ICOT Technical Memorandum: TM-0720

TM-0720

PDSS Manual (Version 1.64e)

by
K. Taki & S. Uchida

May, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

PDSS Manual

(Version 1.64e)

20/2/89

Institute for New Generation Computer Technology

Fourth Laboratory

Contents

1 What is PDSS

PDSS, which stands for PIMOS Development Support System, is a KL1 system to develop the PIMOS. PDSS is widely compatible with the KL1 system found on Multi-PSI V2, besides implementation details, execution speed, etc. The main differences are enumerated below. This is still an approximate list, as the Multi-PSI V2 specifications are not fixed yet.

- Some of the functions implemented through software (e.g atom management) are treated by the compiler. Some atom related operations are available as built in predicates.
- Code management is done by compiler.
- The only resource known by PDSS is the number of performed reductions.
- The I/O device stream has a different form.
- Because PDSS is a single processor system, there is no processor pointing function for process dispersion.
- The priority calculations rules are also a little bit different.

Another function of PDSS is to provide tools for the development of parallel programs. To this end, PDSS has been written in a style which ensures portability and it will be installed onto various UNIX[1] systems. We tried to build PDSS as a handy development tool. We expect it to be improved along the development of PIMOS.

PDSS consists mainly of two parts : one is the language processing system which executes KL1 and the other is the user interface system, called Micro PIMOS. Micro PIMOS is a single user, single task operating system which provides I/O and code management functions to its user. Its description is held in chapter 3. Figure 1 shows PDSS configuration.

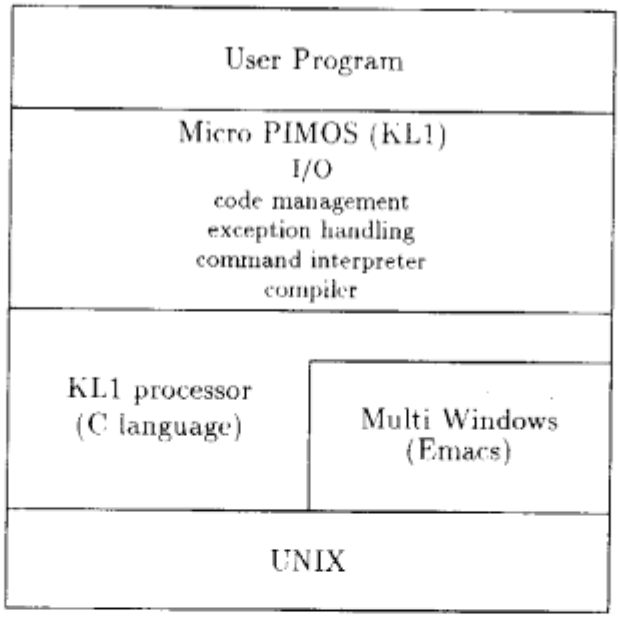


Figure 1: PDSS configuration

In figure 1, we see that a multi-window environment is provided through the GNU-Emacs full-screen editor. Its library has been written in Emacs-LISP.

In PDSS, I/O and code magement functions use a special built-in stream, called device stream. Specifications of this stream can be found in Appendix-1 and Appendix-2. Anyway, the average user doesn't have to use device stream directly, as most necessary facilities are provided in Micro-PIMOS libraries.

[1] UNIX is a trademark of Bell Laboratories.

2 KL1 Language Specification

The language specification of the KL1 dialect executable on PDSS lies in this chapter. Note that there may be differences between this specification and ones found on other systems, such as Multi-PSI V2.

2.1 Outline

KL1 is a language based on GHC (Guarded Horn Clauses) which moreover embeds some extensions related to OS description, modular programming, etc. KL1 also has some restrictions, due to implementation limitations. Its main characteristics are now described :

Sequentiality of guard

Unification of head parameters and execution of guard goals are performed sequentially, from left to right. In the following example, suspension occurs until variable X is instantiated. Note that the following predicate does not fail.

```
Goal:     ?- p(a,X,b).
Clause:   p(a,c,d) :- true | true.
```

Guard restrictions

Only built in predicates can be used within the guard. These predicates are described in section 2.6.

Equality of variables

Equality of unbound variables is not checked in the guard part. Suspension occurs in the following program, until variables X and Y are instantiated, independently from the execution order of the goals of the topmost clause.

```
Goal:     ?- X=Y, p(X,Y).
Clause:   p(A,A) :- true | true.
```

Module functionality

Clustering clauses in several modules allows modular compilation and debugging. In the current version, to each file corresponds a unique module.

Sho-en

An original functional unit, called Sho-en, has been introduced. It is possible to control the execution priority and resource allocation of each Sho-en. OS itself is constructed as such a Sho-en.

Exception handling

Handling of exceptions occuring during the execution of a program is described in KL1, using Sho-en and second-order predicates.

Failure handling

All failures are considered as exceptions within KL1 and execution of a program can be resumed using exception handling facilities.

2.2 Sho-en

A Sho-en is the minimum unit of resource management, priority management and exception handling which exists in the language. Two streams, called control and report streams, are connected to each Sho-en. The control stream is used to control the Sho-en, and can carry various commands. The report stream carries information and requests coming from the Sho-en. Users of Sho-en can handle exceptions if they write programs interpreting the information from the report stream.

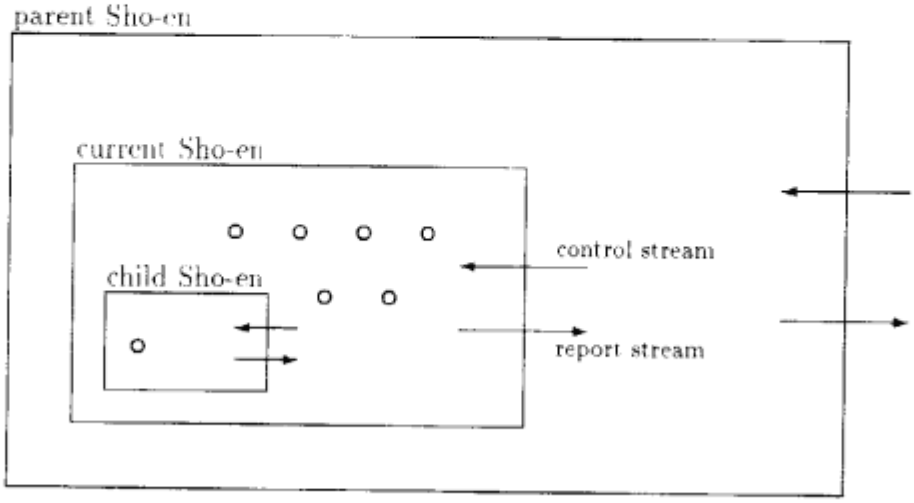


Figure 2: An instant picture of some Sho-ens

Resource management functions

The resource managed by PDSS is the number of performed reductions. It can be seen as a rough mesure of the computing time and memory usage. For all goals which belong to the same Sho-en, it is possible to specify the maximum number of reductions. By default, the system assigns the maximum possible number of reductions. When the Sho-en is generated, i.e when it starts, this amount (or the default amount) is attributed. When reduction allocation is exhausted during the course of execution, an exception **`resource_low`** is inserted in the report stream of the Sho-en. It is possible to increase the reduction resource via an **`add_resource(R)`** command in the control stream, as explained later.

The resource consumption control is performed in a discrete manner : independently from the maximum number of allowed reductions, there is a system dependent reduction granularity according to which control is exerted. Typically, a few thousands reductions. Resource control can be seen as a recursive allocation process : when a Sho-en starts, it is allocated, say, 2000 reductions. When this number is exhausted, a 2000 reductions resource is subtracted from the parent Sho-en and added to the current Sho-en. This may trigger a recursive process, during which reduction allocation is eventually done at the debts of some grand-father Sho-en. During this process, and only at this time, the maximum allocation limit is checked.

Priority management

Another function of the Sho-en is priority management. Each Sho-en holds a record of upper and lower priority bounds, for inner goals. Goals cannot be executed with a priority beyond upper bound and below lower bound. Priority specification is described in section 2.3.

2.2.1 Sho-en Generation

Sho-en generation is performed using the "Sho-en" system module, which contains the predicates execute/7 and execute/8. Below, **`code`** is a three elements vector : **`{module-name,predicate-name, number-of-args}`**. **`arguments`** is a vector with arguments of the goal. (In Multi-PSI V2, **`code`** data type is used for the **`code`**

argument.)

```
execute(module-name, predicate-name, argument, minimum-priority,
        maximum-priority, tag, control, report)

execute(code, argument, minimum-priority,
        maximum-priority, tag, control, report)
```

Above, **minimum-priority** and **maximum-priority** hold the values used to calculate priority bounds within which goals are executed as shown in 3. **Tag** is a bit mask used to filter the exceptions received by the Sho-en. Control stream is unified with **control**, and report stream is unified with **report**. The initial state of the generated Sho-en is **suspend**, and the allowed reduction count is not set.

【ex】 `'Sho-en':execute({primes,do,3},{1,300,PRIMES},0,2,-1,CONTROL,REPORT)`

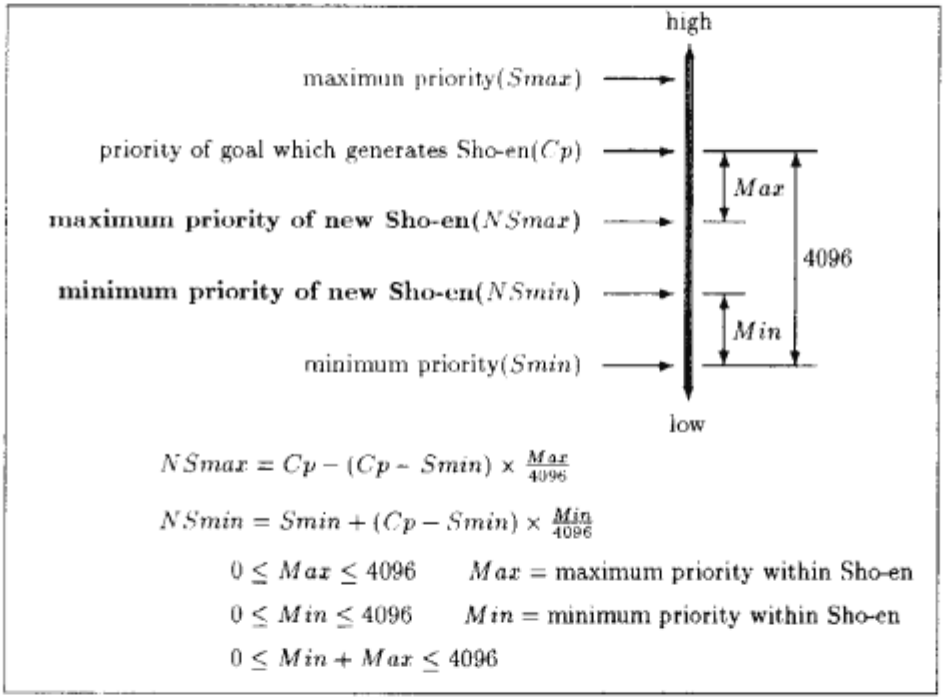


Figure 3: Calculation of Sho-en Priority

2.2.2 Control Stream

Below are the commands which can be inserted in the control stream. When the control stream is closed, the Sho-en is abandonned. Conversely, if the user does never close the stream, execution of Sho-en itself never stops.

start
Activates goal execution in the Sho-en.

stop
Suspends goal execution. Previous command causes execution to resume.

abort

Aborts goal execution once and for all. In particular, start command cannot resume execution.

add_resource(Reduction)

Adds `Reduction` to the current number of allowed reductions. A negative integer means infinity.

allow_resource_report

This command is an answer to the exception `resource_low`. This exception cannot be reported again until this command is inserted.

statistics

Asks statistics about the Sho-en. Information is inserted in the report stream.

2.2.3 Report Stream

The following information can be found in the report stream.

Acknowledgment messages to control stream commands :

Here are the responses to commands put in the control stream.

started

`Start` message has been received.

stopped

`Stop` message has been received.

aborted

`Abort` message has been received.

resource_added

`Add_ressource` message has been received.

resource_report_allowed

`Allow_resource_report` message has been received. Exception `resource_low` can be reported again after this message.

statistics_started

`Statistics` message received. The statistic information itself is reported once collected.

Status information

Here is the information reported whenever Sho-en status changes.

terminated

Execution of Sho-en has finished. If `abort` had been sent previously, this message indicates that the execution has been aborted. Otherwise, it indicates success of all goals.

resource_low

The number of performed reductions is close to the maximum allowed amount, or this amount is not sufficient. When this exception occurs, Sho-en state becomes `suspend`. No other `resource_low` report can occur before that `allow_resource_report` is inserted in the control stream.

Statistic information

Here, we get statistic information about the Sho-en, whenever collection has been done.

statistics(Info)

Unifies the statistic information with `Info`, which is a vector of one element, indicating the number of reductions performed. This number includes reductions performed by children Sho-ens.

Exception information

Here is the description of exceptions which can be reported by a Sho-en. Some of them can specify the handling process for the exception. These have **NewCode** and **NewArg** arguments. If an exception condition is detected by PDSS, the following goal is generated within the Sho-en to handle the exception. Then, system waits for the unification of this goal with **NewCode** and **NewArg**.

```
exception_handling({Module,Predicate,Arity},Argv) :- true |
    apply(Module,Predicate,Argv).                % goal execution
exception_handling(□,_) :- true |
    true.                                        % nothing done
```

failure(Code, Argv, NewCode, NewArgv)
All guards of some goal failed. **Code** is the code of that goal, and **Argv** its arguments. The new code and arguments of the goal chosen by the user, in place of the failing goal which caused exception, should be unified with **NewCode** and **NewArgv**.

unification_failure(X, Y, NewCode, NewArgv)
Unification of **X** and **Y** failed in the body of some clause in the Sho-en. **NewCode** and **NewArg** have the same meaning as above.

exception(ExceptionCode, OpCode, Argv, NewCode, NewArgv)
Exception occured in a built-in predicate, within the body of some clause. **ExceptionCode** is a positive integer which indicates the type of exception. **OpCode** is a positive integer which indicates the operation code of the built-in predicate and **Argv** is the argument. **NewCode** and **NewArg** have the same meaning as above.

exception(ExceptionCode, Code, Argv, NewCode, NewArgv)
Exception occurred in Sho-en. **ExceptionCode** has the same meaning as above. **Code** is the code of the goal which caused exception and **Argv** is the argument. **NewCode** and **NewArg** have the same meaning as above.

exception(ExceptionCode, Number, NewCode, NewArgv)
ExceptionCode has the same meaning as above. **Number** is an integer indicating the device type which caused exception. **NewCode** and **NewArg** have the same meaning as above.

raised(Type, Info, NewCode, NewArgv)
Built-in predicate raise/3 has been executed in the Sho-en. **Type** and **Info** are the parameters used in this predicate.

deadlock(Type, Caller, LockedGoals)
Deadlock has been detected in Sho-en. **Type** is an integer indicating deadlock type. **Caller** is the code of the predicate which causes deadlock or garbage congestion. **LockedGoals** is the list of codes of goals in deadlock.

2.3 Priority

In KL1, it is possible to specify the priority at which each goal is executed. There are logical and physical priorities, and each goal can have its own logical priority. There are different levels of physical priority in the system, and the scheduler converts logical priority into physical priority when it connects goals to the goal stack. (As physical priority is less accurate than logical priority, user should not expect scheduling to reflect exactly the logical priority.) Upper/lower limits of priority in the Sho-en are also logical.

Priority of a goal is specified relatively to its parent goal, or relatively to the Sho-en it belongs to. The former method is called "relative self specification in the belonging Sho-en" and the later is called "rate specification in the belonging Sho-en".

Rate specification in the belonging Sho-en

Goal priority is specified by a value relative to the upper/lower limit of the belonging Sho-en. It is written as follows :

Goal @ priority(*, Rate)

In this case, the goal priority is computed as follows :

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

Relative self specification in the belonging Sho-en

Goal priority is specified by a value relative to the logical priority of the parent goal. This priority cannot exceed the upper/lower limit of the Sho-en.

Goal @ priority($, Rate)

This time, priority is computed as follows, assuming that Cp is the priority of the calling goal :

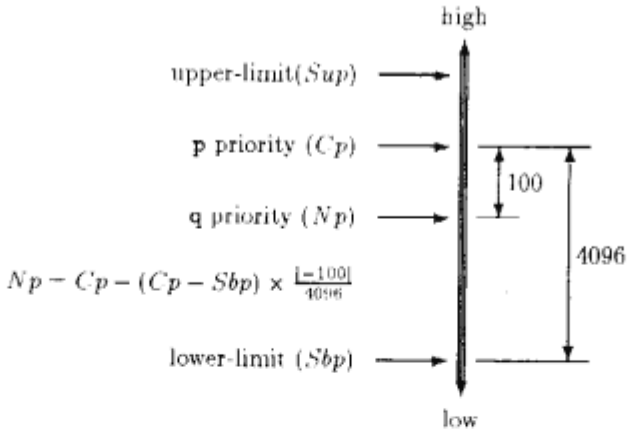
$$Cp + (\text{upper-limit} - Cp) \times \frac{|\text{Rate}|}{4096} \qquad (0 \leq \text{Rate} \leq 4096)$$

or

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

For example, in the following program, suppose that the priority of goal is Cp, the prority Np of goal q is calculated below :

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



2.4 Syntax

Differences between GHC and KL1 are described here. Main differences are concerned with :

- Module definition
- Clause ordering
- Priority specification
- Macros description

The macros are described later in this document.

2.4.1 Module definition

The following is a module definition :

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

This declaration must appear at the head of any module. Furthermore, any predicate defined in this module but used outside of it should be declared as follows :

```
:- public predicate-name/number-of-arguments, ... .
```

Note that predicates executed by a built in predicate **apply** and those specified at Sho-en generation must be declared public. Multiple definitions of a predicate, spreading over several clauses, cannot be split by the definition of a different predicate. Doing otherwise causes display of the message "Assembler: Doubly defined label.".

A goal whose definition pertains to a different module can be used as indicated below :

```
module-name : goal-name
```

2.4.2 Clause ordering

Compilation of KL1 program goes through clause indexing, in order to maximize efficiency. This may in return change the order according to which clauses are selected for evaluation. If clause evaluation ordering is a necessity, the following statements should be used.

Scheduling order

The statement **alternatively\index{alternatively}** can be used to separate two sets of clauses, the first of which should be scheduled with a higher priority. However, if all clauses in the first set are suspended or fail, evaluation of clauses in the second set starts.

```
foo([X|XX],Z) :- true | p(X,XX,Z).
        ...
alternatively.
foo(X,[Z|ZZ]) :- true | q(X,Z,ZZ).
        ...
```

Evaluation order

The **otherwise\index{otherwise}** statement is more straightforward : clauses following it are evaluated only if all of the preceeding clauses failed. Suspension does not trigger anything in this case.

```
foo([X|XX]) :- X==a | pa(X,XX).
foo([X|XX]) :- X==b | pb(X,XX).
        ...
otherwise.
foo(X) :- true | q(X).
        ...
```

2.5 Data types

Here are the data types supported by PDSS :

Unbound variables	····· `A, B, C, _, _abc`
Integers (Range `-2**31` to `2**31-1`)	····· `123, 16'ACE, 8'37`
Atoms	····· `abc, 'ABC'`
Lists	····· `[1,2,3], [1|X]`
Vectors (possibly of dimension 0)	····· `{a,Y,b}, f(X), {}`
Strings (possibly void)	····· `"abc", ""`

When an integer in format x'y is used, the radix base x may vary between 2 and 36, with classical convention for figures. Note that this produces a syntax error on the Prolog based compiler. The form `radix-base#number` should be used instead. Note that the `number` is expressed as a string(e.g. `16#"12AC"`).

As well on the Prolog-based compiler, "..." is considered as a list of 8 bits character code. An expression of the form `string#"..."` should be used to generate a string.

Indeed, unification of two strings occurs if the strings have identical length and contents.

2.6 Built-in predicates

We now give the list of the built-in KL1 primitives supported by PDSS. The following is an example of the format we use :

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

↑ Call format ↑ Valid location for occurence

In this case, **G** means that the predicate can appear in the guard of the clause. Some predicates can occur in the body, in which case the letter **B** is used. **GB** denotes predicates which can occur in both places. Besides, arguments with a ^ are outputs, whereas other arguments are inputs. One should take this into account, because binding an output argument with an already instantiated variable may cause suspension. Also, unification occuring in the guard is passive, whereas unification in the body can be active.

The symbol ***** will sometimes appear after **G**, **B** or **GB**. It means the incompatibility with the predicates on Multi-PSI V2 as follows.

```
G *    : Names are different.
G **   : Specification is differnt.
G ***  : On Multi-PSI V2, this is not a built-in predicate but a function of
         operating system. Specification is different.
G **** : This is not supported on Multi-PSI V2.
```

For some of the predicates described therein, input parameters should verify some domain constraints. Typically, to divide a number by 0 is not a very sound operation. If a domain constraint is not respected, depending on the predicate position, two different things may happen : if the predicate is used within the guard part of a clause, this clause fails. If the predicate is in the body, an exception occurs.

The following syntax for arithmetic macros is based on DEC-10 Prolog. Macros not listed here are described in chapter 2.7.

```
<comparison-f> ::= <f1> <comparison-o> <f1>
<numerical-o>  ::= <variable> ":=" <f1>
<f1>           ::= <f2> | <f1> <numerical-o1> <f2>
<f2>           ::= <f3> | <f2> <numerical-o2> <f3>
<f3>           ::= <term> | <term> <numerical-o3> <term>
<factor>       ::= <integer> | "(" <f1> ")"
<comparison-o> ::= ">" | "<" | ">=" | "=<" | "=:=" | "=\="
<numerical-o1> ::= "+" | "-" | "/\" | "\/" | "xor"
<numerical-o2> ::= "*" | "/" | "<<" | ">>"
<numerical-o3> ::= "mod"
```

2.6.1 Type checking

Unless otherwise specified, if an input variable is still unbound while calling the following predicates, suspension occurs.

wait(X) :: G

As soon as `X` is bound, this predicate succeeds.

atom(X) :: G

As soon as `X` is bound, if `X` is an atom, the predicate succeeds, otherwise it fails.

integer(X) :: G

As above, replacing atom with integer.

list(X) :: G

As above, replacing integer with list.

vector(X, ^Size) :: G

As soon as `X` is bound, if `X` is not a vector, this predicate fails. If `X` is a vector, this predicate succeeds and `Size` is unified with the vector size.

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

This primitive works more or less the same, unless if `X` is not a vector. In this case, exception occurs rather than failure. Also, `NewVector` is unified with a copy of `X`. This is useful to duplicate a vector and avoid inter-process references, which are a pain for the garbage collector.

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

This predicate succeeds if `X` becomes a string. Otherwise it fails. In addition, `Size` is unified with the number of characters in the string, and `ElementSize` is unified with the length of each character, expressed in bits.

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

This primitive, like the vector/3 primitive, makes a copy of the original string. Since this is a body primitive there will be an exception if `X` is bound to something different from a string.

atomic(X) :: G ****

This primitive succeeds if `X` is bound to an atom, an integer or a string. It fails otherwise.

【!】 This predicate may be removed in future version. It is anyhow not supported on Multi-PSI V2.

unbound(X, ^Result) :: B **

This primitive always succeeds but when evaluated, if `X` is unbound, `Result` is unified with `success`. Conversely, if `X` is bound when this primitive is executed, `Result` is unified with the atom `failure`. This primitive never causes suspension.

2.6.2 Comparison

Unless otherwise specified, as in the previous section, whenever the following predicates are called, if one of the input variables is unbound, suspension occurs. Also, for the following predicates which are concerned with

integers, if one of the input variables is not an integer, failure occurs.

Our reader will note the presence of macros after the definition of each predicate.

less_than(Integer1, Integer2) :: G

As one could have guessed, besides suspension or failure cases enumerated above, this predicate succeeds if and only if `Integer1` is less than `Integer2`.

```
X < Y   <=>  less_than(X,Y).
X > Y   <=>  less_than(Y,X).
```

not_less_than(Integer1, Integer2) :: G

Here, greater or equal is the requisite condition.

```
X >= Y   <=>  not_less_than(X,Y).
X =< Y   <=>  not_less_than(Y,X).
```

equal(Integer1, Integer2) :: G

Here, numerical equality is the condition. Note that input parameters have to be integers.

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

not_equal(Integer1, Integer2) :: G

This time, the predicate succeeds if the two input parameters are different integers.

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

not_unified(X, Y) :: G *

If `X` or `Y` are structured, the not_equal predicate is of no help. So, this predicate checks that `X` and `Y` can be unified. If it is the case, the predicate fails. If terms cannot be unified the predicate succeeds. If there are unbound variables in any of the terms, suspension occurs.

```
X \= Y   <=>  not_unified(X,Y).
```

【!】 If `X,Y` are structured terms, unifiability checking is limited in depth. If terms are unifiable until the depth limit, predicate will fail, although a difference may exist deeper in the structures.

2.6.3 Arithmetic operations

If input variables of the following predicates are undefined at call time, suspension occurs. If inputs are not integers, failure or exception occurs, depending on whether the predicate is used in the guard or the body of a clause.

【!】 For the particular case of the first four predicates thereafter, overflow is not detected on PDSS. On multi-PSI V2, it causes failure or exception, like type mismatch.

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

The result of the addition is unified with **`NewInteger`**.

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

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

The result of the subtraction is unified with **`NewInteger`**.

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

multiply(Integer1, Integer2, ^NewInteger) :: GB **

The result of the multiplication is unified with **`NewInteger`**.

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

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

Here `NewInteger` is unified with the division result. If `Integer2` is bound to 0, failure or exception occurs.

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

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

Here `NewInteger` is unified with the rest of the euclidian division result. If `Integer2` is bound to 0, failure or exception occurs.

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

minus(Integer, ^NewInteger) :: GB ****

`NewInteger` is unified with `Integer` with sign exchanged.

[!] Overflow is not detected on PDSS. On Multi-PSI V2, this predicate is not supported.

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

`NewInteger` is unified with the result of logic bitwise shift. `ShiftWidth` should be in the range [0,31] or failure/exception will occur.

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

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

Same stuff, for a right logic shift.

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

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

`NewInteger` is unified with the result of a bitwise logic and operation.

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

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

This time, it is a bitwise or...

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

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

... and now an exclusive or.

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

complement(Integer, ^NewInteger) :: GB

This unifies `NewInteger` with the 2's complement of `Integer`. This is equivalent to exclusive_or (-1, `Integer`, `NewInteger`).

```
Y := \(X)  <=>  complement(X,Y).
```

2.6.4 Vector predicates

If any of the input variables of the following predicates is unbound at call time, suspension occurs, as usual. Three of the following predicates are creating or duplicating vectors. The memory necessary to perform this operation is borrowed from the heap area. If the later is exhausted, an exception occurs.

new_vector(^Vector, Size) :: B

This predicate unifies `Vector` with a freshly allocated vector, filled with zeros. The size of this vector is given by the input parameter `Size`, which should be positive or null. Otherwise, exception will occur.

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

This predicate is used to extract one element from a vector. `Position` indicates the rank of this element, starting from 0. `Element` is unified with the result of extraction. If `Position` is not a relevant integer or if `Vector` does not hold a vector, failure occurs.

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

This is the same primitive, with in addition a copy of the original vector. This is useful to avoid multiple references which could impair garbage collector operations. This time, as this is a body predicate, if `Position` is not a relevant integer (or if `Vector` has a funny structure) exception occurs instead of failure.

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

This predicate, in addition to the function of the previous one, allows the modification of an element at the specified place.

2.6.5 Atom/String predicates

As usual, if an input variable is unbound while calling one of the following predicates, suspension occurs. Also, if the operation domain of one of the following primitive is violated, exception/failure occurs, depending on whether the primitive appears in body or guard. In particular, care should be taken to feed string-type objects into string-concerned predicates!

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

This predicate unifies `String` with a freshly created string. The string is filled with zero. `Size` specifies the length of the string, and `ElementSize` specifies how many bits contains each character, from 1 to 32.

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

This predicate is useful to extract one character from a string. `Position` of the first character is 0. `Element` is unified with the result.

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

This predicate works the same, but it also makes a copy of the input string. The point is again to lighten the task of the garbage collector.

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

This predicate is used to patch a single character in a string. The result of the substitution is unified with `NewString`.

【!】 One should be careful to give as `NewElement` a character with the relevant number of significant bits. Bits in excess are masked and discarded.

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

Here, what we get is a substring extracted from the original string. The original string is copied, into `NewString`, by the way. One has to be careful to suppply meaningful start position and length. The result is unified with `SubString`.

set_substring(String, Position, SubString, ^NewString) :: B ****

This time, we can replace a substring, like we could replace a single character. If `SubString` has not the same bitwise type as `String`, exception occurs. The result of substitution is unified with `NewString`. Exception also occurs if the length of `Substring` + `Position` exceeds length of `String`.

append_string(String1, String2, ^NewString) :: B ****

This predicate unifies with `NewString` the result of concatenating `String2` after `String1`. These input strings should be of the same bitwise type. Otherwise, exception occurs.

make_atom(String, ^Atom) :: B ***

This predicate transforms a string of 8-bits characters into an atom, whose name matches the string contents. This atom is unified with `Atom`.

【!】 On Multi-PSI V2, this is not a built-in predicate but a function of operating system.

atom_name(Atom, ^String) :: B ***

This predicate conversely unifies `String` with a 8 bits string which contains the name of the atom.

【!】 On Multi-PSI V2, this is not a built-in predicate but a system predicate.

atom_number(Atom, ^Number) :: B

Basically, in the system, an identification number is associated with each atom. This predicate unifies `Number` with the id corresponding to the input `Atom`.

2.6.6 Second order function

apply(ModuleName, PredicateName, Args) :: B **

This predicate is used to invoke a predicate whose name is known after some computation. If any of `ModuleName`, `PredicateName` or `Args` are unbound at call time, suspension occurs. If module or predicate are not atoms, or if `Args` is not a vector, exception occurs. Otherwise, the predicate with name `PredicateName` (within the module identified by `ModuleName`) is called, with arguments specified by `Args`.

2.6.7 Stream support

merge(In, ^Out) :: B

This primitive can be used to merge one or more input streams (`In`) and unify the result with `Out`. A vector of streams, if given as one of the input, is divided into its stream components. The following is a partial definition of this predicate :

```
merge([], O) :- true | O=[].
merge([A|I], O) :- true | O=[A|NO], merge(I, NO).
merge({}, O) :- true | O=[].
merge({I}, O) :- true | merge(I, O).
merge({I1,I2}, O) :- true | merge(I1, I2, O).
merge({I1,I2,I3}, O) :- true | merge(I1, I2, I3, O).
                    . . .
merge([], I2, O) :- true | merge(I2, O).
merge(I1, [], O) :- true | merge(I1, O).
merge([A|I1], I2, O) :- true | O=[A|NO], merge(I1, I2, NO).
merge(I1, [A|I2], O) :- true | O=[A|NO], merge(I1, I2, NO).
merge({}, I2, O) :- true | merge(I2, O).
merge(I1, {}, O) :- true | merge(I1, O).
merge({I3,I4}, I2, O) :- true | merge(I3, I4, I2, O).
merge({I3,I4,I5}, I2, O) :- true | merge(I3, I4, I5, I2, O).
                    . . .
                    . . .
```

merge_in(In1, In2, ^In) :: B

This primitive unifies `In` with the vector of streams {`In1`,`In2`}. This is only useful on an older version of the compiler which did not support statements of the form `In`={`In1`,`In2`}.

2.6.8 Special I/O functions

read_console(^Integer) :: G

This predicate unifies `Integer` with a number read from the console window.

[!] The language processor is halted during this operation. This predicate is used mainly for debugging purposes.

display_console(X) :: G

This predicate displays the current value of `X` on the console window, even if this variable is unbound.

put_console(X) :: G

If `X` is an integer, this primitive puts the equivalent ASCII character on the console window. If `X` is an

8-bits character string, the string is put on the screen. If `X` is undefined, or has a funny type, the predicate does nothing. No line feed or carriage return is added.

2.6.9 Other predicates

If an input variable is unbound when calling one of the following predicates, suspension occurs.

raise(Tag, Type, Info) :: B

This predicate causes `Tag` to be logically and-ed with the tag of all ancestor Sho-ens, starting from the current Sho-en until the top, recursively. This process stops as soon as the result of the and operation is not zero. In this case, a message is inserted in the report stream of the current Sho-en. This message is unified with : raised(`Type`, `Info`, `NewCode`, `NewArgv`). If `Tag` is not strictly positive, exception occurs. If `Type` is not a ground-term, suspension occurs.

consume_resource(Red) :: B

This predicate emulates the consumption of computing ressources, as if due to actual reductions. `Red` is the number which is added to the count of performed reductions. If this count exceeds the allowed maximum, the `ressource_low` condition occurs. If `Red` is not an integer, exception occurs.

hash(X, Width, ^Value) :: B **

This predicate unifies value with a hash code computed according to `X` and `Width`. The result is in [0,`Width`-1]. If `Width` is not an integer, exception occurs.

[!] if `X` is not an integer but a vector, the first element of the vector and the vector length are used for hash code calculation.

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

This predicate unifies `ProcessorNumber` with the processor number of the processor executing this predicate. `X` and `Y` are unified with the coordinates of this processor, depending on the topology of the connection network. On PDSS, which emulates execution by a single processor, `ProcessorNumber` is unified arbitrarily with 0, and `X` and `Y` are unified with 1.

2.7 Macros

For ease of writing, several categories of macros have been introduced in KL1.

- Macros for the description of constants.
- Macros for arithmetic comparison.
- Macros for conditional branch.
- Macros for the declaration of implicit arguments.

In the current version, users cannot define their own macros.

2.7.1 Constant description macros

The following macros generate constant numbers.

`Base#"character-string"`

This macro generates an integer number, in the integer `Base` specified before the sharp sign. The base must be from 2 to 36. Figures can be taken in [0,9] and [a/A,z/Z], as most commonly.

`string#"character-string"`

This macro can be used to generate a string of default type. Within PDSS, default type is ASCII stored as 8 bits characters. On Multi-PSI V2, characters are taken within the JIS Kanji set, stored as 2 bytes characters.

`ascii#"character-string"`

This is useful to assert that the generated string is coded in ASCII. within one byte characters.

`#"character"`

This macro generates a character. using the default representation of the system. In this aspect, it is similar to the `string` macro introduced above.

`c#"character"`

This macro asserts generation of an ASCII character. stored as a single byte.

`ascii#character-atom`

This has the same effect, but the character is entered as an atom, not between double quotes. (ex: `ascii#'['`).

`key#lf`

This macro generates a line feed. in ASCII stored as one byte.

`key#cr`

This macro generates a carriage return. in ASCII stored as one byte.

2.7.2 Unification macros

`left-hand = right-hand`

This macro performs unification of left and right hands. It can be used in body and guard.

`left-hand \= right-hand`

This is equivalent to not_unified(`left-hand`, `right-hand`). This macro can be used only in guard part of a clause.

`left-hand := right-hand`

This macro unifies the left hand with the right hand. but if arithmetic macros feature in the right hand, evaluation takes place. This macro, which can be used in guard or body, is similar to the "is" operator of Prolog.

2.7.3 Arithmetic comparison macros

Arithmetic comparison operators can be used in guard, in place of built-in predicates. But arithmetic macros in both hands of the comparison are not evaluated when built-in predicates are used.

You can use the following comparison operators :

Priority	Operator	Expanded pattern
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)

2.7.4 Arithmetic operation macros

Arithmetic macros are using `+,-,*,/`. Expansion is done according to the following rules :

- The right hand of the `:=` macro is evaluated, and the result is unified with the left hand (as for the "is" operator of Prolog).
- Both hands of comparison macros are evaluated and compared to each other.
- In the case of implicit argument macro `<=`, result of evaluation of the right hand side is unified with the argument specified by left hand side.

- `~(expression)` is used to explicitely require expansion of the expression.
  ex: `p(~(X+Y+1))` becomes `add(X,Y,A),add(A,1,B),p(B).`

Macros embedding constants are evaluated during compilation.

The following table summarize which operators can be used. The higher is the priority, the lower is the precedence. It is always possible to make a term by using ( ).

Priority	Operator	Substituted pattern	Generated built-in predicate
500	`X + Y`	Z	add(X,Y,Z)
	`X - Y`	Z	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)
	`X / Y`	Z	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)
term-like	`\(X)`	Z	complement(X,Z)

Backquoting can be used to inhibit expansion :

- ``` ``(term) ```
  Expansion is totally disabled.
- ``` `(term) ```
  Only expansion of the topmost macros is inhibited, for a nested term.

2.7.5 Conditional branch macros

The following is a notation which allows conditional execution within a single clause, as in DEC10 Prolog :

```
foo(X,Y) :- true |
    ( X=:=0 -> p(Y,Z);
      X > 0 -> q(Y,Z);
      otherwise;
      true  -> r(Y,Z) ),
    s(X,Z).
```

If the goal on the left hand side of -> is a condition and if this condition is satisfied, then the goal on the right hand side is executed. The preprocessor generates the following KL1 clauses, from the above example :

```
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 predicate `'$foo/2/0'` has been generated by the preprocessor. More generally, predicates starting with a dollar sign are generated by the preprocessor. The user should not use the same convention!

【!】 Only built-in predicates may be included in a conditional branch.

【!】 The Prolog-based compiler does not support nested conditions, whereas the KL1 compiler does.

2.7.6 Macros for implicit argument passing

It is very unconvenient to rewrite arguments which appear recurrently in the head of several clauses. To lighten this tedium, implicit argument support(through macros) has been provided.

Two kinds of parameter declarations are possible, depending on the scope which is desired. The first one is global, i.e applies to several modules, whereas the second is local to a single module :

```
:- implicit arg-name : type { , arg-name : type , ... }.

:- local_implicit arg-name : type { , arg-name : type , ... }.
```

Here, **arg-name** (atom) is the name of the implicit argument. Type can be : **shared**, **stream**, **oldnew** or **string**.

The global implicit declaration can appear only once in a module, right after the public declaration. Local declarations can appear several times in a module. Each time it appears, it invalidates the previous declaration. To suppress the usage of all implicit arguments, use the following :

```
:- local_implicit.
```

The name space of local and global arguments are the same, so that different names have to be used.

Using `-->` in place of `:-`, means that a predicate uses implicit arguments. They are inserted in the predicate arguments list, before arguments explicitely given by the user. Exact order is as follows :

1. Global arguments
2. Local arguments
3. explicit arguments (order is not changed)

【ex】

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

p(X) --> true | q(X), r.
   %% Here, a and b are added to the argument list.
    ...

:- local_implicit d:oldnew.
   %% At this point a,b and d are added.
    ...

:- local_implicit d:shared, e:stream.
   %% From this point, a,b,d and e are added.
   %% The type of d has changed, from oldnew to shared.
    ...

:- local_implicit.
   %% From this point, only the global args. a and b will be added.
    ...
```

To access a global argument in a clause, & must be put before the argument. If the argument is a string or a vector use :

```
&arg-name(position)
```

to update or access one of its elements. The first element has position 1. The following is a presentation of each type, with some examples.

shared argument type

If no update of a shared argument occurs within a given clause, all goals of the clause share the same instance of the argument. This is illustrated in example a) below. If the value has to be updated in the clause, use the following syntax :

```
&arg-name <= new-value
```

The new value is effective after update. That means that the respective position of update statement and goals in a clause determines whether the old argument value or the new argument value is used. This is illustrated in examples b) to d).

```
【ex】 definition:  :- implicit counter:shared.

  a) before expansion:  p --> true | q, r.
     after expansion:   p(Cnt) :- true | q(Cnt), r(Cnt).

  b) before:  p --> true | &counter <= &counter + 1, q.
     after:   p(Cnt) :- true | add(Cnt,1,Cnt1), q(Cnt1).

  c) before:  p --> true | &counter <= &counter+1, &counter <= &counter*2, q.
     after:   p(Cnt) :- true | add(Cnt,1,Cnt1), multiply(Cnt1,2,Cnt2), q(Cnt2).

  d) before:  p --> true | &counter <= &counter(2), q.
     after:   p(Cnt) :- true | set_vector_element(Cnt,2,Elem,Elem,_), q(Elem).
```

stream argument type

This type is provided to ease output stream management. If no update occurs within the clause, the streams coming from goals are merged into the argument stream. This is illustrated below, in example a). To update the stream, i.e. insert elements, the following syntax should be used :

```
&arg-name <<= [element 1, element 2, ··· ]
```

This is illustrated below, in example b). Note that the relative position of the update within the clause conditions the insertion order, although this may be of little importance for streams.

```
【ex】 definition:  :- implicit window:stream.

  a) before expansion:  p --> true | q, r.
     after expansion:   p(Win) :- true | merge_in(In1,In2,Win), q(In1), r(In2).

  b) before:  p --> true | &window <<= [putb("gazonk")], r.
     after:   p(Win) :- true | Win=[putb("gazonk")|Win1], r(Win1).
```

oldnew argument type

This type calls after a pair of arguments, in a similar fashion to Prolog's DCG, except that arguments are not restricted to difference lists. As an example, when using a vector as an updatable table, to improve efficiency, one often restricts the number of references to 1. To this end, the oldnew argument type is useful. Also, if you

use this type of argument for a difference list, there is a notation to concatenate elements to the list, like for the stream type :

```
&arg-name <<= [element 1, element 2, … ]
```

If the argument is as integer, use the following for update :

```
&arg-name <= new value
```

If argument is a vector, use the following :

```
【element update(1)】
        &arg-name(position) <= new value

【element reference】
        &arg-name(position)     %% cannot also appear on the left side of <=

【element update(2)】
        &arg-name(position) <<= [element 1, element 2, … ]
```

update(1) and update(2) can be used in body part only, as they use the built-in predicate set_vector_element/5. Reference can appear in guard, where it appeals to vector_element/3, or in body, where it uses vector_element/5. The relationship between homonymous elements is similar to that of DCG (from left to right, top to bottom). See example a),b) and f) below.

In (1), replacement is done at the specified position, as illustrated in examples c) and d). In (2), the right side list is inserted in the difference list. New tail is set to the location specified by position. See example e).

There is also a way to refer the current old value of some argument :

```
&arg-name(old)
```

This is useful in particular to access the value of a counter. This is illustrated in example g) below.

```
【ex】 definition:  :- implicit mutter:oldnew.

 a) before expansion:  p --> true | q, r.
    after expansion:   p(Old,New) :- true | q(Old,Mid), r(Mid,New).

 b) before:  p --> true | &mutter <<= [naha], r.
    after:   p(Old,New) :- true | Old=[naha|Mid], r(Mid,New).

 c) before:  p --> true | &mutter(3) <= naha, r.
    after:   p(Old,New) :- true | set_vector_element(Old,3,_,naha,Mid), r(Mid,New).

 d) before:  p --> true | &mutter(1) <= &mutter(3), r.
    after:   p(Old,New) :- true |
                 set_vector_element(Old,3,Elem,Elem,Mid1),
                 set_vector_element(Mid1,1,_,Elem,Mid2), r(Mid2,New).

 e) before:  p --> true | &mutter(2) <<= [naha,uhi,ehe], r.
    after:   p(Old,New) :- true |
                 set_vector_element(Old,2,[naha,uhi,ehe|Cdr],Cdr,Mid),
                 r(Mid,New).

 f) before:  p --> true | &mutter <= &mutter+1, r.
    after:   p(Old,New) :- true | add(Old,1,Mid), r(Mid,New).
```

```
g) before:  p(X) --> true | X = [&mutter(old)|XX], &mutter <= &mutter+1, p(XX).
   after:   p(Old,New,X) :- true | X=[Old|XX], add(Old,1,Mid), p(Mid,New,XX).
```

string argument type

This works basically as the previous type, except that predicates string_element/3 and set_string_element/4 are used instead of vector-based predicates.

Automatic generation of terminating processes

When no user-defined goal is called in the body, the following unifications are automatically performed, depending on the type of declared arguments :

shared type :: Nothing done.
stream type :: Unification with the atom [].
oldnew type :: Old and New are unified.
string type :: Old and New are unified.

Implicit arguments expansion control

If you call a predicate with no implicit arguments from a predicate with implicit arguments, use double braces : {{ predicate(..) }}, in order to suppress argument addition. See example below :

```
:- 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 previous example, three global implicit arguments are declared, with types **stream**, **oldnew** and **shared**. Predicates using --> instead of :- are regarded as having three implicit arguments and are converted at preprocessing time. As an example, loop predicate is expanded as follows :

```
loop(In, Oh,Ot, Cnt, A) :- Cnt =< 0 | In=[], Oh=Ot.
otherwise.
```

```
loop(In, Oh,Ot, Cnt, A) :- true |
    Cnt1 := Cnt-1, In=[getc(X)|In1],
    check(X, Oh,Om, Cnt1),
    loop(In1, Om,Ot, Cnt1, A).
```

Note that the check/4 predicate, used between braces, has no implicit argument, and is expanded as a predicate of arity 4. In order to use some of implicit arguments when calling this predicate, **&** has to be put before the names of the implicit arguments which are explicitely specified in the call.

【!】 Can implicit arguments take any value, declared types notwithstanding. As a matter of fact, the macro processor only expands. If the programmer is not careful enough, errors may be difficult to detect.

2.7.7 Macro library

Macros in system's library to be used should be declared at the top of the module. Declaration goes as follows :

```
:- with_macro macro-definition-name.
```

where **macro-definition-name** is an atom.

The macro definition files are located in a system dependant directory. In this file, macros are defined as follows :

```
fileio#normal      => 0.
fileio#end_of_file => 1.
fileio#read_error  => 2.
fileio#write_error => 3.
```

In the current version, the left part from the sharp sign must be an atom.

3 Micro PIMOS

Micro PIMOS is a very simple operating system which provides various services for KL1 users on PDSS. It is basically designed for single user, single task operations. Services supported by Micro PIMOS follow :

- Command Interpreter.
- I/O Functions (windows, files, etc.)
- Code Management.
- Display of exception information.

On Micro PIMOS, all commands given to the command interpreter are executed within an unit called 'task'. The task is implemented using Sho-en functions described in section 2.2.

【!】 Bits 23:31 in exception tag are reserved for Micro PIMOS. When using functions of Micro PIMOS, user must not modify bits 23:31 in tag of his Sho-en. Beside commands, a way to use Micro PIMOS functions is to issue requests to Micro PIMOS through the user's goal which supervises the Sho-en.

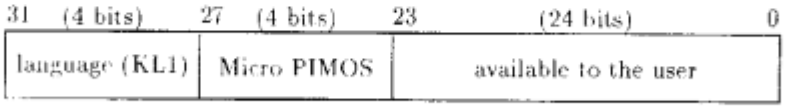


Figure 4: Shoeen exception tag

3.1 Command interpreter

When Micro PIMOS starts, a command interpreter is created to provide user with an interface to PDSS.

When the command interpreter is invoked, it issues a prompt and waits for the next command. Default prompt is `| ?-` or `[debug]?-` when debugging mode is on.

The command interpreter starts by executing the file `"~/.pdssrc"` (if it exists) as a command file. User can set up a convenient working environment via this file. About command file, refer to command take/1.

3.1.1 Command input format

It is possible to write one or more commands in a command line or command file. Depending on the delimiter, commands are executed as follows :

- comma (",")
  Commands both before and after the delimiter are executed in parallel.
- semicolon (";")
  Waits for the termination of commands before the delimiter, and then execute remaining commands. (sequential execution.)
- vertical line ("|")
  After executing the commands before the delimiter, displays the value of variables after this sign. The delimiter between variables is a comma, and `all` stands for all variables.

Lists of commands may be embedded in ( ) to form nested commands.

【ex】
```
| ?- comp("bench");(stat(bench:primes(1,300,P))|P),save(bench).
   % After compiling "bench.kl1", executes the goal 'bench:primes'
   % and saves the code in parallel. During the execution of the
   % goal, reports statistical information and indicates
   % the value of variable 'P' after termination.
```

3.1.2 Commands

Here are the commands supported by the command interpreter, in its current version. Some of the commands expect files to have an extension. If no extension is found, operation is done with the default extension, following the specified filename. Strings or atoms can be used to specify filenames.

Built-in predicates

The command interpreter can execute the built-in predicates which can be described in the body part as a commands. Description using `:=` and arithmetic macros is also possible.

Basic commands

ModuleName:Goal
Executes `Goal` in the module `ModuleName`. Maximum number of reductions is set according to the value of environment variable `reduction`. If the number of performed reductions crosses the limit, the user will be asked whether to continue or to abort.

help
Displays the list of available help commands.

help(Type)
Displays the list of available commands specified by `Type` as follows :
1: builtin, 2: basic, 3: code, 4: dir, 5: debug, 6: env, all.

gc
Invokes GC over the heap area.

gc(all)
Invokes GC for both heap and code areas.

take(FileName)
Executes the command file specified by `FileName`. There is no restriction as to the type of command which can be used in such a file.

`%` or `/* */` are available to mark comments, as in KL1 program.

cputime
Display the CPU time since PDSS started. Unit is millisecond.

cputime(`Time)
Unifies the CPU time since PDSS started with the variable `Time`. The result is an integer and unit is millisecond.

apply(CommandName, ArgsList)
Executes `CommandName` upon each element specified in `ArgList`. ModuleName:PredicateName is also possible for `CommandName`.

stat
Displays the current memory status.

stat(Commands)
Displays the execution time (CPU time) and reduction count of `Commands`.

window(IOStream)
Opens a new window. About commands which deal with I/O streams, refer to the section 3.2. Window name is set automatically.

add_op(Precedence, Type, Operator)
Add an operator to the window of command interpreter.

remove_op(Precedence, Type, Operator)
Deletes an operator from the window of command interpreter.

operator(Operator)

Displays the definition of the operator `Operator` in the window of command interpreter.

operator(OperatorName, ^Definition)

Unifies the definition of operator `Operator`, (format is {Precedence, Type}) in the window of command interpreter with `Definition`.

halt

Terminates PDSS. All windows are closed automatically.

Code commands

comp(FileName)

Compiles the KL1 source file (with extension `.kl1`) `FileName` and loads the result into code area. Trace mode is off for the newly loaded module.

comp(FileName, OutFileName)

Compiles the KL1 source file (with extension `.kl1`) `FileName` and outputs the result into the file (with extension `.asm`) `OutFileName`.

compile(FileName)

Compiles the KL1 source file (with extension `.kl1`) `FileName` and outputs the result into file (with extension `.asm`) `FileName`. Then, loads it into code area and saves into file (with extension `.sav`) `FileName` can also hold a list of files.

load(FileName)

Loads the previously saved file `FileName` (with extension `.sav`) into code area. If such a file does not exist, assembler file (extension is `.asm`) is loaded into code area. Trace mode of the newly loaded module is off.

dload(FileName)

As above, but trace mode of the newly loaded module is on.

save(ModuleName)

Saves the executable code module `ModuleName` to the directory specified by environment variable `savedir`. By default, the directory is `~/.PDSSsave`. It can be changed with the 'ch_savedir' command. `ModuleName` is also used to determine file name. Extension is `.sav`.

save(ModuleName, FileName)

Saves the executable code `ModuleName` to the file `FileName` (with extension `.sav`).

save_all

Saves all loaded modules (except the ones which have already been saved by the `save(ModuleName)` command) into the directory specified by environment variable `savedir`.

ch_savedir(Directory)

Changes the default directory for auto_load and auto_save, to the directory specified by `Directory`. The existence of directory is checked.

listing

Displays information about loaded modules.

listing(^Modules)

Generates a list of all loaded module names and unifies it with `Modules`.

public(ModuleName)

Displays a catalog of public predicates within the module specified by `ModuleName`.

public(ModuleName, ^Public)

Creates a list of information about predicates declared as public, and unifies it with `Public`. Each element of the list is a 2 elements vector of form {predicate name atom, arity}.

Directory commands

cd(Directory)
Changes current directory to the directory specified by `Directory`.

pwd
Displays the pathname of current directory.

pwd(^World)
Unifies the pathname of current directory with `World`.

ls(WildCard)
Displays the pathname of file `WildCard`.

ls(WildCard, ^Files)
Creates a list with the pathnames corresponding to `WildCard` and unifies it with `Files`.

rm(WildCard)
Deletes the file corresponding to `WildCard` from the directory.

Debug commands

trace(ModuleName)
Sets the trace mode on for the code of module `ModuleName`. The debug mode is set to on.

notrace(ModuleName)
Sets the trace mode off for the code of module `ModuleName`.

spy(ModuleName, PredicateName, Arity)
Enables spying of the predicate `PredicateName/Arity` in the module `ModuleName`.

nospy(ModuleName, PredicateName, Arity)
Disables spying of the predicate `PredicateName/Arity` in the module `ModuleName`.

spying(ModuleName)
Displays the list of the predicates currently spied in the module `ModuleName`.

spying(ModuleName,^Spying)
Creates a list of the predicates currently spied in the module `ModuleName`, and then unifies it with `Spying`. Each element of the list is a 2 elements vector formed as {predicate-name-atom, arity}.

debug
Sets debug mode on.

nodebug
Sets debug mode off.

backtrace
Sets display mode on for backtrace information (deadlock information).

nobacktrace
Sets display mode off for backtrace information (deadlock information).

varchk(FileName, Mode, Form)
Checks variables in the KL1 source file (with extension `.kl1`) specified by `FileName` in the mode `Mode`. The result is displayed on the window in the format `Form`. `FileName` can also be a list of files. The definition of `Mode` and `Form` follows :

Mode::	`o` or `one`	····· Displays variables which appear once in a clause.
	`m` or `mrb`	····· Displays variables whose MRB is set.
	`a` or `all`	····· Displays variables of both one and mrb modes.
Form::	`s` or `short`	····· Outputs clauses as a single line.
	`l` or `long`	····· Outputs clauses using line feeds and indentation.

varchk(FileName, Mode)

Checks variables in mode **Mode** and displays in **long** format.

varchk(FileName)

Checks variables in mode **one** and displays in **long** format.

xref(FileName, Mode)

Performs a cross reference check upon the KL1 source file **FileName** (with extension **".kl1"**) and displays result on the window. **FileName** can also be a list of filenames. In this case, references across modules are also checked. **Mode** can be taken amongst the following values :

c or **check**	.....	Checks only predicate calls.
l or **list**	.....	Outputs the reference list (table of definition/reference of predicates).
s or **system**	.....	Outputs the predicates referring to PDSS modules.
b or **builtin**	.....	Outputs predicates referring to built-in predicates.
a or **all**	.....	Outputs all of above predicates.
List	.....	Outputs a reference list for specified elements. Where **List** can include : * Module name. * Built-in predicates. * User-defined predicates.
short	.....	Checks with no display of predicate information.
short(Mode)	.....	Checks according to **Mode** with no display of predicate information.

xref(FileName)

Checks the cross-references in **check** mode.

xref(FileName, Mode, OutFile)

Checks the cross-references and outputs the list to **OutFile**. Any mode is available except **check**.

profile(ModuleName, Mode)

Displays how many times the predicates which are defined in the module **ModuleName** were called and suspended. **ModuleName** can also be a list of modules. **Mode** can be chosen as follows :

c or **call**	..... Sorts according to call count and displays.
s or **susp**	..... Sorts according to suspension count and displays.
other atoms	..... Displays following the order of appearance within the code.

profile(ModuleName)

Executes profile command in **call** mode.

reset_profile(ModuleName)

Resets counts of calls and suspensions for predicates defined in **ModuleName**. **ModuleName** can also be a list of modules.

Environment commands

These commands can be used to change values of environment variables for the command interpreter. The following environment variables are used :

name (atom)	meaning
`world`	Pathname string of current directory.
`trace`	Mode of tracer (`on` or `off`). Initial value is `off`.
`backtrace`	Display mode of backtrace (`on` or `off`). Initial value in on.
`modules`	List of module names in which commands are searched.
`reduction`	Upper limit of the number of reductions assigned when the task was created. The basic allocation unit is 10000 reductions. (0 < number < 100000, Initial value is 10000)
`ucounter`	Counter used to create the names of work files or work windows.
`savedir`	String with the pathname of directory in which save/1 and save_all will produce their effects. Initial value is `~/.PDSSsave`.
`loaddir`	list of pathnames of directories to examine when auto-loading code. Initial value is `[~/.PDSSsave, pathname of library directory, ...]` Note: There are more than two library directories, which may differ from one machine to another.
`auto_load`	Flag for auto-loading (`yes` or `no`). Initial value is `yes`.
`plength`	Maximum length of structure which can displayed in the window of the command interpreter. Initial value is 10.
`pdepth`	Maximum depth of structure which can be displayed on the window of the command interpreter. Initial value is 5.
`pvar`	Displays modes of variables in the window of command interpreter. The value is `nu` or `na`. (`nu` works as 0, 1, 2, ... , and `na` works as A,B,C, ...) Initial value is `nu`.

setenv(Name, Value)

Sets the environment variable **Name** after **Value**. Environment variable is set after that **Name** becomes an atom and **Value** becomes a ground term.

getenv(Name, ^Value)

Unifies the value of environment variable **Name** with **Value**.

printenv(Name)

Displays the value of environment variable **Name**.

printenv

Displays the values of all environment variables of the command interpreter.

resetenv

Initializes all environment variables of the command interpreter.

3.2 I/O functions

Micro PIMOS offers two types of I/O services : window and file I/O services. To use them, Micro PIMOS predicates are provided, which give access to command streams. How commands can be inserted in these streams is now described. [] closes a command stream and, by the way, the associated I/O device. Commands are inserted in command stream by a merger.

3.2.1 Command stream attachment

Windows

window:create(Stream, WindowName, ^Status)

Creates a window with name **WindowName** (8 bits string). Command stream is unified with **Stream**. **Status** is unified with the following terms :

`success`	····· Window successfully created.
`error(cannot_create_window)`	····· Failure : window cannot be opened.
`error(bad_window_name_type)`	····· Failure : **WindowName** is not an 8 bits string.

【!】 When the window is created, it is not in visible state. Use command **show** to make it appear.

window:create(Stream, WindowName)

Creates a window with name `WindowName` (8 bits string). Command stream is unified with `Stream`. If the window cannot be created, the whole task is aborted.

【!】 When the window is created, it is not in visible state. Use command **show** to make it appear.

Files

file:create(Stream, FileName, Mode, ^Status)

Opens file with name `FileName` (8 bits string) with mode `Mode`. `Mode` is an atom chosen among : `r` for read, `w` for write and `a` for append. The command stream of this file is unified with `Stream`. `Status` is unified with one of the following terms :

`success`	····· Open successful.
`error(cannot_open_file)`	····· Cannot open the file.
`error(bad_file_name_type)`	····· `FileName` is not an 8 bits string.
`error(bad_open_mode_type)`	····· `Mode` is not atom.
`error(bad_open_mode)`	····· `Mode` is an atom other than `r`,`w` or `a`.

file:create(Stream, FileName, Mode)

Opens a file as the previous command. `Stream`, `FileName` and `Mode` have same meanings, but if open does not succeed, the whole task is aborted.

3.2.2 Command list

Commands allowed in stream are now listed. These commands are common to window and file, unless otherwise specified.

Input commands

getc(^Char)

Reads an ascii character from I/O device. Value is between 0 and 255. `Char` is unified with the result of input. Upon end of file, `Char` is unified with the atom `end_of_file`.

getl(^String)

Reads one line from I/O device. This line is converted into an 8 bits string, unified with `String`. Upon end of file, `String` is unified with the atom `end_of_file`.

getb(^Buffer, Size)

The number of character `Size` is read from I/O and converted into an 8 bits string, which is unified with `Buffer`. If a carriage return or an end of file is encountered, only characters read before are considered as input. Upon end of file, `Buffer` is unified with the atom `end_of_file`.

gett(^Term)

A string containing one term is read. (A term ends with . + CR or . + space) These characters are analyzed and transformed into a term, which is unified with `Term`. If an error occurs during analysis, if input device is a window, error is output on this window and term input is resumed. If input device is a file, error is displayed on the command interpreter window, then next term is read from the file. At end of file, term is unified with `end_of_file`.

getft(^Term, ^NumberOfVariables)

This command is very similar to the previous one, but variables in the term are analyzed then output as `$VAR(N,VN)`. N is the variable number ($0 \leq N <$ `NumberOfVariables`) and VN is the variable name (8 bits string). `NumberOfVariables` is unified with the number of variables appearing in the term. Upon end of file, `Term` is unified with `end_of_file` and `NumberOfVariables` is unified with 0.

【!】 If end of file has been encountered during the execution of previous commands, successive input commands will return end of file.

Output commands

putc(Char)
Outputs on the I/O device the character with ASCII code `Char`, between 0 and 255.

putl(String)
Outputs the 8 bits string `String` on the I/O device and adds a new line character.

putb(Buffer)
Outputs 8 bits string `Buffer`. No carriage return is added.

putt(Term, Length, Depth)
Outputs term `Term`. If structure depth exceeds `Depth` (> 0) or length exceeds `Length` (> 0), remainder is output as ... This is similar to Prolog's write.

【!】 Atoms are not quoted, so that the result of this command may be unsuited to further read using gett or getft.

putt(Term)
Similar to the previous command, but default value is used for depth and length.

puttq(Term, Length, Depth)
This is similar to putt/3 command, but atoms are quoted when necessary.

puttq(Term)
This is similar to putt/1 command, but atoms are quoted when necessary.

nl
Outputs a new line character.

tab(N)
Outputs `N` ($0 \leq N < 1000$) space characters.

【!】 On Micro PIMOS, I/O is blocked, for efficiency reasons. Buffers are flushed only in the following cases :

- Buffer is full.
- `flush` command has been received.
- I/O device is closed.
- (in the case of windows) some input or show/hide command is received.

Control of input format

Output limitations for structure in putt/1 and puttq/1 commands can be changed as follows :

print_length(Length)
This command changes the default length limit to `Length` (`Length` > 0). Initial value is 10.

print_depth(Depth)
This command changes the default depth limit to `Depth` (`Depth` > 0). Initial value is 10.

print_var_mode(VariableMode)
This command is used to change the output format of terms describing variables. `VariableMode` is the new mode which must be `nu` or `na`. Initial value is `na`.

`na` – Name Mode	::	$VAR(N,VN) → VN (Variable name string) is output.
		$VAR(N) → A,B,C ... is output.
`nu` – Number Mode	::	$VAR(N,VN) → _N (Variable number) is output.
		$VAR(N) → _N (Variable number) is output.

Output buffer commands

The following command control output buffer parameters.

flush(^Status)

This command flushes characters left in buffer. After completion, `Status` is unified with `done`.

buffer_length(BufferLength)

This command changes output buffer length to `BufferLength` (> 0). Initial value is 512 bytes for a window and 2048 bytes for files.

Operators

The following commands are related to operators for parsing.

add_op(Precedence, Type, OperatorName)

This command adds an operator with precedence `Precedence` ($1 \leq$ `Precedence` ≤ 1200), type `Type` (an atom chosen among `fx`, `fy`, `xf`, `yf`, `xfy`, `xfx`, `yfx`) and name `OperatorName` (atom).

remove_op(Precedence, Type, OperatorName)

This command removes an operator. Parameters have the same meaning as in the previous command.

operator(OperatorName, ^Definition)

This command return a list `Definition` of operators matching name `OperatorName` (atom). Each element of the list is in the form (precedence, type).

Grouped processing of commands

do(CommandList)

This command groups the list of command `CommandList` within a single command. Even though merger is used to insert commands in the stream, sequence of commands in `CommandList` is preserved.

Control command

close(^Status)

Closes I/O operation. It is not possible to send other commands after that one. (Only [] can be sent to close the stream.) `Status` is unified with atom `success`.

Window commands

The following commands are effective only for windows.

show

An hidden window will show up when this command is executed.

hide

A visible window will be hidden when this command is executed.

clear

Clears the window space.

beep

Rings the terminal bell.

prompt(^Old, New)

Changes prompt string displayed in executing `gett` or `getft` command. `Old` is unified with current prompt string(8 bits string) and the new prompt becomes `New` (also an 8 bits string). The initial prompt is `"?-"`.

3.3 Directory management

The directory services of Micro PIMOS are available through the directory command stream. This stream is available via a Micro PIMOS predicate, in a similar fashion to I/O services.

Operations on the directory are done by inserting commands into this stream. The stream can be closed with [].

3.3.1 Acquisition of command stream

directory:create(Stream, DirectoryName, ^Status)

Accesses the directory named `DirectoryName` (8 bits string) and unifies the command stream connected to the directory with `Stream`. `Status` can be unified with the following terms :

`success`	····· Access succeeded.
`error(cannot_access)`	····· The directory cannot be accessed.
`error(bad_directory_name_type)`	····· `DirectoryName` is not an 8 bits string.

3.3.2 Commands

The following commands can be inserted into the command stream.

pathname(^PathName)

Unifies the full pathname of the directory (8 bits string) with `PathName`.

listing(WildCard, ^FileNames, ^Status)

Creates the list of pathnames of files corresponding to `WildCard` and unifies it with `FileNames`. `Status` can be unified with the following terms :

`success`	····· List successfully created.
`error(cannot_listing)`	····· List cannot be created.

delete(WildCard, ^Status)

Deletes files corresponding to `WildCard` (8 bits string) from the directory. `Status` can be unified with the following terms :

`success`	····· Deletion successful.
`error(cannot_delete)`	····· Cannot delete the file.

open(Stream, FileName, Mode, ^Status)

Opens the file `FileName` (8 bits string) with mode `Mode` (atom `r` for read, `w` for write or `a` for append) and unifies the command stream connected to the file with `Stream`. `Status` can be unified with the following terms :

`success`	····· Open successful.
`error(cannot_open_file)`	····· Cannot open the file.
`error(bad_file_name_type)`	····· `FileName` is not an 8 bits string.
`error(bad_open_mode_type)`	····· `Mode` is not atom.
`error(bad_open_mode)`	····· `Mode` is an atom other than `r`,`w` or `a`.

3.4 Device Stream for I/O

To use Input/Output device functions directly from Micro PIMOS, one can use the libraries now described. The purpose of the functions therein is to describe other OS than Micro PIMOS (e.g. PIMOS) in KL1. Average user does need device streams shown below.

These device streams are supervised by Micro PIMOS. So if a wrong command is inserted, it only results in the failure of user task, the language processor is unaffected.

3.4.1 Securing Device Stream

User can extract a device stream from Micro PIMOS by using the following predicates. `mpimos_io_device` can also be used as a module name.

mpimos_window_device:windows(Stream)
Unifies the stream which has a function of window device with `Stream`.

mpimos_file_device:files(Stream)
Unifies the stream which has a function of file device with `Stream`.

mpimos_timer_device:timer(Stream)
Unifies the stream which has a function of timer device with `Stream`.

3.4.2 Command

The commands which can be sent to each device stream, stream of opened window, file and directory are just the same as mentionned in Appendix 1. As to the I/O commands for file/window streams, only the commands shown below are allowed.

- Window

Input Only getl(Line, Status, Cdr) is available.
getc/3, getb/4, gettkn/4 are not available.

Output Only putb(Buffer, Status, Cdr) is available.
putc/3, putl/3, putt/5 are not available.

- File

Input Only getb(Size, Bufferm Status, Cdr) is available.
getc/3, getl/3, gettkn/4 are not available.

Output Only putb(Buffer, Status, Cdr) is available.
putc/3, putl/3, putt/5 are not available.

3.5 Code Management

The principal functions for code management on Micro PIMOS now follow.

- Functions to manage the name and information (like the catalog of public predicates and spied predicates) of loaded modules and display this information upon request.
- Auto_load function of modules which are saved by `save(ModuleName)` or `save_all` commands from the command interpreter.

The directory from which `auto_load` is performed is decided after the environment variable `loaddir` of command interpreter. User had better make a directory `~/.PDSSsave` to use the auto_load function, because the default value of first element of both `loaddir` and `savedir` is `~/.PDSSsave`. The value of environment variables can be changed. User can disable the auto load function by setting the environment variable `auto_load` to `no`.

3.6 Displaying Exception Information

The KL1 exceptions handled by PDSS are shown in section Appendix-7. On Micro PIMOS, information about an exception which has occurred within the user task is displayed on the window of command interpreter. The task in which exception has occurred is immediately stopped and its resources (windows and files) are released.

Other exceptions, reported by Micro PIMOS, are handled by Micro PIMOS. Those are consequent to an illegal command to the window, trying to open a file that does not exist, etc. In those cases, as in the case of language definition exceptions, information is displayed on the window of command interpreter and the task is immediately stopped. All resources of the task are released.

4 PDSS Optional parameters

PDSS is usually invoked under GNU-Emacs. This may be seen as the best way to use PDSS, from an execution environment point of view, as all PDSS functions are available. It is possible to execute PDSS on a stand alone basis, but in this case, some functions disappear.

4.1 Usage under GNU-Emacs

To call PDSS under GNU-Emacs, send the following command. Libraries are automatically loaded and PDSS starts.

meta-X pdss *return*

To specify options, type ctrl-U before meta-X. Option contents are described later.

ctrl-U meta-X pdss *return*
PDSS Option?: [**parameter**] *return*

When PDSS starts, a window named "console window" is created. This window is used to trace execution and for input/display at console. Then, several modules are loaded, including runtime support and Micro PIMOS. Then, Micro PIMOS starts. After that, command interpreter window is created and waits for user commands.

When operation is done within GNU Emacs, PDSS input is asynchronous. Therefore the whole system does not hold when input occurs. There is an exception to this for console inputs while tracing. In this case, system halts until input completion. It is possible to control PDSS by striking control keys in the window. These keys are defined in a GNU-Emacs library. Besides following commands, a complete list of supported keys can be found in Appendix-9.

ctrl-C ctrl-C	::	Turns on trace flag.
ctrl-C ctrl Z	::	Sends interrupt code 1.
		In Micro PIMOS, this aborts task.
ctrl-C ctrl-T	::	Sends interrupt code 2.
		In Micro PIMOS, this prints number of reductions performed so far.
ctrl-C !	::	Starts GC.
ctrl-C @	::	Aborts PDSS.
ctrl-C ctrl-B	::	Generates a window buffer menu for PDSS.
ctrl-C ESC	::	Reexecutes PDSS system.
ctrl-C k	::	Removes contents of current window.
ctrl-C ctrl-K	::	Deletes contents of the window created by PDSS.
ctrl-C ctrl-Y	::	Reprints the last input string.
ctrl-C ctrl-F	::	Prints manual of built-in predicates.
ctrl-C f	::	Prints manual of command interpreter.

【!】 When a PDSS window is removed by ctrl-X k, subsequent execution results are not guaranteed to be meaningful.

4.2 PDSS on stand-alone

To use PDSS without GNU-Emacs, type the following command :

pdss [parameter] *return*

Outside of GNU-Emacs, all messages to windows are merged. If any window waits for some input, the whole system stops. Window control keys are not available but, on the other hand, keyboard interrupt is supported.

If ctrl-c is typed, one can enter control commands after the prompt.

4.3 Optional parameters

Optional parameters can be specified at start, to modify the execution environment. Possible parameters follow :

`-hNNN`	:: Size of heap area is **NNN**. Default is 200 kcells.
`-cNNN`	:: Size of code area is **NNN**. Default is 300 kbytes.
file name	:: Uses this file instead of the standard startup file.
`+t/-t`	:: Uses start up file or not. Default is to use it.
`-v`	:: This option changes the way variables appear during trace. By default, variables are printed using their name. If `-v` option is used, they are identified by their relative position to heap bottom. This may change after each GC, so be careful.
`-bNNN`	:: Scheduling politics for goals are changed to breadth-first. **NNN** gives depth limit for TRO.
`-a`	:: Inhibits timer interrupt. This is used when debugging PDSS itself, under dbx.

There are two ways to specify these options :

- Options can be given when starting PDSS. They are treated as arguments of the PDSS command.

```
example-1)
       PDSS Option ?: -h300000 -c50000 -v     (Execution under GNU-Emacs)
example-2)
       [UNIX]% pdss -h300000 -c50000 -v       (Execution on stand alone)
```

- Options can also be specified through an environment variable.

```
example)
       [UNIX]% setenv PDSSOPT "-h300000 -c50000 -v"
       [UNIX]% pdss
```

5 Tracer

The tracer functions supported by PDSS are now described.

5.1 Principle of operation

Basically, in PDSS, trace operations can occur whenever a goal is in one of the following states. These events are called trace points.

- Goal call.
- Suspension due to an uninstanciated argument.
- End of suspension.
- Goal failure.
- Swap out (Caused by interruption or scheduling of a higher priority goal).

There are two ways to operate trace in KL1 : upon predicate execution or upon goal call.

To trace upon predicate execution is to trace when the code, which ones want to trace, is executed. In this case, it is possible to specify trace mode for each module. In the sequel, this mode is called "code trace". It is also to trace each predicate separately. This is called "code spying".

To trace upon goals is to trace, or not to trace, descendant goals of each generated goals. In the sequel, we call this "goal trace". It also possible to limit trace to the descendant goals of specific goals. This is called "goal spying".

Let's see some example. In the following program, we assume that `p(X)` is in trace state and `p(Y)` is not. Then, `q(A,B)` and `r(B)` which are called from `p(X)` are also traced. Conversely, `q(A,B)` and `r(B)` which are called by `p(Y)` are not traced.

```
Goal   :  p(X), p(Y).
Clause:  p(A) :- true | q(A,B), r(B).
```

In PDSS, it is possible to specify before or during execution whether or not code trace is done. On the other hand, goal trace status must be on at first; then, some of the goals can be untraced. Only goals which have both code trace status on and goal trace status on are actually traced.

The four possible cases of spying are the following :

- Code is spied.
- Goal is spied.
- Code or goal are spied.
- Code and goal are spied.

5.2 How to read the display

Trace display contains 4 different information zones :

```
[0012] CALL *$ module:goal(a1,a2,a3)
1      2    3  4
```

1. Identity of the Sho-en to which this goal belongs.
2. Type of trace event :

`CALL` :: Dequeue from goal queue.

`Call` :: Goal called during TRO.

`SUSP` :: Suspension due to an uninstanciated argument.

`Susp` :: Suspension due to priority preemption.

`RESU` :: End of suspension.

`FAIL` :: Goal failure.

`SWAP` :: Swap out.

3. Spy flags :

   `*` :: Code of executing goal is spied.

   `$` :: Goal is spied.

4. Goal

   Terms in argument list which are potentially referred several times (MRB is on) are appended with an `x` mark.

   Variables are shown as follows, according to their type :

   - Ordinary unbound variable :
     First letter is upper case, or underscore, and is followed by a number... `X1`, `_23611`.
   - Some goal waits for instantiation of this variable :
     Format is the same as an ordinary unbound variable, followed by a tilda... `X1~`, `_23611~`.
   - Merger input variable :
     As above, replacing tilda with carret... `X1^`, `_23611^`.

In addition to this description, priority is displayed whenever it changes.

5.3 Commands

The description of tracer commands has the following meta syntax :

Command name :: input format {argument} { [options] }

Help :: ?

Command help.

No Trace :: X

No trace from now on.

No Goal Trace :: x

Turns off trace for the descendants of current goal, i.e. goals called from this goal.

Set Goal Spy :: g

Spies current goal from now on.

Reset Goal Spy :: G

Stops spying current goal, from now on.

Set Module Debug Mode :: d MODULE { MODULE ... }

Sets debug flag on for specified modules. By this mean, code trace is done when predicates from this module are executed.

Reset Module Debug Mode :: D MODULE { MODULE ... }

Effects are opposite to the previous command.

Set Procedure Spy :: p MODULE:PROCEDURE { MODULE:PROCEDURE ... }

Sets trace on for a given predicate in a given module.

Reset Procedure Spy :: P MODULE:PROCEDURE { MODULE:PROCEDURE ... }

Opposite of previous command.

Step :: s [COUNT]

Stops again at next trace point. If `COUNT` is given, stop occurs only after that `COUNT` trace points have passed. Here and in the following, `COUNT` can be considered as a repetition factor.

Step to Next Spied Procedure :: sp [COUNT]

Continues until the next spied predicate is called, then stops.

Step to Next Spied Goal :: sg [COUNT]

As above, but we look for a spied goal.

Step to Next Spied Procedure or Spied Goal :: ss [COUNT]

In this case, any spy case causes stop.

Step to Next Spied Procedure and Spied Goal :: SS [COUNT]

In this case, procedure must be traced and called from a traced goal, to cause stop.

Skip to Next Spied Procedure :: np [COUNT]

This works similarly to **sp** command, but no trace is done until stop.

Skip to Next Spied Goal :: ng [COUNT]

This works similarly to **sg** command, but no trace is done until stop.

Skip to Next Spied Procedure or Spied Goal :: ns [COUNT]

This works similarly to **ss** command, but no trace is done until stop.

Skip to Next Spied Procedure and Spied Goal :: NS [COUNT]

This works similarly to **SS** command, but no trace is done until stop.

Re-Write Goal :: w LENGTH [DEPTH]

This redisplays current goal and arguments, with modified format limits `LENGTH` and `DEPTH`. This is useful when arguments are large.

Where call from :: where

Shows the names of predicate and module which called current traced goal. This is valid only for run-time support routines or built-in predicates (D code).

Monitor Variable :: m VARIABLE_NAME [NAME] [LIMIT]

Monitors the value of variable, whenever it is bound. If the value is a list or a stream, display occurs whenever the top element is bound. Using `NAME`, it is possible to assign a new identifier to the monitored variable, so that the value is shown under that name. `LIMIT` is the number of times value can be shown without stopping the system. Without this parameter, whenever a variable is bound, the value is shown and the tracer waits for a user command.

During display of value, whether the value is a list or not is distinguished :

```
mon#var-name => value %%     ..... In the case of a list.
mon#var-name == value %%     ..... Otherwise.
```

In this situation, the following commands are available :

?	:: Help.
x	:: Stops monitoring this variable or list.
s [COUNT]	:: Goes ahead monitoring value without stop, for `COUNT` times.
w LENGTH [DEPTH]	:: Redisplays value.
m VAR [NAME] [LIMIT]	:: Sets a different monitoring.

Inspect Ready Queue :: ir [PRIORITY]

Shows goals in ready queue. If `PRIORITY` is given, only goals with that physical priority are shown.

Inspect Variable :: iv VARIABLE_NAME

Shows state of specified variable. When state is HOOK or MHOOK (goals are waiting for the instantiation

of this variable), shows the waiting goals. When state is MGHOK (input merger variable), shows merger output variable.

Inspect Sho-en tree :: is

Shows Sho-en tree structure at current time. Horizontal drawing axis is used to represent the parent/children dependency, whereas the vertical axis is used to represent brotherhood. Each Sho-en is described using 5 characters : the first one corresponds to the state of Sho-en and the remaining 4 to its ID. Possible Sho-en status follow :

R :: Ready.
S :: Suspended.
A :: Aborted.

Trace Shoen tree :: ts

Turn on/off the trace flag of Sho-en tree structure. When on, tree structure is shown before and after each modification(e.g. generation, abortion, termination). The format of the description is the same as for above command.

Set Tracer Variable :: set NAME [VALUE]

Tracer variable `NAME` is set to `VALUE`, if present. Otherwise, current value is displayed. Current tracer variables and their default values are now listed.

pv :: Print variable mode. If value is **n**, variables are displayed alphabetically. If **a** is used, relative address are used.

pl :: Print length value.

pd :: Print depth value.

g :: Gate switch which determines whether trace is done or not upon each trace point. Value is made of five characters, each one with value **n**, **t** or **s**. These values correspond to "no trace", "trace (no stop)" and "trace (stop)" respectively. Characters correspond to points CALL/call, SUSP/Susp, RESU, SWAP and FAIL, respectively.

c :: Gate Switch for CALL points. Value is n, t or s.

s :: Gate Switch for SUSP points. Value is n, t or s.

r :: Gate Switch for RESU points. Value is n, t or s.

w :: Gate Switch for SWAP points. Value is n, t or s.

f :: Gate Switch for FAIL points. Value is n, t or s.

6 Dead-lock detection

In PDSS, there are 2 dead-lock detection mechanisms. One acts through global GC, whereas the other tries to detect deadlock during execution. During GC, deadlock is always detected, whereas deadlock check done during execution sometimes fails.

In the following, types of deadlock detected in PDSS and tracer messages are shown.

Dead lock detection occurred during GC : Type=0

Example : when executing the following goal :

```
Goal :: ?- add(X,_,Y), divide(Y,_,Z), modulo(Z,_,_).
```

Following information is shown on the console window.

```
GC 1       HEAP     GREC     PREC     SREC      TOTAL     CODE
Start     15595       77        7       14     139180      725
Deadlock::pdss_runtime_body_builtin:modulo(A~,B,C)
Deadlock::pdss_runtime_body_builtin:divide(D~,E,A~)
Deadlock::pdss_runtime_body_builtin:add(F~,G,D~)
*** Previous goal is deadlock root!
Shoen is terminated by deadlock!
Done       1067       56        3       14      18956      725
GCed      14528       21        4        0     120224        0
          cells  records records records     bytes    bytes
```

The goal appearing before message "Previous goal is deadlock root!" on console window is the root of the data dependance tree. There are some cases where several such trees exist and root is not unique in general. If there are loops structures, there is no root.

Variable referred only by itself (void variable) waiting for instantiation : Type=10

Example : In the following program, after execution of `p(X)`, `q(X)` is executed.

```
Goal :: ?- p(X), q(X).
Clause-1 :: p(_) :- true | true.
Clause-2 :: q(a) :- true | true.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [suspend(HOOKoo/MGHOKo)]
*** Goals waiting for HOOKoo/MGHOKo to be unified:
Deadlock::module:q(A~).
```

Waiting for instantiation of a variable which will never be instantiated by other goals : Type = 11

Example : In the following program, after execution of `p(X)`, `q(X)` is executed.

```
Goal :: ?- p(X), q(X).
Clause-1 :: p(a) :- true | true.
Clause-2 :: q(a) :- true | true.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [suspend(HOOKoo/MGHOKo)]
*** Goals waiting for HOOKoo/MGHOKo to be unified:
Deadlock::module:q(A~).
Deadlock::module:p(A~).
```

Input variable of the merger, referred only by itself, waiting for instantiation : Type=12

Example : In the following program, after execution of `merge(In,Out)`, `q(X)` is executed.

```
Goal :: ?- merge(In, Out), p(In), ….
Clause-1  :: p(a) :- true | true.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [suspend(HOOKoo/MGHOKo)]
*** Goals waiting for HOOKoo/MGHOKo to be unified:
Deadlock::module:p(A^).
Deadlock::pdss_runtime_body_builtin:active_unify(A^,B^).
```

Variable which has a goal waiting for instantiation, and which is unified with a void variable : Type=20

Example : In the following program, after execution of `p(X)`, `q(X)` is executed. This case also occurs if `Y` was not void, but would turn void as a result of the execution.

```
Goal :: ?- p(X), q(X,Y).
Clause-1 :: p(a) :- true | true.
Clause-2 :: q(A,B) :- true | A=B.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [unify(HOOKoo,VOID)]
*** Unification occurred in module:q/2
*** A goal waiting for HOOKoo to be unified:
Deadlock::module:p(A^).
```

Merger input is unified with void variable : Type=21

Example : In the following program, `p(In,_)` is executed after `merge(In,Out)`.

```
Goal :: ?- merge(In, Out), p(In,_), q(Out).
Clause-1  :: p(A,B) :- true | A=B.
Clause-2  :: q([_|Cdr]) :- true | q(Cdr).
```

Following information is shown on the console window.

```
*** Deadlock will occur. [unify(MGHOKo,VOID)]
*** Unification occurred in module:p/2
*** Merger input was abandoned.
*** Goals waiting for merger output:
  3995: module:q(A^).
```

Unifying two variables which have goals waiting for instantiation : Type = 22

Example : In the following program, `r(X,Y)` is executed after `p(X)` and `q(X)`.

```
Goal :: ?- p(X), q(Y), r(X,Y).
Clause-1 :: p(a) :- true | true.
Clause-2 :: q(a) :- true | true.
Clause-3 :: r(A,B) :- true | A=B.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [unify(HOOKoo,HOOKoo)]
*** Unification occurred in module:r/2
*** Goals waiting for HOOKoo to be unified:
Deadlock::module:p(A^).
Deadlock::module:q(B^).
```

Unifying input of merger and variable which has a goal waiting for instantiation : Type=23

Example : In the following program, `q(X,In)` is executed after `merge(In,Out)` and `p(X)`.

```
Goal :: ?- merge(In, Out), p(X), q(X, In), r(Out).
Clause-1 :: p(a) :- true | true.
Clause-2 :: q(A,B) :- true | A=B.
Clause-3 :: r([_|Cdr]) :- true | r(Cdr).
```

Following information is shown on the console window.

```
*** Deadlock occurred. [unify(HOOKoo,MGHOKo)]
*** Unification occurred in module:q/2
*** A goal waiting for HOOKoo to be unified:
Deadlock::module:p(A^).
*** Goals waiting for merger output:
  3995: module:r(B^).
```

Unification of two merger inputs : Type = 24

Example : In the following program, `In1=In2` is executed after `merge(In1,Out1)` and `merge(In1,Out2)`.

```
Goal :: ?- merge(In1, Out1), merge(In2, Out2),
           p(In1,In2), q(Out1), r(Out2).
Clause-1 :: p(A,B) :- true | A=B.
Clause-2 :: q([_|Cdr]) :- true | q(Cdr).
Clause-3 :: r([_|Cdr]) :- true | r(Cdr).
```

Following information is shown on the console window.

```
*** Deadlock will occur. [unify(MGHOKo,MGHOKo)]
*** Unification occurred in module:p/2
*** Merger input was abandoned.
*** Goals waiting for merger output:
  3995: module:q(A^).
  3995: module:r(B^).
```

A variable which has a goal waiting for instantiation is not referred : Type=30

Example : In the following program, `q(X)` is executed after `p(X)`.

```
Goal :: ?- p(X), q(X).
Clause-1 :: p(a) :- true | true.
Clause-2 :: q(_) :- true | true.
```

Following information is shown on the console window.

```
*** Deadlock occurred. [collect(HOOKoo)]
*** Collect_value occurred in module:q/1
*** A goal waiting for HOOKoo to be unified:
Deadlock::module:p(A^).
```

A merger input variable is not referred : Type=31

Example : In the following program, `p(In)` is executed after `merge(In,Out)`.

```
Goal :: ?- merge(In, Out), p(In), q(Out).
Clause-1 :: p(_) :- true | true.
Clause-2 :: q([_|Cdr]) :- true | q(Cdr).
```

Following information is shown on the console window.

```
*** Deadlock will occur. [collect(MGHOKo)]
*** Collect_value occurred in module:p/1
*** Merger input was abandoned.
*** Goals waiting for merger output:
  3995: module:q(A^).
```

Appendix-1 I/O devices

PDSS provides window, file and timer I/O devices. Commands to these devices are inserted in streams. These devices are defined in modules named `pdss_window_device`, `pdss_file_device` and `pdss_timer_device`. This specification is based upon "FEP Host I/O Interface (V0.9)". Full features have not be implemented. Some of the messages or commands are therefore dummy or illegal. Since macro expression `fep#xxxx` is not available, it has to be replaced with atom `'fep#xxxx'`.

Acquisition of device stream

Device stream can be obtained by the predicates listed below. These predicates can be called only once after the emulator has been invoked. Twice or more calls will cause "Device called twice" exception to occur.

pdss_window_device:windows(Stream)
Unifies `Stream` with the command stream of window device.

pdss_file_device:files(Stream)
Unifies `Stream` with the command stream of file device.

pdss_timer_device:files(Stream)
Unifies `Stream` with the command stream of timer device.

Device commands

1. Window device
Window device provides multiple window facility within GNU-Emacs. The following commands can be sent to this device :

create(BufferName, WindowStream, ^Status, Cdr)
Opens a window with buffer name `BufferName` (8 bits string), then unifies its command stream with `WindowStream`. When window is opened successfully, `Status` is unified with `'fep#normal'`. PDSS can't open more than 16 windows at a time. Therefore, it fails when user tries to open too many windows. In this case, `Status` is unified with `'fep#abnormal'`. I/O commands and control commands described below can be inserted in the command stream of a window which has been successfully opened. Note that reset/4 command must be applies before those commands to set up abort and attention lines. The window is automatically closed when its stream is closed.

create(WindowStream, ^Status, Cdr)
Create/3 without buffer name is not available.

get_max_size(X, Y, PathName, ^Characters, ^Lines, ^Status, Cdr)
Always returns `Characters` = 80, `Lines` = 40, `Status` = `'fep#normal'`.

2. File device
This device provides standard facilities of UNIX files. The following commands can be applied to this device :

open(PathName, Mode, FileStream, ^Status, Cdr)
Opens file with path name `PathName` (8 bits string), mode `Mode` (atom:`'fep#read'` = read mode, `'fep#write'` = write mode, `'fep#append'` = append mode), then unifies `FileStream` with the command stream. When the file is opened successfully, `Status` is unified with `'fep#normal'`. Otherwise, `Status` is unified with `'fep#abnormal'`. I/O commands and control commands can be applied to a file which has been successfully opened (reset/4 is also requisite for files. The file is automatically closed when its stream is closed.

directory(PathName, DirectoryStream, ^Status, Cdr)
Opens directory with path name `PathName` (8 bits string) and unifies `DirectoryStream` with its command stream. When open is successful, `Status` is unified with `'fep#normal'`. When it fails, `Status` is unified with `'fep#abnormal'`. Commands can be inserted in a directory stream which

has been successfully created (reset/4 is also requisite for directory.). A directory is automatically closed when its stream is closed.

3. Timer device

Unit of time of the timer device is millisecond, but updates are actually performed each second. The following commands can be sent to the timer device :

get_count(^Count, ^Status, Cdr)
`Count` is unified with the total elapsed milliseconds since 00:00:00 AM. `Status` is unified with `'fep#normal'`.

on_at(Count, ^Now, ^Status, Cdr)
When the time specified by `Count` is reached, Now is unified with `'fep#wake_up'`. `Status` is unified with `'fep#normal'` when the command is over.

on_after(Count, ^Now, ^Status, Cdr)
When duration specified by `Count` has elapsed, `Now` is unified with `'fep#wake_up'`. Status is unified with `'fep#normal'` when the command is over.

Window, file and directory commands

1. Control commands common to windows and files

These are the commands shared by window and file devices.

reset(AbortLine, ^AttentionLine, ^Status, Cdr)
Sets up abort and attention lines. This command should be issued right after the I/O stream has been generated. To abort an I/O request, `AbortLine` must be unified with `'fep#abort'` by the host. Once unified, abort line and attention line must be set up again with reset/4 command. Otherwise, the stream can still be closed with □. `AttentionLine` is unified with interrupt code generated by device (integer). Upon interrupt, I/O should be aborted or attention line should be set again with reset/4 command.

next_attention(^Attention, ^Status, Cdr)
Only attention line is set by this command. This command is used when user does not want to abort I/O after interrupt.

2. Common input commands

There are the commands which can be sent to window or file devices working in read mode.

getc(^Char, ^Status, Cdr)
Reads one character and unifies it with `Char`. When input is completed successfully, `Status` is unified with `'fep#normal'`. If end of file is encountered, `Status` is unified with `'fep#end_of_file'`.
【!】 Not available on Multi-PSI V2 FEP.

getl(^Line, ^Status, Cdr)
Reads one line, converts it into an 8 bits string, then unifies it with `Line`. At this time, newline code is removed. When the input is completed successfully, Status is unified with `'fep#normal'`. If end of file is encountered, `Status` is unified with `'fep#end_of_file'`.
【!】 Not available for files on Multi-PSI V2 FEP.

getb(Size, ^Buffer, ^Status, Cdr)
Reads the number of bytes specified by `Size` (integer), and converts them into an 8 bits string, unified with `Buffer`. If a newline is encountered while reading from a window, input stops at newline character. When the input is completed successfully, `Status` is unified with `'fep#normal'`. If end-of-file is encountered, it is unified with `'fep#end_of_file'`.
【!】 Not available for windows on Multi-PSI V2 FEP.

gettkn(^TokenList, ^Status, ^NumberOfVariables, Cdr)

Reads a string constructed as one term then analyzes this string to extract tokens. The list of generated tokens is unified with `TokenList`.

variable	:: `var(N,String)`
atom	:: `atom(Atom)`
integer	:: `int(Integer)`
string	:: `string(String)`
functor	:: `open(Atom)`
special character	:: atom that has special character as print name.
end	:: `end`

When the input is completed successfully, `Status` is unified with `'fep#normal'`. If end-of-file is found, it is unified with `'fep#end_of_file'`. If an error occured during token analysis, `Status` is unified with `'fep#abnormal'`.

【!】 Not available on Multi-PSI V2 FEP.

3. Common output commands

These commands can be sent to window and file devices opened in write or append more.

putc(Char, ^Status, Cdr)

Writes the character corresponding to `Char` (integer) according to the ASCII code. `Status` is unified with `'fep#normal'`.

【!】 Not available on Multi-PSI V2 FEP.

putl(Line, ^Status, Cdr)

Writes string `Line` (8 bits string) and adds a newline character. `Status` is unified with `'fep#normal'`.

【!】 Not available on Multi-PSI V2 FEP.

putb(Buffer, ^Status, Cdr)

Writes string in Buffer (8 bits string). `Status` is unified with `'fep#normal'`.

putt(Term, Length, Depth, ^Status, Cdr)

Writes the term specified by `Term`, whith maximum length `Length` and maximum depth `Depth`. The part of term which exceeds Length or Depth is printed as .... `Status` is unified with `'fep#normal'`. Since this command uses output function for debugging, variables in `Term` are written like A, B, C with a symbol MRB or HOOK.

【!】 Not available on Multi-PSI V2 FEP.

4. Window control commands

These commands are available only for windows.

close(^Status)

Closes the window. `Status` is unified with `'fep#normal'`.

flush(^Status, Cdr)

No op. `Status` is unified with `'fep#normal'`. Data which have been written are automatically flushed, even if flush/2 is not issued.

beep(^Status, Cdr)

Rings the terminal bell. `Status` is unified with `'fep#normal'`.

clear(^Status, Cdr)

Erases contents of window. `Status` is unified with `'fep#normal'`.

show(^Status, Cdr)

Makes window visible. `Status` is unified with `'fep#normal'`. Since the window stays invisible after creation, one has to make it explicitely visible with this command.

hide(^Status, Cdr)
Makes window invisible. `Status` is unified with `'fep#normal'`.

activate(^Status, Cdr)
Same as show/2.

deactivate(^Status, Cdr)
Same as hide/2.

set_inside_size(Characters, Lines, ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

set_size('fep#manipulator', ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

set_position(X, Y, ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

set_position('fep#manipulator', ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

set_title(String, ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

reshape(X, Y, Characters, Lines, ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

reshape('fep#manipulator', ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

set_font(PathName, ^Status, Cdr)
No op. `Status` is unified with `'fep#normal'`.

select_buffer(BufferName, ^Status, Cdr)
Not available.

get_inside_size(^Characters, ^Lines, ^Status, Cdr)
Always returns `Characters` = 80, `Lines` = 20, `Status` = `'fep#normal'`.

get_position(^X, ^Y, ^Status, Cdr)
Always returns `X` = 0, `Y` = 0, `Status` = `'fep#normal'`.

get_title(^Title, ^Status, Cdr)
Returns name with which the window was created.

get_font(PathName, ^Status, Cdr)
Not available.

5. File control commands

These commands can be used only for files.

close(^Status)
Closes file. `Status` is unified with `'fep#normal'`.

end_of_file(^Status, Cdr)
`Status` is unified with `'fep#yes'` when the end of file has been encountered. Otherwise, it is unified with `'fep#no'`

pathname(^PathName, ^Status, Cdr)
Unifies file pathname with `PathName`. `Status` is unified with `'fep#normal'`.

6. Directory control command

These commands can be used only for directory streams.

pathname(^PathName, ^Status, Cdr)

Unifies pathname of directory with `PathName`. `Status` is unified with `'fep#normal'`

listing(WildCard, FileNameStream, ^Status, Cdr)

Unifies the list of pathnames corresponding to `WildCard` (8 bits string) with `FileNameStream`. `Status` is unified with `'fep#normal'`. `FileNameStream` can include a command next_file_name (FileName, ^Status, Cdr). Then one file name (8 bits string) is returned through `FileNameStream` and `Status` is unified with `'fep#normal'`. When no more files are available, `Status` is unified with `'fep#end_of_file'`.

delete(WildCard, ^Status, Cdr)

Deletes all files corresponding to `WildCard` (8 bits string). PDSS can not recover deleted files. `Status` is unified with `'fep#normal'`.

undelete(WildCard, ^Status, Cdr)

No op. `Status` is unified with `'fep#normal'`.

purge(WildCard, ^Status, Cdr)

No op. `Status` is unified with `'fep#normal'`.

deleted(WildCard, ^FileNameStream, ^Status, Cdr)

Returns a stream from which deleted files corresponding to `WildCard` (8 bits string) can be extracted. This list is always empty. Status is unified with `'fep#normal'`.

expunge(^Status, Cdr)

No op. `Status` is unified with `'fep#normal'`.

Appendix-2 Code device

This device manages code. Code can be manipulated by inserting commands into device stream. (Currently, only Micro PIMOS is allowed to use code device stream, which is not available for the average user.)

assemble(^ModuleName, Module, ^Status)

Assembles the file `FileName` (8 bits string) and loads it into the code area. `ModuleName` is unified with the atom named after the assembled module name. `Status` is unified with `success`, `cannot_open_file`, `memory_limit` or `error`, depending on how the operation has been proceeded.

load_one_module(^ModuleName, FileName, ^Status)

Loads file specified by `FileName` into code area. File format should be either save or assembler format. `ModuleName` is unified with the atom named after the loaded module name. `Status` is unified with `success`, `cannot_open_file`, `memory_limit` or `error`, depending on the course of operations.

save_one_module(ModuleName, FileName, ^Status)

Saves the module `ModuleName` (atom) to file `FileName` (8 bits string). `Status` is unified with either `success`, `cannot_open_file` or `memory_limit`.

remove_module(ModuleName, ^Status)

Deletes module `ModuleName` (atom). `Status` is unified with `success` or `module_not_found`.

debug(Flag, ^Status)

Switches debugging mode on or off. `Flag` is atom `on` or `off`. `Status` is unified with `success`.

backtrace(Flag, ^Status)

Switches backtrace (display of deadlocked goals detected during global GC) on or off. `Flag` is atom `on` or `off`. `Status` is unified with `success`.

trace_module(ModuleName, Mode, ^Status)

Changes trace mode of the module `ModuleName` (atom) to `Mode`. Mode is atom `on` or `off`. `Status` is unified with `success`, `module_not_found` or `undefined_mode`.

get_module_state(ModuleName, ^Mode, ^Status)

Checks trace mode of the module `ModuleName` (atom). `Mode` is unified with `on` or `off`, according to the state of trace mode. `Status` is unified with `success` or `module_not_found`.

spy_predicate(ModuleName, PredicateName, Arity, Mode, ^Status)

Changes trace mode of the predicate `PredicateName/Arity` in module `ModuleName` (atom), to `Mode`. `Mode` is atom `on` or `off`. `Status` is unified with `success`, `module_not_found`, `predicate_not_found` or `undefined_mode`.

all_spied_predicates(ModuleName, ^Predicates, ^Status)

Unifies `Predicates` with a list of information about the predicates spied in the module `ModuleName` (atom). Each element is a 2-elements vector of the form {predicate name atom, arity}. `Status` is unified with `success` or `module_not_found`.

public(ModuleName, ^Public, ^Status)

Unifies `Public` with a list of information about public predicates in the module `ModuleName`. Each element is a 2-elements vector of the form {predicate name atom, arity}. `Status` is unified with `success` or `module_not_found`.

Appendix-3 PIMOS common utilities

These utility programs were developped for PIMOS, but can be used on PDSS as well. When provided modules are called, these utilities are loaded automatically by Micro PIMOS auto-load function.

PIMOS provides the following conversion and store functions are common utilities which can be used in both PIMOS and application programs. When one wishes to use these facilities, he can get the conversion result or object connection stream by calling predicates of the modules provided in PIMOS. User manipulates objects by inserting messages in this stream, through a merger.

- Comparison : a function which generates a total order upon KL1 data.
- Hashing : a classical hash function.
- Pool without key : bag, stack, queue, sorted bag.
- Pool with key : keyed bag, keyed set, keyed sorted bag, keyed sorted set.

1. Comparison

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

Compares `X` and `Y`, then unifies the left hand element of the relation with `S` and the right one with `L`. If `X` = `Y`, `S` is unified with `X` and `L` with `Y` (relation is said to be stable). Besides, if `L` is unified with `X`, `Swapped` is unified with `yes`, and with `no` otherwise.

Definition of the comparison relation :

If the data have different types, order is the type order, i.e. integer, atom, string, list, vector, from left to right. Otherwise, the relation is defined as follows :

- integer……Comparison between integers.
- atom ……Comparison between atom numbers.
- string ……Lexicographic order, if strings are of the same type. Otherwise, type order.
- list ……Comparison of Car. If they are the same, comparison of Cdr, and so on.
- vector ……Comparison of the number of elements. If it is the same for both vectors, proceeds as for lists.

2. Hashing

Standard hash function is provided.

hasher:hash(X, ^H, ^Y)

`H` is unified with a non negative integer holding hash result. `Y` is unified with `X`.

Hash function definition

- integer……Absolute value.
- atom ……Atom number.
- string ……Cb × first-element + Cm × middle-element + Ce × last-element + string-length. Cb, Cm and Ce are the same as KL0 built-in predicates.
- list ……Car hash value + 5 × Cdr hash value.
- vector ……number of elements + sum (for the first, middle and last elements) of ((2 to the power of element rank+ 1) × element hash value)

3. Pool without key

Any KL1 data stored via this mechanism.

Bag

A basic pool. There are only basic functions put and get. To refer an element in the pool we have to extract it, and there is no way to leave it inside pool.

pool:bag(Stream)
Generates a bag object. `Stream` is the command stream associated to it.

Message protocol :

empty(^YorN)
Returns `Y` if the bag is empty. `N` otherwise.

put(X)
Puts `X` into the bag.

get(^X)
Gets `X` from the bag. It is not possible to select a specific element. After this operation, the element is removed from the bag. If no element is in the bag, failure occurs.

get_all(^O)
`O` is unified with the list of all elements in the bag. If none, it is unified with `[]`.

get_and_put(^X, Y)
Pulls out one element and unifies it with `X`, then puts `Y` in its place. If the bag is empty, failure occurs.

Stack

Basically the same as bag, but element order is LIFO

pool:stack(Stream)
Generates a stack object. `Stream` is unified with the control stream.

Message protocol : Same as bag protocol.

Queue

Basically the same as bag, but element order is FIFO.

pool:queue(Stream)
Generates a queue. `Stream` is unified with the control stream.

Message protocol : Same as bag.

Sorted Bag

Works like a bag, but extraction order is 'least element first', according to comparison function.

pool:sorted_bag(Stream)
Generates a sorted bag object, with a standard comparator:sort/5 comparison routine. `Stream` is unified with the command stream.

pool:sorted_bag(Comparator, Stream)
Works the same, but comparison routine is specified by `Comparator`, whose format is {module name atom, predicate name atom, arity}. Sorted bag object, which has a `Comparator` routine, is generated. `Stream` is unified with the command stream. This predicate must have been declared as public, with the same arity and function as comparator:sort/5.

Message protocol :

Same as bag. `get` returns the least one and `get_all` returns a list sorted in ascending order.

Keyed pool

KL1 data are stored with a key by this mechanism.

Keyed Bag

Basic pool with a key. This is based on a hash table.

pool:keyed_bag(Stream)

Generates a keyed bag object, using the standard hash function (hasher:hash/3). `Stream` is unified with the command stream. Initial hash table size is 1.

pool:keyed_bag(Stream, Size)

This works the same as the previous predicate, except that hash size is given by `Size`.

pool:keyed_bag(Hasher, Stream, Size)

With this predicate, it is not only possible to specify hash table size, but also the hash function. `Hasher` is of the form {module name atom, predicate name atom, arity}. The corresponding predicate must have been declared as public, and have same arity and function as hasher:hash/3.

Message protocol :

empty(^YorN)

Returns `Y` if bag is empty and `N` otherwise.

empty(Key, ^YorN)

As above, but subset of elements with key `Key` is examined.

put(Key, X)

Puts `X` into the bag, using key `Key`.

get(Key, ^X)

Unifies `X` with one element with key `Key`. If there are several possible choices, one is picked up at random. After this operation, the chosen element is removed from the bag. If no element with key `Key` is in the bag, failure occurs.

get_all(^O)

`O` is unified with the list of all elements in the bag. Each item in the list is of the form {key, element}. If the bag is empty, `O` is unified with `[]`.

get_all(Key, ^O)

`O` is unified with the list of all elements with key `Key`.

get_and_put(Key, ^X, Y)

Unifies `X` with an element with key `Key`, then replaces it with `Y`. If there is no such element, failure occurs.

Keyed Set

This works like a keyed pool, except that duplicated keys are not allowed.

pool:keyed_set(Stream)

This creates a keyed set. `Stream` is unified with the command stream. Standard hash function(hasher:hash/3) is used. Initial hash table size is one.

pool:keyed_set(Stream, Size)

This works the same, but hash table size is `Size`.

pool:keyed_set(Hasher, Stream, Size)

This works the same, but it is also possible to specify the hash function which should be used. See keyed_bag/3 predicate above.

Message protocol :

empty(^YorN)

Returns `Y` is the bag is empty, `N` otherwise.

empty(Key, ^YorN)

This works the same, but only the subset of elements with key `Key` is analyzed.

put(Key, X, ^OldX)

Adds an element with key `Key` and value `X`. If there is already an element with the same key, its value is updated, and `OldX` is unified with {old value}. Otherwise, `OldX` is unified with {}.

get(Key, ^X)

Unifies `X` with the element with key `Key`. If there is no such element, failure occurs. The element is removed from the set after this operation.

get_all(^O)

All elements are removed from the set, and `O` is unified with a list whose elements are of the form {key, data}. If the set was already empty, `O` is unified with `[]`.

get_all(Key, ^O)

`O` is unified with a list containing element with key `Key`, which is removed from the set. If there is no such element, `O` is unified with `[]`.

get_and_put(Key, ^X, Y)

Replaces element with key `Key` with `Y`. Old value is return in `X`. If there is no element with such a key, failure occurs.

Keyed Sorted Bag

This is similar to sorted bag, but sort is performed only upon key.

pool:keyed_sorted_bag(Stream)

Generates a keyed sorted bag, using standards compare routine(comparator:sort/5). `Stream` is unified with the command stream.

pool:keyed_sorted_bag(Comparator, Stream)

Works the same, but `Comparator` can be used to specify the sort routine. Refer to sorted_bag/2 above.

Message protocol :

It is similar to the one of keyed bag, but data comes out in increasing order of key.

Keyed Sorted Set

This is similar to keyed sorted bag, but identical keys are not allowed.

pool:keyed_sorted_set(Stream)

Generates keyed sorted set object, with standard compare routine(comparator:compare/5). `Stream` is unified with command stream.

pool:keyed_sorted_set(Comparator, Stream)

Works the same but `Comparator` can be used to specify the comparison predicate. See sorted_bag/2 above, for more information.

Message protocol :

This is the same as the one of keyed set, but data comes out in increasing order of key.

Appendix-4 Reserved module names

The following module names are reserved by PDSS, and should not be used. Names marked with * are available.

'Sho-en'
* directory
* file
* window
* mpimos_io_device
monogyny_list_index
mpimos_booter
mpimos_builtin_predicate
mpimos_cmd_basic
mpimos_cmd_code
mpimos_cmd_debug
mpimos_cmd_directory
mpimos_cmd_environment
mpimos_cmd_utl
mpimos_code_manager
mpimos_command_interpreter
mpimos_libdir
mpimos_directory
mpimos_directory_device_driver
mpimos_file
mpimos_file_device_driver
mpimos_file_manager
mpimos_window_device
mpimos_file_device
mpimos_timer_device
mpimos_macro_expander
mpimos_module_pool
mpimos_opcode_table
mpimos_operator_manipulator
mpimos_parser
mpimos_task_monitor
mpimos_unparser
mpimos_utility
* mpimos_varchk
mpimos_window
mpimos_window_device_driver
mpimos_window_manager
* mpimos_xref
* mpimos_pretty_printer
pdss_code_device
pdss_window_device
pdss_file_device
pdss_timer_device
pdss_runtime_active_unify
pdss_runtime_debug
pdss_runtime_exception_handling
pdss_runtime_body_builtin
kl1cmp_blttbl
kl1cmp_command
kl1cmp_compile
kl1cmp_mrb
kl1cmp_normalize
kl1cmp_output
kl1cmp_reader
kl1cmp_register
kl1cmp_macro
kl1cmp_macro_arg
kl1cmp_mtbl
kl1cmp_struct

Appendix-5 Reserved operator names

The following operators are defined for PDSS windows and file input.

```
1200   xfx   :-
1200    fx   :-
1200   xfx   -->
1150    fx   module
1150    fx   public
1150    fx   implicit
1150    fx   local_implicit
1150    fx   with_macro
1100   xfy   ;
1100   xfy   |
1090   xfx   =>
1050   xfy   ->
1000   xfy   ,
 800   xfx   :
 700   xfx   =
 700   xfx   \=
 700   xfx   =\=
 700   xfx   =:=
 700   xfx   ==
 700   xfx   <
 700   xfx   >
 700   xfx   =<
 700   xfx   >=
 700   xfx   :=
 700   xfx   <=
 700   xfx   <<=
 700   xfy   @
 500   yfx   +
 500    fx   +
 500   yfx   -
 500    fx   -
 500   yfx   /\
 500   yfx   \/
 500   yfx   xor
 400   yfx   *
 400   yfx   /
 400   yfx   <<
 400   yfx   >>
 300   xfx   mod
 200    fx   &

 150   xf    ++
 150   xf    --
 100   xfx   #
 100    fx   #
```

Appendix-6 List of built-in predicates

1. Type checking

```
wait(X) :: G
atom(X) :: G
integer(X) :: G
list(X) :: G
vector(X, ^Size) :: G
vector(X, ^Size, ^Vector) :: B
string(X, ^Size, ^ElementSize) :: G
string(X, ^Size, ^ElementSize, ^String) :: B
atomic(X) :: G
unbound(X, ^Result) :: B
```

2. Comparison

```
less_than(Integer1, Integer2) :: G
not_less_than(Integer1, Integer2) :: G
equal(Integer1, Integer2) :: G
not_equal(Integer1, Integer2) :: G
not_unified(X, Y) :: G

The following operators are available :
    <, =<, >, >=, =:=, =\=, \=
```

3. Arithmetic operation

```
add(Integer1, Integer2, ^NewInteger) :: GB
subtract(Integer1, Integer2, ^NewInteger) :: GB
multiply(Integer1, Integer2, ^NewInteger) :: GB
divide(Integer1, Integer2, ^NewInteger) :: GB
modulo(Integer1, Integer2, ^NewInteger) :: GB
minus(Integer, ^NewInteger) :: GB
shift_left(Integer, ShiftWidth, ^NewInteger) :: GB
shift_right(Integer, ShiftWidth, ^NewInteger) :: GB
and(Integer1, Integer2, ^NewInteger) :: GB
or(Integer1, Integer2, ^NewInteger) :: GB
exclusive_or(Integer1, Integer2, ^NewInteger) :: GB
complement(Integer, ^NewInteger) :: GB

:=, <= can be used with the following operators :
    +, -, *, /, mod, <<, >>, /\, \/, xor
```

4. Vector

```
new_vector(^Vector, Size) :: B
vector_element(Vector, Position, ^Element) :: G
vector_element(Vector, Position, ^Element, ^NewVector) :: B
set_vector_element(Vector, Position, ^OldElement, NewElement, ^NewVector) :: B
```

5. Atom/Strings

```
new_string(^String, Size, ElementSize) :: B
string_element(String, Position, ^Element) :: G
string_element(String, Position, ^Element, ^NewString) :: B
set_string_element(String, Position, NewElement, ^NewString) :: B
substring(String, Position, Length, ^SubString, ^NewString) :: B
set_substring(String, Position, Substring, ^NewString) :: B
append_string(String1, String2, ^String) :: B
make_atom(String, ^Atom) :: B
atom_name(Atom, ^String) :: B
atom_number(Atom, ^Number) :: B
```

6. Second order function

```
apply(ModuleName, PredicateName, Args) :: B
```

7. Stream support

```
merge(In, ^Out) :: B
merge_in(In1, In2, ^In) :: B
```

8. Special I/O

```
read_console(^Integer) :: G
display_console(X) :: G
put_console(X) :: G
```

9. Others

```
raise(Tag, Type, Info) :: B
consume_resource(Reduction) :: B
hash(X, Width, ^Value) :: B
current_processor(^ProcessorNumber, ^X, ^Y) :: B
```

Appendix-7 Exception codes

- Illegal Input Type :: 1
  An illegal data type appeared as an input argument of some built-in predicate.
- Range Overflow :: 2
  The range of some input argument of a built-in predicate is incorrect. Zero division is included here.
- Incorrect Priority :: 5
  Assigned priority is outside of the Sho-en bounds.
- Undefined Module :: 6
  An unloaded module has been referred to.
- Undefined Predicate :: 7
  A given predicate does not appear in the required module.
- Failed :: 8
  No candidate clause are selected for goal execution. (Format is failure/4, and no exception is reported.)
- Unify Failed 1 :: 9
  Body unification has failed. (Format is failure/4, and no exception is reported.)
- Unify Failed 2 :: 10
  Data different from list or vector has been input through merger. (Format is failure/4, and no exception is reported.)
- Device Called Twice :: 12
  Device stream acquisition performed twice.

Appendix-8 Reserved Sho-en tags

In current version, the following bits of Sho-en tag are reserved for the KL1 language and for Micro PIMOS usage

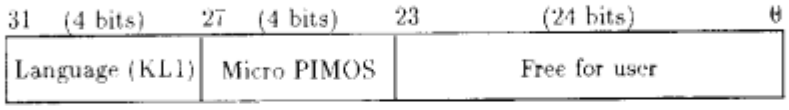


Figure 5: Sho-en exception tag

- bit 31 – Exception while calling a built-in predicate.
- bit 28 – Deadlock detected.
- bit 26 – Message output on Micro PIMOS shell window.
- bit 25 – Error message sent to parent Sho-en.
- bit 24 – I/O stream required from parent Sho-en.

Appendix-9 GNU-Emacs library

There are two library modes in PDSS. The first one is the kl1-mode, used to edit programs, and the second is PDSS-mode, used to run PDSS. Commands defined in each mode are shown below :

1. kl1-mode

ctrl-C ctrl-C

Compiles all the text in the buffer in which command has been executed, as if this text was KL1 source code.

ctrl-C ctrl-R

Copies specified range of text in the buffer `PDSS=COMPILER`.

ctrl-C ctrl-D

Compiles the contents of the buffer `PDSS=COMPILER` as a KL1 program. Then, looks for the assembler file(*.asm) which has the same name as the buffer and updates parts of this file which have changed. Eventually, generates save file.

This command is used with **ctrl-C ctrl-R** to recompile updated parts only. Assembly files should therefore not be deleted.

meta-X pdss-kl1cmp-switch-macro-mode

meta-X pdss-kl1cmp-switch-indexing-mode

meta-X pdss-kl1cmp-switch-debug-mode

meta-X pdss-kl1cmp-switch-system-mode

Changes options of the Prolog version compiler. Commands with no argument work as toggle switches, while arguments 1/0 corresponds to on/off. Detailed meaning and initial values of these options are described in Appendix-10. Above commands correspond to `e`, `i`, `d` and `s` options, resp.

When using KL1 version compiler, these commands are not available.

【!】 Information and prompt are output in buffer `PDSS=COMPILE`, but basically, user does not need to type anything in this buffer.

ctrl-C ctrl-F

Displays the manual of built-in predicates.

2. PDSS-mode

a. Window/buffer operations

meta-.

Displays a candidate string, beginning with `?-` and matches the previous string which has been entered. This is used to repeat the last interpreted command.

ctrl-C ctrl-Y

Redisplays previous input.

ctrl-C k

Deletes all text in the buffer.

ctrl-C ctrl-K

Deletes all text in all PDSS-mode buffers.

ctrl-C ctrl-B

Displays buffer menu of PDSS mode buffer.

ctrl-C m

Looks for the pattern module-name:predicate-name from the beginning of current line, and start insert at its current position. This is convenient to set variable name when setting variable monitor in the tracer.

ctrl-C ctrl-F

Displays built-in predicate manual.

ctrl-C f

Displays command manual for command interpreter.

ctrl-X k

Kills buffer, but gives a warning if PDSS is running.

b. KL1 program control

ctrl-C ctrl-Z

Inserts 1 into the attention stream of the KL1 window process that corresponds to current buffer. This is treated as a task stop request by Micro PIMOS.

ctrl-C ctrl-T

Puts 2 in the same buffer as above. This causes display of statistic information from Micro PIMOS.

c. Emulator control

ctrl-C !

Garbage collection request.

ctrl-C @

Stops PDSS system, but the buffers used as Micro PIMOS windows are left untouched.

ctrl-C ESC

Restarts PDSS.

3 Mode independent command

ctrl-C ctrl-P

Displays next PDSS-mode buffer in current window. The PDSS buffer group is managed as a circular list. So, if user repeats this command, all buffers are displayed one by one.

ctrl-C p

This is almost the same as the previous command, but display occurs in the other window.

Appendix-10 Using command procedures for compiling

This is the description of the command procedure to compile a KL1 program, used as a UNIX command. It may be useful to compile it within a makefile. There are two versions of this command : one for the KL1/KL1 compiler and the other for the KL1/Prolog compiler. Basic usage rules are the same, but some available options are different.

Command :

pdsscmp [options] file names ...

Options :

`+e / -e` :: Macro expansion is performed or not. Default is to perform it. KL1/Prolog only.

`+i / -i` :: Indexing code is generated or not. Default is not to generate it. KL1/Prolog only.

`+d / -d` :: Deadlock detection code is generated or not. Default is to generate it. Speed performance is a little bit worse. KL1/Prolog only. (on KL1/KL1 compiler, this code is always generated.)

`+a / -a` :: Assemble is performed or not. Default is to perform it. When performed, an assembler file (xxx.asm) and a save file (xxx.sav) are generated. Otherwise, only assembler file is generated.

`+s / -s` :: Compiles for Micro PIMOS or for user. Default is to compile for user. System-mode private built-in predicates can be used in the first case. Built-in predicates in this manual can be compiled with the user version. KL1/Prolog only. (All built-in predicates can be compiled in KL1/KL1.)

`-o=PATH` :: Changes output directory to `PATH`. Current working directory is the default.

File name :

xxx.asm :: Assembles an assembler-file (xxx.asm) and creates a save file (xxx.sav).

xxx.kl1 :: Compiles a source file (xxx.kl1), makes assembler file and then assembles it to make save file.

xxx :: same as xxx.kl1.

Examples :

- To compile and assemble the two source files append.kl1 and queen.kl1, and then to make append.asm, append.sav, queen.asm and queen.sav in the current directory :

```
pdsscmp  append.kl1 queen.kl1    or
pdsscmp  append queen
```

- To compile and assemble append.kl1 and assemble queen.asm :

```
pdsscmp  append.kl1 queen.asm
```

- To compile and assemble all .kl1 files in the directory `source` and then to put assemble and save files in directory `object`:

```
pdsscmp  -o=object source/*.kl1
```

Appendix-11 Sample program

```
:- module sample.
:- public primes/2, primes/1.

primes(N, PL) :- true | gen(2, N, NL), sift(NL, PL).

primes(N) :- true |
    gen(2, N, NL), sift(NL, PL),
    window:create([show|Window], "sample"),
    outconv(PL, Window).

gen(Max, S):- true | gen(1, Max, S).
gen(N, Max, S) :- N =< Max, M := N+1 | S=[N|S1], gen(M, Max, S1).
gen(N, Max, S) :- N > Max | S=[].

sift([P|L], S) :- true | S=[P|S1], filter(P, L, K), sift(K, S1).
sift([],    S) :- true | S=[].

filter(P, [Q|L], K) :- Q mod P=:=0 |          filter(P, L, K).
filter(P, [Q|L], K) :- Q mod P=\=0 | K=[Q|K1], filter(P, L, K1).
filter(P, [],    K) :- true        | K=[].

outconv([P|PL], W) :- true | W=[putt(P),nl|W1], outconv(PL, W1).
outconv([], W) :- true | W=[putb("END"),getc(_)].


| ?- sample:primes(10,PL).

yes.
| ?- sample:primes(10,PL)|PL.
PL = [2,3,5,7]

yes.
| ?- sample:primes(10).
2
3
5
7
END

yes.
| ?- halt.
```

Appendix-12 What to do if a bug is found out...

1. When you find system bugs, please inform the PDSS development group. E-mail address is :

   pdss@icot21.icot.junet

   In your mail, include the following information :

   a. PDSS (emulator, Micro PIMOS) version number.

   b. Compiler version number.

   c. The program in which you found the bug.

   d. How to start it, and what happens.

   e. Execution log and weird points.

2. If it is a bug of a program of your own, go at the least through the following list :

   a. Have you done varchk?

   b. In case of deadlock, if there are the following goals in the incrimined part, you may have forgotten to close command stream to file or window, or you may have requested the output of undefined variables. Check your code.

      ```
      mpimos_file:xxxxxx( ... )      or
      mpimos_window:xxxxxxx( ... )
      ```

   c. Always in the case of deadlock, if the following goals are in the locked part, there must be multiple reference paths to the input streams of a merger and you have forgotten to close some stream to the merger. Check your program.

      ```
      pdss_runtime_body_builtin:active_unify(X^, Z^)
      pdss_runtime_body_builtin:active_unify(Y^, Z^)
      ```

Index