ICOT Technical Memorandum: TM-0884

TM-0884

PIMOS 1.5 Introductory Manual

by
T. Chikayama & K. Suzaki

May, 1990

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

PIMOS 1.5
Introductory Manual

Contents

Chapter 1

Introduction

This manual is intended to give the novice user an introduction to use the PIMOS operating system version 1.5 on either a Pseudo Multi-PSI or an actual Multi-PSI machine. When refering to either one, Pseudo-Multi or Multi-PSI, it is written as "(Pseudo) Multi-PSI." Most operations described in this manual apply to either system, but when one doesn't it is explicitly stated. This text first explains how to set up the PIMOS environment. Then it gives examples showing how to develop, compile, execute, debug, and evaluate KL1 programs. Finally it describes some of the basic PIMOS utility programs.

Before reading this manual, you should have finished installing the KL1 system (PIMOS, CSP, and CAL) on the PSI machine's SIMPOS operating system. The "(Pseudo) Multi-PSI System Administration Manual"[1] gives detailed instructions on how to do this. It is also assumed that you are familiar with the following SIMPOS operations as explained in the "Basic PSI/SIMPOS Operating Manual (1),(2)."

- Basic Operations

  Logging in/out.
  Using the SYSTEM MENU, selecting items, and opening windows using the mouse.
  Shutting down the PSI-II.

- User Registration

  Registering users, registering new items in the SYSTEM MENU, and defining logical names of directories by editing the "login.com" file.

- The PMACS Editor

  Editing text files.

- The File Manipulator

  Creating directories and copying files.

This manual is composed of the following three major sections.

1. Installing PIMOS on the PSI-II, Booting the System, Login, and Shutdown (Chapters 2 and 3)
   These chapters show how to install and boot PIMOS, and gives the steps for an individual user to log on and shut down the machine.

2. Writing and Executing KL1 Programs under PIMOS. (Chapters 4 to 7)
   These chapters show how to write a KL1 program, how to distribute processes over multiple processors, and how to execute and debug programs. They also show how to deal with some common problems you may encounter.

3. PIMOS Utilities (Chapters 8 to 11)
   This section describes how to use the shell, the compiler, the Listener, and several other basic PIMOS utilities

While reading this manual you should have a (Pseudo) Multi PSI machine available to perform each of the operations illustrated. If you need a more detailed explanation of the PIMOS facilities refer to the "PIMOS 1.5 Operating Manual"[3]. For information on developing more elaborate KL1 programs, consult the "KL1 Programming Manual"[5].

Now, onto the Multi-PSI world....

Chapter 2

Preparing To Run PIMOS

This chapter explains how to register a PIMOS user and set up their environment before booting PIMOS.

2.1 User Registration

Before your personal user name is registered, you can login in to the PSI using the following temporary login names. The password for each account is the same as its login name.

pmpsi: A Pseudo Multi-PSI user
mpsi: A Master FEP user on Multi-PSI
slave: A Slave FEP user on Multi-PSI

After you have logged in, you will notice the following two PIMOS related items in the SYSTEM MENU.

- Pseudo Multi-PSI or Multi-PSI
  Select this to boot PIMOS.
- CAL
  Select this to use CAL, the Compiler, Assembler, Linker facility, to incorporate a KL1-program into PIMOS as a utility by linking it with the PIMOS kernel object.

The above user names should only be used temporarily. Before developing KL1-programs on PSI, you should create a new username in the following way.

1. Register the new user name.
   (On a Multi-PSI, this must be done on the Master FEP.)

   If you have already logged in to SIMPOS, log out. Log in as "superuser" with the password "superuser". Select the following sequence of menu items on the console, starting with "others" from the SYSTEM MENU.

   others → user → maintenance → register

   Register the user name in the usual way by filling in the blanks presented in the window that appears. Pay careful attention to the rank and initial mode, being sure to use the following values.

For Pseudo Multi-PSI machine users :
rank= general, initial mode = general
For Multi-PSI machine users :
rank= general, initial mode = general

After completing registration of the new user name, login under that name. For a more detailed guide to user maintenance, refer to the "Basic System Administration Manual for PSI/SIMPOS Systems"[6].

2. Set up the new user's environment.

   The following example shows how to set up the environment for a new user name, "feldmark", by copying the "login.com" file from a standard user's directory and revising it for the new environment.

   (a) Copy the file "login.com" from user pmpsi to feldmark's home directory using the File Manipulator.

```
On a Pseudo Multi-PSI system:
copy file >sys>user>pmpsi>login.com
     to   >sys>user>feldmark>*.*

On a Multi-PSI system:
copy file >sys>user>mpsi>login.com
     to   >sys>user>feldmark>*.*
```

   (b) Edit this new copy of login.com using the PMACS editor.

   - Confirm the following SYSTEM MENU registration classes in the login.com file.

```
menu :-
    items_list(
        ....................................
%%%  On a Pseudo Multi-PSI Machine use this:
     {"Pseudo Multi-PSI",  mpsicsp##pseudo_csp_main_program},

%%%  On a Multi-PSI Machine use this:
     {"Multi-PSI",         mpsicsp##csp_main_program},
              ....................................
%%%  CAL Cross System (Use this on either machine)
     {"CAL",               cal##cal_manipulator},
```

   - Change the definition of the logical name "me" to your own user name.

```
define :-
%%%%        "me" := ["user:pmpsi"],
            "me" := ["user:feldmark"],
          ....................
```

   - Confirm the following logical name definitions :

```
On a Pseudo Multi-PSI machine :
---
       "root"  := ["user:pmpsi"],
```

```
        "psys"  := [">sys>csp>SYSTEM.DIR"],
        "pmpsi" := [">sys>csp>PMPSI.DIR"],
        "pimos" := [">sys>csp>PIMOS.DIR"],
---
On a Multi-PSI machine :
---

        "root"  := [">sys>csp"],
        "mpsi"  := ["root:MPSI.DIR"],
        "msys"  := ["root:SYSTEM.DIR"],
        "pimos" := [">sys>csp>PIMOS.DIR"],
---
```

(c) When you have finished confirming the above information and making the necessary changes, log out and log back in order to reset the parts of your environment that you have changed.

(d) Check that the item "Pseudo Multi-PSI" or "Multi-PSI" and the item "CAL" appear in your SYSTEM MENU.

(e) Do the following only on a real Multi-PSI machine. (Never create this file on a Slave FEP.) Edit the file:

msys:MPSI.CONFIG (i.e `>sys>csp>SYSTEM.DIR>MPSI.CONFIG`)

This sets up a new work address for each FEP according to the explanation in Section 6.3.5 of the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2]. Note that you should change only the network addresses and should never change the other parameters in this file.

2.2 Customization

- Create the following PIMOS system directories in your home directory :

```
me:SYSTEM.DIR   (Pseudo Multi-PSI only)
me:PIMOS.DIR
```

- In your "login.com" file, redefine the logical name "pimos". On a Pseudo Multi-PSI machine, also redefine the logical name "psys". Both changes are shown below:

```
%%%  "pimos" := ["sys>csp>PIMOS.DIR"],
     "pimos" := ["me:PIMOS.DIR"],
%%%  "psys"  := [">sys>csp>SYSTEM.DIR"],
     "psys"  := ["me:SYSTEM.DIR"],      (Pseudo Multi-PSI only)
```

- If you are using a Pseudo Multi-PSI machine, and copy the following files from "psys:" into your "SYSTEM.DIR" directory.

```
MPSI.CONFIG
MPSI.PARAM
boot.init
usercom.init
```

- Copy the following files from the user "pimos:" to your "PIMOS.DIR" directory. (Do not copy the file "pimos.mac" which may also be in the "pimos" directory.)

```
pimos.conf
pimos.kbn
pimos.ldb
pimos.sym
pimos.users
```

- Copy the file "pimos.users" and make the following modifications to the definitions in this file.

```
%%% [system(*,*,">sys>user>ShellUser",*,
    [system(*,*,">sys>user>feldmark",*,

%%% [system(*,*,">sys>user>ListenerUser",*,
    [system(*,*,">sys>user>feldmark",*,
```

  This file is called the "User Environment File," and it defines system login names and tasks to be run when you boot PIMOS.

  You can register any names or tasks (including your own programs) in this file. For more details about the "User Environment File," refer to Section 12.3 in the "PIMOS 1.5 Operating Manual"[3].

- On the Pseudo Multi-PSI, you can change things like the number of pseudo processing elements, the system configuration, etc. via the configuration file :

```
psys:MPSI.CONFIG
```

  For more detail, refer to chapters 4 and 6 in the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

- Once more, log out and log back in to reset the parts of your environment that you have just changed.

Chapter 3

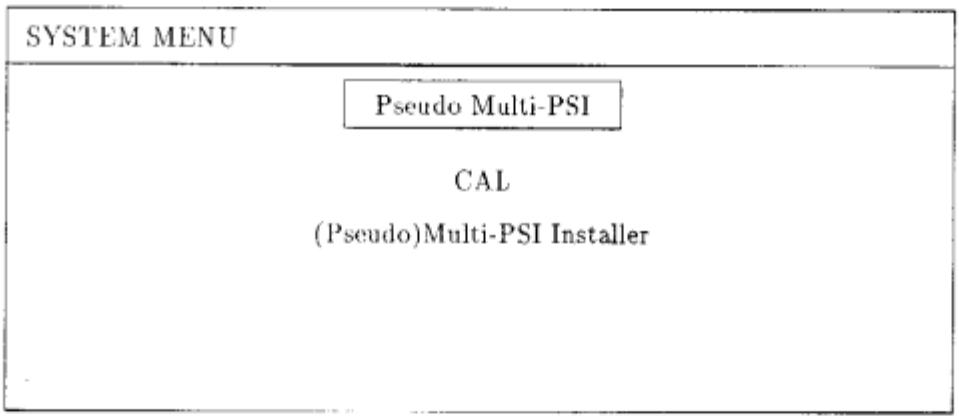
Booting Up, Logging Into, and Shutting Down PIMOS

This chapter explains how to boot up, login to, and shut down your (Pseudo) Multi-PSI machine, along with what to do if booting fails.

3.1 Booting CSP on (Pseudo) Multi-PSI

3.1.1 Pseudo Multi-PSI

On a Pseudo Multi PSI machine, select the item "Pseudo Multi-PSI" in the system menu that appears after logging in and clicking the mouse's left-button.



After the "Multi-PSI CSP" program starts up, you will see the "Monitor Panel" in the upper-left corner of your console and the "CSP Command Window" on the right-hand side of the panel.

The first time this sequence is performed each time after booting SIMPOS, it may take a long time since many CSP programs must be loaded.

The following messages appear in the "CSP Command Window".

```
Initiating all PEs from configuration file MPSI.CONFIG
Current PEs are
    0, 1, 2, 3, 4, 5, 6, 7
Current Default PEs are
    0, 1, 2, 3, 4, 5, 6, 7
PE auto power on... All PEs power ready.
PEs initiation completed
```

```
$
```

The $ that appears is the CSP prompt symbol.

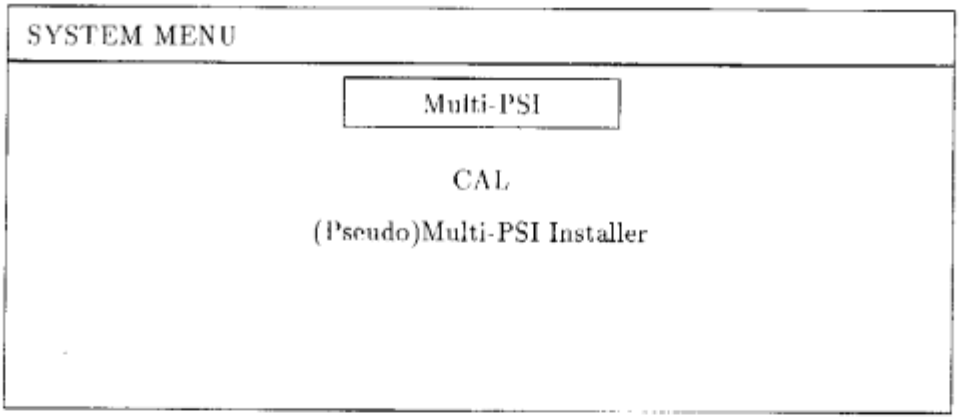
3.1.2 Multi-PSI

This sections explains how to boot a Multi-PSI machine in which all FEPs are connected by a "PSI-net." Details on how to boot with other configurations can be found in Chapters 4 and 6 of the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

1. Turn on the power switches for the master and all slave FEPs. Confirm that all FEPs are functioning and SIMPOS is running.
2. Login as "slave" on all slave FEPs, using "slave" as the password.
3. The following message should appear (in both Japanese and English) on each slave display to indicate that it is waiting for a response from the master FEP.

   ただいまマスタ CSP からの通信待機中です. しばらくお待ち下さい.
   Now Waiting Orders from Master CSP. Please wait ...

4. Login to the master FEP with your own account.
5. Select the item "Multi-PSI" from the SYSTEM MENU.



6. Confirm that CSP has started up on all FEPs.
7. The following message appearing on a slave FEP indicates that the slave configration file "sys:MSI.CONFIG" is missing, but this message may be ignored.

   Configuration File doesn't exist. MPSI.CONFIG.
   Slave FEP-CSP Program is Starting !!!

8. On the master FEP, you will see the "Monitor Panel" in the upper-left corner of your console and the "CSP Command Window" on the right hand side of the panel.

The first time this sequence is performed each time after booting SIMPOS, it may take a long time since many CSP programs must be loaded.

9. The following messages will appear in the "CSP Command Window".

```
Initiating all PEs from configuration file MPSI.CONFIG
Current PEs are
    0, 1, 2, 3, 4, 5, 6, 7
Current Default PEs are
    0, 1, 2, 3, 4, 5, 6, 7
PE auto power on... All pes power ready.  Memory Configuration...
PEs initiation completed

$
```

   If you have not set the Auto-Boot switch (explained in the next section) the CSP prompt symbol, $, as shown above, also appears.

All CSP commands can be displayed by typing "?" to this prompt, and each command is also described in detail in the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

3.2 Setting the Auto-Boot Switch

Booting PIMOS can be done automatically by setting the auto-boot switch in the CSP Command Window. From the CSP Command Window, type the following command.

```
$ panel
```

When you do this, the "CSP Console Panel" should appear. The CSP Control Panel is made up of several sub-menus, which you should set as shown below.

1. Auto-Boot switch definitions:

CLEAR	[ON]	OFF
INIT	[ON]	OFF
IMPL	[ON]	OFF
Patch	[ON]	OFF
- - - - - - - - - - - - - - - - - -		
IPL	[ON]	OFF
START		
[PIMOS]	ON	OFF

2. Boot file name definitions:

   - Pseudo Multi-PSI:

CLEAR	FILE (.mbn)	[(Don't care on Pseudo)]
INIT	FILE (.mbn)	[(Don't care on Pseudo)]
INIT	FILE (.mbn)	[(Don't care on Pseudo)]
IMPL	FILE (.mbn)	[(Don't care on Pseudo)]
IMPL	FILE (.kbn)	[pmpsi:kl1kp.kbn]
PATCH	FILE (.com)	[pmpsi:kl1pat.com]
IPL	FILE (.kbn)	[pimos:pimos.kbn]
SYMBOL	FILE (.sym)	[pimos:pimos.sym]
START	ADDRESS	[202]

- Multi-PSI :

CLEAR	FILE (.mbn)	mpsi:MCLEAR.MBN
INIT	FILE (.mbn)	mpsi:MINIT.MBN
INIT	FILE (.mbn)	mpsi:MIDAT.MBN
IMPL	FILE (.mbn)	mpsi:MPSI.MBN
IMPL	FILE (.kbn)	(Don't care on MPSI)
PATCH	FILE (.com)	mpsi:MPSIPAT.com
IPL	FILE (.kbn)	pimos:pimos.kbn
SYMBOL	FILE (.sym)	pimos:pimos.sym
START	ADDRESS	2000

3. Select the "AUTO" setting by clicking on it once with the left mouse button.

   Power On Auto Boot [AUTO] CONSOLE

4. Next, select "SAVE" to save the settings you have choosen.

   [SAVE] EXIT

5. Finally, exit by selecting the "EXIT" box.

   SAVE [EXIT]

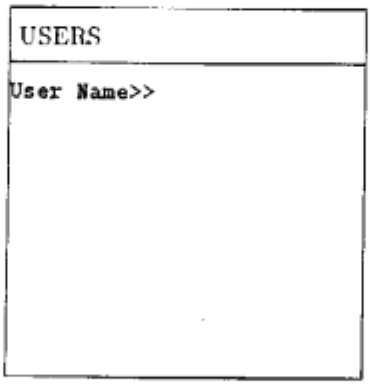
3.3 Booting PIMOS from CSP

To bootstrap PIMOS from CSP, type the "boot" command into the "CSP Command Window". If you are using a real Multi-PSI machine, all processors will automatically turn on.

```
$ boot
```

3.4 Logging In

After booting PIMOS, the "Users Window"appears as shown below via which you can use the PIMOS utilities.



You can type any user name that is registered in the file.

```
pimos:pimos.users
```

For a detailed explanation of this User Environment File, refer to Section 12.3 of the "PIMOS 1.5 Operating Manual"[3].

The following three user names are initially available:

```
ShellUser, ListenerUser, shutdown
```

If you want to use the Shell, explained in Chapter 8, log in as ShellUser.

```
User Name>>ShellUser
```

If you want to use the Listener, explained in Chapter 9, log in as ListenerUser.

```
User Name>>ListenerUser
```

Using only these two utilities, you can compile, execute, and debug KL1 programs.

Note that PIMOS has no command to logout, and that you may login simultaneously as many times as you wish. For example, you can login twice as ShellUser in order to use two Shell utilities.

The third user name, "shutdown," is explained in the next section.

3.5 Shutting Down

To shutdown PIMOS, type "shutdown" into the Users Window.

```
User Name>> shutdown
```

Answer "yes" when asked if you really want to shut down. (Or no if you have made a mistake and didn't really intend to shut down PIMOS.)

```
Really(yes/no)? yes
```

You will also be asked to confirm that you want to leave CSP. Type "y".

```
Do you want to exit from CSP ? [Y/N] : y
```

You have now shut down (Pseudo) Multi-PSI. On Multi-PSI, the power to all processors will automatically be turned off. On both Pseudo Multi-PSI and Multi-PSI, you should also manually turn off the power to the PSI-II.

For more details about the shutdown process, refer to the "PIMOS 1.5 Operating Manual"[3]. For more information about CSP and Multi-PSI's power supply, refer to the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

3.6 Returning to CSP from PIMOS

If for some reason you want interrupt PIMOS and return to CSP, you can do so in either of the following ways:

1. Select the CSP Command Window by clicking the left mouse button twice and type a control-C to the CSP prompt.

   ```
   $ Control-C
   ```

2. Alternatively, this sequence of commands is also available.

- Click the left mouse button while in the Monitor Panel.
- Push the "abort" key.
- Select the following from the window that appears.

  "Return to csp top level"

To return to PIMOS, use the "go" command in response to the CSP prompt.

```
$go
```

3.7 If Booting Fails

In Pseudo Multi-PSI, PIMOS may fail to boot due to problems with the way the following PIMOS kernel objects were linked by the CAL system.

```
pimos:pimos.kbn
pimos:pimos.ldb
pimos:pimos.sys
```

- "Copying GC Memory Shortage" message
  If a message of the form "Copying GC Memory Shortage" appears in the CSP Command Window on a Pseudo Multi Machine, you should change the configuration of your pseudo machine to one that has fewer processors. Refer to the last section of Chapter 2 for how to do this. Then enter the command "exit" in the CSP Command Window and reboot.
- User program bugs
  Another possible reason for not being able to boot PIMOS is that a user program linked to a PIMOS kernel object has a bug. In this case you should remove the three PIMOS kernel object files mentioned above, and make new copies of the originals from the directory `">sys>csp>PIMOS.DIR"`. Shutdown PIMOS and try to reboot. If successful, you may debug your program(s) and relink them using CAL.
- Too many programs linked to PIMOS
  If the programs linked to the kernel objects have no bugs, it may be the case that you have used CAL to link them into PIMOS too many times. This will cause PIMOS to fail because of too many patches to the kernel objects. In this case you should remove the three PIMOS kernel object files mentioned above, make new copies of the originals from the directory `">sys>csp>PIMOS.DIR"`, and use CAL to recompile and link your programs. Shutdown PIMOS and try to reboot.
- If none of the above solutions work, once again, delete and recopy the above kernel object files and reboot PIMOS. Then, compile your programs with the PIMOS Native-Compiler, not with CAL. Confirm that your programs run correctly before trying to to compile and link them to the kernel objects again with CAL.

  If you still cannot boot PIMOS, you may have found a bug in either PIMOS or CAL. Please submit a bug report along with copies of the programs you have linked to the kernel objects as described in the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

Chapter 4

Programming in KL1

The parallel logic programming language KL1 has many features that can be used to efficiently describe parallel algorithms and systems. It is based on the parallel logic programming language, GHC, developed at the Institute for New Generation Computer Technology (ICOT) as part of the Japanese Fifth Generation Computer Project.

This chapter explains how to develop KL1 programs on a (Pseudo) Multi-PSI machine, assuming that you know how to write simple programs in Prolog.

4.1 Features of KL1

The following section outlines the main features of KL1.

4.1.1 Selection and Execution of Clauses

Each KL1 clause is divided into two parts by an operator, |, called the "commit operator". The goals on the left-hand side of this commit operator form the "guard" part, while those on the right of the commit operator form the "body" of the clause. Any type of goal may be used in the body part of a clause, but only built-in predicates, described later in this section, may be used in the guard part. The first goal of a clause is called the "head" predicate, in the example below, "a".

```
a(X) :- integer(X) | b1(X,Y), b2(Y).        %(0-1)
a(X) :- X=3        | c1(X,Y), c2(Y,Z), c3(Z). %(0-2)
a(X) :- list(X)    | d1(X,Y), d2(Y), d3(Y).   %(0-3)
```

All clauses with the same head are in an OR-relationship. That is, the predicate succeeds if the first clause is true, OR the second clause is true, OR etc.) The guard part plays an important role in selecting one clause to be executed from all those in this OR-relationship. Suppose the goal "?- a(3)" is called from another clause. Each of the above definitions, (0-1), (0-2), and (0-3) can be tested in parallel, but only the two guards in (0-1) and (0-2) (integer(3), 3=3) will succeed. The KL1 system selects and executes one clause at random from those whose guard part succeeds, and discards the rest. All goals from the body of this clause are then activated in a random order. In this case, either the body part from clause (0-1) or from clause (0-2) will be activated.

Suppose that the clause (0-2) was selected. The goals c1, c2, and c3 will be called in an order irrelevant to their order in clause (0-2). In fact, they may all be executed in parallel on different processors. How to force goals to be executed parallel is explained in the next chapter.

4.1.2 Failure of Goals

The commit operator, |, in KL1 programs works like the cut operator, !, in Prolog. The failure of a goal in a guard part on the left-hand side of the commit operator can be thought of as causing the guard part of another (randomly selected) clause to be tried. But if a body goal on the right hand side of a commit operator fails, it means that the parent goal also fails. The failure of the parent goal causes a failure of the grand-parent goal. This continues until all the goals in the KL1 program have failed.

4.1.3 Kl1 Data Types

The following data types are available in KL1.

- Variable (`Var,_`, etc.)
- Integer (`138, 16'8A`, etc.)
- Floating point (`1.23, 1.0E10`, etc.)
- Atom (`a, 'ABC'`, etc.)
- List (`[1,2,3], [a,[A,b],c|Y]`, etc.)
- Vector (`{a,b,c}, {}, a(b)`, etc.)
- String (`"abc", ""`, etc.)
- Module (`module#foo`, etc.)

Refer to Section 2.1.3 of the "PIMOS 1.5 Operating Manual" [3] for more details on these data types.

4.1.4 Built-in Predicates

There are three types of built-in predicates in KL1. The first type can only be used in the guard part, the second can only be used in the body, and the third can be used in both the guard part and the body. All PIMOS built-in predicates are listed in Section 2.2 of the "PIMOS 1.5 Operating Manual"[3].

4.2 A Sample KL1 Program

Next is an actual KL1 program, which should be entered into a file named "sample1.kl1". Note that any file containing a KL1 program must have the ".kl1" extension at the end of its name.

```
:- module sample1.                          %(1-1)
:- public sieve/3.                          %(1-2)

sieve([],Even,Odd):- true |                 %(1-3)
        Even=[],
        Odd=[].
sieve([N|List],Even,Odd):-                  %(1-4)
        N mod 2 =:= 0 |
```

```
        Even=[N|Tail],
        sieve(List,Tail,Odd).
sieve([N|List],Even,Odd):-                %(1-5)
        N mod 2 =\= 0 |
        Odd=[N|Tail],
        sieve(List,Even,Tail).            %(1-6)
```

Here you can see the use of the "=:=" operator, which performs a comparison of the two values on either side of it. For an explanation of these kinds of operators in KL1, refer to Section 2.1.6 of the "PIMOS 1.5 Operating Manual"[3].

Line (1-1) declares the module name of this program to be "sample1". Line (1-2) declares that the predicate named "sieve" may be called from external modules and that it has 3 arguments. Both of these two declarations must be placed at the top of any file that contains a KL1 program.

We can call any goal in a module from any other module or from the Listener prompt (see Chapter 9) by using the following format.

```
module_name:goal_name
```

To call the goal "sieve" from the Listener type :

```
?- sample1:sieve([2,5,6,8,0,9],Even,Odd).
```

This goal will be matched with the clauses (1-4) and (1-5), but only the guard part of (1 4) will succeed. The body part of (1-4) will then be activated, resulting in the unification :

```
Even = [2|Tail]
```

and the recursive call of the goal :

```
sieve([5,6,8,0,9],Tail,Odd).
```

As in the above example, all instantiation of output variables (such as `Even=[N|Tail]`) must be done in the body part of each clause. This restriction comes from the synchronization mechanism of KL1 processes, and will be explained in the next section.

The goal `sieve([5,6,8,0,9],Tail,Odd)` called recursively will be successfully matched with clause (1-5), and the goals of its body part :

```
Odd=[5|Tail], and sieve([6,8,0,9],Odd,Tail)
```

will be executed. Finally, the result of this program will be the instantiations `Odd=[5,9]`, `Even=[2,6,8,0]`.

Clause (1-5) in the above program could be replaced with the following one:

```
otherwise.
sieve([N|List],Odd,Even):-                %(1-5')
        true |
        Odd=[N|Tail],
        sieve(List,Tail,Even).
```

The `otherwise` statement is a simple expression indicating negation. If inserted among clauses that are placed in OR-relationships, the KL1 system selects the clauses under the `otherwise` statement only when all clauses above it fail. So, by using the `otherwise` statement, it is not necessary to describe the negative conditions of the guard parts of all the other clauses.

Here is another example of how to use the `otherwise.` statement.

```
:- module sample2.
:- public check/2.

check([], Out):- true | Out = []
check([M|In], Out):- atom(M) |
        Out = [atom|Out1], check(In, Out1).
check([M|In], Out):- integer(M) |
        Out = [integer|Out1], check(In, Out1).
check([M|In], Out):- string(M,_,_) |
        Out = [string|Out1], check(In, Out1).
check([M|In], Out):- list(M) |
        Out = [list|Out1], check(In, Out1).
check([M|In], Out):- vector(M,_) |
        Out = [vector|Out1], check(In, Out1).
otherwise.
check([_|In], Out):- true |                          %(2-1)
        check(In, Out).
```

Clause (2-1) is a definition for ignoring exceptional input data. The program executes as follows.

```
?- sample2:check([1,a,"ds",23.5,{a,b}],X).
X = [integer, atom, string, vector]
yes.
?-
```

The goals defined under the `otherwise` statement were responsible for ignoring the data 23.5 in the input list.

4.3 Stream Communication and Process Synchronization

Consider the following example :

```
:- module sample3.
:- public go/3, generate/2.

go(Max,Even,Odd):- true |
        generate(Max,Numbers),                       %(3-1)
        sample1:sieve(Numbers,Even,Odd).             %(3-2)

generate(0,Numbers):- true |
        Numbers=[].
otherwise.
generate(N,Numbers):- true |
        Numbers=[N|NewNumbers],
        N1 := N-1,
        generate(N1,NewNumbers).
```

When we call the goal "go" with an appropriate number as the first argument, it activates the two subgoals "generate" and "sample1:sieve." The goal "generate" generates a list of integers in the variable "Numbers." The goal "sieve" defined in the module "sample1" divides the list "Numbers" into two lists "Even" and "Odd." Both predicates "generate" and "sieve" continue to run by calling themselves recursively. We call predicates that do this "processes".

Now assume that the two process are running in parallel on different processors. These processes share the variable "Numbers" which is instantiated by the process "generate" and is read by the process "sieve". We call this kind of communication between processes by shared variables "stream communication."

When the process "sieve" runs faster than the process "generate," "sieve" waits for data in the shared variable "Numbers" to be instantiated by "generate." This synchronization is performed automatically by the KL1 system.

In the above program, consider the process "sieve" called with no data in the first argument :

```
sieve(List,Even,Tail)                          %(1-6)
```

This goal could be matched with clauses (1-3), (1-4), or (1-5) by one of the following unifications :

```
List = [] or List = [N|List']
```

Unifications that try to instantiate a variable in the calling goal are suspended until the variable becomes instantiated by another process (in this case "generate".) This mechanism controls the process "sieve" and prevents it from running faster than the process "generate." It provides us with a natural facility for describing process synchronization.

4.4 Merging Streams

To merge the output from the streams of two or more subprocesses into one stream, we can use a "merge" process that is easily defined by a KL1 program, but available as a built-in predicate in KL1 since it is used so frequently.

The "merge" predicate takes an arbitrary number of streams as input and merges them into one output stream. If data is sent on any input stream, the process will immediately forward the data on the output stream. Unlike the "append" program in Prolog, the sequence of data on output stream of a "merge" process is unrelated to the order of data on the input streams. The builtin predicate, "merge," is called in the following way.

```
merge(In,Out)
```

The first argument must be instantiated to a vector of input streams such as:

```
In = { In1,In2, ... }
```

The second argument is a single output stream.

The following example shows how to use the merge predicate.

```
:- module sample4.
:- public flatten/2.
flatten([],Out):- true |                    %(4-1)
        Out=[].
flatten([Top|Tail],Out):- true |            %(4-2)
        flatten(Top,X1),
        flatten(Tail,X2),
```

```
        merge({X1,X2},Out).
flatten(A,Out):- atom(A),A\=[] |        %(4-3)
        Out=[A].
```

This program receives a list, possibly containing other lists, as the first argument. In the second argument, it returns a list of all atoms that were anywhere in the input list. This invocation,

```
?- sample3:flatten([a,[b,c],[d,[e,f]]],X).
```

returns the following list :

```
X = [ c,a,f,e,d,b ]
```

Note that the order of the atoms a,b,c,... cannot be determined by looking at the program because the order of execution of the two goals "flatten" in clause (4-2) cannot be determined.

The goal merge({X1,X2},Out) will immediately send data received from the input streams "X1" and "X2" into the output stream "Out". When the goal "flatten" has consumed all data in the first argument, the clause (4-1) will put [] in the stream "Out." This instantiation unifies the "Cdr" part of the list with [], and the stream is "closed." Thus, merge will close its output stream when all of its input streams are closed.

Execution of the goal "flatten" in the program "sample3" will generate many instances of merge. But generating too many instances of merge will degrade the efficiency of your KL1 programs. We can modify the original "flatten" example and come up with the following program that avoids this problem.

```
:- module sample5.
:- public flatten/2.
flatten(In,Out):- true |
        flatten1(In,X),
        merge(X,Out).
flatten1([],Out):- true |
        Out=[].
flatten1([Top|Tail],Out):- true |
        flatten1(Top,X1),
        flatten1(Tail,X2),
        Out = {X1,X2}.
flatten1(A,Out):- atom(A),A\=[] |
        Out=[A].
```

4.5 I/O Interface

This section shows how to use the window interface utilities and how to do file I/O within KL1 programs. Details can be found in Sections 3.5, 4, 5, and 8.3 of the "PIMOS 1.5 Operating Manual"[3].

4.5.1 Standard I/O Devices

Any KL1 program can access a window of the Shell or the Listener by using the Standard I/O device represented by the streams Standard-Input, Standard-Output, Standard-Input/Output, Message-Output, and Standard-Interaction.

In order to perform I/O, messages are sent to these streams requesting operations such as "getc(C)" to read a character, "putc(C)" to output a character, etc. A complete list of these messages can be found in Section 3.5 of the "PIMOS 1.5 Operating Manual"[3].

The following module shows how to access the streams Standard-Input, Standard-Output, and Message-Output.

```
:- module std_io.
:- public create/3.

create(Input,Output,Message):- true|
    shoen:raise(pimos_tag#shell,get_std_in,Input),
    shoen:raise(pimos_tag#shell,get_std_out,Output),
    shoen:raise(pimos_tag#shell,get_std_mes,Message).
```

This module can be called as :

```
?- std_io:create(Input,Output,Message),
   Input=[ ... ], Output=[ ... ], Message=[ ... ].
```

Which results in the following streams :

- Input : A Standard-Input device stream
  Accepts any message provided by a `buffer:input_filter`.
- Output : A Standard-Output device stream
  Accepts any messages provided by a `buffer:output_filter`.
- Message : A Message-Output device stream
  Accepts any messages provided by a `buffer:output_filter`.

An explanation of the buffer and the filter utilities can also be found in Section 3.5 of the "PIMOS 1.5 Operating Manual"[3].

4.5.2 Window I/O

The following module creates a window for input and output.

```
:- module window.
:- public create/1.

create(Window):- true|
    shoen:raise(pimos_tag#task,
        general_request,General_Request_Device),
    General_Request_Device=
        [window(normal(Window_Request_Device,_,_))],
    Window_Request_Device=
        [create(normal(Window_Device,_,_))],
    Window_Device=[
%       set_size(mouse,_),
        set_size(char(50,25),_),
```

```
        set_position(mouse,_),
%        set_position(at(0,0),_),
        set_title("Interactive Window",_),
        activate(_)| Stream ],
    buffer:interaction_filter(Window,Stream).      %(5-1)
```

This module can be called as :

```
?- window:create(Window),
   Window=[ putl("Hit return to exit."), getl(_) ].
```

You will now see the frame of a window on your display. Click the mouse-button to confirm its position. If you hit the return key, the window will disappear immediately.

The variable "Window" becomes a stream to which we can send messages prepared by an I/O-filter to manipulate the window that just appeared. See Section 3.5.4 in the "PIMOS 1.5 Operating Manual"[3] for further details.

4.5.3 Files I/O

File I/O is performed in a similar manner to window I/O. A stream representing the file is created, and messages requesting file operations like "putc(C)" and "getc(C)" are sent to initiate I/O. The following module allows input and output to be done on a file.

```
:- module file.
:- public open/4.

open(FileName,Mode,File,Status):- true|
    shoen:raise(pimos_tag#task,
        general_request,General_Request_Device),
    General_Request_Device=
        [file(normal(File_Request_Device,_,_))],
    File_Request_Device=
        [open(FileName,Access)],
    ( Mode=r ->                                    %(6-1)
        Access=read(Result),
        ( Result=normal(Stream,_,_) ->
            buffer:input_filter(File,Stream),      %(6-2)
            Status=success
        ; otherwise
        ; true -> Status=Result )
    ; Mode=w ->
        Access=write(Result),
        ( Result=normal(Stream,_,_) ->
            buffer:output_filter(File,Stream),     %(6-3)
            Status=success
        ; otherwise
        ; true -> Status=Result )
    ; Mode=a ->
        Access=append(Result),
        ( Result=normal(Stream,_,_) ->
```

```
        buffer:output_filter(File,Stream),      %(6-4)
        Status=success
    ; otherwise
    ; true -> Status=Result ) ).
```

Here, the expression in (6-1) is a "Macro symbol" for conditional selection of execution. "Macro symbols" are explained in Section 2.1.6 of the "PIMOS 1.5 Operating Manual"[3].

The arguments in the above goal, `open(FileName, Mode, File, Status)` have the following meanings.

- FileName : The name of the file to open, specified as string data.
  `"sample.kl1"`, `">sys>user>feldmark>sample.kl1"`, etc.
- Mode : The mode for opening the file. (r, w, a)
  r → read, w → write, a → append
- File : A stream to the file opened.
- Status : Other information concerning the opening of this file. (success, failure, etc.)

This predicate returns a stream to the file which accpets any message provided by the following filter utilities :

- Mode = r
  Messages from `buffer:input_filter`
- Mode = w,a
  Messages from `buffer:output filter`

4.5.4 A Sample Program Using the I/O Interface

Below is a sample program that uses the PIMOS I/O interface. Before compiling this program, compile the three modules above, "std_io", "window", and "file". This program prints out the contents of the named files in a new window, and is executed as follows.

```
?- sample6:demo.
```

Input the name of any file when the program asks you to. Next, confirm the position of the window by clicking the mouse button when the window frame appears. The contents of the file are then displayed in the window.

```
:- module sample6.
:- public demo/0.

demo:- true|
    std_io:create(In,Out,Mes),
    In=[prompt("File name ? "),getl(FName)],
    Out=[putt('Opening file : '),putl(FName)],
    Mes=[putt(Status),nl],
    type_file(FName,Status).

type_file(FName,St):- string(FName,_,_)|
```

```
    file:open(FName,r,File,St),
    ( St=success ->
        window:create(Win),
        File=[getl(Line)|RF],
        type(Line,RF,Win)
    ; otherwise
    ; true -> true ).

type(-1,File,Win):- true|
    File=[],
    Win=[prompt("*** Hit Return key to exit.***"),
            getl(_)].
otherwise.
type(L,File,Win):- true|
    Win=[putl(L),flush(_)|W],
    File=[getl(N)|F],
    type(N,F,W).
```

4.6 Deadlock

As mentioned in the section "Stream Communications and Process Synchronizations", instantiation of undefined variables in a guard part will be suspended by the KL1 system. If the variable is not instantiated by any other processes, the suspended execution will never resume. We call this situation "Deadlock".

There are many situations where deadlock can occur. For example, if we modify clause (4-1) in the module sample4 like this:

```
flatten([],[]):- true|true.                    %(4-1')
```

The goal `?- flatten([],X1)` which activates this clause will be suspended because of an attempt to instantiate the variable X1 with []. This will cause a deadlock to occur because X1 is never instantiated by another process.

Or when we rewrite the same clause as :

```
flatten([],Out):- true|true.                   %(4-1")
```

a deadlock will occur because the variable "Out" that is read from a merge will never be instantiated to anything.

Chapter 6 shows how to detect deadlock in a KL1 program.

Chapter 5

Load Distribution

This chapter shows briefly how to do load distribution in KL1. In the current version of PIMOS, a processor number must be explicity assigned to a goal in order to make it execute on a different processor.

5.1 Implementing Load Distribution

Any KL1 body goal may be assigned to a specific processor by attaching an experssion containing the PE number in the following manner, where PE is an integer greater than or equal to zero.

```
goal@process(PE)
```

The following sample program illustrates this concept.

```
:- module sample7.
:- public foo/0.

foo:- true |
        a@processor(0),
        a@processor(1),
        a@processor(2).
a :- true | true.
```

Processor numbers may also be variables. Execution of the corresponding goal is suspended until the variable is instantiated to a number.

```
:- module sample8.
:- public foo/3.

foo(P1,P2, P3):- true |
        a@processor(P1),
        a@processor(P2),
        a@processor(P3).
a :- true | true.
```

5.2 An Example of Load Distribution

The following is a more complex example of load distribution.

```
:- module sample9.
:- public distribute/1.

distribute(N):- true |
        current_processor(_,X,Y),                    %(1)
        PEs := X*Y,                                  %(2)
        fork(N, PEs, 0)@priority(*,4096).            %(3)

fork(0, PEs, PE):- true | true.
otherwise.
fork(N, PEs, PE):- true |
        PeNo := PE mod PEs,
        job_ext@processor(PeNo),                     %(4)
        NextPE := PE+1,
        N1 := N-1,
        fork(N1, PEs, NextPE).

job_ext :- true | job@priority(*,2000).                    %(5)
job :- true | true.
```

In line (1), "current_processor" is a built-in predicate that returns the number of the processor on which it was executed along with the number of processors available in the horizontal(X) and vertical(Y) directions. In line (2), `PEs` becomes the total number of processors in the machine. In line (3), the goal `fork` that distributes the goal `job_ext` to other processors is called with the highest priority. (An explaination of priority can be found in Section 2.1.4 of the "PIMOS 1.5 Operating Manual"[3].) In line (4), the goal `job_ext` represents an actual job that is distributed to the processor numbered `PeNo`. Finally, in line (5), the goal `job` is called with lower priority than that of the goal `fork`.

If this program is executed in the following way,

```
?- sample9:distribute(8).
```

the `job` will be executed 8 times. If only 4 processors are available, the goals will be distributed to processors number 0 through 3, in the order `0,1,2,3,0,1,2,3` .

5.3 Efficient Load Distribution

In order to do load distribution in your programs efficiently, you should keep in mind the following fact. Since each processor in the Multi-PSI machine has its own private memory, communication between two processors is necessary whenever a goal on one processor reads data from a goal on another processor. If you distribute goals that need to communicate large amounts of data, the resulting communications overhead may degrade the parallel performance of your program.

The following suggestions may help.

- Never distribute goals that only have a small amount of work to do.
- If one goal must read data from another goal, it is desireable for them to be on the same processor.

- Databases that must be accessed by many goals should probably be distributed to each processor.
- Make any data that will be read by goals on other processors as compact as possible.
- Give high priority to any goal that will distribute other goals. Give low priority to goals that will themselves be distributed.

For more details about load distribution, see Section 11.5 of "KL1 Programming Manual"[5].

Chapter 6

Running KL1 Programs

This chapter explains the operations necessary to compile, execute, and debug KL1 programs along with how to evaluate their performance.

6.1 Compiling KL1 Programs

You can compile KL1 programs contained in files with a `.kl1` suffix using either of the following two utilities.

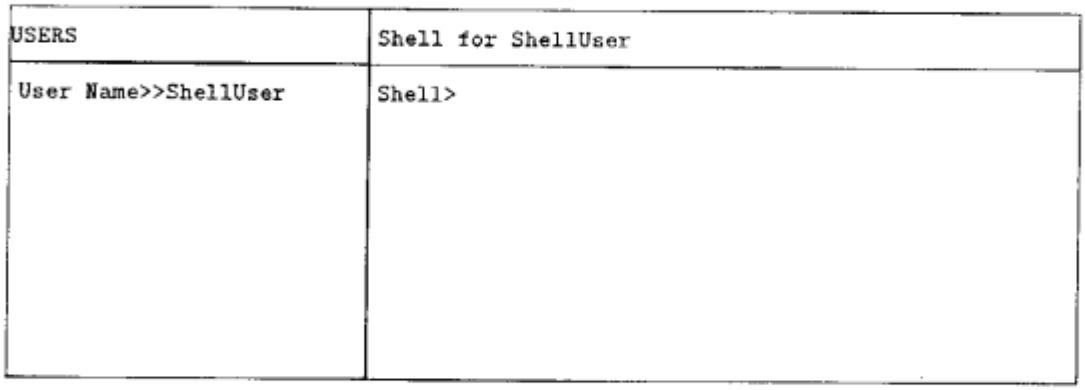
- The PIMOS resident Native-Compiler
- The CAL Cross-Compiler

When debugging, you should use the Native-Compiler. The CAL Cross-Compiler will be explained in the Chapter 10.

6.2 How to Use the PIMOS Native-Compiler

6.2.1 Starting It Up

To start up the Native-Compiler from a Shell, login to PIMOS as "ShellUser" in the Users Window.



You can find out what the current directory is by typing the "cd" command in the Shell Window.

```
Shell> cd
Illegal filename for take .              : Ignore this line.
```

```
The value of the task:directory is "sys>user>feldmark".
Shell>
```

To change current directory type :

```
Shell> cd "Directory Name"
```

Next, start up the Native-Compiler with the "compile" command.

```
Shell> compile
```

The following prompt appears in the window.

```
** KL1 Compiler **
COMPILE>
```

Type the name(s) of your KL1 program(s), omitting the ".kl1" extension. Two or more names can be input on the same line separated by commas. For example, you can compile the programs in the files "sample1.kl1", "sample2.kl1", and "sample3.kl1" with the following command.

```
COMPILE> sample1, sample2, sample3
```

To compile files in the other directories, type their names along with their path-names, in string data form, like :

```
">sys>user>your-name>directory-name>file-name"
```

If you need to change the current directory, exit the compiler by typing `!exit` at the prompt, change the current directory with the "cd" command, and start up the Native-Compiler again.

On a Pseudo Multi-PSI machine, compiling too many files at the same time may cause a memory shortage error. If this happens, divide the files into two or more groups that can be compiled at the same time, observing the following rules.

- Group together files (modules) that call each other.
- Compile lower groups before compiling upper groups. In other words, first compile groups that do not call other groups. Next, compile groups that call other groups that have already been compiled.

For example :

```
Compile File : **psi::>sys>**>sample1.kl1.1

Compile Module : sample1
sieve/3
Compile Succeeded : sample1

Compile File : **psi***::>sys>***>sample2.kl1.1

Compile Module : sample2
check/2
```

```
Compile Succeeded : sample2

Compile File : **psi***::>sys>***>sample3.kl1.1

Compile Module : sample3
go/3
generate/2
Compile Succeeded : sample3

"sample1" Loaded
"sample2" Loaded
"sample3" Loaded

Compilation(s) Succeeded
COMPILE>
```

To exit from the compiler and return to the shell prompt, type :

```
COMPILE>!exit
```

6.2.2 Batch Compilation

You can also compile your programs directly from the Shell Prompt.

```
Shell> compile(["sample1","sample2","sample3"])
```

This command is useful in a "Command File". (See Chapter 8.)

6.3 Executing Programs

KL1 programs can be executed with either of the following two utilities.

- The Listener (see Chapter 9)
- Shell (see Chapter 8.)

Listener utilities are very useful for debugging your programs.

6.3.1 Starting up the Listener

Two methods are available to start up the Listener.

- Login to PIMOS as "ListenerUser" from a Users Window.

- Type the "listener" command at the Shell Prompt, and set the position and size of the window that appears after clicking the mouse button.

USERS	Shell for ShellUser
`User Name>>`	`Shell> listener &`

If a program called from a Listener runs into trouble, you can kill both it and the Listener together if the Listener was called from a Shell, since any jobs called from a Shell can be managed by Shell commands. (See Chapter 8.) It is adviseable to start the Listener from a Shell.

6.3.2 Executing a Goal

A Listener plays a role similar to that of a Prolog interpreter. We can call any goal in any KL1 program by typing its name in the following form to a Listener prompt. Note that it must be followed by a period.

?- module-name:goal.

For example:

```
?- sample1:sieve([1,2,3],E,O).
 E = [2]
 O = [1,3]
 yes.
```

Values instantiated to any variables are stored in something called a "Variable Pool" (see Chapter 9) that corresponds to and is created at the same time as each individual Listener. In this case, the variables "E" and "O" retain the values that were set above. To show this fact, try the following unification.

```
?- X = [E,O].
 X = [[2],[1,3]]
 yes.
```

If you call another goal with a variable currently in the Variable Pool by mistake, the goal may fail. This is any easy mistake to make. For example, the following goal called with the variables "E" and "O" from the Variable Pool will certainly fail.

```
?-  sample1:sieve([5,2,3],E,0).
 Unification failure>> 5
                With > 1
                Pe   > 0
no.
```

It fails because the program tried to unify the variable "O" that had already been bound to `[1,3]` with `[5|Tail]`. You can free all bindings kept in the Variable Pool by typing a period at the Listener prompt.

```
?- .
```

If you do not need a Variable Pool at all, type the `forget` command at the Listener prompt. (The inverse command is `remember`)

```
?- forget.
```

6.4 Debugging Your Programs

Two tools, the Tracer and the Inspector are available to help debug your KL1 programs.

6.4.1 The Tracer

Unlike Prolog, KL1 goals are not executed in a sequential order. At any particular moment, it is not possible to determine which goal will be executed next. But by using the Listener's Tracer tool, it is possible to select specific goals to be traced, even while many goals are executing in parallel, thus allowing you to debug without worrying about the order of execution.

To use the Tracer, enter the following command.

```
?- trace.
yes.
{trace}
?-
```

This puts you into trace mode where a variety of information regarding the execution of a goal can be displayed. For example, execution of the predicate "go" in the module "sample3" can be traced like this:

```
?- sample3:go(3,E,0).
 @0@4096 0   sample3:go(3,A,B)              (a)
 1     * (1)   generate(3,C)                (b)
 2     * (2)   sample1:sieve(C,A,B)?        (c)
```

In line (a), there are three numbers: `@0@4096 0`. The first number "0" is the number of the processor on which the goal is executed. The second number "4096" is the priority given to the goal. The last number "0" is a goal number to identify the goal in this trace.

On lines (b) and (c), we see subgoals that are called from the goal "go". The numbers, "1" and "2", at the front are goal numbers. The mark, *, indicates that trace-mode is active for the goal. The parenthesized numbers, (1) and (2), are used in some trace commands to indicate the subgoal to which the trace command is applied. (A detailed explanation of all

the information displayed by the Tracer may be found in Section 10.6.2 of the "PIMOS 1.5 Operating Manual"[3].)

In the above example, both subgoals (1) and (2) are now in trace-mode. We can reverse the mode of any subgoals that do not need to be traced by typing the `t` command followed by their number, as in:

```
t N1,N2,...
```

For example, typing the following command at the prompt in the line (c),

```
2     * (2)   sample1:sieve(C,A,B)? t 1     (c)
@0@4096 0   sample3:go(3,A,B)
1       (1)   generate(3,C)                  (d)
2     * (2)   sample1:sieve(C,A,B)?          (e)
```

removes the * from subgoal (1) in line (d). This means that subgoal (1) is now in notrace-mode. Typing the same command again will return the goal to trace-mode. To continue to trace, hit the "return" key at the prompt in line (e).

```
@0@4096 2   sample1:sieve([3,2,1],A,B)
                B= [3|D]
3     * (1)     sieve([2,1],A,D)?
```

We can finish tracing a sequence of goals by typing the `x` command if all subgoals are in trace mode. This has the effect of reversing the modes and continuing execution.

```
3     * (1)     sieve([2,1],A,D)? x
 E = [2]
 O = [1,3]
yes.
?-
```

In this case, all execution has finished, but in other cases, some goals in trace-mode may remain to be executed, and the tracer will show what goal is to be executed next.

Display of trace information can be stopped with the "notrace" command, and a help menu can be displayed with the "help" command. For more information on the trace command, see Section 10.6.3 of the "PIMOS 1.5 Operating Manual"[3].

The Tracer has two facilities for spying on goals and tracing specific types of goals selected by the user.

- Reduction Spy
  Report the reduction of selected goals.

- Fork Spy
  Report the reduction of a parent goal with a selected goal as a child.

To select spy mode for a goal, type :

```
?- spy module-name:predicate-name/number-of-arguments.
```

Next, to select which spy mode to use, type one of the following :

```
?- spy_reduction. ( or just sr.)
?- spy_fork. ( or just sf.)
```

The next example illustrates tracing using the spy facilities.

```
?- spy sample1:sieve/3.
Spypoint set on sample1:sieve/3
 yes.

?- sr.
 yes.

{SpyReduction}
?- sample3:go(3,E,0).
<0> sample3:go(3,A,B) -->
@0@4096     + sample1:sieve([3,2,1],A,B)
                    B=[3|C]
1   + * (1)      sieve([2,1],A,C)?
```

Trace commands are also available in spy mode.

We can continue until the execution of the next spied goal with the following command.

- `sr` command : Trace until the next goal with Reduction Spy mode.
- `sf` command : Trace until the next goal with Fork Spy mode.

To reset a goal's spy mode, type :

```
?- nospy module-name:predicate-name/number-of-arguments.
```

To reset the spy mode of all goals, type :

```
?- nospy.
```

The "nospy" command resets the spy mode of a goal(s), but does not actually take you out of spy mode. To exit spy mode completely, i.e. to return to the Listener top level, use the "notrace" command.

```
?- notrace. (or just ntr.)
```

6.4.2 The Inspector

The Inspector is a tool for displaying any piece of KL1 data or registering it in the Variable Pool. The Inspector can be called from the top level of the Listener or from the prompt of the Tracer. A more detailed explanation of the Inspector appears in Chapter 11 of the "PIMOS 1.5 Operating Manual"[3].

- Invoking the Inspector from the Listener.
  You can see the data bound to any variable in the Variable Pool. To display the names of variables kept in the pool, simply type :

```
?- list.
```

Next, if you want to see the data bound to the variable "X", invoke the Inspector by typing :

```
?- inspect(X)
```

If "X" is bound to {1,2,3,a(b)}, then the following will appear.

```
{1,2,3,a(b)}>
```

The Inspector can be applied to any command. For example, in order to show the fourth element of this data item, we can use the `me` command. (Note that data elements are numbered starting at zero.)

```
{1,2,3,a(b)}> me 3
3 :  a(b)
```

- Invoking the Inspector from the Tracer.
  To invoke the Inspector from the Tracer, type,

  `ins` goal_number

  although the goal number may be omitted. For example:

```
?- sample3:go(3,E,0).
 @@@4096 0   sample3:go(3,A,B)
 1     * (1)   generate(3,C)
 2     * (2)   sample1:sieve(C,A,B)? ins 1
 generate(3,A)>
```

The commands that can be used from the Inspector are displayed by typing the `help` command on the Inspector prompt. To exit the Inspector, type the `exit` command.

6.5 Using the Re-Linker

After debugging a program, you must re-compile any modules you modified. Unmodified modules do not need to be recompiled. If modules that have not been recompiled call modules that have been recompiled, you must call the Re-Linker to link these modules again. The Re-Linker can be executed in the Shell as shown here.

```
Shell>relinker([sample1, sample2,...])
```

The elements of the argument list are names (as atoms) of modules.

The information for executing a KL1 program generated by compiling it is normally lost when you shutdown PIMOS. In order to not have to recompile every program you use every time you reboot PIMOS, you can save this information into a file using the Unloader utility. The Unloader can also be executed in a Shell as follows.

```
Shell> unload([sample1,sample2,...],"module.unl")
```

This will save the information needed to execute the modules sample1, sample2, ... in the file "module.unl". The name of the file must be string type data. The next time you boot PIMOS, you can load the information into PIMOS using the Loader, which can also be executed in the Shell. For example :

```
Shell>load("module.unl")
```

Now you can call any of the modules sample1, sample2, ... from the Shell or from the Listener.

6.6 Dealing with Deadlock

When a running program does not show any response, deadlock may have occurred between goals. You can check for deadlock in your program by trying either of the following.

- Check if the color of the Monitor Panel is black when "Silent Mode" in the CSP Console Panel is OFF. (See Section 7.2 of the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].)
- Suspend the execution of your program several times by typing `Control-C`. Use the "r" command to check whether or not the number of reductions are increasing. (See Chapter 9.)

6.6.1 Invoking the Garbage Collector

In the current version of PIMOS, deadlock can also be detected by the Garbage Collector. The Garbage Collector is automatically invoked when available memory space becomes short, but if you suspect that your program is deadlocked, you can invoke it yourself by either of the the following two methods.

- Type `Control-C` in the Listener, and use the "g" command.
- Use the "gc" command in Shell.

You can call the Garbage Collector from anywhere. If deadlock is detected, it will be reported in the window which the program was started from. If this was the Shell, you must hit the return key once in order to see the message.

6.7 Evaluating Program Performance

The following facilities can be used to evaluate the performance of your programs.

- Reporting resource consumption (the number of reductions) and total execution time of a goal called from the Listener.

  To get this information, give the following command to the Listener after executing a goal.

  ```
  ?- statistics.
  ```
  (or just st.)

  For example:

```
?- sample3:go(3,E,O).
 E = [2]
 O = [1,3]
 24 reductions
 118 msec
 yes.
```

To reset this display mode, type `nostatistics.` or just `nst.`

- Performance-Meter (Only on Multi-PSI.)
  The Performance-Meter is a tool for showing the average work load of each processor on a Multi PSI machine. To invoke it, type the following command in the Shell window.

  ```
  Shell> pmeter
  ```

  An explanation of the Performance-Meter can be seen in Appendix B of the "PIMOS 1.5 Operating Manual"[3].

Chapter 7

Dealing with Problems

This chapter explains briefly how to deal with PIMOS or SIMPOS problems that could be caused by KL1 programs. Greater detail on this subject can be seen in Chapter 12 of the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2].

7.1 Problems with PIMOS

- If PIMOS doesn't appear to be running correctly (because of bugs in your program or for other reasons) you can stop PIMOS execution by typing `Control-C` in the "CSP Command Window". You will then see a CSP prompt, `$`, in the window. If typing `Control-C` has no effect, click a mouse button on the "Monitor Panel" in the upper-left corner of your display and push the `ABORT` key. Next, select the menu item "Return to CSP Toplevel" in the window that pops up. You will then get a prompt in the "CSP Command Window." Input the `exit` command at this prompt.

  ```
  $ exit
  ```

  to terminate CSP. You can reboot PIMOS by selecting "(Pseudo) Multi-PSI" from the SYSTEM MENU.

- If a KL1 program consumes all the PIMOS memory or causes a PIMOS error, PIMOS will abort execution and a `$` prompt will appear in the "CSP Command Window". You should then shutdown PIMOS with the `exit` command and reboot from the SYSTEM MENU.

- When the cause of a problem is a shortage of memory, the following error message will be displayed in the "CSP Command Window":

  ```
  Copying GC Memory Shortage : xxx words
  ```

  Sometimes this type of error can be avoided in Pseudo Multi-PSI by changing the information that specifies the number of processors in your configuration file. See Chapters 4 and 6 in the "Multi-PSI/V2 Console System (CSP) Operating Manual"[2] for directions on how to do this.

7.2 Problems with SIMPOS

SIMPOS execution may halt because of problems with one of your programs. In this case, the CONSOLE Window (which first appears on the display when SIMPOS is booted) will appear with messages and a prompt something like the following :

```
%F-SRGHLT, CPU HALT BY STOP REGISTER

MPC = ....                PMPC = ....
CIR0 = ....
CIR1 = ....

  ..................

$
```

You can recover from this situation with the following operations.

1. Type the deb command to the prompt to enter the PSI-CSP debugging mode.

   ```
   $deb
   ```

2. Input go to the > prompt that appears on the CONSOLE.

   ```
   >go
   ```

3. Next, push the CTRL key and the BREAK key at the same time. The CONSOLE Window will disappear. Return to the CSP Command Window by clicking the mouse button on it if necessary.

4. If there is not a $ prompt in the CSP Command Window, type Control-C to make the prompt appear.

5. Input the exit command at the $ prompt to terminate CSP.

6. Reboot PIMOS by selecting "(Pseudo) Multi-PSI"from the SYSTEM MENU.

- If SIMPOS does not respond correctly after operation 3 above, push the BREAK key to return to the > prompt. Next, perform the following operations to shutdown SIMPOS and reboot it.

  ```
  >go/tr
  @q simpos
  @< cr >
  DC> OK to SHUT DOWN ??  [Y/N] -> [1/0]
  RC> 1
  .....
  >q
  $bo
  ```

- If operations 4 or 5 above fail and you cannot terminate CSP, select the item "process" from the SYSTEM MENU to start up the "Process Manipulator." (See the "Basic PSI/SIMPOS Operating Manual (1),(2)"[7].) In the "Process Manipulator" window, select (Pseudo_) multi_psi_main_program and execute the exterminate command. You may then shut down SIMPOS.

- If all the above operations fail, push the RESET button on the PSI-II and cross your fingers.

Chapter 8

The Shell

This chapter illustrates the essential functions of the Shell.

8.1 Setting Up an Initialization File

As explained in Chapter 2, the Shell wakes up with its default directory set to the user's home directory specified in the "User Environment File." In this directory, you can create a file `shell.com` with commands to do things like set environment variables or executes programs. The commands in this file are executed automatically when the shell starts up.

Following is a small sample `shell.com` file.

```
set history = 20
set rscinc = infinite
set prompt = "% "
listener(mouse,char(50,20),"font:test_11") &
```

8.2 Basic Shell Commands

The following examples show how to use some of the most common shell commands.

- Terminate the Shell
  `Shell> exit`
- Change the current directory
  `Shell> cd ">sys>csp>SYSTEM.DIR"`
- Execute a command file (which may contain any Shell command(s))
  `Shell> take "test.com"`
- Display a list of Shell commands
  `Shell> help`

8.3 The History Function

The history function is available after setting the length of the history stack.

```
Shell> set history=20
```

You should put this command in your `shell.com` file.
You can see contents of the history stack by typing:

```
Shell> history
```

If you want to use one of the commands that is displayed in the history stack, type its number at the prompt. Typing "0" executes the last command you entered. The following executes the third command on the history stack.

```
Shell> 3
```

8.4 Executing Tasks

You can enter any Shell command or call any goal of a KL1 program from the Shell prompt. But you cannot use any variables when you call a KL1 goal from Shell, so you cannot call goals that return values through variables.

8.4.1 Task Job Control and I/O

- A task can be executed either in the foreground or in the background of the Shell.
- Every task has a "Standard Input," "Standard Output," and "Standard Message Output," all of which are set by default to the Shell's window. You can redirect any of them to or from another file if necessary.
- You can combine two or more tasks by means of pipes |. Tasks connected by pipes are called a "job."

8.4.2 Examples of Tasks

- Executing a Listener in the foreground
  `Shell> listener`
- Executing a Listener in the background
  `Shell> listener &`
- Executing multiple tasks
  `Shell> listener & listener & compiler`
- Executing a user's KL1 program
  `Shell> sample6:demo`
- Executing a task with Standard Input redirected from a file
  `Shell> cat <= file("me:sample1.kl1")`
  This task displays the contents of a file. (The `cat` command recognizes a blank line as end of input.)
- Executing a task with Standard Output redirected to a file
  `Shell> cat => file(^"file.new")`
  Keyboard input is written to the file. (Hitting the `return` key will terminate this command.)
  The character ^ placed in front of the file name is a command to set the output mode to "write mode". If you omit the character, the default mode is "append mode."
- Executing a task with Standard Input redirected from a window and Standard Output redirected to a file
  `Shell> cat <= window("window name") => file(^"file.new")`

- Executing a task with Standard Message Output redirected to file
  `Shell> compile(["sample1","sample2","sample3"]) -> file(^"compile.log")`
  This task does a batch compile. (See Section 7.2.4 of the "PIMOS 1.5 Operating Manual"[3].)
- Combining two tasks with a pipe
  `Shell> grep(":-") <= file("sample1.kl1") | lc`
  This job will count the number of lines in the file that contains the string ":-".

8.5 Job Control

The following commands are available for controlling jobs in the Shell.

- The "Control-C" command moves a job running in foreground to the background.

```
Shell> compile
COMPILE> ^C
Shell>
```

- The "status" command displays information on all jobs.

```
Shell> status
       1 --> running compile
```

- The "stop JobNo" command stops the job numbered "JobNo".
  `Shell> stop 1`
- The "fore JobNo" command brings the job numbered "JobNo" into the foreground and resumes it if it was stopped.
  `Shell> fore 1`
- The "back JobNo" command puts the job numbered "JobNo" into background and resumes it if it was stopped.
  `Shell> back 1`
- The "kill JobNo" command terminates the job numbered "JobNo".
  `Shell> kill 1`
- The "kill all" command terminates all jobs.

8.6 Programs as Shell Utilities

You can turn any KL1 program into a Shell utility by giving it a module name to be used as a Shell command name, and giving it a single public predicate named go. (See Section 8.3 in the "PIMOS 1.5 Operating Manual"[3].) The "echo" utility can be implemented by the following KL1 program.

```
:- with_macro pimos.
:- module echo.
:- public go/1.

go(Arg):- true|
        shoen:raise(pimos_tag#shell,get_std_out,Out),
Out = [putt(Arg),nl,flush(_)].
```

This progran can be used just like any other Shell utility, in the following manner.

```
Shell> echo([a,b,c])
[a,b,c]
Shell> echo("a b c")
"a b c"
```

Chapter 9

The Listener

Chapter 6 introduced Listener functions to aid debugging, for detecting deadlocks, and to manipulate the Variable Pool. This chapter describes several more functions of the Listener.

9.1 Basic Listener Commands

The following are examples of how to use some of the most common Listener commands.

- Terminating the Listener
  ```
  ?- exit.
  ```

- Calling a Public Predicate

  ```
  ?- module_name :  goal.
  ```

- Calling a Local Predicate
  We can call local predicates that are not declared as public predicates in the following way :

  ```
  ?- module_name ::  goal.
  ```

  For example :

  ```
  ?- sample9::fork(100,32,0).
  ```

- Calling Built-in Predicates
  We can also call built-in predicates which are available in the body part of clauses. For example :

  ```
  ?- current_processor(PE,X,Y), PEs := X*Y.
  ```

- Using the Help Menu

  ```
  ?- help.
  ?- help all.
  ?- help basic.
  ```
  etc.

- Viewing statistics
  The "statistics" command (or just "st") turns ON the mode that displays the amount of resources consumed (number of reductions) and the elapsed execution time of a goal called from the Listener. (The inverse command is "nostatistics" or just "nst".)

- Setting the Default Module Name

  ```
  ?- default_module "module name".
  ```

  This allows you to omit the module name when calling predicates from this module. For example :

  ```
  ?- default_module sample9.
  ```

  Calling a public predicate :

  ```
  ?- distribute(10).
  ```

  Calling a local predicate :

  ```
  ?- :fork(10,8,0).
  ```

- Other Commands
  Explanations of other commands are given in Section 10.5 of the "PIMOS 1.5 Operating Manual"[3].

9.2 Suspending Execution

All programs called from the Listener can be suspended by hitting the attention key, `Control-C`. While the execution of a program is suspended, the following commands are available for displaying information about that program.

- *< cr >* : Resume execution
- `g` : Call the Garbage Collector and resume execution.
- `r` : Display the amount of resources consumed (number of reductions.)
- `s` : Display the the execution state of the program (task.)
- `@` : Display the state of the program (task) resources.
- `t` : Display the names of goals for which execution is being traced.
- `k` : Display the names of goals whose execution is suspended.
- `a` : Abort execution.
- `b` : Break (interrupt) execution. (`exit.` returns from a break.)
- `h` : Display the help menu.

9.3 Changing Standard I/O Devices

The default device for Standard I/O is the Listener window. You can change it to a file or new window in the following manner.

Set Standard Output to the file `demo.out`

```
?- sample6:demo => file(~demo.out).
```

Set Standard Input to a new window

```
?- sample6:demo <= window("any name").
```

Set Standard Message Output to the file demo.mes

```
?- sample6:demo -> file(~"demo.mes").
```

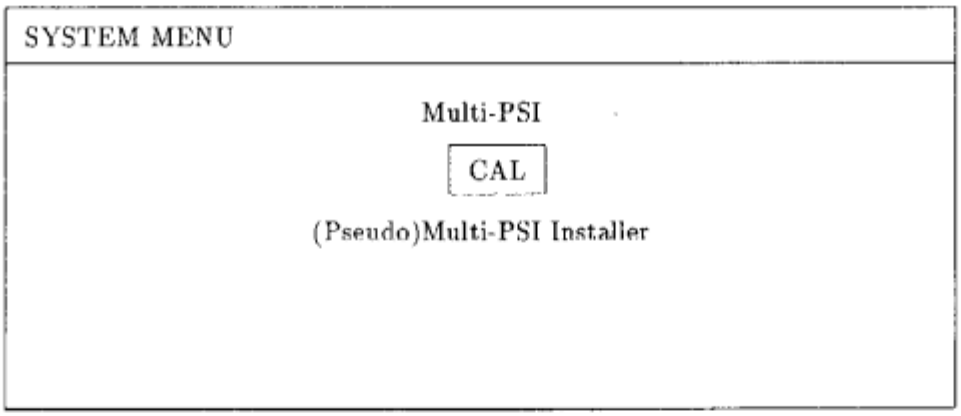
Chapter 10

CAL

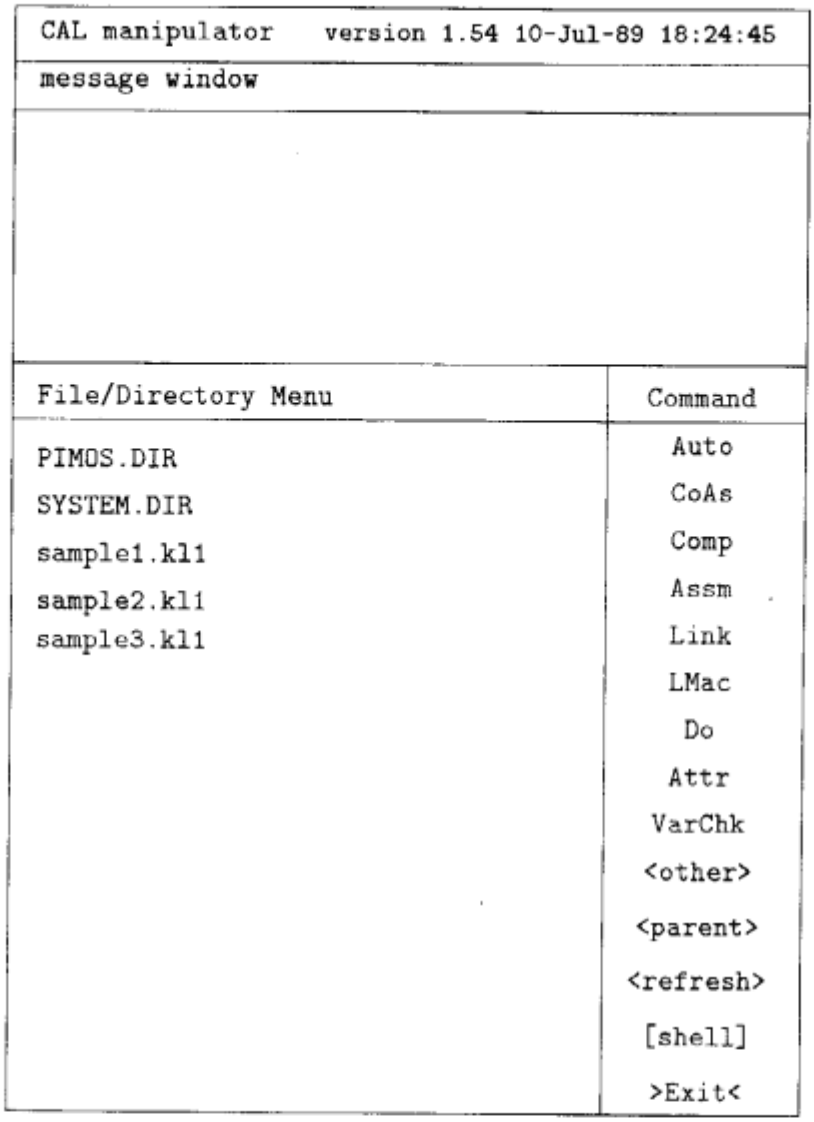
CAL is a system for compiling, assembling, and linking KL1 programs to make the binary objects (denoted by files with the `.kbn` suffix) that can be loaded into Multi-PSI CSP. All programs for PIMOS are implemented in KL1 and compiled with the CAL system. If you compile a KL1 program and link it to PIMOS using the CAL system, it can be loaded into CSP when you boot PIMOS.

10.1 Starting Up CAL

1. Select the item "CAL" from the SYSTEM MENU



2. After about a minute, the mouse arrow will change into the form for determining the position of a window. Click the mouse button to select a window and its size.



The three windows shown here have the following functions.

- Message Window
  This is the window used to display information from executing CAL commands and for inputing commands.
- File/Directory Menu Window
  This is the window for displaying KL1 source files (`*.kl1`), code files (`*.rkp`) created by compiling and assembling, and subdirectorics under the current directory. You use the mouse to select a source file(s) to be compiled and assembled by the "CoAs" command in the "Command" window.
- Command Window
  This window contains the command menu. Any command can be executed by clicking a mouse button on it.

10.2 How to Use CAL

Programs are usually compiled and assembled at the same time, producing a code file (`*.rkp`) for each file. Next, they are linked to the PIMOS object file **`pimos:pimos.kbn`** using a CAL function for partial linking.

1. If you want to change the directory, select the item **"<other>"** in the **File/Directory Menu**, and type in the name of the directory in the **message window**. For example:

```
other>me:
```

2. Select the names of all files to be compiled and assembled by clicking on them. You can move the **File/Directory** menu with the same mouse operations as the "File Manipulator."

3. Select the item **CoAs** in the **Command** window to perform the compilation and assembly.

4. Status information is displayed in the **Message Window**. The following illustrates what might appear during a successful execution of the "CoAs" command.

```
**psi***::>sys>user>***>sample1.kl1.1
*Compile Start ....
furui/3,END
*Assemble Start ....
End.
*Output File Name
**psi***::>sys>user>***>sample1.rkp

*CoAs end !!
```

5. Next, select **Link** from the **Command** window.

6. Type the following command in the **Message Window** to link the compiled program to the PIMOS object code.

```
>> plink pimos:pimos.kbn

ok.
P>>
```

7. Load all the (`*.rkp`) code files produced above one by one.

```
P>> load  sample1.rkp
**psi***::>sys>user>***>sample1.rkp.1
 StartPC:XXXXXXXX
  EndPC:YYYYYYYY

 ok.
P>> load  sample2.rkp
.....
```

You can also use the "`load *`" command to load all `*.rkp` code files in the current directory.

8. Save the new configuration of your programs using the "psave" command.

```
P>> psave
Saving ....
**>pimos.kbn
**>pimos.ldb
**>pimos.sym

 ok.
>>
```

9. Exit from the linker.

```
>> exit
```

10. If you compile your program again after debugging it, select the following command in the `Command` window to refresh the "`File/Directory Menu.`"

```
<refresh>
```

You can terminate CAL with the `>Exit<` command from the `Command` window.
Note: Each time you link your program to the PIMOS object code, the object code is patched. Linking your program to PIMOS too many times may cause PIMOS to fail to load into CSP because the patch has become too large. Loading may also fail when your program has a bug. You can deal with this kind of problem by referring to the last section in Chapter 3 of this manual.

10.3 Choosing Between the Native-Compiler and CAL on PIMOS

- When you are developing and debugging KL1 programs, you should always use the Native-Compiler on PIMOS.
- If a program is bug-free and you will use it frequently, you may want to link it into PIMOS by using CAL.
- In the current version of PIMOS, programs compiled by CAL run faster than those compiled with the Native-Compiler.

10.4 The Variable Checker

CAL has a feature that will report in the "message window," any variable that is only used once in a KL1 clause. This can be very useful since a variable referenced only once can easily be the cause of deadlock.

In order to use the Variable Checker, after selecting a file in the "File/Directory" menu, select "VarChk" from the "Command" menu.

For more details about CAL, see "Using the Multi-PSI/V2 Cross-System (V2/CAL)"[4].

Chapter 11

Other Utilities

This chapter explains how to use two other PIMOS resources, Pools and the Timer device.

11.1 Pools

A Pool is a database for storing and retrieving KL1 data. It is controlled just like an I/O device, by first creating a stream representing the Pool, and then sending messages to that stream.

There are two types of Pools, those that use keys to index data, and those that do not. The following types of Pools are available in PIMOS.

- Pools without a key : bag, stack, queue, sorted_bag
- Pools with a key : keyed_bag, keyed_set, keyed_sorted_bag, keyed_sorted_set

The control messages available to each type are described in Section 3.3 of the "PIMOS 1.5 Operating Manual"[3].

The following example illustrates the basic functions of the frequently used bag, keyd_bag, and keyed_set types of Pools. More details may be found in Section 3.3 of the "PIMOS 1.5 Operating Manual"[3].

- Bag : This is the most basic kind of pool, and can only store and retrieve data. A bag is created by :

  ```
  pool:bag(S)
  ```

  Data is stored and retrieved from this bag by sending messages to the stream variable S as in :

  ```
  ?- pool:bag(S),
     S=[empty(A), put("str"), put(atom), put({vect}),
        get(B), get_all(C), get_if_any(D)].
  A = yes
  B = {vect}
  C = [atom,"str"]
  D = {}
  S = [ ... ]
  yes.
  ```

  This examples first asks whether the Pool is empty or not and gets the answer **yes** in the variable A. Next, it stores three data pieces of data "str" (a string), atom (an atom), and

{vect} (a vector) into the Pool. Next, the message get(B) is sent in order to extract one piece of data. The item {vect} is returned in the variable B. The next message get_all(C) retrieves all data still in the Pool. The last message get_if_any(D) can be sent at any time, even if the Pool is empty. This message retrieves data in the same way as messages like get( ), but if the pool is empty, it returns { } whereas get ( ) and other goals will simply fail.

- Keyed Bag : This is a pool that indexes data via a key, using a hash table to store the keys.

```
pool:keyed_bag(S)
```

For example :

```
?- pool:keyed_bag(S),
   S=[empty(A), put("atom",atom), put(str,"str1"),
      put(str,"str2"), put({vect},v(1)),
      empty(str,B), get(str,C), get_if_any("atom",D),
      get_and_put({vect},E,v(100)), get_all(F) ]
A = yes
B = no
C = "str2"
D = {atom}
E = {v(1)}
F = {{{vect},v(100)},str("str1")}
S = [ ... ]
yes.
```

Again the first message asks whether the Pool is empty or not and gets the answer yes. But this time, the first three pieces of data are each associated and stored with a key. The string "atom" is the key for atom, str is the key for both "str1" and "str2", and v(1) is the key for {vect}. Next the Pool is queried again as to whether or not it is empty, with respect to items with the key str, to which the answer is no. Next, data with the key "atom" is retrieved via the get_if_any message without asking whether or not the Pool is empty. The get_and_put message retrieves a piece of data with the key {vect} replacing it with v(100) and returning it and the key to the Pool. The last message get_all retrieves all data remaining in the Pool.

- Keyed Set : A Pool that can store only one piece of data with each key.

```
pool:keyed_set(S)
```

For example :

```
?- pool:keyed_set(S),
   S=[put(key,"str",A), put(key,atom,B), get(key,C)].
A = {}
B = {"str"}
C = atom
S = [ ... ]
yes.
```

In a keyed_set, the message put stores an item with a key and returns the old data that used to be associated with the key onto the optional third argument. In this case, the first message put returns {} to the variable A because there was no data with the key key. The second message returns the old data "str" in the variable B.

11.1.1 Using Pools

The following program puts each line of a text file into a "Keyed Bag" and allows the user to retrieve any line accoring to its line number. Before compiling this program, you must have finished compiling the module file from Chapter 4.

```
:- module sample10.
:- public go/1.
:- with_macro pimos.

go(Name):- true|
        file:open(Name,r,[getl(L)|F],success),
        std_inter(IO)@priority($,-100),
        IO=[putt('Number of Lines = '),putt(NL),nl,
            prompt("> "),gett(T)|IO1],
        pool:keyed_bag(Pool),
        read_file(L,F,P1,1,NL),
        display(T,IO1,P2),
        merge({P1,P2},Pool).

std_inter(IO):- true|
        shoen:raise(pimos_tag#shell,get_std_inter,IO).

read_file(-1,F,P,N,NL):- true|
        F=[],P=[],NL:=N-1.
otherwise.
read_file(L,F,P,N,NL):- true|
        P=[put(N,L)|P1],
        F=[getl(L1)|F1],
        N1 := N+1,
        read_file(L1,F1,P1,N1,NL).

display(exit,IO,P):- true|
        IO=[],P=[].
display(N,IO,P):- integer(N)|
        P=[get_if_any(N,L)|P0],
        ( L={St} ->
          P0=[put(N,St)|P1],
          IO=[putt(N),putt(' : '),putl(St),
              prompt("> "),gett(T)|IO1]
        ; otherwise
        ; true -> IO=[prompt("> "),gett(T)|IO1],
                  P1=P0 ),
        display(T,IO1,P1).
otherwise.
display(_,IO,P):- true|
```

```
IO=[prompt("> "),gett(T)|IO1],
display(T,IO1,P).
```

The program can be used like this.

```
?- sample10:go("sample10.kl1").
Number of Lines = 42
> 5.
5 : go(Name):- true|
> 7.
7 : std_inter(IO)@priority($,-100),
> 1.
1 : :- module sample10.
> exit.
 yes.
```

11.2 The Timer

The Timer is another PIMOS resource just like files, windows, etc. Creating the following module will illustrate how it can be used.

```
:- module timer.
:- public create/1.

create(Timer):- true|
        shoen:raise(pimos_tag#task,
                general_request,General_Request_Device),
        General_Request_Device=
                [timer(normal(Timer_Request_Device,_,_))],
        Timer_Request_Device=
                [create(normal(Timer,_,_))].
```

Calling this module as,

```
?- timer:create(Timer), Timer = [...].
```

will return a stream to the Timer in the variable `Timer`. You can then use Timer functions by sending the following messages to this stream.

- `get_count(^Result)`
  This returns the elapsed time since 0:00 AM (midnight) in milliseconds. The variable `Result` is instantiated to `normal(Count)` where the variable `Count` is the actual time.

- `on_at(Count,^Result)`
  When given a time in milliseconds since 0:00 AM (midnight) in the variable `Count`, the variable `Result` will be instantiated to the value `normal(Now)`, and the variable `Now` will be instantiated to the atom `wake_up` at the actual time specified by `Count`.

- `on_after(Count,^Result)`
  When given an amount of time in milliseconds in the variable **Count**, the variable **Result** will be instantiated to the value **normal(Now)**, and the variable **Now** will be instantiated to the atom **wake_up** after the amount of time specified by **Count** elapses.

11.2.1 Using the Timer

The following alarm program shows how to use the Timer. In this program, sending the message **on_at(Count,^Result)**, returns the value **normal(Now)** to the variable **Result**. The variable **Now** will be instantiated to **wake_up** when the specified time arrives. The alarm process detects the instantiation and sends a message to Message-Output.

```
:- module alarm.
:- public go/3.

go(H,M,Mes):-
        integer(H),integer(M), string(Mes,_,_)|
        Time := ( H*3600 + M*60 )*1000,
        timer:create( [on_at(Time,St)] ),
        alarm( St, Mes ).
alarm( normal(wake_up), Mes ):- true|
        shoen:raise( pimos_tag#shell, get_std_mes, Stream),
        Stream=[ putl(Mes) ].
```

Before compiling this program, you must compile the module **timer** above. This program can be used as a Shell utility, (see Chapter 8) and is executed as follows.

```
Shell> alarm(17,30,"### Time is up. ###")&
Shell> ......

Shell> (Shell Command or <cr> )
### Time is up. ###
```

In this example, the message **### Time is up. ###** will be appear following the first carriage return pressed after 17:30.

Reference

[1] (Pseudo) Multi-PSI System Administration Manual

[2] Multi-PSI/V2 Console System (CSP) Operating Manual

[3] PIMOS 1.5 Operating Manual

[4] Using the Multi-PSI/V2 Cross-System (V2/CAL)

[5] KL1 Programming Manual: Introduction, Beginning Level, and Intermediate Level

[6] Basic System Administration Manual for PSI/SIMPOS Systems

[7] Basic PSI/SIMPOS Operating Manual (1),(2)