CL-PVM: A Common Lisp Interface to PVM

Liwei Li* and Paul S. Wang†
Department of Mathematics and Computer Sciences
Kent State University

September 15, 1995

Abstract

CL-PVM is a set of Common Lisp functions that interfaces Common Lisp (KCL, AKCL, or GCL) to the C-based library of PVM. Generally, there is one CL interface function to each PVM C library function. The CL function calls its C-based counter part and relays data to and from the C function. This interface is complete and allows Lisp-based programs to take part in a PVM arrangement and thus facilitates the combination of symbolic, numeric, graphics, and other useful systems in a distributed fashion. The design, implementation, and usage of the CL-to-PVM interface are explained and examples are given.

1 Introduction

Parallel Virtual Machine (PVM) is a software package that integrates a heterogeneous network of computers and workstations to form a single parallel/concurrent computing facility[1]. PVM consists of two parts: a run-time server and a set of library functions. The PVM server manages the virtual parallel machine made up by a select set of host machines. The configuration is recorded in a *hostfile* and can be altered dynamically. The server also

*lil@mcs.kent.edu pwang@mcs.kent.edu

†Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650

enables user control of the virtual machine from any component host and allows initiation of tasks anywhere on the virtual machine.

A PVM task uses the PVM library functions to interact with other tasks: sending and receiving messages, initiating subtasks, detecting errors, etc. The PVM version 3.0 library is written in C allowing direct calls from C programs. There is also a Fortran 77 interface to give F77 programs access to the PVM library.

CL-PVM provides a Common Lisp interface to enable Lisp-based programs to partake in PVM applications. A wide variety of useful Lisp programs exists including symbolic computation systems, expert systems, artificial intelligence systems, knowledge-based systems, and many more.

With CL-PVM, a Lisp user can invoke the library routines directly from the Lisp toplevel or from Lisp programs. Figure 1 shows a sample PVM with three components:

- `Monkey`: a Sun4 workstation
- `Condor`: an HP 9000/730
- `Nimitz`: a Sun 4/670

Each PVM host has a *PVM daemon* process to support the control and operation of the virtual machine. Distributed tasks on the PVM utilize PVM protocols to interact with one another. The C-based PVM library supports the protocols. Lisp or Fortran processes access these C routines through appropriate interface functions.

This document describes the design, implementation, and usage of the CL[1] interface to PVM. The appendix contains a *User's Guide*.

2 Implementation Strategy

2.1 Accessing C Codes from CL

When a CL program is compiled, an intermediate C file is generated. This C file is then compiled by the C compiler to produce the object code[5]. Because

[1]In this document, the wording Common Lisp (CL) refers to UNIX-based KCL, AKCL, or GCL.

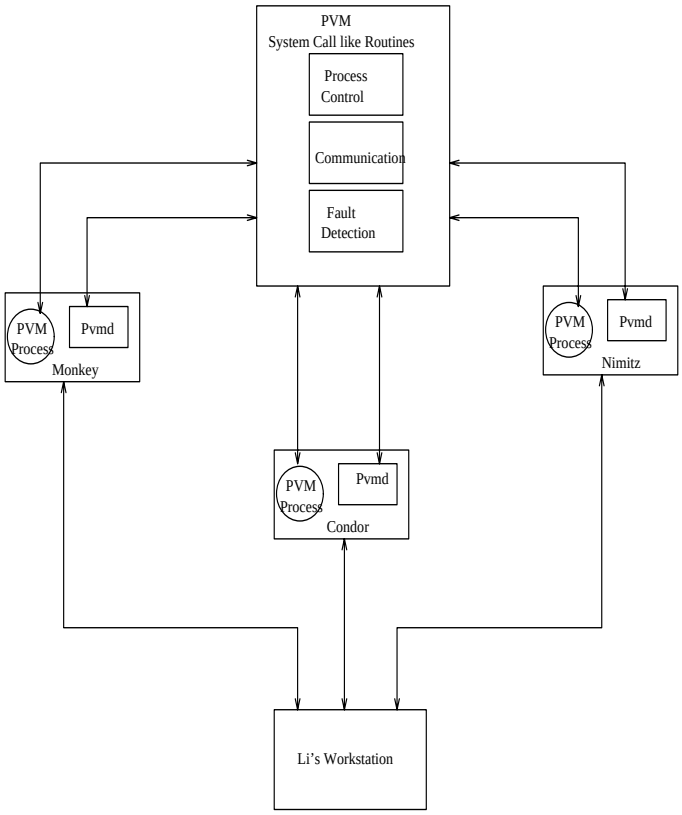

Figure 1: PVM with three machines: `Monkey`, `Condor` and `Nimitz`

of this arrangement, CL allows embedded C codes inside Lisp functions or files as *literal* lines. These lines are given as Lisp string constants and become part of the intermediate C file when the Lisp file is compiled. CL provides a set of mechanisms including `CLINES`, `DEFCFUN`, and `DEFENTRY` for interfacing to C functions [5]:

- CLINES
  Syntax: `(clines string*)`
  KCL specific: The KCL compiler embeds `string`s into the intermediate C language code. The interpreter ignores this form.

- DEFCFUN
  Syntax: `(defcfun header n element*)`

KCL specific: Defines a C-language function which calls Lisp functions and/or handles Lisp objects. `header` gives the header of the C function as a string. N is the number of the main stack entries used by the C function, primarily for protecting Lisp objects from being garbage-collected. Each `element` may give a C code fragment as a string, or it may be a list:

```
((symbol arg*) place*)
```

which, when executed, calls the Lisp function named by `symbol` with the specified arguments and saves the value(s) to the specified places. The DEFCFUN form has the above meanings only after compiled; The KCL interpreter simply ignores this form.

- DEFENTRY
  Syntax: `(DEFENTRY name arg-types c-function)`
  The compiler defines a Lisp function whose body consists of a calling sequence to the C language function specified by `c-function`. The interpreter ignores this form. The `arg-types` specifies the C types of the arguments which `c-function` requires. The list of allowed types is (`object char int float double string`). Code will be produced to coerce from a Lisp object to the appropriate before passing the argument to the `c-function`. The `c-function` should be of the form `(c-result-type c-fname)` where the `c-result-type` is a member of (`void object char int float double string`). The `c-fname` may be a symbol (in which case it will be made all lower case) or a string. If `c-function` is not a list, then `(object c-function)` is assumed.

With these utilities, functions with mixed C and Lisp codes can be written to interface Lisp and C.

2.2 Interfacing `pvm_spawn`

To illustrate the interface method, the interface to the PVM library function `pvm_spawn` is presented as an example. The `%` is a Lisp character macro that quotes a line as a string. The `;` is used for comment lines.

```
;; Interface to the PVM routine:
;;   int ntask = pvm_spawn( char *task, char **argv, int flag,
;;                          char *where, int ntask, int *tids )
;;   starts new PVM processes
;;
;;   tids is a return value -- the list of task ids spawn
;;
(defCfun "int call_spawn(tk, av, flag, we, ntask, tid)
   object tk, av, we, tid; int flag, ntask;" 1
;; tk, av and we are Lisp stings
;; fg and nk are Lisp fixnums
;; return value of call_spawn is a list of task ids
%  char *task, *args[20], *where, *tmp, *avs;
%  int i, *tids;
%  char **argv = NULL;
%  object s;
;;
;;   Interfacing CL string.
;;
%    task=(char *)calloc(((tk->st).st_dim)+1, sizeof(char));
%    strncpy(task, (tk->st).st_self, (tk->st).st_dim);
%    task[(tk->st).st_dim]='\0';
;;
;;   Interfacing CL string.
;;
%    where=(char *)calloc(((we->st).st_dim)+1, sizeof(char));
%    strncpy(where, we->st.st_self, we->st.st_dim);
%    where[we->st.st_dim]='\0';
;;
;;   Interfacing CL string that contains multiple arguments
;;   separated by blank or tab.
;;
%    avs   =   (av->st).st_self;
%    i     =   (av->st).st_dim;
%    if ( i > 0 )
%    {   argv = args;
%        tmp = (char *)calloc(i+1, sizeof(char));
```

```
%        strncpy(tmp, avs,i);
%        tmp[i]='\0';
%    /* fill in argv */
%        argv[0] = strtok(tmp, " \t");
%        for ( i=1; i < 20 && argv[i-1] != NULL; i++ )
%            argv[i] = strtok(NULL, " \t");
%    /* Refinement: check for too many args */
%    }
;;
;;   Allocates C-integer array for the task id information.
;;
%    tids = (int *)calloc(ntask, sizeof(int));
;;
;;   Call the PVM-C library routine.
;;
%    ntask=pvm_spawn(task, argv, flag, where, ntask, tids);
;;
;;   Allocates Lisp array for the tids and fill it with the
;;   content of C-array tids.
;;
     (make-array (int ntask)) "vs[0]")
%    for ( i=ntask-1; i >= 0; i-- )
%    {
        (fillnum "vs[0]" (int tids[i]) (int i))
%    }
;;
;;   Avoid tid to be accidentally garbage collected.
;;
     (set tid "vs[0]")
%    Creturn(ntask);
)

(defentry pvm-spawn(object object int object int object) (int call_spawn))
```

As declared by the `DEFENTRY`, the CL-PVM function `pvm-spawn` receives six arguments in a Lisp function call and passes them to the interface function

`call_spawn`. The CL-PVM documentation specifies the data types of each of the arguments as:

- `tk` — a Lisp string
- `av` — a Lisp string containing multiple parts separated by SPACE or TAB
- `flag` — a Lisp integer
- `we` — a Lisp string
- `ntask` — a Lisp integer
- `tid` — a Lisp *object* that is used to receive a Lisp array of integer processes ids returned by `call_spawn`

The major tasks of an interface function like `call_spwan` are to convert incoming arguments from Lisp types to C types and to convert return values from C types to Lisp types. The four data types `integer`, `character`, `float`, and `double` are converted automatically between the C and CL. Little effort is required to treat these four data types. On the other hand, the general Lisp `object` type requires careful treatment. The `object` type can represent any Lisp *s-expression* and encapsulates many different types. At the C level, a CL `object` is a `union`:

```
union lispunion
{
  struct fixnum_struct FIX;
  struct shortfloat_struct SF;
  struct longfloat_struct LF;
  struct character ch;
  struct symbol   s;
  struct cons     c;
  struct array    a;
  struct vector   v;
  struct string   st;
  struct ustring  ust;
  struct bitvector bv;
```

```
  struct structure str;
  struct cfun     cf;
  struct cclosure cc;
  struct dummy    d;
  struct fixarray fixa;
  struct sfarray  sfa;
  struct lfarray  lfa;
};
```

This C union contains the complete CL data types where each has its unique structure. Data types in this union are enumerated so that the correct computation and manipulation can be performed upon the appropriate instance of the union object. For instance, in the CL `pvm-spawn` the first object is a CL simple string that specifies the name of the task to be spawned. The second object is another CL string that supplies the required arguments for the task. The arguments are delimited by space or tab within the CL string. The next CL object indicates some particular host machine on which the task has to be spawned. It is a CL simple string. The last object is a CL object which is bound to a Lisp array filled when the call to pvm-spawn returns. Without loss of generality, we assume that Lisp programmers call `pvm-spawn` in the following format:

```
(pvm-spawn "Slave" "1 2 3 4" 1 "monkey.mcs.kent.edu" 2 'status)
```

We must split the Lisp string `"1 2 3 4"` into four parts. Assuming a blank or tab is the delimiter, we create a temporary C type array of strings `tmp` to hold the four parameter values to the task. In order to obtain the substring values of the Lisp object `task,argv,where` a reference is made to the Lisp structures in C (see Appendix A):

`/usr/local/include/cmpinclude.h`.

Since`object` is a union, address references are necessary in order to reach the actual beginning of the substring. The temporary integer array `status` is utilized to hold the returning status values of each adding action. After the call returns, the content of `status` is copied into Lisp `array`. To prevent a Lisp object from being unexpectedly garbage collected, we save the object on stack `vs[0]` which is recognized by the garbage collector[5]. The second parameter `1` at the end of the `DEFCFUN call_spawn` is used to reserve `1`

such place for each call to the C function. Figure 3 illustrates the general structure of a `DEFCFUN` function. Such CL functions can be compiled by the Lisp compiler to form part of a Lisp system or be loaded dynamically into an existing Lisp system. Figure 2 illustrates this approach. The PVM C library consists of two parts: `libpvm3.a` includes the regular routines and `libgpvm3.a` includes the PVM group communication and management functions.

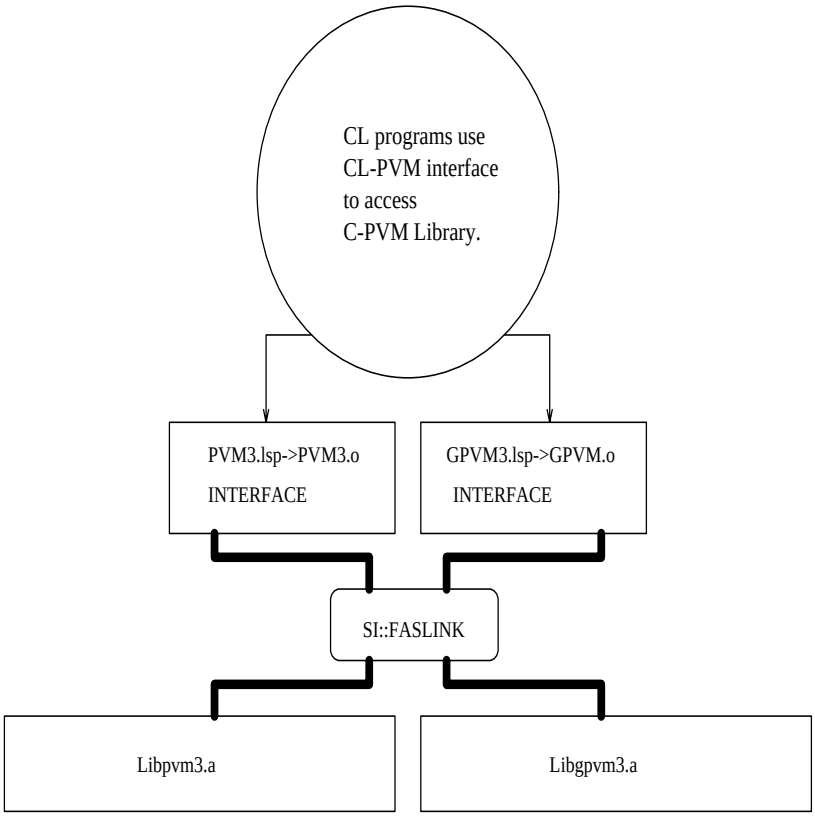

Figure 2: Lisp programs access PVM C-libs through interface library

`DEFENTRY` defines an entry for a C function in the Lisp environment. Implicit data conversion is assumed. C-type allowed in Lisp interface are `int char`, `float`, `double`, `object`, `void`. It is sufficient to use `DEFENTRY` to handle the primary data types such as int,float,etc. More complex data types require extra handling before and after the entry point. The CL `pvm-spawn`

utilizes another CL macro that allows the co-existences of C and CL-like code. The `DEFCFUN` facility allows us to write a mixed C-Lisp function and pass data back and forth between the two environments.

2.3 Interfacing `pvm_packf`

The general form of `DEFCFUN` is

```
(DEFCFUN "type cfn_name(args) arg-type-list;" n
     statement1
     statement2
     ...
)
```

The statements can be either in C (in double quotes) or in the special form of Lisp as explained in Section 2.1. A complicated CL-PVM function requires the usage of the other CL-PVM routines. We now show how to use `DEFCFUN` and other CL-PVM functions to interface the PVM library routine `pvm-packf`[1]. Before the implementation our primary focus is on the design of the CL-PVM interface. For instance PVM-C `pvm_packf` receives a action-description string and then parse the variable length argument list to invoke the proper data packing functions. After examine fortran interface carefully, we decide to design our CL interface according to the following guidelines:

- One-to-One interface to PVM-C function if feasible in CL environment.
- Maintain the PVM-C function prototype as close as possible.

The CL-PVM `pvm-packf` is an example where minimum alteration is necessary when PVM-C `pvm_packf` is interfaced. The `packf` function has been re-coded in Lisp according to its C implementation. In C program, programmers invokes the function in the following fashion:

```
double darry[4];
int count, *iarry;
...
info = pvm_initsend( PvmDataDefault );
pvm_packf("%+ %d %*d %4lf", PvmDataRaw, count, count, iarry, darry);
info = pvm_send(tid, msgtag );
```

The `packf` function sends data in PvmDataRaw mode. It packs the integer `count`. When the '*' is parsed it reads in the second `count` as the number item to be packed for the following integer array. Finally, it packs four doubles into the buffer. The CL-PVM `packf` function behaves in the same way. In CL a list object is received as the argument for `pvm-packf` . The first element of the list is treated as the description string and parsed for actions to be taken for the rest of the list. A typical usage is given below:

```
(pvm-initsend 0)
(pvm-packf
 '("%+ %d %*d %4lf" PvmDataRaw 4 4 '(1 2 3 4) '(2.3 4.5 6.7 8.9)))
(pvm-send tid msgtag)
```

The CL-PVM `pvm-packf` implementation is given below. Some portions of the code are omitted to save space.

```
(Clines
%#include <string.h>
%#include <ctype.h>
)
;;
;; int info = pvm_packf( const char *fmt, ... )
;;
(defCfun "int call_packf(lst)
    object lst; " 0
;;
;;  Variable declaration section omitted.
;;
;;  Check for the length of the input list
;;
    ((list-length lst) lstlen)
%   parac=fix(lstlen);
;;
;;  Interface the format string if parameter
;;  list is not empty.
;;
%   if (parac > 0)
%   {
```

```
      ((car lst)  first)
      ((cdr lst) rest)
%     tmp=(first->st).st_self;
%     sz=(first->st).st_dim;
%     if (sz > 0)
%     { fmt=(char *)calloc(sz+1, sizeof(char));
%       strncpy(fmt, tmp, sz);
%       fmt[sz]='\0';
%       argpt=fmt;
%     }
%   }
;;
;;  Retrieve the initsend parameter and call
;;  pvm-initsend and pack the format string
;;  into the buffer.
;;
%   if (argpt[0]=='%' && argpt[1]=='+')
%   {
      ((car rest) car_arg)
      ((cdr rest) rest)
%     inits=fix(car_arg);
      (pvm-initsend (int inits))
%     argpt +=2;
%   }
    (pvm-pkstr first)
;;
;;  Parse the format string to retrieve the
;;  correspondent data type to pack into
;;  the buffer.
;;
%   while (*argpt != '\0')
%   {
;;
;;   If '%' is seen parse the following
;;   information;
;;
%    if (*argpt++=='%')
```

```
%    { intcnt=1;
%      intstd=1;
%      isv=1;
;;
;;     Extract the proper information off
;;     the head of the list if '*' is
;;     encountered.
;;     Extract the number if digit is
;;     encountered.
;;
%      if (*argpt=='*')
%      {
         ((car rest) car_arg)
         ((cdr rest) rest)
%        intcnt=fix(car_arg);
%        isv=0;
%        argpt++;
%      }
%      else if (isdigit(*argpt))
%      {   intcnt=atoi(argpt);
%          isv=0;
%          while(isdigit(*++argpt));
%      }
;;
;;     Extract the proper information off
;;     the head of the list if '.' is
;;     encountered.
;;     Extract the number if digit is
;;     encountered.
%      if (*argpt=='.')
%      { isv=0;
%        if (*++argpt='*')
%        {
           ((car rest) car_arg)
           ((cdr rest) rest)
%          intstd=fix(car_arg);
%          argpt++;
```

```
%         }
%         else if (isdigit(*argpt))
%         {
%             intstd=atoi(argpt);
%             while(isdigit(*++argpt));
%         }
%       }
;;
;;      initialize the proper variable.
;;
%       for (cc =1; cc;)
%       { switch (*argpt++)
%          { case 'h':
%                vh = 1; break;
%            case 'l':
%                vl=1;   break;
%            case 'u':
%                vu = 1; break;
%            default:
%                argpt--;
%                cc=0;
%          }
%       }
        ((car rest) lar_arg)
        ((lst2arr lar_arg) car_arg)
        ((cdr rest) rest)
;;      Extract proper information
;;      according to the data type
;;      flag.
;;
%       switch(*argpt++)
%       { case 'c':
           (pvm-pkbyte car_arg (int intcnt) (int intstd))
%             break;
%         case 'd':
%          if (vl)
            (pvm-pklong car_arg (int intcnt) (int intstd))
```

```
%          else if (vh)
             (pvm-pkshort car_arg (int intcnt) (int intstd))
%          else
             (pvm-pkint car_arg (int intcnt) (int intstd))
%          break;
%         case 'f':
%          if (vl)
             (pvm-pkdouble car_arg (int intcnt) (int intstd))
%          else
             (pvm-pkfloat car_arg (int intcnt) (int intstd))
%          break;
%         case 'x':
%          if (vl)
             (pvm-pkdcplx car_arg (int intcnt) (int intstd))
%          else
             (pvm-pkcplx car_arg (int intcnt) (int intstd))
%             break;
%         case 's':
           (pvm-pkstr car_arg)
%       }
%     }
%   }
)
```

Data type conversion is the key issue. From a mathematician's point of view, we must define a set of functions which maps from the Lisp set of data types to C set of data types[3] and moreover, whose inverse function maps from C set of data types to Lisp set of data types. Figure 3 also illustrates this point. Only a portion of C data types are used in PVM C routines, thus the conversion function are determined based on the involving C data types. Figure 4 shows the formation of the conversion functions. We are not using the word *function* according to its strict mathematical definition. When an arrow is pointed to a dotted box instead of a specific data type, it implies that such data type maps to all the data types within the box. Appendix B provides a list of the PVM C function headers and Appendix C contains the Lisp interface headers.

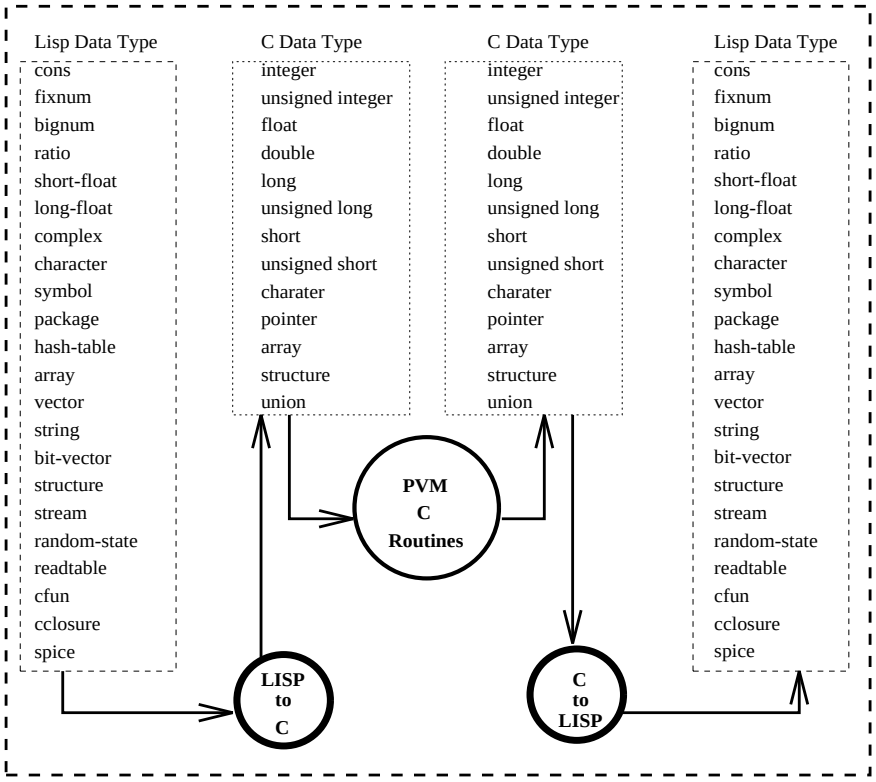

Figure 3: Data type conversion contains two parts: Lisp set of data type maps to C set of data type and C set of data type maps to Lisp set of data type.

3 Testing the CL-PVM

The Lisp interface routines to PVM can be invoked interactively from the Lisp toplevel. This feature also provides an easy way to test the CL-PVM.

Whenever a user invokes Lisp from the system prompt, at the end of the Lisp starting up program it checks and see if the file `init.lsp` exists at the current directory. If so, it loads it into the current Lisp environment. This provides us a way of customization. We can integrate the MAXIMA commands with initial Lisp environment and make it function as a symbolic computational utility. One can also initialize it with PVM C routines and their Lisp interface counterparts to attach the PVM message passing capability to it. A sample `init.lsp` is given below:

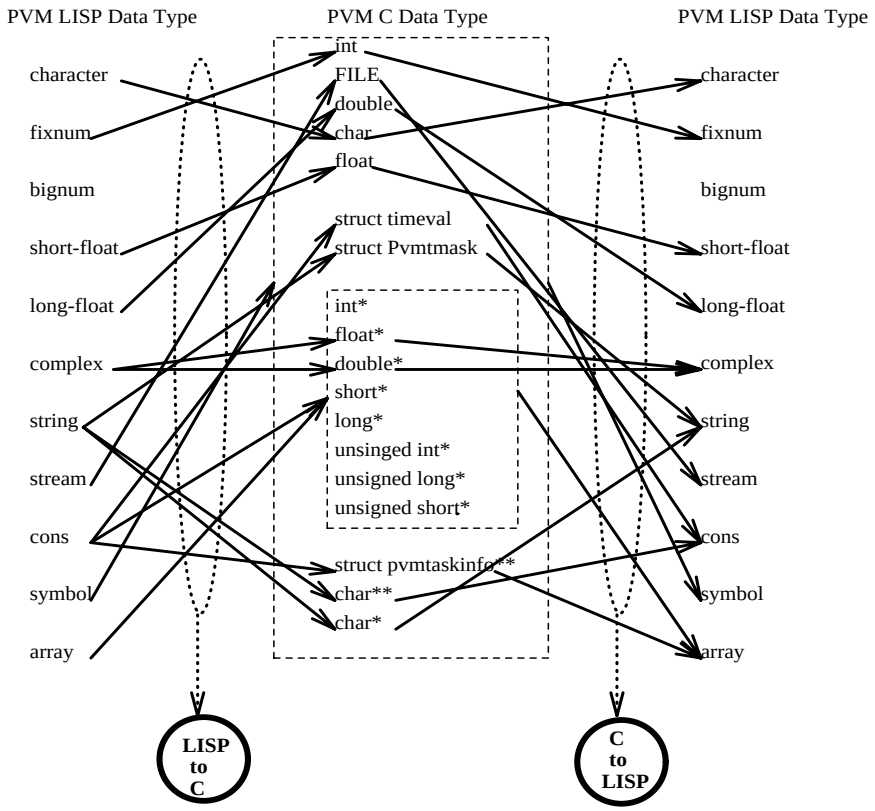

Figure 4: The definition of "Lisp to C" and "C to Lisp" mapping function

```
;;;;;;;;;;;;;;;; init.lsp  ;;;;;;;;;;;;;;;;;;;;
(si::faslink "clpvm.o" "/vol/pvm/lib/SUN4/libpvm3.a -lc")
```

`Si::faslink` loads both the `libpvm3.a` library routines and their Lisp interfaces into the Lisp environment. The `clpvm.o` file is the resulting object file by compiling clpvm.lsp with `lc` compiler.

When the setup is complete, one can perform the following simple test. We begin with a PVM session on the machine *Monkey*. Next, start the Lisp engine. CL-PVM routines are loaded with the *init.lsp* into the Lisp environment as explained in the previous paragraph. To construct a simple PVM, we add two more hosts: *Condor and Nimitz* to our PVM. *Condor* is an HP machine and Nimitz is a Sun processor. Figure 5 shows in detail the input and the output of `pvm-addhosts`.

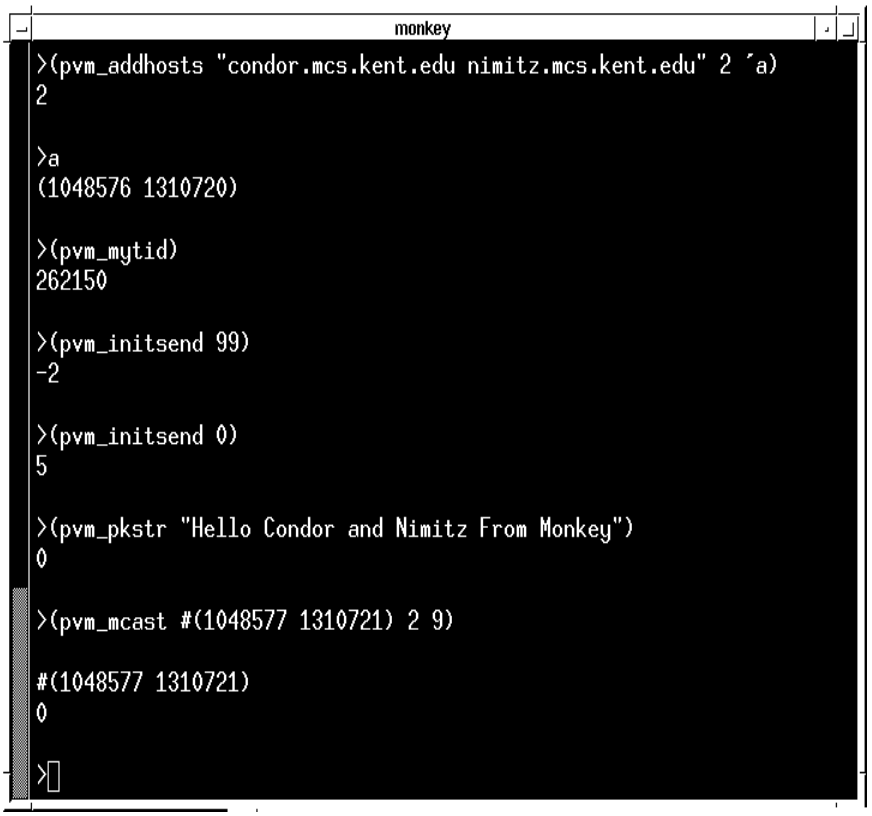

Figure 5: Invoking PVM routines interactively from host Monkey

Once the other two hosts are joined, we start the Lisp on everyone of them. We view the PVM configuration by invoking `pvm-config` on each host. The variables `a, b, c` are hosting variables that bring back useful information after the call. After the initial checkup we interactively performed a multi-casting from *Monkey* to *Nimitz* and *Condor* by sending a message " Hello Condor and Nimitz from Monkey" to the other two hosts. Figure 6 and Figure 7 capture the interaction on *Nimitz* and *Condor* respectively. The interacting capability of the Lisp environment actually provides us a convenient way of testing and debugging our interface routines. Entering CL-PVM environment is similar to typing in "pvm" at the shell prompt where one can monitor the performance of the PVM interactively. Moreover, the CL-PVM

environment not only perform the same task as a PVM console but also open up all the other library routines for interaction.

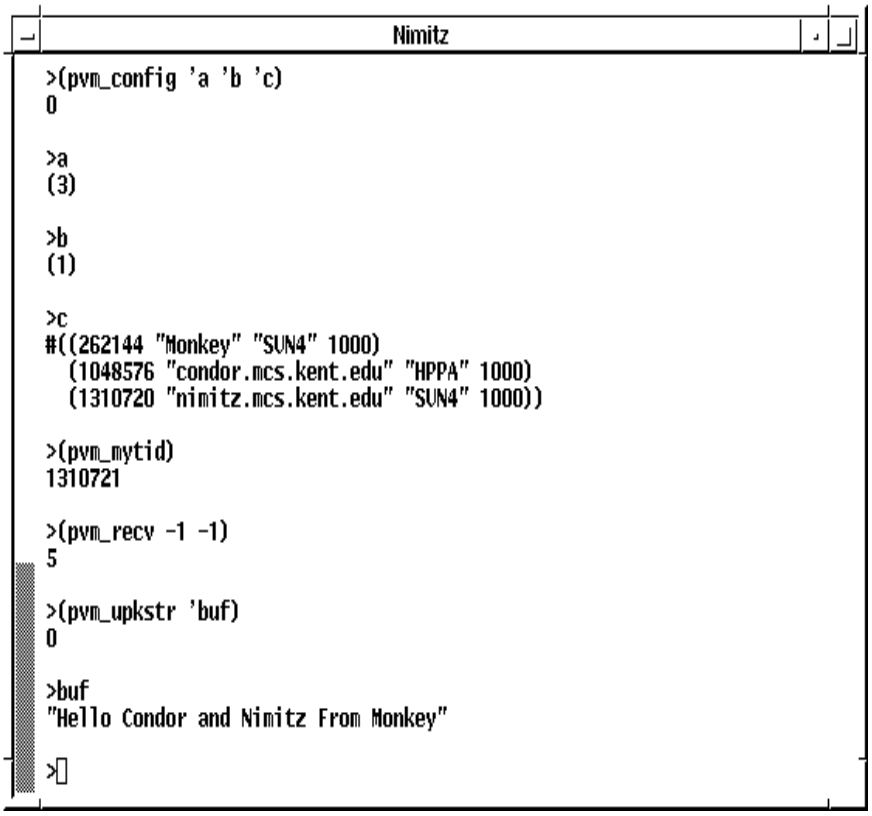

Figure 6: Capture of the interaction on host Nimitz

4 Linking Lisp and C Programs

Testing CL-PVM can be divided into three tasks: testing Lisp-C, C-Lisp, and Lisp-Lisp interactions. The tests can be performed with five simple programs:

`hello.c, hello.lsp,`
`hello_other.c, hello_other.lsp,`
`master.lsp, slave.c`

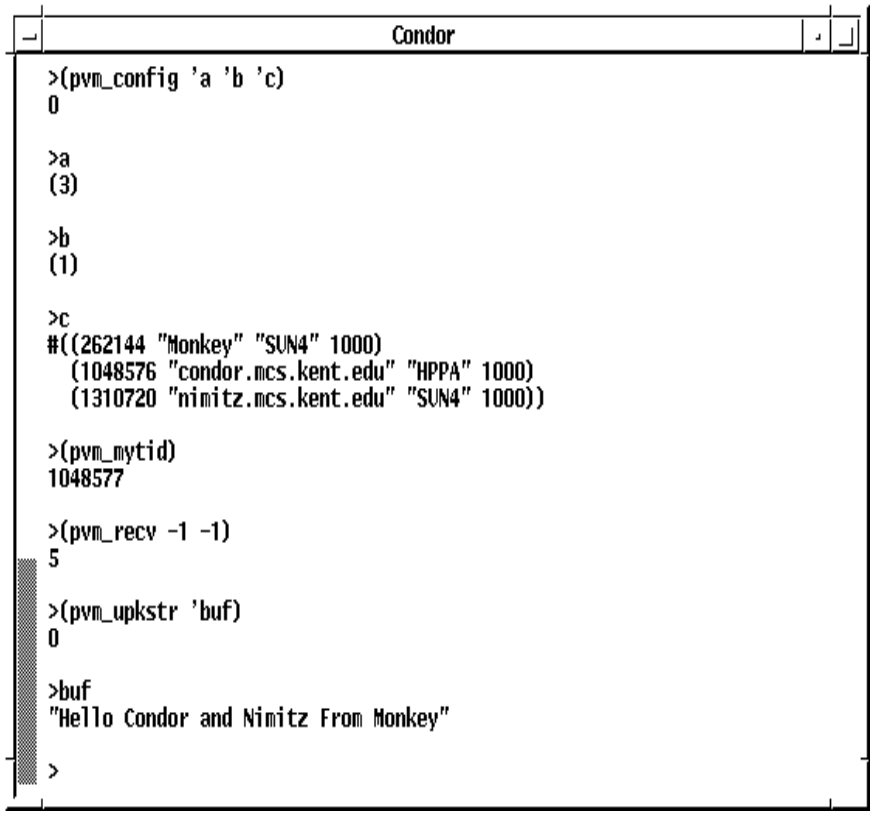

Figure 7: Capture of the interaction on host Condor

The C versions of these test programs have been written by the developers of the origin PVM3. The rest of the programs are written in Lisp (see below) to perform the same tests.

```
;;;;;;;;;;;;;;;;;;  hello.lsp  ;;;;;;;;;;;;;;;;;;
(defvar buf (make-string 100))
(defun hello()
(pvm-mytid)
(cond ( (not (equal
             (pvm-spawn "hello_other" "" 0 "nimitz.mcs.kent.edu" 1 'psids)
             1 ))
        (princ '|can't start hello_other|))
      (t
```

```
        (pvm-bufinfo (pvm-recv -1 -1) 'len 'code 'tid)
        (pvm-upkstr 'buf)
        (princ buf))))

;;;;;;;;;;;;;;  hello_other.lsp  ;;;;;;;;;;;;;;;;
(defvar buf "Hello World From Lisp")
(defun hello_other()
(pvm-initsend 0)
(pvm-pkstr buf)
(pvm-send (pvm-parent) 1)
(pvm-exit))
```

The `pvm_spawn` spawns process(es) on the specified hosts. It obviously works whether the executable has been generated by C, Fortran or Lisp. For this function it is only important to test if spawning from a Lisp process is successful. The program given below spawns three other slave processes and perform simple arithmetic operations.

```
;;;;;;;;;;;;;;;;;;  master.lsp  ;;;;;;;;;;;;;;;;;;;;
(defun master1()
(pvm-mytid)
(cond ( (not (equal (pvm-parent) 'PvmNoParent))
        (print "How many slaves to start 1-32?") (terpri)
        (let ((in (read)))
        (setq nproc in)))
      (t
        (pvm-config 'nhost 'narch 'hostp)
        (setq nproc (car nhost))
        (if (> nproc 32) (setq nproc 32))))
        (setq numt (pvm-spawn SLAVENAME "" 0
              "nimitz.mcs.kent.edu" nproc 'tids))
(cond ( (< numt nproc)
        (print "Trouble spawning slaves,
               Aborting. Error Codes are:") (terpri)
        (print tids)(terpri)
        (loop
           (if (equal (car tids) nil) (return (pvm-exit)))
           (pvm-kill (car tids))
```

```
          (setq tids (cdr tids)))
      ))
(setq n 100)
(pvm-initsend 0)
(pvm-pkint  nproc 1 1)
(pvm-pkint  tids nproc 1)
(pvm-pkint  n 1 1)
(pvm-pkfloat (make-array '(100) :initial-element
      (float 1.0 0.1s0) :element-type 'short-float)  n 1)
(pvm-mcast  tids nproc 0)
(setq msgtype 5)
(setq cnt 0)
(loop
    (if (equal cnt nproc) (return (pvm-exit)))
    (pvm-recv -1 msgtype)
    (pvm-upkint  'who 1 1)
    (pvm-upkfloat 'result 1 1)
    (print "From")
    (print who)
    (print "Result=")
    (print result)(terpri)
    (setq cnt (+ cnt 1))))
```

To invoke a Lisp executable file to test `hello_other.lsp` a simple shell script `cltest` can be constructed:

```
## cltest script

/usr/local/bin/akcl <<EOF
     (setq *load-verbose* nil)
     (load "hello_other.lsp")
     (hello_other)
     (bye)
EOF
```

And the `cltest` script is the executable file spawned by PVM.

5 Conclusion

CL-PVM is a working interface that allows CL programs to participate in PVM applications. The work is part of the overall effort at Kent [2] to interface symbolic computation systems with other systems to construct *problem solving environments* [**?**]. The CL-PVM implementation exploits the fact that CL is implemented in C and has macros to embed C codes. The CL compiler becomes the bridge that connects Lisp and C codes. The data type conversion relies on knowledge of the C structure used for Lisp objects. CL-PVM, as it stands, provides a minimum, lowest-level interface to the C routines of PVM. The name and arguments of each CL-PVM routine correspond directly to its PVM counterpart. This is a logical and sensible design. However, the convention makes usage of the CL-PVM routines less than natural for Lisp programmers who are more at home with Lisp objects such as lists and s-expressions and less so with strings separated by white space, for example. A natural extension of the current work is to build a layer on top of CL-PVM to supply convenient features for distributed Lisp applications that completely hides the C interface layer. The approach here can also be used to interface CL to the developing *Message Passing Interface*[2] standard.

A C structure header of Lisp

```
/*
(c) Copyright Taiichi Yuasa and Masami Hagiya, 1984.  All rights reserved.
Copying of this file is authorized to users who have executed the true and
proper "License Agreement for Kyoto Common LISP" with SIGLISP.
*/
#include <stdio.h>
#include <setjmp.h>
#define TRUE    1
#define FALSE   0
typedef int bool;
typedef int fixnum;
typedef float shortfloat;
typedef double longfloat;
typedef union lispunion *object;
```

```
#define OBJNULL ((object)NULL)
struct fixnum_struct {
        short   t, m;
        fixnum  FIXVAL;
};
#define fix(x)  (x)->FIX.FIXVAL
#define SMALL_FIXNUM_LIMIT      1024
struct fixnum_struct small_fixnum_table[];
#define small_fixnum(i) (object)(small_fixnum_table+SMALL_FIXNUM_LIMIT+(i))
struct shortfloat_struct {
        short           t, m;
        shortfloat      SFVAL;
};
#define sf(x)   (x)->SF.SFVAL
struct longfloat_struct {
        short           t, m;
        longfloat       LFVAL;
};
#define lf(x)   (x)->LF.LFVAL
struct character {
        short           t, m;
        unsigned short  ch_code;
        unsigned char   ch_font;
        unsigned char   ch_bits;
};
struct character character_table[];
#define code_char(c)    (object)(character_table+(c))
#define char_code(x)    (x)->ch.ch_code
#define char_font(x)    (x)->ch.ch_font
#define char_bits(x)    (x)->ch.ch_bits
enum stype {
        stp_ordinary,
        stp_constant,
        stp_special
};
struct symbol {
        short   t, m;
```

```
        object  s_dbind;
        int     (*s_sfdef)();
#define s_fillp         st_fillp
#define s_self          st_self
        int     s_fillp;
        char    *s_self;
        object  s_gfdef;
        object  s_plist;
        object  s_hpack;
        short   s_stype;
        short   s_mflag;
};
struct cons {
        short   t, m;
        object  c_cdr;
        object  c_car;
};
struct array {
        short   t, m;
        short   a_rank;
        short   a_adjustable;
        int     a_dim;
        int     *a_dims;
        object  *a_self;
        object  a_displaced;
        short   a_elttype;
        short   a_offset;
};
struct vector {
        short   t, m;
        short   v_hasfillp;
        short   v_adjustable;
        int     v_dim;
        int     v_fillp;
        object  *v_self;
        object  v_displaced;
        short   v_elttype;
```

```
        short   v_offset;
};
struct string {
        short   t, m;
        short   st_hasfillp;
        short   st_adjustable;
        int     st_dim;
        int     st_fillp;
        char    *st_self;
        object  st_displaced;
};
struct ustring {
        short   t, m;
        short   ust_hasfillp;
        short   ust_adjustable;
        int     ust_dim;
        int     ust_fillp;
        unsigned char
                *ust_self;
        object  ust_displaced;
};
struct bitvector {
        short   t, m;
        short   bv_hasfillp;
        short   bv_adjustable;
        int     bv_dim;
        int     bv_fillp;
        char    *bv_self;
        object  bv_displaced;
        short   bv_elttype;
        short   bv_offset;
};
struct fixarray {
        short   t, m;
        short   fixa_rank;
        short   fixa_adjustable;
        int     fixa_dim;
```

```
        int     *fixa_dims;
        fixnum  *fixa_self;
        object  fixa_displaced;
        short   fixa_elttype;
        short   fixa_offset;
};
struct sfarray {
        short   t, m;
        short   sfa_rank;
        short   sfa_adjustable;
        int     sfa_dim;
        int     *sfa_dims;
        shortfloat
                *sfa_self;
        object  sfa_displaced;
        short   sfa_elttype;
        short   sfa_offset;
};
struct lfarray {
        short   t, m;
        short   lfa_rank;
        short   lfa_adjustable;
        int     lfa_dim;
        int     *lfa_dims;
        longfloat
                *lfa_self;
        object  lfa_displaced;
        short   lfa_elttype;
        short   lfa_offset;
};
struct structure {
        short   t, m;
        object  str_name;
        object  *str_self;
        int     str_length;
};
struct cfun {
```

```
        short   t, m;
        object  cf_name;
        int     (*cf_self)();
        object  cf_data;
        char    *cf_start;
        int     cf_size;
};
struct cclosure {
        short   t, m;
        object  cc_name;
        int     (*cc_self)();
        object  cc_env;
        object  cc_data;
        char    *cc_start;
        int     cc_size;
        object  *cc_turbo;
};
struct dummy {
        short   t, m;
};
union lispunion {
        struct fixnum_struct
                        FIX;
        struct shortfloat_struct
                        SF;
        struct longfloat_struct
                        LF;
        struct character
                        ch;
        struct symbol   s;
        struct cons     c;
        struct array    a;
        struct vector   v;
        struct string   st;
        struct ustring  ust;
        struct bitvector
                        bv;
```

```
        struct structure
                        str;
        struct cfun     cf;
        struct cclosure cc;
        struct dummy    d;
        struct fixarray fixa;
        struct sfarray  sfa;
        struct lfarray  lfa;
};
enum type {
        t_cons = 0,
        t_start = t_cons,
        t_fixnum,
        t_bignum,
        t_ratio,
        t_shortfloat,
        t_longfloat,
        t_complex,
        t_character,
        t_symbol,
        t_package,
        t_hashtable,
        t_array,
        t_vector,
        t_string,
        t_bitvector,
        t_structure,
        t_stream,
        t_random,
        t_readtable,
        t_pathname,
        t_cfun,
        t_cclosure,
        t_spice,
        t_end,
        t_contiguous,
        t_relocatable,
```

```
        t_other
};
#define type_of(obje)   ((enum type)(((object)(obje))->d.t))
#define endp(obje)      endp1(obje)
object value_stack[];
#define vs_org          value_stack
object *vs_limit;
object *vs_base;
object *vs_top;
#define vs_push(obje)   (*vs_top++ = (obje))
#define vs_pop          (*--vs_top)
#define vs_head         vs_top[-1]
#define vs_mark         object *old_vs_top = vs_top
#define vs_reset        vs_top = old_vs_top
#define vs_check        if (vs_top >= vs_limit)  \
                                vs_overflow();
#define vs_check_push(obje)  \
                        (vs_top >= vs_limit ?  \
                         (object)vs_overflow() : (*vs_top++ = (obje)))
#define check_arg(n)  \
                        if (vs_top - vs_base != (n))  \
                                check_arg_failed(n)
#define MMcheck_arg(n)  \
                        if (vs_top - vs_base < (n))  \
                                too_few_arguments();  \
                        else if (vs_top - vs_base > (n))  \
                                too_many_arguments()
#define vs_reserve(x)   if(vs_base+(x) >= vs_limit)  \
                                vs_overflow();
struct bds_bd {
        object  bds_sym;
        object  bds_val;
};
struct bds_bd bind_stack[];
#define bds_org         bind_stack
typedef struct bds_bd *bds_ptr;
bds_ptr bds_limit;
```

```
bds_ptr bds_top;
#define bds_check  \
        if (bds_top >= bds_limit)  \
                bds_overflow()
#define bds_bind(sym, val)  \
        ((++bds_top)->bds_sym = (sym),  \
        bds_top->bds_val = (sym)->s.s_dbind,  \
        (sym)->s.s_dbind = (val))
#define bds_unwind1  \
        ((bds_top->bds_sym)->s.s_dbind = bds_top->bds_val, --bds_top)
typedef struct invocation_history {
        object  ihs_function;
        object  *ihs_base;
} *ihs_ptr;
struct invocation_history ihs_stack[];
#define ihs_org         ihs_stack
ihs_ptr ihs_limit;
ihs_ptr ihs_top;
#define ihs_check  \
        if (ihs_top >= ihs_limit)  \
                ihs_overflow()
#define ihs_push(function)  \
        (++ihs_top)->ihs_function = (function);  \
        ihs_top->ihs_base = vs_base
#define ihs_pop()       (ihs_top--)
enum fr_class {
        FRS_CATCH,
        FRS_CATCHALL,
        FRS_PROTECT
};
struct frame {
        jmp_buf         frs_jmpbuf;
        object          *frs_lex;
        bds_ptr         frs_bds_top;
        enum fr_class   frs_class;
        object          frs_val;
        ihs_ptr         frs_ihs;
```

```
};
typedef struct frame *frame_ptr;
#define alloc_frame_id()        alloc_object(t_spice)
struct frame frame_stack[];
#define frs_org         frame_stack
frame_ptr frs_limit;
frame_ptr frs_top;
#define frs_push(class, val)  \
        if (++frs_top >= frs_limit)  \
                frs_overflow();  \
        frs_top->frs_lex = lex_env;\
        frs_top->frs_bds_top = bds_top;  \
        frs_top->frs_class = (class);  \
        frs_top->frs_val = (val);  \
        frs_top->frs_ihs = ihs_top;  \
        setjmp(frs_top->frs_jmpbuf)
#define frs_pop()       frs_top--
bool nlj_active;
frame_ptr nlj_fr;
object nlj_tag;
object *lex_env;
object caar();
object cadr();
object cdar();
object cddr();
object caaar();
object caadr();
object cadar();
object caddr();
object cdaar();
object cdadr();
object cddar();
object cdddr();
object caaaar();
object caaadr();
object caadar();
object caaddr();
```

```
object cadaar();
object cadadr();
object caddar();
object cadddr();
object cdaaar();
object cdaadr();
object cdadar();
object cdaddr();
object cddaar();
object cddadr();
object cdddar();
object cddddr();
#define MMcons(a,d)     make_cons((a),(d))
#define MMcar(x)        (x)->c.c_car
#define MMcdr(x)        (x)->c.c_cdr
#define CMPcar(x)       (x)->c.c_car
#define CMPcdr(x)       (x)->c.c_cdr
#define CMPcaar(x)      (x)->c.c_car->c.c_car
#define CMPcadr(x)      (x)->c.c_cdr->c.c_car
#define CMPcdar(x)      (x)->c.c_car->c.c_cdr
#define CMPcddr(x)      (x)->c.c_cdr->c.c_cdr
#define CMPcaaar(x)     (x)->c.c_car->c.c_car->c.c_car
#define CMPcaadr(x)     (x)->c.c_cdr->c.c_car->c.c_car
#define CMPcadar(x)     (x)->c.c_car->c.c_cdr->c.c_car
#define CMPcaddr(x)     (x)->c.c_cdr->c.c_cdr->c.c_car
#define CMPcdaar(x)     (x)->c.c_car->c.c_car->c.c_cdr
#define CMPcdadr(x)     (x)->c.c_cdr->c.c_car->c.c_cdr
#define CMPcddar(x)     (x)->c.c_car->c.c_cdr->c.c_cdr
#define CMPcdddr(x)     (x)->c.c_cdr->c.c_cdr->c.c_cdr
#define CMPcaaaar(x)    (x)->c.c_car->c.c_car->c.c_car->c.c_car
#define CMPcaaadr(x)    (x)->c.c_cdr->c.c_car->c.c_car->c.c_car
#define CMPcaadar(x)    (x)->c.c_car->c.c_cdr->c.c_car->c.c_car
#define CMPcaaddr(x)    (x)->c.c_cdr->c.c_cdr->c.c_car->c.c_car
#define CMPcadaar(x)    (x)->c.c_car->c.c_car->c.c_cdr->c.c_car
#define CMPcadadr(x)    (x)->c.c_cdr->c.c_car->c.c_cdr->c.c_car
#define CMPcaddar(x)    (x)->c.c_car->c.c_cdr->c.c_cdr->c.c_car
#define CMPcadddr(x)    (x)->c.c_cdr->c.c_cdr->c.c_cdr->c.c_car
```

```
#define CMPcdaaar(x)    (x)->c.c_car->c.c_car->c.c_car->c.c_cdr
#define CMPcdaadr(x)    (x)->c.c_cdr->c.c_car->c.c_car->c.c_cdr
#define CMPcdadar(x)    (x)->c.c_car->c.c_cdr->c.c_car->c.c_cdr
#define CMPcdaddr(x)    (x)->c.c_cdr->c.c_cdr->c.c_car->c.c_cdr
#define CMPcddaar(x)    (x)->c.c_car->c.c_car->c.c_cdr->c.c_cdr
#define CMPcddadr(x)    (x)->c.c_cdr->c.c_car->c.c_cdr->c.c_cdr
#define CMPcdddar(x)    (x)->c.c_car->c.c_cdr->c.c_cdr->c.c_cdr
#define CMPcddddr(x)    (x)->c.c_cdr->c.c_cdr->c.c_cdr->c.c_cdr
#define CMPfuncall      funcall
#define cclosure_call   funcall
object simple_lispcall();
object simple_lispcall_no_event();
object simple_symlispcall();
object simple_symlispcall_no_event();
object CMPtemp;
object CMPtemp1;
object CMPtemp2;
object CMPtemp3;
#define Cnil    ((object)&Cnil_body)
#define Ct      ((object)&Ct_body)
struct symbol Cnil_body, Ct_body;
object MF();
object MM();
object Scons;
object siSfunction_documentation;
object siSvariable_documentation;
object siSpretty_print_format;
object Slist;
object keyword_package;
object alloc_object();
object car();
object cdr();
object list();
object listA();
object coerce_to_string();
object elt();
object elt_set();
```

```
frame_ptr frs_sch();
frame_ptr frs_sch_catch();
object make_cclosure();
object nth();
object nthcdr();
object make_cons();
object append();
object nconc();
object reverse();
object nreverse();
object number_expt();
object number_minus();
object number_negate();
object number_plus();
object number_times();
object one_minus();
object one_plus();
object get();
object getf();
object putprop();
object remprop();
object string_to_object();
object symbol_function();
object symbol_value();
object make_fixnum();
object make_shortfloat();
object make_longfloat();
object structure_ref();
object structure_set();
object princ();
object prin1();
object print();
object terpri();
object aref();
object aset();
object aref1();
object aset1();
```

```
char object_to_char();
int object_to_int();
float object_to_float();
double object_to_double();
int FIXtemp;
#define CMPmake_fixnum(x) \
((((FIXtemp=(x))+1024)&-2048)==0?small_fixnum(FIXtemp):make_fixnum(FIXtemp))
#define Creturn(v) return((vs_top=vs,(v)))
#define Cexit return((vs_top=vs,0))
double sin(), cos(), tan();
```

B Headers of PVM C Library Routines

```
int info = pvm_addhosts( char **hosts, {int nhost, int *infos} )
int info = pvm_advise( int route )
int cod = pvm_archcode( char *arch )
int info = pvm_barrier( char *group, int count )
int info = pvm_bcast( char *group, int msgtag )
int info = pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid)
int info = pvm_catchout( FILE *ff )
int info = pvm_config( int *nhost, int *narch, struct pvmhostinfo **hostp )
int cc = pvm_delete( char *name, int index )
int info = pvm_delhosts( char **hosts, int nhost, int *infos )
int info = pvm_exit( void )
int info = pvm_freebuf( int bufid )
int info = pvm_gather( void *result,void *data,int count,
          int datatype,int msgtag,char *group, int rootginst)
int inum = pvm_getinst( char *group, int tid )
int val = pvm_getopt( int what )
int bufid = pvm_getrbuf( void )
int bufid = pvm_getsbuf( void )
int tid = pvm_gettid( char *group, int inum )
int size = pvm_gsize( char *group )
int info = pvm_halt( void )
int info = pvm_hostsync( int host, struct timeval *clk,struct timeval *delta)
int bufid = pvm_initsend( int encoding )
```

```
int cc = pvm_insert( char *name, int index, int data )
int inum = pvm_joingroup( char *group )
int info = pvm_kill( int tid )
int cc = pvm_lookup( char *name, int index, {int *data} )
int info = pvm_lvgroup( char *group )
int info = pvm_mcast( int *tids, int ntask, int msgtag )
int bufid = pvm_mkbuf( int encoding )
int mstat = pvm_mstat( char *host )
int tid = pvm_mytid( void )
int info = pvm_notify( int what, int msgtag, int cnt, int *tids)
int bufid = pvm_nrecv( int tid, int msgtag )
int info = pvm_packf( const char *fmt, ... )
int info = pvm_pkbyte( char *xp, int nitem, int stride )
int info = pvm_pkcplx( float *cp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp, int nitem, int stride )
int info = pvm_pkint( int *ip, int nitem, int stride )
int info = pvm_pkuint( unsigned int *ip, int nitem, int stride )
int info = pvm_pkushort( unsigned short *ip, int nitem, int stride )
int info = pvm_pkulong( unsigned long *ip, int nitem, int stride )
int info = pvm_pklong( long *ip, int nitem, int stride )
int info = pvm_pkshort( short *jp, int nitem, int stride )
int info = pvm_pkstr( char *sp )
int tid = pvm_parent( void )
int info = pvm_perror( char *msg )
int info = pvm_precv( int tid, int msgtag, char *buf, int len},
           int datatype, int *atid, int *atag, int *alen )
int bufid = pvm_probe( int tid, int msgtag )
int info = pvm_psend( int tid, int msgtag, char *buf, int len, int datatype )
int status = pvm_pstat( int tid )
int bufid = pvm_recv( int tid, int msgtag )
int (*old)() = pvm_recvf( int (*new)( int bufid, int tid, int tag ))
int info = pvm_reduce( void (*func)(),void *data, int count,
            int datatype,int msgtag, char *group, int rootginst)
int cc = pvm_reg_hoster()
int cc = pvm_reg_rm( struct pvmhostinfo **hip )
```

```
int cc = pvm_reg_tasker()
int info = pvm_scatter( void *result, void *data,int count,
          int datatype, int msgtag,char *group, int rootginst)
int info = pvm_send( int tid, int msgtag )
int info = pvm_sendsig( int tid, int signum )
int oldset = pvm_serror( int set )
int info = pvm_getmwid( int bufid )
int info = pvm_setmwid( int bufid, int waitid )
int oldval = pvm_setopt( int what, int val )
int oldbuf = pvm_setrbuf( int bufid )
int oldbuf = pvm_setsbuf( int bufid )
int info = pvm_gettmask( int who, Pvmtmask mask )
int info = pvm_settmask( int who, Pvmtmask mask )
int numt = pvm_spawn( char *task, char **argv, int flag,
          char *where, int ntask, int *tids )
int info = pvm_start_pvmd( int argc, char **argv, int block )
int info = pvm_tasks( int where, int *ntask, struct pvmtaskinfo **taskp )
int dtid = pvm_tidtohost( int tid )
int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )
int info = pvm_unpackf( const char *fmt, ... )
int info = pvm_upkbyte( char *xp, int nitem, int stride)
int info = pvm_upkcplx( float *cp, int nitem, int stride)
int info = pvm_upkdcplx( double *zp, int nitem, int stride)
int info = pvm_upkdouble( double *dp, int nitem, int stride)
int info = pvm_upkfloat( float *fp, int nitem, int stride)
int info = pvm_upkint( int *ip, int nitem, int stride)
int info = pvm_upkuint( unsigned int *ip, int nitem, int stride)
int info = pvm_upkushort( unsigned short *ip, int nitem, int stride)
int info = pvm_upkulong( unsigned long *ip, int nitem, int stride)
int info = pvm_upklong( long *ip, int nitem, int stride)
int info = pvm_upkshort( short *jp, int nitem, int stride)
int info = pvm_upkstr( char *sp )
```

C CL-PVM Routines

```
(pvm-addhosts object int object)
```

```
(pvm-advise int)
(pvm-archcode object)
(pvm-barrier object int)
(pvm-bufinfo int object object object)
(pvm-catchout object)
(pvm-config object object object)
(pvm-delhosts object int object)
(pvm-freebuf int)
(pvm-hostsync int object object)
(pvm-initsend int)
(pvm-joingroup object)
(pvm-lvgroup object)
(pvm-reg_rm object)
(pvm-gettmask int object)
(pvm-settmask int object)
(pvm-spawn object object int object int object)
(pvm-tasks int object object)
(pvm-trecv int int object)
(pvm-packf object)
(pvm-unpackf object)
(pvm-delete object int)
(pvm-getopt int)
(pvm-getrbuf)
(pvm-getsbuf)
(pvm-halt)
(pvm-insert object int int)
(pvm-kill int)
(pvm-lookup object int object)
(pvm-mcast object int int)
(pvm-mkbuf int)
(pvm-mstat object)
(pvm-notify int int int object)
(pvm-nrecv int int)
(pvm-pkbyte object int int)
(pvm-pkcplx object int int)
(pvm-pkdcplx object int int)
(pvm-pkdouble object int int)
```

```
(pvm-pkfloat object int int)
(pvm-pkint object int int)
(pvm-pkuint object int int)
(pvm-pkushort object int int)
(pvm-pkshort object int int)
(pvm-pkstr object)
(pvm-pkulong object int int)
(pvm-pklong object int int)
(pvm-parent)
(pvm-perror object)
(pvm-precv int int object int int object object object)
(pvm-probe int int)
(pvm-psend int int object int int)
(pvm-pstat int)
(pvm-recv int int)
(pvm-reg_hoster)
(pvm-reg_tasker)
(pvm-send int int)
(pvm-sendsig int int)
(pvm-getmwid int)
(pvm-setmwid int int)
(pvm-setopt int int)
(pvm-setrbuf int)
(pvm-setsbuf int)
(pvm-start_pvmd int object int)
(pvm-tidtohost int)
(pvm-upkbyte object int int)
(pvm-upkcplx object int int)
(pvm-upkdcplx object int int)
(pvm-upkdouble object int int)
(pvm-upkfloat object int int)
(pvm-upkint object int int)
(pvm-upkuint object int int)
(pvm-upkushort object int int)
(pvm-upkulong object int int)
(pvm-upklong object int int)
(pvm-upkshort object int int)
```

```
(pvm-upkstr object)
(pvm-recvf)*
(pvm-reduce)*
(pvm-gather object object int int object int)*
(pvm-scatter object object int int object int)*

* : not implemented
```

References

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, 1993.

[2] S. Gray, N. Kajler, and P. Wang, "MP: A Protocol for Efficient Exchange of Mathematical Expressions," Proceedings, ISSAC'94, Oxford UK, ACM Press, pp. 330-335, July, 1994.

[3] P. Wang, *An Introduction to ANSI C on UNIX*, PWS Publishing Co. Boston, MA., 1991.

[4] R. Wilensky, *Common LISPcraft*, W. W. Norton & Co., N.Y., 1986.

[5] T. Yuasa and M. Hagiya, "Kyoto Common Lisp Dictionary", Kyoto University, KCL on-line documentation" 1986.