_out)

Messages offered by buffer:output_filter

Messages available for standard output (std_mes)

Messages offered by buffer:output_filter

Messages available for standard interaction (std_inter)

Messages offered by buffer:interaction_filter

Messages available for standard I/O (std_io)

- For standard input

  getb(Size, ^StringWithStatus)

  prompt(String, ^Status)

- For standard output

  putb(String, ^Status)

  flush(^Status)

- For standard interaction

igetb(Size, ^StringWithStatus)

iprompt(String, ^Status)

iputb(String, ^Status)

iflush(^Status)

[Caution]The functions of messages igetb/2, iprompt/2, iputb/2, and iflush/1 are the same as messages getb/2, prompt/2, putb/2, and iflush/1, respectively, except that the former are sent to standard interaction.

Abort/attention control messages

reset(^AbortAttentionAndIdWithStatus)

All attentions caused before sending this reset/1 message are processed by the shell. This is done also when the user has acquired std_io to execute attention processing by himself/herself. The user can get attention control after sending reset/1. After this, the user can execute the desired attention processing using the four messages below. Once acquired, attention control can never be discarded.

switch(^Status)

The user detecting the attention uses this message to move his/her job forcedly to the background.

next_attention(^AttentionWithStatus)

resend(Id, ^Status)

cancel(Id, ^Status)

Refer to the I/O device manual for details of messages reset/1, next_attention/1, resend/2, and cancel/2.

8.3.3 Shell utility program samples

This section shows simple samples of the shell utility program.

8.3.4 Tee program

This program copies the contents of standard input to the standard output and as standard message output:

```
:- with_macro pimos.
:- module tee.
:- public go/0.

go :- true |
    shoen:raise(pimos_tag#shell, get_std_in, In),
    shoen:raise(pimos_tag#shell, get_std_out, Out),
    shoen:raise(pimos_tag#shell, get_std_mes, Mes),
    In = [prompt(string#"TEE>"), get1(Line) | NIn],
    loop(Line, NIn, Out, Mes).

loop(Line, In, Out, Mes) :- string(Line, 0, _) |
    In = [], Out = [], Mes = [].
loop(Line, In, Out, Mes) :- integer(Line), Line = -1 |
    In = [], Out = [], Mes = [].
```

```
loop(Line, In, Out, Mes) :- string(Line, L, _), L > 0 |
    In = [prompt(string#"TEE>"), get1(NLine) | NIn],
    Out = [put1(Line), flush(_) | NOut],
    Mes = [put1(Line), flush(_) | NMes],
    loop(NLine, NIn, NOut, NMes).
```

Echo program

This program writes the argument to the standard output.

```
:- with_macro pimos.
:- module echo.
:- public go/1.

go(Arg) :- true |
    shoen:raise(pimos_tag#shell, get_std_out, Out),
    Out = [putt(Arg), n1, flush(_)].
```

Chapter 9

SHELL UTILITY PROGRAM

The shell utility program was designed to be started from the shell. The respective command names correspond to the module names. Since all predicates are go, they are omitted from the formats below. They do not need to be included in commands during input. These modules have no relation to the main program of the shell. They are exactly identical with 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.

9.1 GC Start

When the free space in memory runs short, Garbage Collection (GC) is automatically initiated. The following commands start GC regardless of the amount of free space:

gc

gc

Instructs the processors to initiate GC. The instruction is presented for all processors except the one on which the program is active.

gc

gc(PE)

Instructs the processors to initiate GC. The instruction is presented for the processors that have the processor numbers specified in argument PE. Exception: If atom was specified in PE, the processor on which the program was started is examined; it is instructed to initiate GC.

seqgc

seqgc

Instructs all processors to initiate CG. The instruction is presented in ascending order of the processor numbers. Exception: For the processor on which the program is active, the instruction is presented last of all.

seqgc

seqgc(PE)

Instructs the processors to initiate GC. The instruction is presented for the processors that have the processor numbers specified in argument PE. Exception: If atom was specified in PE, the processor on which the program was started is examined; it is instructed to initiate GC. This command executes exactly the same function as gc/1 mentioned above.

9.2 Status Display

9.2.1 Displaying the resource tree

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

rs

rs

Examines the resource tree about the child resources of the shell that started the program, then displays the contents.

rs

rs(Id)

Examines the resource tree about the child resources of resource Id, then displays the contents. See the general_request_stream protocol for specifying argument Id.

rsall

rsall

Examines the resource tree about all PIMOS-managed resources, then displays its contents.

9.2.2 Displaying the task tree

The following commands display only task information about the resource tree:

ts

ts

Examines the resource tree about the subtasks of the shell that started the program, then displays its contents.

ts

ts(Id)

Examines the resource tree about the subtasks of task Id, then displays its contents. See "general_request_stream protocol" for specifying argument Id.

tsall

tsall

Examines the resource tree about all PIMOS-managed tasks, then displays its contents.

9.3 Terminates PIMOS

The following command terminates the whole of PIMOS:

shutdown

shutdown

PIMOS Terminates.

9.4 Copy

The following commands copy data between standard I/O devices:

cat

cat

Read data line by line from the standard input, and outputs it to the standard output.

tee

tee

Reads data line by line from the standard input, and outputs it to the standard output and standard message output.

9.5 Module-related commands

9.5.1 Displaying usage of the modules

The following commands display usage of the modules:

usage

usage

Displays usage of the usage command on the standard output.

usage

usage(ModuleName)

Displays usage of the module specified in ModuleName (atom). Actually, the usage/1 predicate of the module that has the module name specified in argument is executed. The information displayed contains a list of two-element vector, which consists of (i) the format character string returned to the argument and (ii) the argument list. Thus, if the specified module does not contain predicate usage/1, or the value returned as the argument is invalid, the event is regarded as an error.

[Caution]Predicate usage is defined in few modules offered by the current PIMOS version.

9.5.2 Library function

The library function supports program development under PIMOS. The currently offered library function is only the function to examine defined predicates within a module.

Inquiry about the module definition predicates

Module named module is a shell program that outputs the predicates defined in the specified module and the number of arguments to the standard output. The predicates are sorted by the predicate name and the number of arguments, then output in the format of **predicate- name/number-of-argument**.

module

module(ModuleName)

Outputs the following information to the standard output: the names of the predicates defined in the module specified in ModuleName (atom) and the number of arguments.

[Example]Invoke the library function

```
SHELL> module(foo)
```

9.6 Commands to Character Strings and Text Files

9.6.1 Echo

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

echo

echo(X)

Outputs the contents specified in argument X to the standard output.

9.6.2 Retrieving a character string

The following commands retrieve a character string

grep

grep(Pattern)

Reads data line by line from the standard input. If a line contains the character string specified in argument Pattern, this command outputs the contents of the line to the standard output.

grep

grep(PathnameWithWildcard, Pattern)

Reads data line by line from a file the name of which matches the path name containing the wild card specified in argument PathnameWithWildcard (character string). If a line contains the character string specified in argument Pattern, this command outputs the contents of the line to the standard output. The wildcard may be included in the directory.

[Caution]Unlike the command name, currently this command does not support retrieval processing of regular expression. This may be revised in future.

9.6.3 Counting the number of lines

The following command counts the number of lines.

lc

lc

Reads data line by line from the standard input, then outputs the number of lines to the standard output.

9.6.4 Displaying the directory

The following commands display the contents of directory:

ls

ls

Lists the files of the current directory.

ls

ls(PathnameWithWildcard)

Lists the files the names of which match the path name containing the wildcard specified in argument PathnameWithWildcard (character string). The wildcard may be included in the directory.

Chapter 10

Listener

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

10.1 Function Summary

The listener has the functions outlined below.

Calling of Execution Codes

The listener is able to execute goal sequences consisting of user-defined predicates and built-in predicates from the listener top level. The listener manages this execution as a task. In addition, the listener also allows the making of inquiries and so on regarding the amount of resources consumed and the state of resources of the task, interrupting execution of the task by pressing the attention key.

Providing of Information Regarding Program Execution

When the execution of a task is completed, the listener reports the amount of resources that have been consumed and the execution time for that particular execution. In addition, in the case there are variables contained in the goal sequence, the listener also reports those values.

Tracing Function

The tracing function provided by the listener allows the process of progression of the reduction of the specified goal to be viewed. In addition to the normal step tracing function, a spying function, which traces only the calling of the specified predicate, and a function which temporarily pauses goal execution, are also available.

Deadlock Detection

When execution of the user program becomes deadlocked, the listener reports this deadlocked state. The deadlocks that are reported include user-defined predicate and built- in predicate deadlocks, as well as merger deadlocks.

Inspect Function

The listener is also equipped with an inspect function for analyzing the contents of data having complicated structures. By using this function, it is possible to analyze the values of variables and analyze the state of the number of arguments of the goal during trace execution (see chapter 10).

Standard Input/Output

At the time of starting of a task, 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, when not specified, the stream of the listener window will be given. As a result of this function, debugging of programs like those that are started from the shell can be performed easily.

10.2 Operating Method

10.2.1 Starting

The listener starts from the shell. It waits for input of the goal sequence at the "?- " prompt. This state is referred to as the listener top level. The goal sequence desired to be executed can be input from the top level, and trace information and execution results are displayed on the listener window. The following two goals are for starting the listener.

listener

listener

This starts the listener. The listener window can be created at any desired position and to any desired size using the mouse.

[Example]Starting up the listener from the shell.

```
prompt> listener &
```

listener

listener(Position, Size, Font)

This starts the listener and creates the listener window according to the specified position, size and font. The following values can be specified for each of the number of arguments.

Position: `mouse` or `at(X,Y)` (`X, Y` are the number of dots.)

Size: `mouse` or `char(Characters, Lines)` (`Characters`, `Lines` are the number of characters and the number of lines, respectively.)

Font: The name of the path of the font file (character string)

[Example]Starting up the listener specifing the position, size and font from the shell.

```
prompt> listener:go(mouse,char(64,16),"font:test_11") &
```

[Caution]The listener window is generated on the FEP specified with the environment variable task:node.

10.2.2 Input of Goal Sequence

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

[Caution]Since these multiple goals are executed in parallel, there is no significance to the order in which the goals are input.

Notation Arguments

Data notation of arguments of the goal is the same as KL1 data notation in the program. Notation of arguments using a macro is also possible.

End of Input

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

Calling of Public Predicates

Goals which call user-defined predicates that are declared public predicates (public declaration) in the program are specified with the following format in the same manner as calling between modules within the program.

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

[**Example**]Calling of a public predicate

```
?- layered:n_queens(8,Ans).
```

[**Hint**]In regard to user-defined predicates which declare a default module, input is possible by omitting the name of the module and entering only the predicate name and arguments. A description of the setting of default modules is given in 10.5.2.

Calling of Local Predicates

Goals which call user-defined predicates that are not declaring public predicates in the program are specified with the following format.

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

[**Example**]Calling of a local predicate

```
?- layered::queens(1,8,begin,Ans).
```

Calling of Built-in Predicates

Built-in predicates are described in the same manner as within the body of the program clause. Macros for calling built-in predicates can be used in the same way.

[**Example1**]Calling of a built-in predicate

```
?- predicate_to_code(module#layered,n_queens,2,Code).
```

[**Example2**]Integer operation

```
?- Priority := (Pri-Min)/(Max-Min)*4096.
```

[**Example3**]Floating point number operation

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

Starting of a Multiple Number of Goals

In the case of starting a multiple number of goals at one time, a "," (comma) is used to separate each of the goals.

Example[1] Starting of a multiple number of goals

```
?- test::jump(Jump,Step), test:hop(Hop), test:step(Hop,Step).
```

Example[2] Mixing of built-in predicates

```
?- current_processor(_,X,Y), PEs := X*Y, layered:n_queens(PEs,Ans).
```

Standard Input/Output

Although the standard input/output is normally set in the listener window, it can be changed as desired during input of the goal sequence. A detailed description of this is given in 10.3.

10.2.3 Variable Pool

The listener stores those variables used by the user for which the value has been determined by execution in a location referred to as the **variable pool**. The values of variables stored in the variable pool are bound, and can be used as many times as desired as arguments of the goal at the listener top level. The loading of variables into the variable pool is subject to the following restrictions:

Variables stored in the variable pool: Those variables in which the name of the variable starts with a capital letter.

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

Those variables in which the name of the variable starts with a capital letter are stored in the variable pool, and their values are reported at the time of end of execution.

Variables not stored in the variable pool: Those variables for which the name of the variable starts with an underline.

```
_, _X, _Priority, _X1, _Ans_1, __XX, ...
```

Those variables for which the name of the variable starts with an underline are not stored in the variable pool, and their values are not reported at the time of end of execution.

[Example]Variables that are not stored in the variable pool

```
?- _X=2, XX := _X*_X.
 XX=4
 yes.
```

List of Stored Variables

A command is available which displays the names and values of variables that are stored in the variable pool. A detailed description of this command is given in 10.5.3

Releasing of Values

".", "`cr`" (period-return) should be entered in order to release values.

10.2.4 Interruption of Execution

The user can interrupt execution of a task by pressing the attention key ("**Ctr-C**"). The following key operations can be performed for execution of a task that has been interrupted.

< *cr* >: continue
`g`: gc & continue
`r`: consumed resource
`s`: status
`@`: status all
`t`: list traced goals
`k`: list kept goals
`a`: abort
`b`: break
`h`: help

The following provides a description of the meanings of each of these operations.

Continue (return)
Continue execution.

Gc & Continue (g)

This starts the garbage collector for all processors and then continues execution.

[Hint]In cases like that which occurs when execution of the user program has become deadlocked, it is convenient to perform this operation. An additional description of deadlock is given in 10.4.

Consumed resource (r)

This inquires as to the amount of resources that were consumed (number of reductions) up until the time execution of the task was interrupted, and displays `unknown` when the amount of resources consumed is unknown.

Status (s)

This inquires as to the status of the task. The following information is displayed.

```
%%% STATUS %%%
id      : 203
name    : "task_35"
status  : normal([stop(interruption)])
limit   : 75000000 reductions
consume :10129 reductions
[children] (id name status limit consume)
33 "task_1" normal ([stop(parent)]) 75000000 unknown
```

The meanings of each of the fields are as indicated below.

id: Relative resource identifier of task that is executing the goal.

name: Task name. Serial number of task after the listener is started.

status: Status of task. Consists of the following types of status:

normal: Currently in normal execution.

resource_low: The amount of resources is becoming low.

terminating: Currently in terminating processing.

unknown: Status is unknown.

limit: This is the upper limit of the amount of consumed resources.

consume: This is the amount of resources that were consumed (number of reductions) up until the time execution of the task was interrupted. When this is unknown, `unknown` is displayed.

children: When information pertaining to child tasks has been obtained, this provides that same information for all subject child tasks.

Status all (@)

This inquires as to the resource status of the task. Examples of the information that is displayed are 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 "*66/1*"
18 window "*63/1*"
19 file ">sys>csp>PIMOS.DIR>pimos.mac"
20 timer "timer"
23 atom "atom"
```

The meanings of each of these fields are as indicated below.

id: The relative resource identifier of the task that is executing the goal.

type: The type of OS-defined resource.

name: Task name. Serial number of task after listener has been started.

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

List traced goals (t)
This lists those goals that have not been reduced at the time this command is executed from among those goals that have been executed following specification of trace. A detailed description of this is given in 10.6.

List kept goals (k)
This displays the list of goals for which execution has been paused during trace execution. A detailed description of this command is given in section 10.6.

Abort (a)
This aborts execution.

Break (b)
This returns the listener to the top level state while execution remains interrupted. At this time, the interruption level is displayed at the top level. `exit` is entered in order to go down to the interruption level.

[Example]Calling and return of top level during interruption of execution.

```
?? Interrupt (h for help): b

{1}
?- exit.
```

Help (h)
This displays the help menu.

10.2.5 Termination

`exit` is entered from the listener top level. When the inspector has been started together with the listener, the inspector will also terminate when the listener terminates.

10.3 Standard Input/Output

The listener has available four devices (streams) consisting of standard input, standard output, standard message output and standard interaction for tasks. With respect to standard input, standard output and standard message output, the user can specify the desired device. What is different from the shell is that since the present execution environment of the task in the listener is only the "foreground", the standard output and standard message output behave in the same manner.

[Caution]Current `std_io` is not supported.
Although standard input/output is normally set in the listener window, in the case this is specified by the user, designation is performed at the listener top level describing after the goal sequence using the operators indicated below.

Standard Input: `<=`(device)

Standard Output: `=>`(device)

Standard Message Output: `->`(device)

In addition, the device is designated in the following manner:

File: `file(file path name)`

The file path name is a character string.

Window: `window(window title)`

The window title is a character string.

A maximum of one each of standard input, standard output and standard message output can be specified for each task (goal sequence that is entered once).

[Example1]Using the standard input/output of the listener.

```
?- test1:start, test2::start.
```

[Example2]Designating a file as a standard message output.

```
?- test1:start, test2::start -> file("mes.doc").
```

[Example3]Designating standard input, standard output and standard message output.

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

Refer to Chapter 8 in regard to the method for acquiring the standard input/output stream from within the task, as well as message protocol.

10.4 Deadlock Detection

10.4.1 Summary

Deadlock refers to the execution of a certain goal being interrupted and the occurrence of a deadlocked state in which restarting is definitely not possible within the range of normal processing in KL1. 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 of the group of goals that are in a dependent relationship with the data to be generated by the deadlocked goal will become deadlocked also. Therefore, the number of deadlocked goals generally tends to be large. What is necessary when actually debugging programs in which deadlock may occur is 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 PIMOS are all maximum goals. In addition, in the case the group of goals in the cause-and- effect relationship involving deadlock are forming a cyclic structure, since any goal has the same value in the dependent relationship, only one of those goals is reported. In looking at the behavior of program execution, although the detection of a deadlock in the source program is an extremely difficult task, the efficiency of program debugging is improved dramatically as a result of this function. The deadlocks that are reported consist of user- defined predicate and built-in predicate deadlocks as well as merger deadlocks.

[Hint]We recommend using the variable checking function of the cross system, V2/CAL (or PDSS) as a measure against the occurrence of deadlocks. There is considerable effectiveness even in checking only those variables that only occur once in the program clauses (so-called void variables). Furthermore, a variable checking function is scheduled to be loaded into the PIMOS main unit sometime in the future.

10.4.2 Starting of Deadlock Detection

At present, deadlock detection is performed at the time batch garbage collection (GC) is executed. Therefore, in order to perform deadlock detection, the garbage collector has to be started when the user program has become deadlocked. Although the garbage collector is begun automatically when there is a shortage of memory space area, in order to forcibly perform deadlock detection, it is also necessary to forcibly start the garbage collector. The following method is used to forcibly start the garbage collector.

1. Execution of the user program is interrupted (by entering attention using Ctr-C), and the garbage collector is started using the commands available during interruption of execution. Refer to 10.2.4 for further details.

2. The garbage collector is started from the shell.

[Hint1]The following methods are used for confirming that execution of the program has stopped:

1. The "multi PSI-CSP status monitoring panel" becomes black when the CSP silent mode is off.

2. Repeatedly interrupt execution of the user program (by entering attention using Ctr-C), and confirm that the amount of resources consumed by execution is not increasing using the commands available during interruption of execution. Refer to section 10.2.4 for further details.

[Hint2]In addition to deadlock detection by batch GC, a deadlock detection function using MRB GC is scheduled to be added in the future. Through the use of this detection method, not only the maximum goal, but the goal that created the maximum goal (the so-called "killer") is also detected during program execution.

[Caution]Deadlock is not detected in the following cases:

[Example]Effect of variable pool

```
?- add(X,1,_).
```

Although the execution of built-in predicate add/3 is interrupted when the variable X is undefined, since 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 the user performs debugging of deadlocks, it is important to make effective use of variables whose names begin with an underline.

10.4.3 Types of Deadlocked Goals

The types of deadlocked goals that the listener reports are as follows:

1. User-defined predicates

   [Example]Deadlock of a user-defined predicate

   ```
   *** Deadlock is detected in GC.
   Goal> rscfilter:genreq_filter(A,B)
     Pe> 0
    continue or abort[a]?
   ```

   rscfilter:genreq_filter/2 in processor 0 has become deadlocked.

2. Built-in predicates

   [Example]Deadlock of a built-in predicate

```
*** Deadlock is detected in GC.
Builtin> add(1,A,B)
     At> listener_cmd_builtin:do_command/1
     Pe> 0
 continue or abort[a]?
```

Built-in predicate add/3 being called in the body of listener_cmd_builtin:do_command/1 in processor 0 has become deadlocked.

3. Merger

[**Example**]Merger deadlock

```
?- merge({_,_,_,[abc],[256]},OUT).
 ?? Interrupt (h for help): g
 *** Merger deadlock is detected in GC.
 *** 3 merger inputs are abandoned.
 At> listener_cmd_builtin:do_command/1
 Pe> 0
  continue or abort[a]? <cr>
 OUT = [abc,256]
 yes.
```

Built-in predicate merge/2 that is called in the body of listener_cmd_builtin:do_command/1 in processor 0 became deadlocked due to three input streams being abandoned without being closed.

10.4.4 Deadlock Processing

Deadlock Processing by the Listener

When a deadlock is detected, the listener performs the following processing according to the type of deadlock.

User-defined predicate and built-in predicate deadlocks: The listener replaces the goal that has become deadlocked with a goal that will definitely be successful (which can be considered to be the true goal).

[**Caution**]Because of the above, there is strong possibility that the group of goals which is in a dependent relationship with the data to be generated by the goal that has become deadlocked will also become deadlocked in the form of a chain reaction. The manner in which deadlocks are handled by the listener is scheduled to be changed considerably in the future.

Merger Deadlocks: All of the input streams that have been abandoned by the merger that has become deadlocked are closed (by unifying empty lists).

Commands that can be Inputted During Report of a Deadlock

When a report of a deadlock is obtained, the listener will ask the user whether execution is to be continued or aborted.

abort (a): Execution is aborted.

[**Caution**]Even in the case program execution is aborted, a multiple number of deadlocks may be reported. This occurs in the case when a multiple number of maximum goals have been detected by one operation of the garbage collector. Therefore, there is no dependent relationship between the multiple number of deadlocked goals that are reported.

continue (when anything except "a" is entered): Execution is continued.

[**Caution**]The user interface at the time of reporting of deadlocks is scheduled to be modified considerably in the future.

10.5 Commands

There are several commands available for performing program debugging and conversational program execution.

10.5.1 Basic Commands

ModuleName:PublicGoal

This executes the public goal that is the public predicate of the module indicated with ModuleName.

ModuleName:LocalGoal

This executes the local goal that is the local predicate of the module indicated with ModuleName.

help [he]

This displays the list of help commands.

help Type [he]

help(Type) [he]

This displays the list of type commands designated with Type (atom). The following items can be designated for type.

basic: Basic commands

env: Environment commands

varpool: Variable pool operational commands

debug: Debugging commands

all: All commands

display(Term, Depth, Length) [disp]

This displays arbitrary KL1 data designated for Term in the listener window under the restrictions of the Depth, the maximum depth of the displaying structure, and Length, the maximum length (number of elements).

display Term [disp]

display(Term) [disp]

This displays arbitrary KL1 data designated for Term in the listener window under the conditions of the default depth and length.

exit [ex]

This takes the user one step out of the nested execution level. When the level is zero (top level, when level is not displayed), the listener is terminated.

10.5.2 Environment Commands

variable_value [vv]

This turns on the mode which reports the values of variables appearing in the goal sequence for which execution has been successful.

[Example]When variable_value is on.

```
?- layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ...]
 152 reductions
 432 msec
 yes.
```

novariable_value [nvv]

This turns off the mode which reports the values of variables appearing in the goal sequence for which execution has been successful.

[Example]When variable_value is off.

```
?- layered:n_queens(4,Ans).
 152 reductions
 410 msec
 yes.
```

statistics [st]

This turns on the mode which reports information relating to task execution. The information that is reported consists of the amount of resource consumed by the task (reductions) and the time required for execution (in millisecond units).

[Example]When statistics is on.

```
?- layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ...]
 152 reductions
 432 msec
 yes.
```

[Caution]A slight amount of listener overhead is contained in the number of reductions that is displayed.

[Hint]The PIMOS real-time timer is used for the measurement of execution time (in parallel systems, the total of CPU operating time does not have much meaning). As a result, in the event the task is interrupted, that interruption time is also included in the execution time.

nostatistics [nst]

This turns off the mode which reports information related to task execution.

[Example]When statistics is off.

```
?- layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ...]
 yes.
```

default_module [dfm]

default_module ModuleName [dfm]

default_module(ModuleName) [dfm]

This sets the default module for the module that has been designated with ModuleName (atom). Predicates in which the default module has been declared can be executed omitting input of the module name at the listener top level. Initial values are not set (no default module). In the case ModuleName is omitted, the listener will wait for input by the user following display of the current default module values. Furthermore, calling of user-defined predicates in the case of omission of the module name is as indicated below.

Calling of Public Predicates

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

[Example1]Calling of a public predicate that declares the default module

```
?- n_queens(8,Ans).
```

[Caution]In calling of user-defined predicates in which the module name has been omitted, when the public predicate to be called collides with listener commands (or built-in predicates), the listener command is given priority.

Calling of Local Predicates

```
: LocalPredicateName (Arg, ...)
```

[Example2]Calling of a local predicate that declares the default module

```
?- :queens(1,8,begin,Ans).
```

print_depth [pd]

print_depth Depth [pd]

print_depth(Depth) [pd]

This sets the default of the output restriction on depth of the structure to the value indicated with Depth (Depth > 0). The initial value is 4. Furthermore, when the structure to be output exceeds the output restricting value of depth, an "`&`" is displayed.

print_length [pl]

print_length Length [pl]

print_length(Length) [pl]

This sets the default of the output restriction on length of the structure (number of elements) to the value indicated with Length (Length > 0). The initial value is 10. Furthermore, when the structure to be output exceeds the output restriction value of length, a "**...**" is displayed.

print_string_length [psl]

print_string_length Length [psl]

print_string_length(Length) [psl]

This sets the default of the output restriction on the number of elements of the string to the value indicated with Length (Length > 0). The initial value is 20. Furthermore, when the number of elements of the string to be output exceeds the output restriction value, a "**??**" is displayed.

print_atom_length [pal]

print_atom_length Length [pal]

print_atom_length(Length) [pal]

This sets the default of the output restriction on the number of characters of the printed name of the atom to the value indicated with Length (Length > 0). The initial value is 40. Furthermore, when the number of characters of the printed name of the atom to be output exceeds the restriction value, a "??" is displayed.

print_variable_mode [pv]

print_variable_mode Mode [pv]

print_variable_mode(Mode) [pv]

This changes the output mode of those items which express variables to the mode indicated with Mode. Either name of address is given to Mode with atom. The initial value is name.

name	-	Name Mode	::	A,B,C, ...
address	-	Address Mode	::	$UNDEF(Pe, Address)

set_minimum_priority [setmin]

set_maximum_priority [setmax]

set_minimum_priority Min [setmin]

set_maximum_priority Max [setmax]

set_minimum_priority(Min) [setmin]

set_maximum_priority(Max) [setmax]

Each of these set the upper and lower limits of the priority range of the task to the values designated for Min and Max. Min and Max must be such that $0 le$Min < Max $\leq$ 4000 and be designated relative to the priority range given by the listener itself. The initial values are 0 and 4000, respectively.

minimum_priority [min]

maximum_priority [max]

priority [pri]

This asks for the upper and lower limits of the priority range of the task.

environmevnt [env]

This asks for the values of the environment variables of the listener. The initial values of each of the environment variables are as indicated below.

[Example]Display of environment variables when the listener is started

```
?- environment.
 Minimum priority         : 0
 Maximum priority         : 4000
 Default module           : 0
 Variable value           : on
 Statistics               : off
```

```
Remembrance              : on
Print depth              : 4
Print length             : 10
Print string length      : 20
Print atom length        : 40
Print variable mode      : name
Next task number         : 1
```

[Caution]The display of the default module having a value of zero indicates that the initial value is not set.

10.5.3 Variable Pool Operational Commands

Period-Return

This releases all of the variables stored in the variable pool. Period and return refer to the input of empty goals consisting only of "." and "cr".

list [l]

This displays a list of the names of variables stored in the variable pool.

value [v]

This displays the variables stored in the variable pool together with their values.

remember [rem]

This turns on the mode which remembers the values of variables that have newly appeared at the listener top level. Furthermore, even if this mode is turned on, those variables whose names begin with an underline are released when the task is completed.

forget [fgt]

This turns off the mode which remembers the values of variables that have newly appeared at the listener top level. Once this command has been entered, the values of variables that newly appear at the listener top level will be released. The values of variables that have already been bound are not released until a "period-return" is executed at the listener top level.

10.5.4 Debug Commands

Refer to 10.6 in regard to the concepts presented below that relate to the tracing system.

trace [tr]

This is used to enter the trace execution mode. The following prompt is output in the trace mode.

[Example]Trace mode prompt

```
{Trace}
?-
```

spy_reduction [sr]

This is used to enter the execution mode of the reduction spy. The following prompt is output in the reduction spy mode.

[Example]Reduction spy mode prompt

```
{SpyReduction}
?-
```

spy_fork [sf]

This is used to enter the execution mode of the fork spy. The following prompt is output in the fork spy mode.

[Example]Fork spy mode prompt

```
{SpyFork}
?-
```

notrace [ntr]

This is used to leave the trace and spy execution modes and return to the normal execution mode.

spy ModuleName: PredicateName/Arity

spy(ModuleName, PredicateName, Arity)

This makes the predicate to be the PredicateName/Arity in the module indicated with ModuleName, the spypoint predicate. Designation is possible for both public and local predicates. Furthermore, a wild card can be used for ModuleName, PredicateName and Arity. The wild card is atom "*".

[Example1]Designation of spypoint predicate using wild cards

```
?- spy(listener, *, *).
 Spypoint set on listener:*/*
 yes.
 The spy is set on all predicates having the module name "listener".
```

[Caution]In the case of using wild cards in the present designation of spy using the ModuleName:PredicateName/ Arity notation, the wild card must be surrounded by commas "(","(")". Furthermore, since "/*...*/" is a comment, if "/*" is mistakenly entered, close the comment with "*/".

[Example2]Designation of spypoint predicate using wild cards

```
?- spy listener:(*)/(*).
```

An analysis of the inclusive relationship between wild cards and spypoint predicates that have already been set (or that have been newly set) is performed for designation of spypoint predicates using wild cards.

[Example3]Analysis of inclusive relationship with wild cards

```
?- spy(listener, *, *).
 Spypoint set on listener:*/*
 yes.

?- spy(listener, listener, *).
 There already is a spypoint on listener:listener/*
 yes.
```

spy PredicateName/Arity

spy(PredicateName, Arity)

The sets the spy in the case the default module has been omitted. Wild cards can be used for Predicate, Name and Arity.

nospy ModuleName:PredicateName/Arity

nospy(ModuleName, PredicateName, Arity)

This canceles the predicate to be PredicateName/Arity in the module indicated with ModuleName from the spypoint predicate group. Wild cards can be used for ModuleName, PredicateName and Arity. The wild card is atom "*".

[Caution]Predicates in which the spy is set using the wild card cannot be canceled individually. In order to delete, the entire designation which used the wild card has to be deleted.

[Example]Case of not being able to delete the spypoint predicate

```
?- spy(listener, listener, *).
 Spypoint set on listener:listener/*.
 yes.

?- nospy(listener, listener, 10).
 Spypoint can not reset on listener:listener/10
 yes.

?- nospy(listener, *, *).
 %%% Reset spied pattern (listener:*/*) %%%
 listener:listener/*
 yes.
```

nospy PredicateName/Arity

nospy(PredicateName, Arity)

This cancels the spy when the default module has been omitted. Wild cards can be used for PredicateName and Arity.

nospy

This cancels all spypoint from spied predicates. This is the same as the case of executing `nospy(*,*,*)`.

spied [sd]

This displays the list of predicate patterns that are the spypoints of the spy.

[Example]Displaying the list of spypoint patterns

```
?- spy(*, go, *), spy(tee, *, *)
 Spypoint set on *:go/*
 Spypoint set on tee:*/*
 yes.

?- spied.
 %%% SPIED PATTERNS (2 patterns) %%%
 *:go/*
 tee:*/*
 yes.
```

spying [si]

This analyzes the spypoint patterns that are registered and displays a list of those predicates that are spypoint predicates.

[Example1]Displaying the list of spypoint predicates

```
?- spy(*, go, *), spy(tee, *, *)
 Spypoint set on *:go/*
 Spypoint set on tee:*/*
 yes.

?- spying.
 %%% SPIED PATTERNS (2 patterns) %%%
*:go/*
tee:*/*
 %%% SPYING PREDICATES (69 predicates) %%%
boot_strapper:
  go/2 (public) - *:go/*
top_task:
  go/2 (public) - *:go/*
  go/4 - *:go/*
       ...
       ...
compiler:
  go/0 (public) - *:go/*
 ...(10/69) ...?
```

The list of spypoint predicates is displayed in groups of 10 predicates each. The word (public) appears in parentheses is additionally displayed for public predicates. In addition, the designated pattern of the corresponding spy is also displayed. Those commands that can be entered during display of the list are indicated below.

< *cr* >	:	continue
N(>0)	:	number of list
q	:	quit
h	:	help

The following provides descriptions of the meanings of each of these operations.

Continue :: return

Continues display of the list.

Change number of list :: N (N>0)
This displays only the number of predicates designated with N. Once this command has been executed, all spypoint predicates will be displayed in groups of N each.

Quit :: q
Terminates displaying of the list.

Help :: h
Provides help.

spying ModuleName [si]

spying(Module Name) [si]

This displays the list of predicates that are spypoint predicates within the module designated with ModuleName. Furthermore, The commands that can be entered during display of the list are the same as those of spying/0.

[Example]Displaying the list of spypoint predicates designating a module.

```
?- spy(*, go, *), spy(tee, *, *).
 Spypoint set on *:go/*
 Spypoint set on tee:*/*
 yes.

?- spying(tee)
 %%% SPIED PATTERNS (2 spied patterns) %%%
*:go/*
tee:*/*
 %%% SPYING PREDICATES (2 spied predicates) %%%
tee:
  go/0 (public) - tee:*/*, *:go/*
  loop/3 - tee:*/*
 yes.
```

inspect Term [ins]

inspect(Term) [ins]

This performs calling of the inspector from the listener top level. It inspects arbitrary KL1 data designated with Term. The use of this function allows investigation of the contents of data having complicated structures (see Chapter 11).

10.6 Tracer

10.6.1 Summary

The important feature of a function which traces execution for the purpose of program debugging is that it suppresses the tracing of those portions that are not necessary thereby allowing concentration of attention on the target of debugging. In ordinary sequential languages, the execution of one portion of the program cannot be begun until the execution of another portion of the program that was started from one location within the program is 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 also, execution is 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 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 designated 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) that 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:

```
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 the following 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 corresponds to the designating of not performing tracing for arbitrary portions of the tree within this tree structure.

Trace

When the goal for which tracing has been designated has been reduced, that reduction is output.

[Example]Trace

```
{Trace}
?- network:go(Size).
@0@4096 0     network:go(A)
                  current_processor(B,C,D)
                  E:=C*D
                  A=E
                  F:=A*A
                  merge(G,H)
1     * (1)     observer(F,H)
2     * (2)     goal_distributor(0,A,G)?
```

The reduction of network:go/1 that was executed in the trace mode is output.

Spy

In this tracer, spy refers to the performing of tracing as the target of reduction of a specific predicate only. In this tracer, two types of spies are realized.

Reduction Spy

This is a spy which reports the 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 will not be reported as long as that suspension is not canceled.

[Example]Reduction spy

```
?- spy layered:colors/1.
   ...
{SpyReduction}
?- layerd:map(Ans).
<0> layered:map(A) -->
@0@4096     + colors(B)
                  [red,yellow,blue,white]=B?
```

Reduction is reported when that goal that corresponds to layered:colors/1 for which the spy is set is reduced.

Fork Spy

This spy reports reduction when the goal, which has a goal that is the target of the spy as the goal of the body (child goal), is reduced.

[Example]Fork Spy

```
?- spy layered:colors/1.
   ...
{SpyFork}
?- layered:map(Ans).
<0> layered:map(A) -->
@0@4096       area/3
1   + * (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, that reduction is reported.

[Caution]In the case of executing fork spy, it is not possible to know the status of the argument of the goal (parent goal) which is directly calling the detected spied goal or brother goals of the goal for reasons relating to implementation.

10.6.2 Trace Output

Precautions Relating to Trace Output

The trace output will not always agree with that described by the user with respect to the following items:

1. There are cases when the execution sequence of built-in predicates and unification may be changed by the compiler.
2. There are cases when built-in predicates that have been described by the user may be expanded into a plural 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 notated with a capital letter as the first letter. Although the identify of variables having the same name is guaranteed within a tracer, in the event the garbage collector is started during tracing, the same variables will be displayed with different names.

Trace Output

The following types of outputs can be performed with the trace function.

```
@0@4096 20    layered:queen(2,4,[1*L,2*T,3*E1|F1],X)                (Ex.1)
                  X=[2*H1|I1]
                  J1:=2+1                                           (Ex.2)
30 + * (1)      filter([1*L,2*AT,3*E1|F1],2,1,H1)                   (Ex.3)
31   * (2)      queen(J1,4,[1*L,2*T,3*E1|F1],I1)?

<2> layered:queen(1,4,begin,[1*D|E]) -->                           (Ex.4)
@0@4096     + filter(begin,1,1,D)                                  (Ex.5)
                  D=begin?

<3> layered:queens(2,4,C,A) -->
@0@4096       queen/4                                              (Ex.6)
4  +  (1)       filter(C,1,1,D)?


@0@4096 6     job:create(createJob( & ,Q, ...)
11    * (1)     @pe(0) <= create1(createJob(...),Q,...)?           (Ex.7)

@0@4096 0     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)?   (Ex.8)


@0@4096 1   listener:go(mouse,mouse,...)
                  calculate_priority_from_minimum(4092,A)
              ...
10   *  (9)  @pri(A) => get_command(prompt,K,...)?                 (Ex.9)
```

[Example1]Trace goal

```
@0 @4096 20 layered:queen(2,4,[1*L,2*T,3*E1|F1],X)
(a) (b)  (c)          (d)
```

[Example2]Body built-in predicate

```
        J1:=2+1
          (e)
```

[Example3]Child goal

```
31   *    (2)  queen(Ji,4,[1*L,2*T,3*E1|F1],Ii)?
(c) (f)   (g)              (h)               (i)
```

[Example4]Ancestor goal

```
<2> layered:queen(1,4,begin,[1*D|E]) -->
(j)           (k)
```

[Example5]Goal detected by reduction spy

```
@0@4096      + filter(begin,1,1,D)
            (l)       (m)
```

[Example6]Parent goal detected by fork spy

```
@0@4096        queen/4
                 (n)
```

[Example7]Child goal with processor designation

```
11     * (1)      @pe(0) <= create1(createJob( ...),Q, ...)?
                   (o)
```

[Example8]Child goal with designation of execution priority (Relative self specification in the belonging sho-en)

```
                     calculate_priority_from_current(-3000,B)
1        (1)       @pri(B) => queen_n([1,2,3,4],[],[],A,4,0,0,4,1)?
                   (p)
```

[Example9]Child goal with designation of execution priority (Rate specification in the belonging sho-en)

```
                   calculate_priority_from_minimum(4092,A)
                  ....
10     * (9)       @pri(A) => get_command(prompt,K, ...)?
                   (p)
```

The meanings of each of the fields in the trace displays in the examples above are as indicated below.

(a) Processor number: The number of the processor that reduced the goal.

(b) Priority: The priority of the goal in the belonging sho-en which it belongs (proportioning within the affiliated domain).

(c) Goal number: The goal identification number. The number that is given exclusively to the goal by the tracer during execution of the goal in the trace mode. Management of these numbers makes it possible to know which goal has been reduced.

(d) Trace goal: The reduced goal.

(e) Body built-in predicate

(f) Trace mark: This mark indicates that a trace is designated for the child goal.

(g) Child goal serial number: A serial number that is given to child goals of the trace goal. This is used when designating child goals with the trace command.

(h) Child goal: Child goals are displayed with the module name omitted when the trace goal and the module name are the same.

(i) Prompt: This indicates that the tracer is waiting for input of a trace command.

(j) Ancestor goal number: This is the goal number of the ancestor goal. It is used in the same manner as explained in (c).

(k) Ancestor goal: The goal that corresponds to the ancestor of the goal that was detected by the spy (this may be the parent in some cases). In other words, this means that the user executed this goal in the spy mode.

(l) Spy mark: This mark is attached when the goal corresponds to spypoint predicate.

(m) Spy goal: The spypoint goal that was detected as a result of execution of the reduction spy.

(n) Parent goal: This is the goal that corresponds 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 Arity. The parent goal is displayed with the module name omitted when the ancestor goal and the module name are the same.

(o) Destination processor: This is displayed when the child goal is a goal that designates a processor.

(p) Execution priority designation: This is displayed when the child goal is a goal that designates execution priority. The tracer displays the predicates indicated below together with the execution priority designation corresponding to the designation method of execution priority.

- Case of Relative self specification belonging sho-en domain:

```
calculate_priority_from_current(Rate,X)
...
@pri(X) => Goal
```

- Case of Rate specification in the belonging sho-en domain:

```
calculate_priority_from_minimum(Rate,X)
...
@pri(X) => Goal
```

The respective "Rate" are the proportion of the priority designated by the user.

[Caution]In the case a current goal with execution priority designation is detected by the fork spy, the predicate which expresses the designation method of execution priority is not displayed. This is scheduled to be modified in the future.

10.6.3 Trace Commands

The following describes the commands that can be input following display during tracing. The meanings of the headings for each command are as indicated below.

```
Command Name :: Input Format {Argument}
```

Although whether or not execution of a child goal is to be performed again in the trace mode can be designated individually using the commands, there are modes available in regard to which is to be the default. In the explanations below, the mode in which the child goal is traced in the case of default is designated as the **verbose mode**, and the mode in which the child goal is not traced in the case of default is designated as the **terse mode**. For either of these modes, child goals for which trace will be performed as long as there is no particular designation will be displayed with trace mark ("*").

Continue :: Return

Continues execution. Execution is performed in accordance with the trace mode that is set for the child goal.

[Example1]Continues execution in the verbose mode (all traces)

```
@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)? <cr>
```

Child goals (1) and (2) are traced.

[Example2]Continues execution in the terse mode (suppression of tracing)

```
@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)? <cr>
```

Execution is performed suppressing tracing of child goals (1) and (2).

[Example3]Continues execution

```
@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)? <cr>
```

Child goal (2) is traced although tracing of child goal (1) is suppressed.

Toggle trace mode all and Continue :: x

Invents the trace mode and continues execution. The trace modes set for all of the child goals are inverted.

[Example1]Continues execution in the verbose mode (suppression of tracing)

```
@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)? <cr>
```

The tracing of child goals (1) and (2) is suppressed.

[Example2]Continues execution in the terse mode (all traces)

```
@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)? <cr>
```

Both child goals (1) and (2) are traced.

[Example3]Inverse execution of the trace mode

```
@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)? <cr>
```

Although child goal (1) is traced, the tracing of child goal (2) is suppressed.

Toggle trace mode and Continue :: N1,N2,...

Inverts the trace mode individually and continues execution. Execution is performed inverting the trace modes of child goals designated with child goal serial numbers N1, N2,... Child goals that are not designated are executed in accordance with individually set trace modes.

[Example1]Cancels the trace mode individually and continues execution

```
@0@4096 0     layered:map(A)
1     * (1)     area(ok),begin,B)
2     * (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? 1,2,3
```

Child goals (4), (5) and (6) are traced although tracing of child goals (1), (2) and (3) is suppressed.

[Example2]Designates the trace mode individually and continues execution

```
@0@4096 0     layered:map(A)
1       (1)     area(ok),begin,B)
2       (2)     area(on(ok),B,C)
3       (3)     area(on(on(ok)),C,D)
4       (4)     area(on(off(on(ok))),D,E)
5       (5)     area(off(on(on(ok))),E,F)
6       (6)     area(on(on(on(ok))),F,A)? 1,2,3
```

Although child goals (1), (2) and (3) are traced, tracing of child goals (4), (5) and (6) is suppressed.

Toggle default trace mode :: t

This inverts the automatic setting mode of tracing for the child goal. At present, the verbose mode is inverted to the terse mode, and the terse mode is inverted into the verbose mode.

[Example]Inverts automatic setting mode of tracing

```
@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)? t
@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)?
```

The automatic setting mode of tracing is changed from the verbose mode to the terse mode.

Toggle trace mode :: t N1,N2,...

Inverts the trace mode individually. The trace mode is inverted for child goals designated with child goal serial numbers N1,N2,... There is no change in the automatic setting mode of tracing.

[Example]Inverts the trace mode individually

```
@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)? t 1
@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)?
```

The designation of tracing of child goal (1) is canceled.

Apply reduction spy all and Continue :: sr

All child goals are executed by the reduction spy. Those descendent goals that are targets of the spy are not traced until they are reduced.

[**Example**]Executes all child goals by the reduction spy

```
@0@4096 0     layered:map(A)
1     * (1)     area(ok),begin,B)
2     * (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? sr
```

All child goals are executed by the reduction spy.

Apply reduction spy and Continue :: sr N1,N2,...

This designates the reduction spy and continues execution. Those child goals designated with child goal serial numbers N1,N2,... are executed by the reduction spy. Those child goals that were not designated are executed in accordance with the status of the trace mark.

[**Example1**]Designates reduction spy in the verbose mode

```
@0@4096 0     layered:map(A)
1     * (1)     area(ok),begin,B)
2     * (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? sr 1,2,3
```

The reduction spy is applied to child goals `(1)`, `(2)` and `(3)`, and child goals `(4)`, `(5)` and `(6)` are traced.

[**Example2**]Designates reduction spy in the terse mode

```
@0@4096 0     layered:map(A)
1       (1)     area(ok),begin,B)
2       (2)     area(on(ok),B,C)
3       (3)     area(on(on(ok)),C,D)
4       (4)     area(on(off(on(ok))),D,E)
5       (5)     area(off(on(on(ok))),E,F)
6       (6)     area(on(on(on(ok))),F,A)? sr 1,2,3
```

The reduction spy is applied to child goals `(1)`, `(2)` and `(3)` while tracing is suppressed for child goals `(4)`, `(5)` and `(6)`.

Apply fork spy all and Continue :: sf

All child goals are executed by the fork spy.

[**Example**]Executes all child goals by the fork spy

```
@0@4096 0     layered:map(A)
1       (1)     area(ok),begin,B)
2       (2)     area(on(ok),B,C)
3       (3)     area(on(on(ok)),C,D)
4       (4)     area(on(off(on(ok))),D,E)
5       (5)     area(off(on(on(ok))),E,F)
6       (6)     area(on(on(on(ok))),F,A)? sf
```

All child goals are executed by the fork spy.

Apply fork spy and Continue :: sf N1,N2,...

This designates the fork spy and continues execution. Those child goals designated with child goal serial numbers N1,N2,... are executed by the fork spy. Those child goals that were not designated are executed in accordance with the status of the trace mark.

[Example1]Designates fork spy in the verbose mode

```
@0@4096 0     layered:map(A)
1     * (1)     area(ok),begin,B)
2     * (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? sf 1,2,3
```

The fork spy is applied to child goals `(1)`, `(2)` and `(3)`, and child goals `(4)`, `(5)` and `(6)` are traced.

[Example2]Designates fork spy in the terse mode

```
@0@4096 0     layered:map(A)
1       (1)     area(ok),begin,B)
2       (2)     area(on(ok),B,C)
3       (3)     area(on(on(ok)),C,D)
4       (4)     area(on(off(on(ok))),D,E)
5       (5)     area(off(on(on(ok))),E,F)
6       (6)     area(on(on(on(ok))),F,A)? sf 1,2,3
```

The fork spy is applied to child goals `(1)`, `(2)` and `(3)` while tracing of child goals `(4)`, `(5)` and `(6)` is suppressed.

keep(cancel)all :: k

This is used to designate the keeping of execution of all child goals or cancel the designation. This command operates in the manner of a toggle switch, and in the case that holding of execution has already been designated for a designated sub-goal, that designation is canceled and the default trace mode is set. To use this function, you can change the scheduling for the execution of goals actively.

[Example]Designates keeping of execution or cancels designation

```
@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)? k  % Keep Designation
@0@4096 3     layered:queens(2,4,C,A)
                  G:=2+1
6     k (1)     queen(1,4,C,H)
7     k (2)     queens(G,4,H,A)? k  % Designation Cancellation
@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)?
```

The letter "k" in front of the sub-goal serial number indicates that keeping of execution of that goal has been designated.

[Hint]The following three goal states exist during tracing. No single goal can be in a multiple number of states.

1. Executed within Sho-en.
2. Waiting for reporting or reporting within the tracer.
3. State in which execution is not being kept.

keep(cancel)goals :: k N1,N2,...
This designates keeping of execution of child goals or cancels designation.

[Example]Designates keeping of execution and cancels designations

```
{trace}
?- layered:map(Ans).
@0@4096 0     layered:map(A)
1     * (1)     area(ok),begin,B)
2     * (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? k 1,2,3
                                % Designation of Keeping of Execution
@0@4096 0     layered:map(A)
1     k (1)     area(ok),begin,B)
2     k (2)     area(on(ok),B,C)
3     k (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (6)     area(on(on(on(ok))),F,A)? k 3
                                % Cancellation of Keep Designation
@0@4096 0     layered:map(A)
1     k (1)     area(ok),begin,B)
2     k (2)     area(on(ok),B,C)
3     * (3)     area(on(on(ok)),C,D)
4     * (4)     area(on(off(on(ok))),D,E)
5     * (5)     area(off(on(on(ok))),E,F)
6     * (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) - Execution in the trace mode.
```

List kept goals :: kg
This displays a list of those goals for which execution is being kept.

[Example]Displays the list of kept goals

```
@0@4096 4     layered:area(on(off( & )),D,E)
7     * (1)     colors(G)
8     * (2)     areai(on(off(on(ok))),D,G,E)? kg
 %%% KEEPING GOALS (3 goals) %%%
k:1 layered:area(ok,begin,B)
k:2 layered:area(on(ok),B,C)
k:3 layered:area(on(on(ok)),C,D)? tr 1    % Execution of Trace Mode
```

The "number" of k:"number" is the goal number. Those commands that can be input during display of the list are as indicated below.

```
N,...      :  execute
tr N,...   :  trace
sr N,...   :  spy reduction
sf N,...   :  spy fork
ins N      :  inspect goal
pd <D>     :  print depth
pl <L>     :  print length
psl <L>    :  print string length
pal <L>    :  print atom length
pv         :  print variable mode
q, <cr>    :  quit
h          :  help
```

Execute goals :: N1,N2,...
This executes kept goals designated with goal numbers N1,N2,... suppressing tracing.

Trace goals :: tr N1,N2,...
This executes kept goals designated with goal numbers N1,N2,... in the trace mode.

Execution goals by reduction spy :: sr N1,N2,...
This executes kept goals designated with goal numbers N1,N2,... in the reduction spy mode.

Execute goals by fork spy :: sf N1,N2,...
This executes kept goals designated with goal numbers N1,N2,... in the fork spy mode.

Inspect goal :: ins N
This calls the inspector and inspects the goal designated with goal number N.

Change print depth :: pd Depth
This changes the output restriction value of the depth of the structure. The word Depth may be omitted.

Change print length :: pl Length
This changes the output restriction value of the length of the structure. The word Length may be omitted.

Change print string length :: psl Length
This changes the output restriction value of the number of elements of the string. The word Length may be omitted.

Change print atom length :: pal Length
This changes the output restriction value of the number of characters of the atom. The word Length may be omitted.

Toggle print variable mode :: pv
This switches the variable display mode (between name and address).

Quit :: q or return
This quits the displaying of the list.

Help :: p
This is used to request help.

[Hint]The list of kept goals can also be viewed when execution of the program has been interrupted (Ctr-C).

Notrace anymore :: ntr
There is no more tracing beyond that point.

Set predicate spy on trace goal :: s
When the predicate of the trace goal is not registered as the spypoint predicate, this registers that predicate as the spypoint predicate.

[Example]Registers of the predicate of the trace 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)? s
@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)?
```

layered:queens/4 is registered as the spypoint predicate.

Reset predicate spy on trace goal :: s

When the predicate of the trace goal is registered as the spypoint predicate, this deletes that predicate from the spypoint predicate.

[Example]Deletes the trace goal predicate from 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)? s
@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)?
```

layered:queens/4 is deleted from the spypoint predicate.

[Caution]Those predicates that are registered as spypoint predicates by wild cards cannot be deleted.

Set predicate spy on subgoal :: s N

This registers the predicate designated with child goal serial number N as a spypoint predicate.

[Example]Registers of the predicate of the trace 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)? s 1
@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)?
```

layered:queen/4 is registered as the spypoint target.

Reset predicate spy on subgoal :: s N

This deletes the predicate designated with child goal serial number N from the spypoint predicate.

[Example]Deletion of child goal predicate from 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)? s 1
```

```
@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)?
```

layered:queen/4 is deleted from the spypoint predicate.

List spied patterns :: sd
This displays the list of predicate patterns that are the targets of the spy.

List spied predicates :: si
This displays the list of predicates that are the targets of the spy. Refer to 10.5.4 for those commands that can be input during display of the list.

List spied predicates of specified modules :: si ModuleName
This displays a list of the predicates that are the targets of the spy in the module designated with ModuleName. Refer to 10.5.4 for those commands which can be input during display of the list.

List traced goals :: tg
This displays a list of those goals that have still not been reduced at the point of execution of this command from among the goals for which trace has been designated for execution. If goals that are likely to become deadlocked have been designated for tracing, those goals will always be contained in this goal group.

[Example]List of unexecuted goals that are targets for tracing

```
 %%% TRACED GOALS (5 goals) %%%
t:14 layered:area1(on(off( & )),D,J,E)
t:12 layered:area1(on(on(ok)),C,I,D)
t:10 layered:area1(on(ok),B,H,C)
t:16 layered:area1(off(on( & )),E,K,F)
t:8  layered:area1(ok,begin,G,B)
 ... (5/5) ...?
```

Furthermore, the "number" of t:"number" is the goal number. Those commands that can be input during display of this list are as indicated below.

< *cr* >	:	continue
N(>0)	:	number of list
N(<0)	:	back
t, 0	:	back to top
ins ID	:	inspect goal
kg	:	list kept goals
pd <D>	:	print depth
pd <L>	:	print length
psl <L>	:	print string length
pal <L>	:	print atom length
pv	:	print variable mode
q	:	quit
h	:	help

Continue :: return
Continues display.

Change number of list :: N (N>0)
This displays only the number of goals designated with N. From that point on, the goals are displayed in groups of N each.

Back goals :: N (N<0)
This displays only the number of goals designated with N going backwards from the bottom to the top.

Back to top :: t,0
This repeats the display starting with the first goal.

Inspect goal :: ins ID
This calls the inspector and inspects the goal designated with the goal number ID.

List kept goals :: kg
This displays the list of goals for which execution has been kept.

Change print depth :: pd Depth
This changes the output restriction value of the depth of the structure. The word Depth may be omitted.

Change print length :: pl Length
This changes the output restriction value of the length of the structure. The word Length may be omitted.

Change print string length :: psl Length
This changes the output restriction value of the number of elements of the string. The word Length may be omitted.

Change print atom length :: pal Length
This changes the output restriction value of the number of characters of the atom. The word Length may be omitted.

Toggle print variable mode :: pv
This switches the variable display mode (between name and address).

Quit :: q
This quits the displaying of the list.

Help :: p
This is used to request help.

[Hint]The list of unexecuted goals that are the targets of tracing can also be viewed when execution of the program has been interrupted (Ctr-C).

Inspect trace goal :: ins
This calls the inspector and inspects trace goals (see Chapter 11).

Inspect subgoal :: ins N
This calls the inspector and inspects the goal designated with the child goal serial number (N). When no number is designated, the trace goal is inspected (see Chapter 11).

Change holophrasting :: holo
This changes the environment variable values related to holophrasting. It changes the output restriction values of the depth and length of the structure, the number of elements of the string and the number of characters of the 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
This changes the output restriction value of the depth of the structure. The word Depth may be omitted.

Change print length :: pl Length
This changes the output restriction value of the length (number of elements) of the structure. The word Length may be omitted.

Change print string length :: psl Length
This changes the output restriction value of the number of elements of the string. The word Length may be omitted.

Change print atom length :: pal Length
This changes the output restriction value of the number of characters of the atom. The word Length may be omitted.

Toggle print variable mode :: pv
This switches the variable display mode (between name and address).

Print environment variables :: env
This displays the values of the environment variables related to the tracer.

Re-display :: disp Depth,Length
This re-displays the reduction temporarily changing the values of Print-Depth and Print-Length to Depth and Length.

Break to listener :: b
This is used to enter the top level at which the listener is nested one level deeper while tracing or execution remains interrupted. At this time, the interrupted level will be displayed at the top level.

Abort :: a
This aborts execution.

Help :: h
This provides a simple explanation of commands that can be used with the tracer.

10.7 Error Handling

When a goal designated by the user results in some type of error such as the goal failing or an error in handling with respect to an exception, the listener will display information pertaining to the error and forcibly terminate the task that is executing that goal.

Error Messages Displayed by the Listener

`Nonexistent module`	: Module does not exist	(Ex.1)
`Nonexistent public predicate`	: Public predicate does not exist	(Ex.2)
`Nonexistent local predicate`	: Local predicate does not exist	(Ex.3)
`Local predicate`	: Local predicate executed as public predicate	(Ex.4)
`Public predicate`	: Public predicate executed as local predicate	(Ex.5)
`Default module is not set`	: Default module is not set	(Ex.6)
`Nonexistent command`	: Command (built-in predicate) does not exist	(Ex.7)
`Illegal argument`	: Illegal input of argument of command	(Ex.8)
`Illegal goal(variable)`	: Input of variable as a goal	(Ex.9)
`Syntax error`	: Syntax error in input	(Ex.10)
`Illegal specification`	: Error in specification of standard input/output device	(Ex.11)
`Cannot create device`	: Standard input/output device cannot be created	(Ex.12)
`Failure`	: All the guards of candidate clauses have failed	(Ex.13)
`Unification failure`	: Body unification has failed	(Ex.14-15)
`Raised`	: An exception unknown to the listener has occurred	(Ex.16)
`Trace exception occurred`	: A trace exception unknown to the listener has occurred	(Ex.17)
`Spy exception occurred`	: A spy exception unknown to the listener has occurred	(Ex.18)
`Priority rate X is illegal input`	: Illegal designation of goal execution priority	(Ex.19)
`Priority rate X is range overflow`	: Designation of goal execution priority is not within the legal range	(Ex.20)
`Processor number X is illegal input`	: Illegal designation of goal execution processor	(Ex.21)
`Cannot throw the goal at processor X`	: Designation of goal execution processor is not within the legal range	(Ex.22)
`Exception`	: An exception has occurred in execution of a built-in predicate	(Ex.23-29)
`Deadlock is detected in GC`	: A deadlock has been detected	
`Merger deadlock is detected in GC`	: A merger deadlock has been detected	
`Firmware error`	: Occurrence indicates a bug of firmware	

[Example1]Module does not exist

```
?- hash:hash({1,2,3}),Hash,Data).
 Nonexistent module>>hash
 no.
```

The module `hash` does not exist.

[Example2]Public predicate does not exist

```
?- pool:keyed_sort_bag([put(2,b),put(1,a),get_all(Content)])
 Nonexistent public predicate>>pool:keyed_sort_bag/1
 no.
```

`keyed_sort_bag/1` does not exist in the module `pool`.

[Example3]Local predicate does not exist

```
?- layered::area(Area).
 Nonexistent local predicate>>layered::area/1
 no.
```

`area/1` does not exist in the module `pool`.

[Example4]Local predicate executed as public predicate

```
?- layered:queen(1,4,being,List).
 Local predicate>>layered:queen/4
 no.
```

`queen/4` of the module `layered` is a local predicate.

[Example5]Public predicate executed as local predicate

```
?- layered::n_queens(4.Ans).
 Public predicate>>layered::n_queens/2
 no.
```

`n_queens/2` of the module `layered` is a public predicate.

[Example6]Default module is not set

```
?- :queen(1,4,begin,List).
 >> Default module is not set.
 no.
```

A local predicate was attempted to be executed with syntax which omitted the name of the module despite the default module not being set.

[Example7]Command does not exist

```
?- string("listener",Size,Elm).
 Nonexistent command>>string("listener",A,B)
 no.
```

The command `string("listener",A,B)` does not exist (because it is a built-in predicate for guards only).

[Example8]Illegal input of argument of command

```
?- setmin(0), setmax(infinite).
 Illegal argument>>set_maximum_priority(infinite)
 no.
```

The argument of `infinite` of the command `set_max_priority` is an illegal input.

[Example9]Input of variable as a goal

```
?- X.
 Illegal goal(variable)>>X
 no.
```

Input of a variable as a goal is not allowed.

[Example10]Syntax error in input

```
?- 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 ***`.

[Example11]Error in specification of standard input/output device

```
?- tee:go -> file("mes.doc") => windoe("OUT").
 Illegal specification>> =>(windoe("OUT"))
 no.
```

`windoe("OUT")` cannot be designated for the standard output device.

[Example12]Standard input/output device cannot be created

```
?- tee:go <= file("in.doc") => windoe("OUT").
 Cannot create device>> <=(file("in.doc"))
 no.
```

The standard input/output device cannot be created because the file "`in.doc`", the standard input device, does not exist.

[Example13]All the guards of candidate clauses have failed

```
?- pool:keyed_set([put(1,abc)]).
 Failure>> keyed_bag:keyed_set/5
    Goal > keyed_set( code$hasher:hash/3$207 ,[put(1,abc)],0,0{0})
      Pe > 0
 no.
```

All of the candidate clauses of predicate `keyed_set/5` of module `keyed_bag` in processor 0 have failed (because `put/2` does not exist in the protocol of `pool: keyed_set/5`).

[Example14]Body unification has failed

```
?- {a,b,c}={a,d,c}.
 Unification failure>> d
                With > b
                  Pe > 0
 no.
```

Unification of `d` and `b` failed in processor 0.

[Caution]In the case the unification of two structures fails at the element level, that element is displayed.

[Example15]Body unification has failed

```
?- {1,2,3}={1,2,3,4}.
 Unification failure>> {1,2,3}
                With > {1,2,3,4}
                  Pe > 0
 no.
```

Unification of {`1,2,3`} and {`1,2,3,4`} has failed in processor 0.

[Example16]An exception unknown to the listener has occurred

```
?- shoen:raise(pimos_tag#shell,get_std_io,IO), IO=[putb("TEST",_9].
 Raised>> get_std_io
   Data > A
     Pe > 0
 no.
```

The exception `Info=get_std_io`, `Data=IO` of `shoen:raise(Tag,Info,Data)` has occurred in processor 0.

[Example17]A trace exception unknown to the listener has occurred

```
?- module:get_code(layered,n_queens,2,Cde), apply_tracing(Cde,{4,Ans},1).
 >> Trace exception occurred.
 Trace ID> 1
 SubGoals>[internal( code$layered:queens/4$668,{1,4,begin,A})]
       Pe> 0
 no.
```

A trace exception of trace identifier 1 unknown to the listener has occurred in processor 0.

[Example18]A spy exception unknown to the listener has occurred

```
?- apply_spying(Cde,{4,Ans},{layered,filter,4},1).
 >> Spy exception occurred.
 Reduction> layered:queen/4
   SubGoal> internal( code$layered:filter/4$668 ,begin(1,1,A))
    Spy ID> 1
        Pe> 0
 no.
```

A spy exception of spy identifier 1 unknown to the listener has occurred in processor 0.

[Example19]Illegal designation of goal execution priority

```
?- pragma_test:go(1,rate).
 >> Priority rated "rate" is illegal input.
 Pragma> @ priority(*,rate)
     At> pragma_test:go/2
     Pe> 0
 no.
```

The execution priority `@ priority(*,rate)` designated for the goal being called in the body of `pragma_test:go/2` in processor 0 is an illegal input.

[Example20]Designation of goal execution priority is not within the legal range

```
?- pragma_test:go(2,-100).
 >> Priority rate "-100" is range overflow.
 Pragma> @ priority(*,-100)
     At> Pragma_test:go/2
     Pe> 0
 no.
```

The execution priority `@ priority(*,-100)` designated for the goal being called in the body of `pragma_test:go/2` in processor 0 is not within the legal range (designation using negative numbers is not possible rate specification in the belonging Sho-en).

[Example21]Illegal designation of goal execution processor

```
?- pragma_test:go(3,pe7).
 >> Processor number "pe7" is illegal input.
 Pragma>> a/0 @ processor(pe7)
     At> pragma_test:go/2
     Pe> 0
 no.
```

Execution processor number `pe7` designated for goal a/0 being called in the body of `pragma_test:go/2` in processor 0 is an illegal input.

[Example22]Designation of goal execution processor is not within the legal range

```
?- pragma_test:go(3,8).
 >> Cannot throw the goal at processor "8".
 Pragma> a/0 @ processor(8)
     At> pragma_test_go/2
     Pe> 0
 no.
```

Execution is not possible due to execution processor 8 designated for goal a/0 which is being called in the body of `pragma_test:go/2` in processor 0 not existing (for example, in the case of an 8 processors configuration).

[Example23]The argument of the built-in predicate is illegal

```
?- predicate_to_code(layered,n_queens,2,Code).
 Exception>> illegal_input_at_arg1
   Builtin > predicate_to_code(layered,n_queens,2,A)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

The first argument of built-in predicate `predicate_to_code/4` that is executed with `listener_ cmd_ builtin:do_command/1` in processor 0 is illegal.

[Example24]The argument of the built-in predicate is not within the legal range

```
?- new_string(String,10,64).
 Exception>> range_overflow
   Builtin > new_string(A,10,64)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

The argument of the built-in predicate `new_string/3` that is executed with `listener_cmd_ builtin: do_command/1` in processor 0 is not within the legal range (element sizes larger than 32 bits cannot be designated for the third argument).

[Example25]Index of structure out of bounds

```
?- new_vector(Vector,3),
   set_vector_element(Vector,3,Old,new,NewVector).
 Exception>> out_of_bounds
   Builtin > set_vector_element({0,0,0},3,A,new,B)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

The designation of the element location of built-in predicate `set_vector_element/5` that is executed with `listener_cmd_builtin:do_command/1` in processor 0 is outside of the structure.

[Example26]Overflow of integer operation

```
?- current_priority(Pri,Min,Max), Rate := ((Pri-Min)*4096)/(Max-Min).
 Exception>> integer_overflow
   Builtin > multiply(-289851296,4096,A)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

An overflow has occurred in the operation on the built- in predicate `multiply/3` that is executed with `listener_ cmd_builtin:do_command/1` in processor 0.

[Example27]Division by zero

```
?- X := 512/0
 Exception>> integer_zero_division
   Builtin > divide(512,0,A)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

Division by zero was performed in operation on the built-in predicate `divide/3` that was executed with `listener_cmd_builtin:do_command/1` in processor 0.

[Example28]Invalid operation on floating point number

```
?- X $:= 0.0/0.0.
 Exception>> invalid_floating_point_operation
   Builtin > floating_point_divide(0.000000E+0.000000E+0,A)
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

Built-in predicate `floating_point_divide/3` that was executed with `listener_cmd_builtin:do_command/1` in processor 0 is an invalid operation.

[Example29]Mismatch in arity of high- order predicate

```
?- 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,A,B})
        At > listener_cmd_builtin:do_command/1
        Pe > 0
 no.
```

The built-in predicate `apply/2` executed with `listener_cmd_builtin:do_command/1` in processor 0 does not match arity given in `comparator:sort/5`.

Chapter 11

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 tracing. (Refer to Chapter 10).

11.1 Outline of Functions

The inspector has the following functions:

Data inspection and display: The intended data is analyzed according to the user's instruction and displayed. Any KL1 data item can be analyzed and inspected. To specify the data analysis/display position, the element number, or the car, cdr, or tail parameter can be used depending on the data type.

Variable name and variable pool: A variable name can be given to data which is found during inspection. The variable name is registered in the variable pool and used to refer to the data thereafter.

The inspector can refer to its own local variable pool and the variable pool of the listener who invokes the inspec- tor, and registers data in them.

Output interruption: The user can interrupt output of the inspector during the processing. With the user's instruction received, thereafter, output may be restarted or canceled.

Nested call of the inspector: The inspector can be nested for calling. The nesting depth is indicated at the time of invokation and termination.

Input/output destination switching: When the inspector is invoked, it is given input/output authority from the invokation source process. Normally, input/output authority is returned to the invokation source process when the inspector terminates. Input/output authority is also returned to the invokation source process when the input/output destination of the inspector is switched to a new window. Thereafter, the switched inspector and invokation source processes become independent of each other and run in parallel.

11.2 Data Display Format and Analysis/Display Position Specification Procedure

11.2.1 Data display format

The inspector displays data in the following format:

Integer:

`integer` : Integer value

Floating-point number:

`floating_point` : Floating-point number

Atom:

`atom` : Atom print name

Character string:

`string` : Character string

List:

Element number : Element value

or,

`car` : car value of the list
`cdr` : cdr value of the list
`tail` : tail value of the list

Vector:

Element number: Element value

Module:

`module` : `module$`module name`$`module size

or,

Element number: Element value

Code:

`code` : `code$`module name : code name/number of arguments`$` module size

Undefined variable:

`>Unbound`

Others:

(Data tag) value

11.2.2 Analysis/display position specification procedure

To analyze or display the portion of complex structured data needed by the user, specify the analysis or display position as follows:

Integer: More intensified analysis cannot be done.

Floating-point number: More intensified analysis cannot be done.

Atom: More intensified analysis cannot be done.

Character string: Specifies the element number of a character string.

List: The list consists of car and cdr. The inspector normally refers to the elements in the list and specifies the element number counted from the first element. However, to analyze or display the contents of the list in the form of car or cdr, the 'car' or 'cdr' value of an atom can be specified as the parameter. To analyze or display the last part of the list, the 'tail' parameter is supported.

Vector: Specifies the vector element number.

Module: Specifies the module element number.

Code: More intensified analysis cannot be done.

Undefined variable: At the point of the time, more intensified analysis cannot be done.

Others: More intensified analysis cannot be done.

11.3 Operating Procedure

11.3.1 Invokation

The inspector is invoked with a command input from the top level of the listener or a trace command during tracing. When the inspector is invoked during tracing, the goal deprived of the processor number and module name is passed for inspection at the time of invokation. When the inspector is invoked at the top level of the listener, any KL1 data item specified by the user is passed.

The data to be inspected, which is passed at the time of invokation, is put into the local variable pool of the inspector under the variable name of `ROOT`. This is used as the root of retrieval.

[Example1]Invokating the inspector at the top level of the listener

```
?- inspect(a).
a>
```

[Example2]Invokating the inspector during tracing

```
{trace}
?- qw:queens(4,Ans,1,1,0).
@0@4096 0   qw:queens(4,A,1,1,0)
1  *1) original_list(1,4,B)
2  *2) queen(4,B,A,1,0,1)? i
queens(4,A,1,1, ... )>
```

11.3.2 Command input

When the inspector is invoked, a prompt is displayed, waiting for the user to input a command. On the prompt, the contents of data kept at that stage are displayed briefly.

[Example]Displaying the prompt when kept data is `{0,1,2,3,4,5,6,7,8.9}`

```
{0,1,2,3, ... }>
```

11.3.3 Variable pool

The data item found during inspection can have a variable name. The variable names are registered in the variable pool, and can be used to refer to data thereafter. The inspector can refer to its own local variable pool and a variable pool of the invokation source, listener, and register data in them. Normally, the local variable pool of the inspector is used. When a user refers to the listener variable pool or register data in it, he/she must pad the variable name with the leading "_". The variable name is deprived of "_" when it is registered in the listener variable pool or referred to.

In giving a variable name to data, the variable name which has already been registered cannot be used. This is a difference from unification. If the invokation source, 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).

11.3.4 Setting the environment for output to the window

The inspector has the following five environment parameters for data output:

- Output restriction value related to the structure depth
- Output restriction value related to the structure length
- Output restriction value related to the character string length
- Output restriction value related to the atom print name length
- Parameter related to the character string output format

Four of the above parameters are shared with the listener. The 'print_depth', 'print_length', 'print_string_length', or 'print_atom_length' command are used to refer to or set the parameters for the input/output destination window currently displayed. If a parameter is modified within a nested process, the parameter is not returned to the original value when the process terminates.

In the parameter related to the character string output format, the character print format or code format is specified as the character string output format. The parameter is valid within the inspector. The 'print_string_mode' command is used to refer to or modify the parameter value. The character print format is set as the initial value when the inspector is invoked.

11.3.5 Switching of the input/output destination

When the 'fork' command is executed, a new window is created and the input/output destination of the inspector is switched to the window. The original window waits for the invokation source process to input a command. Thereafter, the switched inspector and the invokation source process are independent each other and run in parallel.

The initial value of the environment parameter for output of a new window created is the same as that of the original window.

11.3.6 Output interruption

A user can interrupt an output operation of the inspector by entering `^C` at the keyboard. Thereafter, the output operation may be restarted or canceled according to the user's instruction. To restart the output operation, a linefeed code or `c` is entered at the keyboard. When `t` is entered, the output operation is canceled and the system returns to the command input level of the inspector.

[Example] 100 elements are output from 20th element of the vector consisting of 1000 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

 ?? Interrupt (h for help): h

   <cr>: continue
      t: return to inspector top level
      c: continue
      h: help

 ?? --------- (h for help): t
{0,1,2,3, ... }>
```

11.3.7 Termination

Enter the 'exit' command. However, when the invokation source listener terminates, the inspector terminates at the same time.

11.4 Command

In a command the user enters, the command name is specified in the first argument, and the needed parameters, in the second and subsequent arguments. A comma is used to partition the parameter arguments. The commands have abbreviations. The entry of an abbreviation also enables the command to run normally.

Some command 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 restriction value related to the structure depth, 4, as the output restriction value related to the structure length, and the character output format, as the character print format.

11.4.1 Basic command

exit [ex]

Terminates the inspector and returns the window input/output authority to the invokation 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 invokation source process.

inspect Term [i]

Suspends execution of the inspector currently invokation and starts up a new inspector.

[Example]Nested inspector call

```
[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}> me
0 : 0
1 : c(A,b(B,0))
2 : a(A,a(B,a(C,a(D, & ))))
3 : 0
{0,c(A,b(B,0)),a(C,a(D, & )),0}> down 2
0 : a
1 : A
2 : a(A,a(B,a(C,a(D,0))))
a(A,a(B,a(C, & )))> exit
>Inspector break level: 0
[put(a,A),put(b,B),put(a,C),put(c,D), ... ]>
```

help Type

Displays the command list of the type specified in **Type** (atom). The type includes the following:

basic: Basic command

insp: Inspection command

env: Environment command

varpool: Variable pool manipulation command

all: All commands

[Hint]The types can be displayed with help/0.

help

Displays the list of the command types which can be specified with help/1.

11.4.2 Inspection command

me M, N

Displays **N** elements from the **M**-th element of the data currently kept.

If the **M** value exceeds the number of elements in the data, an error occurs. If the number of elements to be displayed does not reach **N**, only the elements available are displayed.

A negative integer can also be specified as **N**. In this case, **N** elements are retrieved from the **M**-th element in the minus direction and displayed.

me M

Displays the **M**-th element of the data currently kept. If the **M** value exceeds the number of data elements, an error occurs.

me

Displays elements starting from the 0-th element of the data currently kept by the output restriction value related to the structure length.

If the number of elements that can be displayed does not reach the specified number, only the elements available are displayed.

[Example1]Displaying the contents of vector `{0,1,2,3,4,5,6,7,8,9}`

```
{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
{0,1,2,3, ... }> me 9,-3
7 : 7
```

```
8 : 8
9 : 9
{0,1,2,3, ... }>
```

[Example2]Displaying the contents of list `[0,1,2,3,4,5,6,7,8,9|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 string "`0123456789`"

```
"01234?? "> me
string : "01234 ... "
"01234?? "> me 5
string : " ... 5 ... "
"01234?? "> me 8,10
string : " ... 89"
"01234?? "> me 9,-3
string : " ... 789"
"01234?? ">
```

down M **[emptiness]**

Traces the specified portion of data to one deeper level.

The command can be omitted. The entry of parameter **M** only will do.

[Example]'Down' manipulation on data `p({0,{1,{2,{3,{4}}}}})`

```
p({0,{1, & }})> me
0 : p
```

```
1 : {0,{1,{2,{3, & }}}}
p({0,{1, & }})> down 0
atom : p
p> up
p({0,{1, & }})> down 1
0 : 0
1 : {1,{2,{3,{4}}}}
{0,{1,{2, & }}}>
```

up N [u]

Traces data upward by **N** levels.

up [u]

Traces data upward by one level.

show Term [sh]

Shows the value of the specified data, **Term**, briefly.

monitor M Name [m]

Monitors and displays the condition of instanciation at the level of depth 1 of the specified portion of data. After the display, all commands of the inspector with some exceptions can be executed on the data. (Use the help command to display the excutable command name list.) Then, the deeper levels can be monitored and the condition can be displayed.

Any character string can be specified in **Name** as a remainder of the data (The inspector has no concern in the character array.).

monitor M [m]

Monitors and displays the instanciation condition at the level of depth 1 of the specified portion of data. After the display, all inspector commands with some exceptions can be executed on the data. (Use the help command to display the executable command name list.)

[Hint]Currently, the monitor condition is displayed in another window, which causes inconveniences to the user. This problem will be resolved in the future.

[Example]Displaying the monitor until variable `A` is instanciated to `[empty(no)]`

```
p(A)> monitor 1 list
p(A)>
```

Monitor window:

```
list == [A|B] ? m car message
list == [A|B] ? m cdr list
list == [] ?
message == empty(A) ? m 1 YorN
message == empty(A) ?
YorN == no ?
>Monitoring window will be closed ?
```

history N

Displays the history of data tracing for the past **N** operations. The position of the data traced with the 'down' command and the data list at that time are displayed.

history

Displays the history of data tracing for the past several operations. The position of the data traced with the 'down' command and the data list at that time are displayed. The output restriction value related to the structure length is used as the number of trace operations.

[Example]Displaying the history of data tracing when 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 to be inspected. The contents of variable `ROOT` registered in the local variable pool of the inspector are replaced with the new inspection data.

11.4.3 Environment command

print_depth D [pd]

Sets the output restriction value related to the structure depth in **D**.

print_depth [pd]

Refers to or sets the output restriction value related to the structure depth.

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_length L [pl]

Sets the output restriction value related to the structure length in **L**.

print_length [pl]

Refers to or sets the output restriction value related to the structure length.

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 restriction value related to the character string length in **L**.

print_string_length [psl]

Refers to or sets the output restriction value related to the character string length.

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 restriction value related to the atom print name length in **L**.

print_atom_length [pal]

Refers to or sets the output restriction value related to the atom print name length.

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 character string output format. The output format is divided into two types: character print format (character) and code format (code). The two types of formats are toggled. The initial value displayed when the inspector is invoked is the character print format.

[Example]Output when the character string output format is changed

```
"ab"> me
string : "ab"
"ab"> psm
 Print mode for string: character => code
"ab"> me
string : string#{9057,9058}
"ab"> psm
 Print mode for string: code => character
"ab">
```

print_mode [pm]

Changes the data output format. The output format is divided into the general format (general) and the format displaying the tag and value (tag-and-value). The two types of formats are toggled. The initial value displayed when the inspector is invoked is the general format.

environment [env]

Displays the environment variable values 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>
```

11.4.4 Variable pool manipulation command

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 cannot be used. This is a difference from unification.

list Type [ls]

Lists the variable names and data of the variables registered in the inspector local variable pool and listener variable pool. Any of local (the variables contained in the inspector local variable pool only are handled), listener (the variables contained in the listener variable pool only are handled), or all (all variables contained in both the inspector local variable and listener variable pools are handled) can be specified in **Type**.

The variable names of variables registered in the listener variable pool are displayed with the leading -s.

list [ls]

Lists the variable names and data of variables registered in the inspector local variable and listener variable pools.

The variable names of variables registered in the listener variable pool are displayed with the leading "_"-s.

Period return

When "." and "cr" are entered, all 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

- Substitute current data a for variable A with the 'name' command.
- Use the list command to display the variable 'list'. Variables A and T are the user-defined local variables of the inspector, ROOT, a variable indicating the data passed when the inspector is invoked, and _L and _Table, variables registered in the listener variable pool.
- When the '.' command is executed, all user-defined inspector local variables are released.

```
a> name A
a> list
 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> .
 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>
```

Chapter 12

Administration

This chapter explains how to booting or shut down PIMOS, and describes the environment file related to the booting.

12.1 Booting and Shutdown

12.1.1 Booting procedure

This subsection describes how to start up PIMOS. The starting up procedure varies depending on whether or not a PIMOS environment file (refer to 12.2.1) resides. The procedures are as follows:

When an environment file resides:

1. Use the CSP of the multi-PSI system to enter as follows:

   ```
   $go/pimos↩
   ```

   it's not a arrow, it's a signal that shows a CR code of keyboard where, the underlined value designates the user input. The multi-PSI system is booting automatically in the auto-boot mode. For the environment setting up and CSP operation, refer to the "Multi-PSI Console System Operation Manual".

2. An initial task is invoked according to the environment file. At this stage, the user environment set-up program (refer to 12.2.2) is usually invoked. The subsequent steps are explained on the assumption that the program has been invoked.

3. The user environment input window appears displayed under the CSP Monitor Panel for the multi-PSI system.

4. Enter the registered user name in the window (assuming it to be "ICOT") as follows:

   ```
   User Name>>ICOT↩
   ```

5. Then, the task corresponding to the user (example, shell) is starts.

   [Caution]PIMOS does not support a logic similar to logout at present. Therefore, the user environment input window is always displayed.

When an environment file does not reside:

1. Use the CSP of the multi-PSI system to enter as follows:

```
$go/pimos↩
```

When the autoboot mode is set the booting process is automatically performed.

2. A window appears under for the multi-PSI system. The window is called as a PIMOS console window. A general user uses the window at the time of booting or shutdown only.

3. In the console window, enter as follows:

```
Module>>shell↩
Pred>>go↩
PE>>↩
FEP>>↩
```

Then, the shell is started.

4. The mouse cursor is changed to the one for window creation. Create a window for the shell. Every user program runs on the shell. For use of the shell, refer to chapter 8.

12.1.2 Shutdown procedure

The shutdown behavior is different depending on whether the environment file exists or not.

When the environment file exists:

1. Use the shell to enter as follows:

```
prompt>shutdown↩
```

Namely, execute shutdown:go.

2. Then, the window appears. It is asked for confirmation if shutdown is intended. Enter the response.

```
Really(yes/no) ?yes↩
```

[Caution]To cancel the shutdown procedure, enter no.

3. With this operation, all tasks run on PIMOS terminate. The resource amount consumed since PIMOS was booted is indicated in the CSP window.

4. Finally, CSP asks whether or not to terminate the multi-PSI system. To terminate it, enter as follows:

```
 Do you want to exit from CSP ? [Y/N] :y↩
```

Then, the CSP window goes out and the system terminates. To restart the multi-PSI system, enter as follows:

```
 Do you want to exit from CSP ? [Y/N] :n↩
$boot↩
```

When the environment file does not exists:

1. Use the PIMOS console window (a window created first when PIMOS is booted) to enter as follows:

```
Module>>!exit↩
```

2. With this operation, all tasks run on PIMOS terminate. The resource amount consumed since PIMOS was booted is indicated in the CSP window.

3. Finally, CSP asks whether or not to terminate the multi-PSI system. To terminate it, enter as follows:

```
Do you want to exit from CSP ? [Y/N] :y↩
```

   Then, the CSP window goes out and the system terminates. To restart the multi-PSI system, enter as follows:

```
Do you want to exit from CSP ? [Y/N] :n↩
$boot↩
```

12.2 How to Set Up Booting Procedure

When PIMOS is started up, a user-specified task can be invoked. The starting up procedure is as follows:

12.2.1 How to set the PIMOS environment

When PIMOS starts up, if the environment file exists in FEP of FEP number 0, a task is invoked according to the contents of the file.

```
pimos:pimos.conf
```

The contents of the environment file are as follows.

with_console: Specifies use of the PIMOS console function. This data item must be set normally. If another with_console or without_console appears in the middle of the file, it is ignored.

without_console: Do not specify this item, because it is intended for engineers who develop PIMOS.

system (Fep, PE, Name, Goal): Specifies invoking of the specified task. The arguments have the following meanings:

Fep: Specifies the default FEP number (integer) for a task. This value is set in the environment variable, task:node.

PE: Specifies the PE number (integer) under which the initial goal of a task runs.

Name: Specifies a task name (string).

Goal: Specifies the initial goal of a task in the format of "module_name:predicate".

[Caution]if the predicate argument contains an error, the task is not invoked. The item must not contain a variable. (Be sure that the output variable does not contain a value including an undefined variable which makes unification a success.)

12.2.2 Contents of the environment set-up file for release

```
with_console
system(0, 0, "login", users:go).
```

Meaning: The PIMOS console function is used. The user environment set-up program, users:go, is invoked as an initial task by specifying number 0 as FEP, number 0 as PE, and login as the task name.

12.3 User Environment Set-up Program

12.3.1 Setting up the user environment

As mentioned in the previous section, when users:go is executed, the user environment set-up program is invoked. The following describes the set-up procedure using the previous example:

1. Boot PIMOS according to the environment file mentioned previously.
2. The user environment set-up window (entitled as USERS) is appears in FEP specified in the environment file.
3. When the user name registered in the user environment file of PIMOS is entered, the corresponding task starts. For example, when the default user environment set-up file is used, to invoke the shell, enter as follows:

```
User Name>>ShellUser↩
```

[Caution]PIMOS does not support a logic similar to logout at present. Therefore, the user environment set-up window is always displayed.

12.3.2 User environment file

The user environment set-up program invokes a task in accordance with the contents of the user environment file in FEP`#0` every time a user name is entered.

```
pimos:pimos.users
```

The contents of the user environment file is describes in the following format:

```
user(UserName, Tasks )
```

The data items specify the invokation of a task corresponding to the user name entered in the user environment set-up window. The arguments have the following meanings:

UserName: User name (string). When this name is used to set the environment, this character string is set in environment variable, 'task:user'.

Tasks: Item list written in the following format:

```
system(Resv, Fep, Dir, PE, Name, Goal)
```

The arguments have the following meanings:

Resv: * is specified all the time. If any other symbol is used, the operations are not always the one that you intend.

Fep: Specifies the default FEP number for a task with an integer or *. If * is entered, the FEP number set in the user environment set-up program is used. The value is set in environment variable, 'task:fep'.

Dir: Specifies the initial directory name (string) of the program.

PE: Specifies the PE number, under which the initial goal of a task runs, with an integer or *. If * specified, the PE number set in the user environment set-up program is used.

Name: Specifies the task name (string).

Goal: Specifies the initial goal of a task in the format of "module name: predicate".

[Caution]If the predicate argument contains an error, the task is not invoked. The item must not contain a variable. (Be sure that the output variable does not contain a value including an undefined variable which makes unification a success.)

12.3.3 Contents of the user environment file for release

```
user("shutdown",[system(0,0">sys>user>pimos",0,"shutdown,shutdown:go)]).
user("ShellUser",[system(*,*,">sys>user>ShellUser",*,"shell",
                         shell:go(xx,xx,xx,xx,xx))]).
user("ListenerUser",[system(*,*,">sys>user>ListenerUser",*,"listener",
                         listener:go(at(xx,xx),char(xx,xx),xx))]).
user("ICOT",[system(*,*,">sys>user>ICOT",*,"shell"
                         shell:go(xx,xx,xx,xx,xx))]).
```

Meaning: When the environment is set with the user name, ShellUser, the shell is activated. When ListenerUser is entered, the listener is invoked. When shutdown is specified, shutdown:go is run to terminate PIMOS.

Appendix A

How to create a string I/O device on FEP

As a string I/O device, you can create your own device. String I/O device is written in ESP and runs on FEP. This appendix describes how to define such a device using ESP.

A device name for string I/O operations is generally specified in the format of "package_name##class_name". Therefore, the class corresponding to the device name must be defined. The class must meet the following rules.

[Hint]The following rules are valid only for the class corresponding to a device name (class of a string I/O device and the other classes except the inherited mpsicsp##as_pimos_string_io_device). For a device called by the string I/O device, any class can be defined.

[Hint]The classes are automatically loaded when the devices are created, once their templates are saved.

Inherited class: To obtain the function as a PIMOS device (data conversion between KL1 and KL0), the class, mpsicsp##as_pimos_string_io_device, is inherited.

Method or slot which must not be defined: The following methods and slots are already defined in the class, mpsicsp##as_pimos_string_io_device. Therefore, you must not define it.

Class method: :creat_string_io_device/2

Instance method: :goal/1, :io/5, :kill_myself/1, :die/1, :check_instruction/4, :refute_message/2

Instance attribute slot: self, string_io_device, status, message, simpos_device_stream, node_manger, node_device_id, error_receiver, attention_address, abort_status, abort_address

Method defined: The following instance method is defined:

:io(Instance, Command, FromMulPString,^ToMulPString)

The arguments are as follows:

Instance: Instance object of the device

Command: Integer. ID indicating the type of an I/O instruction.

FromMulPString: 16-bit string. Data sent from the Multi-PSI (KL1 side). Usually, data needed for execution of an I/O instruction is entered.

ToMulPString: 16-bit string. Data sent to the Multi-PSI (KL1 side) after an I/O instruction is executed. Normally, the result of executing an I/O instruction is entered.

This method defines that an I/O instruction specified in Command is to be executed. When the execution fails for some reason, the method should be 'fail' so that the status of message from KL1 indicates "abnormal". If no data is needed to be sent to the KL1 with an I/O instruction, 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 general I/O instructions.

[**Hint**]Otherwise, assuming that the device is initialized when it is created, if the creation fails, it cannot be determined whether the registration of that device has not been done or that the initialization of the device fails.

Termination processing: A process of the string I/O device terminates when the stream of KL1 is closed. If you want to process any data before termination, you must define a instance method :cut_off/1 for that purpose.

Example of Programming

The following is a example of device program. In this program, commands are entered in the menu window and echoed back to another window.

ESP program on FEP

```
class string_io_example has
nature

mpsicsp##as_pimos_string_io_device ;
instance
attribute
echo_window, read_menu ;
:io(Obj, 0, FromMulPString, ToMulPString) :- !,
    :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) ;
:init(Obj, Input, "") :-
    ItemList = [
  {"test", common,0,_},   {"analysis",common,1,_},
  {"manipulate",common,2_}, {"check",common,3_},
  {"exit",common,4,_}],
    :create(#menu, [title("MENU"),items_list(ItemList)], Menu),
    :activate(Menu), Obj!read_menu := Menu,
    string_element(Input, 0, Sizex),
    string_element(Input, 1, Sixey),
    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?men(Obj, Input, String) :-
       W = Obj!echo_window,
       :putl(W,Input), :flash(W),
       :read(Obj!read_menu, Type),
       new_string(String, 0, Type) ;

    :write_to_window(Obj, Input, "") :- !,
       W = Obj!echo_window,
       :putl(W, Input), :flash(W) ;
```

```
    :cut_off(Obj) :- !,
W = Obj!echo_window,
:putl(W, "Terminate Device"), :flash(W) ;

end.
```

KL1 program on the Multi-PSI

```
:- 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#"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) :- true |
execute(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(4, string#"manipulate start ...end", FromFEP)|STGIO2],
```

Appendix B

Performance Meter

The performance meter is a tool to observe which processor run fully and which processor is idle. The following sections explain how to use the performance meter.

[Caution]The performance meter does not run correctly under Pseudo-Multi-PSI.

B.1 Start up

The performance meter is called as a command from the shell. pmeter

pmeter

Starting up the performance meter.

B.2 Setting up

After the performance meter has been activated, a parameter setting window is appeared. Enter the following data:

Measurement interval: Specifies the measurement interval value with an integer in msec. units. Usually 2000. (If a value less than 2000 is specified, the results are not always correct.)

FEP number setting the display window: Any integer of 0 to 3. For the system incorporating a slave FEP, the slave FEP number (1 to 3) is entered normally. Enter 0 for the system which incorporates a master FEP only.

FEP number setting the measurement timer: Any integer of 0 to 3.

For the system incorporating two or more slave FEPs, a slave FEP different from that for the display window is usually assigned to reduce a measurement error. When one slave FEP is incorporated, the same slave as that for the display window is assigned. When only a master FEP is incorporated, enter 0.

B.3 Display Start

After the parameter values have been set, the setting window goes out. Then the performance meter program is automatically loaded and the window is set up. After that, when a user program is run, the performance meter displays the execution status.

[Caution]The performance meter uses a KL1 goal with a lowest priority for measurement. Therefore, if the lowest priority is set to a user program and run with the performance, the data that you can see with performance meter has error.

Appendix C

PROTOCOL DEFINITIONS

C.1 Outline

To issue a request to PIMOS, a user program sends a message via the stream. In PIMOS, a specification language (hereafter, protocol definition language) is used to define protocols for messages. A part of PIMOS is created from protocol definitions written in this language using the computer for this language. The protocol definition languages are outlined and the practical protocol definitions are presented in the following sections.

C.2 Protocol Definition Language

C.2.1 Grammar

The grammar of the protocol definition languages is as follows:

```
definition ::= name[(formals)] = [inheritance] body.

inheritance ::= {super &}
      super ::= invocation
       body ::= [sign] (invoation | {enumeration})
 invocation ::= name [(actuals)] | parameter
enumeration ::= pattern {; pattern}
    pattern ::= atom [(actuals)]
    formals ::= parameter {, parameter}
    actuals ::= body {, body}
  parameter ::= name
       sign ::= + | -



|:    Select either of the parameters.
{ }:  Repeat the parameter zero or more times.
[ ]:  Omissible
```

C.2.2 System Defined Protocol

The system defined protocol includes atom, integer, string, code, module, ground, any, list(p), vector (p), vector ([p1, p2, —, 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 instanciated. 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, — pN]). vector(p) is defined that a vector of any length and the protocol of each element follows the p. vector ([p1,

p2, — pN]) is defined that a vector of size N and that p1, p2, — pN represent the protocols of elements.

C.2.3 Description of grammar

This section makes brief explanation about the grammar mentioned in the previous section.

Protocol definition: A protocol can be defined in the following format:

```
Protocol name = Protocol definition
```

In the protocol definition, a message pattern or a protocol name to be referred to can be described. The message pattern is encircled with { } and partitioned with ;.

Inheritance

Multiple inheritance of protocols is available when the protocol names are specified.

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 referred to is different from the one the person has set.

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 the actual value corresponding to each parameter when it is referred to. The value is passed to the right hand side and expanded to be an actual definition.

C.3 How to read the protocol definition

This section explains how to read the 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) ;
        suicide(-status) ;
        abort(rscpath, -status) ;
        number_of_feps(-status(integer)) ;

        setenv(envname, value, -status) ;
        getenv(envname, -status(value)) ;
        getenv(-status(envs)) ;
        unsetenv(envaname, -status) ;

        window(fepnodeno, -rscstream(winreq_command)) ;
        file(fepnodeno, -rscstream(filereq_command)) ;
        timer(fepnodeno, -rscstream(timerrreq_command)) ;
        string_io(fepno, nodeno, -rscstream(strioreq_command)) ;

        window(fepno, nodeno, -rscstream(winreq_command)) ;
        file(fepno, nodeno, -rscstream(filereq_command)) ;
        timer(fepno, nodeno, -rscstream(strioreq_command)) ;
```

```
        string_io(fepno , nodeno, -rscstream(strioreq_command)).

        window(-rscstream(winreq_command)) ;
        file(-rscstream(filereq_command)) ;
        timer(-rscstream(timerreq_command)) ;
        string_io(-rscstream(timerreq_command)) ;
        module(-rscstream(modulereq_command)) ;
        atom(-rscstream(atomreq_command))
        }.

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 or the top-level device of the resource tree. Messages can be sent to the general_request_device via the stream. The right hand side of stream(genreq_command) indicates that 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))

setenv(envname, value, -status)

represent a message for setting a value in an environment variable, having 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.

window(-rscstream(winreq_command))

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 C.1: Message protocals corresponding to target devices

Target of messages	Name of protocal
general_request_device	genreq_command
window_request_device	winreq_command
file_request_device	filereq_command
timer_request_device	timereq_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( ) is a parameter. When the protocol is referred to, a value is passed. In this example, the value represents winreq_command. Then, any of the following values is returned to window(-rscstream(winreq_command)):

- normal(-stream(winreq_command),-abort,attention)
- abnormal
- aborted

The first argument in normal( ) is window_request_stream, the second argument, an abort line, and the third argument, an attention line. The message which is defined as 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 abort line, while normal( ) is returned from PIMOS to the user.

Thus, reading protocol definitions leads to understand message protocols.

To help check the definitions of messages, C.1 lists the destinations of messages and the protocol names used in the protocol definitions.

C.4 Definition of Protocal

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% general_request_device (Top Level)
%
% (1)inquiring the status
% (2) inquiring and setting of parameters
% (3) getting request_stream to each device
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

genreq_filter = stream(genreq_command).

genreq_command = ask_task_command & {
        execute (execode, argv, taskname, minprio, maxprio, tag, initrsc, -task);
        execute (execode, argv, taskname, minprio, maxprio, tag,-task);
        suicide)-status);
        abort(rscpath,-status);
        number_of_feps(-satus(integer));
```

```
        setenv(envname, value, -status);
        getenv(envname,-status(value));
        getenv(-status(envs));
        unsetenv(envname,-status);
        window(fepnodeno,-rscstream(winreq_command));
        file(fepnodeno,-rscstream(filereq_command));
        timer(fepnodeno, -rscstream(timerreq_command));
        string_io(fepnodeno,-rscstream(strioreq_command));
        window(fepno nodeno ,-rscstream(winreq_command));
        file(fepno, nodeno, -rscstream(filereq_command));
        timer(fepnodeno, -rscstream(timerreq_command));
        string_io(fepnodeno,-rscstream(strioreq_command));
        window(-rscstream(winreq_command));
        file(-rscstream(filereq_command));
        timer(-rscstream(timerreq_command));
        string_io(-rscstream(strioreq_command));
        module(-rscstream(modulereq_command));
        atom(-rscstream(atomreq_command))
}.

ask_task_command = {
        id(-status(integer));
        absolute_id(-status(rscpath));
        status(rscpath, rsckind, verbosity,-status(rscinfo))
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Messages relevant to the abort line and attention line
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

aa_command = {
        resent(-status(-abort, attention, msgid));
        next_attention( -status(attention));
        resend( msgid, -status);
        cancel( msgid, -status)
}.

% task

task = {
        normal( -control, -report );
        abnormal;
        aborted
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Messages relevant to:
%
%       (1) window_request_device
%       (2) file_request_device
%       (3) timer_request_device
%       (4) string_io_request_device
%       (5) module_request_device
%       (6) atom_request_device
%
```

```
%%%
```

```
        flush( -status)
}.

% Messages relevant to the window

window_command = input_command & output_command & aa_command & {
        beep( -status) ;
        set_size( winsize, -status) ;
        set_position( winpos, -status) ;
        set_title( title, -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))
}.

% file common message

cfile_command = {
        pathname(-status(pathname))
}.

% Messages relevant to the input file

rfile_command = cfile_command & input_command & aa_command & {
        end_of_file( -status(eof))
}.

% Messages relevant to the output file

wfile_command = cfile_command & aa_command &
        output_command.

dir_command = cfile_command & aa_command & {
        listing( wildcard, -status(list(filename))) ;
        delete( wildcare, -status) ;
        undelete( wildcard, -status) ;
        deleted( wildcard, -status(list(filename))) ;
        purge( wildcard, -status) ;
        expunge( -status)
}.

% Messages relevant to the timer

timer_command = aa_command & {
        get_count( -status(count)) ;
        on_at( count, -status(now)) ;
        on_after( count, -status(now))
}.
```

```
% Messages relevant to the string_io device

strio_command = aa_command & {
        iod( io_command, tofepstring, -status(fromfepstring))
}.

% Messages relevant to the module

module_command = aa_command & {
        get_code( modulename, predname,arity,-retrieval_status(code,ground)) ;
        get_module( modulename, -retrieval_status(module,atom)) ;
        update_module(modulename, module, -status) ;
        remove_module( modulename, -status) ;
        get_all_modules( -status(moduleinfo))
}.

% Messages relevant to the atom

atom_command = aa_command & {
        get_atom_name(atom, -retrieval_status(atomname,dkey)) ;
        intern(atomname, -status(atom))
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Messages to the task
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% user -> PIMOS (messages flowing to the control stream)

control = stream(control_command).

control_command = ask_task_command & aa_command & {
        stop ;
        allow_resource_report ;
        statistics ;
        abort(rscpath, -satus) ;
        abort(-status) ;
        add_resource(hrsc,lrsc)
}.

% PIMOS -> user (messages flowing the report stream)

report = -stream(report_command).

report_command = {
        statistics_started ;
        resource_report_allowed ;
        resource_added;
        resource_low ;
        aborted ;
        terminated ;
        exception(exckind, excinfor, -ncode, -nargv) ;
        statistics(hstat,lstat) ;
        return_resource(hrsc,lrsc) ;
```

```
        stopped( -reason_stream, -cancel)
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Definitions for data that has become a message's argument
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscpath = {[rscid | rscpath];
        * ;
        []
        }.
rscid = integer & { * }.
        rsckind = rscname & {
        * }.
rscname = {
        general_request ;
        window_request ;
        file_request ;
        timer_request ;
        string_io_request ;
        module_request ;
        atom_request ;
        task ;
        window ;
        file ;
        timer ;
        string_io ;
        module ;
        atom ;
        director
}.

verbosity = {
        brief ;
        detail
}.

reason_stream = stream(atom).
rscinfo = list(rscinfo2).
rscinfo2 = brief_info & detail_info.
brief_info = vector([rscid, rscname, name, bchildren]).
detail_info = vector([rscid, name, task_status, limit, consume, dchildren]).
task_status = vector([atom, stops]).
stops = list(stopreason).
stopreason = {stop(atom)}.
bchildren = list(brief_info).
dchildren = list(detail_info).
msgid = integer.
winpos = {
        mouse ;
        at(x,y)
        }.
winsize = {
        mouse ;
        char(charnum, linenum)
```

```
        }.
moduleinfo = list(moduleinfo2).
moduleinfo2 = vector([atom, module]).
now = {wake_up; aborted}.
attention = {attention : []}.
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.
count = integer.
io_command = integer.
tofepstring = string.
fromfepstring = string.
modulename = atom.
predname = atom.
arity = integer.
dkey = atom & string.
atomname = string.
limit = vector([hrsc,lrsc]).
consume = vector([hrsc,lrsc]) & {unkown}.
initrsc = vector([hrsc,lrsc]).
name = string.
eof = {yes ;
        no
```

```
        }.
envs = list(env).
env = vector([envname,value]).
fepnodeno = integer & string.
nodeno = string.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% User defined protocols with parameters
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscstream(p) = {
        normal(-stream(p), -abort, attention) ;
        abnormal ;
        aborted
}.

status = {
        normal ;
        abnormal ;
        aborted
}.

status(p) = {
        normal ;
        abnormal ;
        aborted
}.

status(x,y) = {
        normal(x,y) ;
        abnormal ;
        aborted
}.

status(x,y,z) = {
        normal(x,y,z) ;
        abnormal ;
        aborted
}.

retrieval_status(x,y) = {
        normal(x) ;
        abnormal ;
        aborted
        non_existent(y)
}.

eofstatus(p) = {
        normal ;
        abnormal(p) ;
        aborted
        end_of_file
}.
```