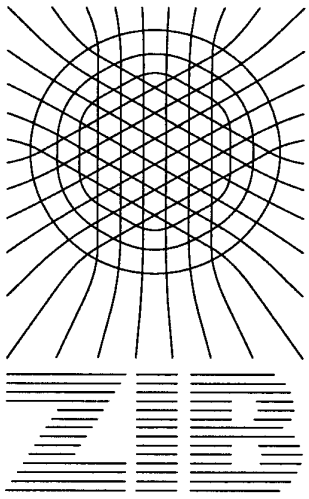


Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Herbert Melenk Winfried Neun

Portable Standard LISP Implementation for CRAY X+MP Computers.

Release of PSL 3.4 for COS.

Technical Report 87-2

Portable Standard Lisp

Implementation for Cray X-MP Computers

Release of PSL 3.4 for COS

Herbert Melenk
Winfried Neun

Konrad-Zuse-Zentrum für Informationstechnik
Berlin

September 1986

Technical Report TR 87-2

Cooperative Project of ZIB Berlin and Cray Research

Portable Standard Lisp (PSL) is a portable implementation of the programming language LISP constructed at the University of Utah. The version 3.4 of PSL was implemented for Cray X-MP computers by Konrad-Zuse-Zentrum Berlin; this implementation is based to an important part on the earlier implementation of PSL 3.2 at Salt Lake City, Los Alamos and Mendota Heights.

Table of Contents

0. Introduction

1. Usage of COS PSL 3.4
   1.1 Getting Started
   1.2 Dataset Handling
   1.3 Memory Management
   1.4 Compiler and Disassembler
   1.5 Libraries
   1.6 Calling COS Commands
   1.7 Reprieve Mechanism
   1.8 I/O Functions for Word Addressable Files
   1.9 Counting
   1.10 List of Available Utilities

2. Installation Guide
   2.1 Distribution Tape
   2.2 Description of the file PSLINST
   2.3 Presetting System Variables and Memory Sizes
   2.4 Lists of Modules

3. Implementation Details
   3.1 COS Specific Features
   3.2 Mapping of PSL to the Cray X-MP Architecture
   3.3 Memory Layout
   3.4 I/O Operations
   3.5 Batch Processing
   3.6 Arithmetic
   3.7 Cray Specific Compiler Features
   3.8 Test Assistance

0. Introduction

The work of implementing Portable Standard Lisp (PSL) version 3.4 for Cray 1 and Cray X-MP computers has led to a first distributable release, named "Release 5". The present implementation is a continuation of the work started at Utah and Los Alamos implementing PSL 3.2 for Cray computers. Apart from the general difference between versions 3.2 and 3.4 it differs from its predecessor by the following aspects:

Security

Provisions were taken to **protect** the running system against **overflow** of one of the stacks. The **reprieve handling** was enlarged in order to continue LISP after system interrupts whenever possible (exception: time overflow in batch mode). Some **diagnostic** features giving information about problem cause and location and for analysing a running system are incorporated.

Usability

A **dynamic memory management** was incorporated into PSL; all relevant portions of the memory can be enlarged and shrinked on request; the target area for compiled code is **enlarged automatically** in case of lack. The **access path** scheme for files was enlarged: local files can be processed as well as permanent datasets residing under the own or a foreign owner id. Because of the wide spread usage of batch operation in the Cray world a **batch adapted behaviour** was implemented for PSL if running in batch mode (e.g. generating standard answers for system questions, including a bracket counter into the echo printing of input). Because of the absence of a directory hierarchy in COS and in order to increase speed when loading compiled modules a **library facility** based on word addressable files was integrated into PSL. The COS **command language statements** for dataset access and dataset staging control are available in PSL via the dataset management subprograms of the COS library and a LISP specific interface: a LISP user in batch or interactive mode can invoke a COS command in its original syntax as a string or in a format open to LISP calculations.

Completeness

All LISP language features and most utilities described in the PSL manual are available for Cray PSL. The tools for enlarging a local PSL version are included. PSL is completely self bootstrapping on a Cray computer without need for foreign machine execution.

Execution Speed

The execution speed was augmented by several means: the function **cons** is compiled **inline**, many global control **variables** are held in **registers**, the **stack frame** technique was **refined**, unnecessary **memory access** operations are **deleted** in many cases, on system level **vector instructions** are used by handcoded routines (LAP level), FORTRAN calls were removed from **arithmetic** completely, the loading of modules was

speeded up by the **library concept** (reduction of system overhead when opening and closing files), some **system routines** heavily used were **reformulated** adapted to the code production strategy of the compiler or were **coded by hand** using the LAP language level.

The following document includes three parts describing **usage**, **installation** and **implementation specifics** of this version.

1. Usage of COS PSL 3.4

The following description assumes PSL to be installed due to our installation guide (cf. chapter 2).

1.1 Getting Started

PSL 3.4 resides in a file on disk as absolute binary program. The program is started by the following commands:

```
ACCESS,DN=PSL34,ID=PSL,OWN=ask your computing center.
PSL34.
```

PSL expects input from $IN and produces output on $OUT.

```
Remark: In batch mode the modes of operation are set automatically:
        (on echo)          % echo of input to $OUT
        (off usermode)     % system functions may be redefined without
                           % extra confirmation
        (on parens)        % parenthesis depth counter printed
```

Sample job:

```
JOB,...
ACCOUNT,...
ACCESS,DN=PSL34,ID=PSL,OWN=ask your computing center.
PSL34.
/EOF
(load big)    % yes, big numbers do work!
(expt 2 100)
(quit)
```

1.2 Dataset Handling

A COS-userjob may use several types of datasets and modes to process a dataset, e.g.:

- datasets which are local to the job (the DN has up to 7 uppercase characters and is alphanumeric) either saved or not,

- permanent datasets which are not local to the job (the DN has up to 15 arbitrary characters, including non alphanumeric characters, an ID is optional and the Ownership-value can be different from the jobs default OWN),

- datasets which reside on frontend-systems and must be acquired before use and/or must be saved after use.

An access to a permanent dataset may not be possible because the file is locked by another UQ access. In this case it will be nice to wait if a batchjob tries the access, otherwise it wont be so nice to hang up terminal session.
Messages about files accessed or saved are very useful if you are debugging the system, but the running system dont need to show every file accessed to the user.

PSL 3.4 (CRAY) is able to use the above three types of datasets. The standard open processing is:

(OPEN name 'INPUT) :

PSL checks whether the DN is up to 7 uppercase characters long and alphanumeric. If so, it is inquired whether the dataset is already local. If this is true, the dataset is opened.

Otherwise, an ACCESS-command is issued for the filename as PDN with an generated DN (FTnnPSL), an user-supplied ID field (default 'PSL'L) and the jobs default OWN. If the ACCESS is successful, the file is opened. Otherwise a second ACCESS command is issued with an user-suppliable OWN-field. If this second ACCESS succeeds, the file is opened, otherwise LISP function conterror is called.

(OPEN name 'OUTPUT) :

A new dataset is created with generated DN (FTnnPSL) in any case.

(CLOSE filedescriptor) :

If the dataset was opened for write access, the dataset is closed and saved. Otherwise the dataset is closed. The dataset is released, if it was non local before open processing.

Setting the ID or OWN field is managed via call to

```
(COS-Set-Id   string)        resp.
(COS-Set-Own  string)  before  OPEN.
```

Note : the variable owner-of-psl* is bound to a string containing the installation default, so a user must not know that default value. (cos-set-own owner-of-psl*) resets the ownership value.

The parameters remain valid until a new setting takes place.

Waiting for file access and messages:

The default for access operation is :
no message
wait iff batchmode.
To alter this, use:

(cos-waitio T) or (cos-waitio NIL) resp.
(cos-msgio T) or (cos-msgio NIL)

cos-waitio is called as side-effect by function batch? to preset the value after system startup (for (batch?) see batch processing).

1.3 Memory Management

The CRAY-1/X-MP systems are real memory computers, and from there results the necessity to save memory whereever it is possible and not too expensive. The biggest parts of LISP memory are Heap, Bps, Stack and Bindingstack, each of an individual size depending on the LISP application running. The initial size of heap, bps, stack and binding stack can't be too big, because in this case any COS/LISP-Job would behave like an "elephant" in the COS-scheduler, causing a very bad priority and disturbing the other participants.

Bytheway: another big part of memory is the hash table, which is implemented as quarter words (parcels) in CRAY PSL. So memory requirements for the hash table was reduced.

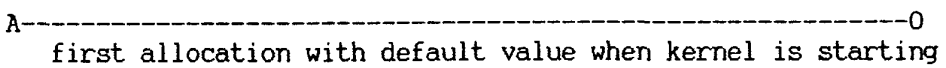
On the other hand many LISP applications need large memory portions for Stack, Bps, Heap or Bindingstack. For example running an interactive job you cant really predict which options must be loaded.

The PSL/COS Interface offers dynamic allocation for Stack, Heap, Bps and Bindingstack via calls to COS heap-manager.

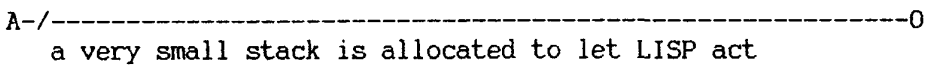
Memory for bps, heap, binding stack and stack is allocated in one portion with the size of (Plus initial-bpssize initial-heapsize initial-stacksize initial-bndstksize 30). Bpssize, Stacksize Bndstksize and Heapsize are fluids with an appropriate initial value to satisfy the memory requirements during system startup.

When any memory portion needs to be enlarged (or shrinked) the memory block is enlarged (shrinked) via call to the COS heap manager. Then the memory layout is redone, memory portions are moved (for this purpose a vector copy has been incorporated) and pointers are updated.

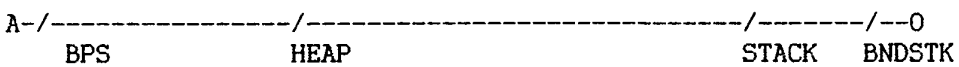
The memory layout is as follows:



first allocation with default value when kernel is starting



a very small stack is allocated to let LISP act



BPS HEAP STACK BNDSTK

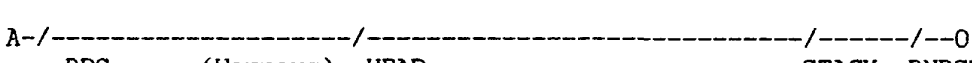
the initial memory layout is done in LISP as first effort

LISP-functions are supplied to enlarge /shrink current allocation for Bps, Heap, Bndstk or Stack:

```
(Set-Stack-size  new-value)
(Set-Heap-size   new-value)
(Set-Bps-size    new-value)
(Set-Bndstk-size new-value)
```

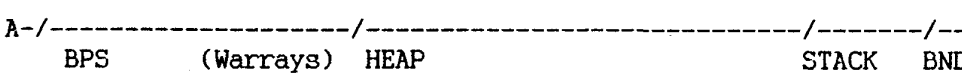
The current values are values of the fluids stacksize, heapsize, bndstksize and bpssize.

When the user wants to enlarge e.g. BPS the following operations will happen: (Set-Bps-size (plus Bpssize 20000)) or loading fails because of the lack of BPS and enlarging of BPS is done automatically:



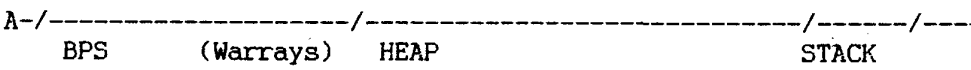
BPS (Warrays) HEAP STACK BNDSTK

1. Memory is enlarged by COS heap manager, if memory is available:



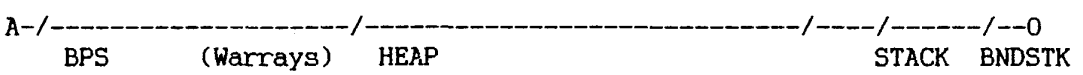
BPS (Warrays) HEAP STACK BNDSTK

2. Movement of Bndstk, update of CatchStack:

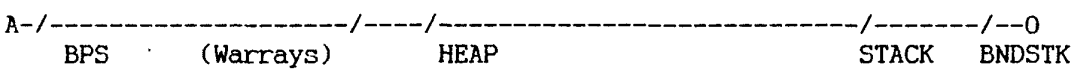


BPS (Warrays) HEAP STACK BNDSTK

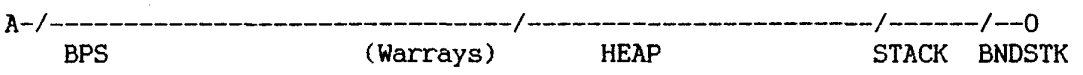
3. Movement of Stack, update of CatchStack:



4. Relink of Heap and Warrays via garbage-collector techniques, movement of heap:



5. Movement of Warrays, update of register pointers into Warray space:



Everything is fine again.

No abnormal situation should occur while memory is moved, because any pointer can be damaged in the moment.

There is a special utility called counting in the compiler, which counts memory references to fluids and globals and function calls. It allocates an array called SYMCNT for counting located above the BNDSTK. This makes the situation described above a bit more complicated because another memory portion has to be moved.

1.4 Compiler and Disassembler

The Cray PSL compiler has an additional final LAP ("LISP Assembly Program") path for code optimization on instruction basis. This path can be influenced by the value of the variable

*LapOpt

Possible values are:

T	full LAP optimization
NIL	no LAP optimization
(opt1 opt2 ...)	list of selected optimization options.

Note that the values T and NIL can be set by using the forms (ON LapOpt) and (OFF LapOpt).

A trace of the lapopt changes on the code can be achieved by turning on the switch

Lapopttrace.

At the moment the following options are implemented:

LapOptLoads1	first package of Load optimization: redundant load and move instructions are removed from the code.
LapOptLoads2	second package of load optimization: instructions for loading from memory and instructions using loaded information are moved in the code in order to use the loading time for other independant instructions whenever possible.

At the moment only the scalar instructions of the CRAY X-MP computers are produced by the compiler. An experimental upgrade of the compiler for the usage of vector instructions is included in the PSL library. If the vector instructions are to be used an additional module has to be loaded explicitly:

```
(load vector-instructions)
```

The code generated by the compiler can be inspected by setting one of the PSL standard list variables or by using the disassembler. The function "disassemble" converts compiled and system functions from machine code back to a form similar to Cray assembly language. The text is produced for the standard output file OUT*. The assembly form contains references to LISP wherever possible, e.g. explicit names of fluid variables, functions called, names of system registers etc. This code can be understood only if the Cray assembly language is known. The program "disassemble" resides in a load module with the same name. It can be invoked by

```
(load disassemble)
(disassemble 'myFirstFunction)
(disassemble 'mySecondFunction)
 . . .
```

If "disassemble" is used to learn about the PSL implementation itself, it is recommended to turn off the lap optimization because it is rather difficult to find the connection to a source program if the code has been rearranged by the optimizer.

1.5 Libraries

A group of precompiled functions and data structures can be collected to a load module within PSL. A load module can be loaded into a running PSL; this loading is much faster than the compilation of its constituents from the sources. A load module is constructed by computing definitions (function definitions, puts, setqs etc.) between a starting call to FASLOUT and an ending call to FASLEND.

A load module resides in a sequential file the name of which is given in the FASLOUT call (it will be suffixed by ".b" automatically). Normally each of these files will be an individual dataset to the operating system. Whithout regarding user applications the PSL system itself contains already some hundred load modules, which will be loaded to a running PSL in part. In order to reduce the number of datasets and the number of open calls a library concept has been introduced to PSL. A library is a named collection of load modules in one dataset which can be opened as a whole. Libraries and individual datasets are fully compatible: a load module can be part of a library and it can be an individual dataset; the dynamic selection of the source for the loading is done due to precedence rules.

1.5.1 Definition of a Library

A library is a dataset, which contains a number of sequential binary files. The library contains at any time an actual directory, which describes the files by name and word position within the dataset. There are no restrictions to the file names (they have to be simply PSL strings), but only names with respect to the naming conventions of load modules make sense.

The directory resides always at the end of a library and the head of a library holds a pointer to the actual directory. When the library is opened the directory is read into memory as a LISP structure and it will be interpreted for load and update purposes. In case of update all new information is written behind the actual directory. The new directory will be written as the last action; so the library remains logically unchangend during the whole update process and an intermediate abort will not affect the integrity of the library.

1.5.2 Open and Close of Libraries

A library is opened by a call to the macro

```
(LIBOPEN name mod)

   with mod = INPUT / UPDATE / NEW
```

or

```
(LIBOPEN name)

   with implicit mod = INPUT
```

The parameters are quoted automatically by the macro. The same effect can be formulated by a call to the expr function

```
(LIBOPEN1 name mod)
```

which expects evaluated parameters (e.g. QUOTE-forms).

If a library is opened for INPUT or UPDATE, it must contain a valid directory; in this case modules can be loaded from this library. The value of CurrentInputLib* is set to the library name. If a library is opened UPDATE or NEW, new files can be added to the library. The name of

the library is placed in front of CurrentOutputLib*. In the case of NEW an existing library will be overwritten totally. NEW is the only possible value for a non existing library.

A library is closed by a call to the macro

```
(LIBCLOSE name)
```

or the EXPR function

```
(LIBCLOSE1 name)
```

In the case of library modifications the directory is updated in memory. It will be written to the file at close time. By a close call all connections to the library are given up.

Special case: A call

```
(LIBCLOSE T)
```

causes all currently open libraries to be closed. The library software maintains a variable *AllLibraries*, which contains the names of all currently open libraries. This variable should not be modified.

1.5.3 Loading from Libraries

The loading of modules is done by explicit or implicit (autoload) calls to LOAD. LOAD looks for the value of the PSL variable

```
LoadDirectories*
```

which is a list of strings; LOAD takes these strings as prefixes for the filename. The concept of LoadDirectories* has been enlarged for the libraries: Additional to strings the special list form

```
(LIB  name)
```

can be member of the list LoadDirectories*. In this case the library with the given name is inserted to the LOAD search path. The variable LoadDirectories* is updated automatically by LIBOPEN and LIBCLOSE calls: an open for an INPUT or UPDATE library places the new element in front of LoadDirectories* (highest priority) and a close deletes the library reference from the variable. Besides this automatic feature the variable can be manipulated "by hand",

e.g.:

```
(LIBOPEN MINE)
(LIBOPEN UTLIB)
(setq LoadDirectories* (cons "" LoadDirectories*))
  % value of LoadDirectories: ("" (LIB UTLIB) (LIB MINE))
(LOAD CALCU)
```

The first element of LoadDirectories* is the empty string and so the filename "calcu.b" is tried first as individual dataset. If not found, the library UTLIB is looked up for the file and the library MINE afterwards. The first occurence of the load module is accepted. If the

load module is not found anywhere the loading stops with error. If one of the libraries in LoadDirectories* is not open, it is simply ignored.

```
(setq LoadDirectories* '( (LIB UTLIB) "" ))
```

In this case the library is searched first and the single file dataset second.

1.5.4 Standard Libraries

Two standard libraries are part of a COS PSL installation. The library NONKERNELLIB contains modules being part of the PSL main program. Normally they are loaded during the installation phase and the user of PSL does not need a reference to this library. The second library with the name UTLIB contains those parts of PSL which are loaded to an application program on explicit or implicit request at runtime. UTLIB contains the PSL compiler and the important utilities. The library UTLIB must be open at PSL run time. Otherwise the compiler and the utilities cannot be used.

If COS PSL is installed in the standard manner, the library UTLIB is opened automatically at start time of PSL. So if the user does not use additional private libraries with complicated loading hierarchies he can ignore the presence of UTLIB. UTLIB is installed with a central owner id in a shareble manner. If due to a complicated situation UTLIB has to be opened "by hand" (e.g. in a private SAVESYSTEM context), the user has to make shure, that the correct owner id for UTLIB is known to the system. The owner id is passed to the system by the function COS-SET-OWN. The name of the PSL owner id is value of the variable OWNER-OF-PSL* which is set to the necessary value by the normal installation job, which does a COS-SET-OWN for this value too.

1.5.5 Libraries at SAVESYSTEM Time

It is very important to close **all** libraries (the lib UTLIB too!) at SAVESYSTEM time. Otherwise the directory in memory would be saved with the application but the file connection is lost. This closing should be performed explicitly. If not done savesystem forces a closing itself. In the third parameter of SAVESYSTEM a list of expressions is fixed which will be performed when the saved application will be started again. Typically the reopen calls for the closed libraries can be part of this list.

e.g.

```
(libclose T)                              % close all libs
(savesystem "new algebra" "NEWALG"
   '( (cos-set-own OWNER-OF-PSL*)   % owner for UTLIB
      (libopen UTLIB)
      (cos-set-own "MATRIXID")      % another owner
      (libopen MATRIXLIB)
      (cos-set-own OWNER-OF-PSL*))) % reset to standard value
```

1.5.6 Producing New Library Elements

If the list CurrentOutputLib* is not empty (that means a library has been opened NEW or UPDATE) a FASLOUT-FASLEND sequence automatically

produces a new file in the first element of CurrentOutputLib*. If CurrentOutputLib* contains no library name, an individual dataset is produced as is done in other PSL implementations.

Besides producing new load modules library members can be produced by copying existing modules:

```
(FaslCopy mod1 mod2)
  or
(FaslCopy mod1)   with mod2 = mod1
```

copies the module with the name mod1 (string) to become a module with the name mod2 (string). Mod1 is taken from the CurrentInputLib* library or from the individual dataset (if not in the library). Mod2 is produced in the first CurrentOutputLib* or (if that is not present) as an individual dataset. So three types of copies are possible:

```
library to library
indiv. dataset to library
library to indiv. dataset.
```

Warnings:

- A copy from a library to itself is prohibited.
- For FaslCopy the full filename has to be specified (including the suffix ".b" which indicates that the object contains binary information in fasl format) in the correct spelling regarding uppercase / lowercase letters (normally lowercase letters only).
- The value of CurrentInputLib* is manipulated by LOAD directly. So this value should be set explicitly in front of any FaslCopy operation.

1.5.7 Deleting and Renaming of Members

A call to the expr function

```
(LibDeleteMember lib memb)
```

deletes the element with the name memb from the currently open library lib. If the library has been opened for INPUT, the element is discarded from the current execution. If the library has been opened for UPDATE or NEW, the reference to the member is removed from the library permanently at LibClose time. Please note the rules for module names mentioned above.

A call to the expr function

```
(LibRenameMember lib oldname newname)
```

renames the member with the name "oldname" to be named by "newname". If the library is open for input only, the renaming is only local to the PSL run. A renaming in a library opened for update or new will be local to the PSL run and permanent after libclose.

1.5.8 Member List

If a library is open the directory is connected to the library id via the property Lib-membs. If printed this directory is understandable although only sorted by history, e.g.

```
(mapc (get 'MATRIXLIB 'Lib-membs) (function Prin2T))
```

With the same technique a total copy from a complete library to another can be taken by using

```
(function (lambda(x) (FaslCopy (car x)))
```

instead. This technique is needed if a library has grown by updating modules in the library; only the living members are copied and unused space is ignored.

1.5.9 Library Files

The open handling for library files corresponds to the handling of the permanent datasets with two exceptions:

Libraries cannot be opened when they are already local, they are always accessed with a generated name.

A new library is saved once when close processing takes place. Otherwise it is closed and COS will extend the allocation automatically if the file has been updated (to the end-of-file).

Note that library files are wordaddressable files and that they must have uppercase names.

1.5.10 COS Operations on Libraries

A library is a word addressable dataset. This has to be taken into account if the dataset is handled by command language. A copy of a library is possible by the following jcl

```
ACCESS,DN=lll,PDN=mmm,ID=PSL.
ASSIGN,DN=lll,U.
COPYU,I=lll,O=nnn,NS.
SAVE,DN=nnn,PDN=ooo,ID=PSL,PAM=R.
```

A library dataset can be processed by FORTRAN wordaddressable io. For distribution purposes a library can be unloaded to an ASCII file by a FORTRAN program PSLLUNL. A FORTRAN program PSLLRST restores a library from an ASCII file. PSL itself is distributed with the kernel as CAL and FORTRAN sources and the libraries NONKERNELLIB and UTLIB in unloaded form.

1.5.11 LISP Implementation of Libraries

The internal operation of LOAD leads in the case of binary files to a call of FASLIN which does the actual loading. FASLIN has been modified to use a FASL-io instead of the former binary io. The FASL-io switches

between the conventional binary io and the library access during the opening of the load file: if the file is member of the CurrentInputLib*, this lib member is taken. The variable CurrentInputLib* itself is triggered by LOAD, when it tests the library membership of a given name due to the load directories. The contacting functions used by LOAD are

(LibP name)	test if name is an open library and set CurrentInputLib* to this name
(LibMembp name)	test if name is member of the CurrentInputLib*

FASLOUT is modified in a quite similar (but simpler) manner.

The libraries are constructed as word addressable files. The first words of these files have the following meaning:

1. address of the directory
2. high address of the dataset (first word to write)
3. magic number in order to test libraryship

The directory has the following structure:

first word: number of entries
following the entries in sequence:
  length of the entry name in bytes minus 1 (PSL conv.)
  entry name as byte string on word boundary (PSL conv.)
  starting address of member (word number in dataset)
  end address of member (last word number in dataset).

During processing of a library an id with the library name as printname is used to contact the library. This id has the properties:

Lib-fid	file id of the lib for io-operations via the word addressable io
Lib-op	operation type of the lib: 0 input 1 update but not yet changed 2 new or update with change (directory to be written at close time)
Lib-Membs	List of members; one element is a three element list with name (string) starting word ending word.

During a current i/o stream to a library the operations are controlled by a file identifier. The identifier is produced during opening of the member and is the sole reference during the io process. It is a list with the following elements:

1. Id "LIB" distinction between lib-io and binary io
2. current position
3. name of the library
4. file id of the library
5. first position of the member
6. last position of the member
7. member name (only in case of write access)

The new member is put to the member list only by member close (FASLEND). An old member reference with the same name is overwritten. An aborted write access to the library does not cause a logical modification (some additional but unused space may be allocated for the file; this will be used by a later successful write operation).

1.6 Calling COS Commands

Several routines have been written to interface with COS. In order to give the user of PSL great flexibility in dataset processing, e.g. a dataset must be acquired before usage or should be saved on a frontend system after usage, or a permanent dataset management command is to be executed while PSL is running.

The following functions can be called:

```
ACCESS, ADJUST, DELETE, MODIFY, SAVE, PERMIT
ACQUIRE, DISPOSE, FETCH, SUBMIT
ASSIGN , RELEASE
```

The parameters for these functions are similar to those of COS Jcl.

The result of the function will be the status code returned from the operation. e.g

```
DISPOSE,DN=MYFILE,DF=TR,TEXT='station-dependent text',NOWAIT.
```

corresponds to

```
(COS-DISPOSE '(DN MYFILE DF TR TEXT "station-dependent text" NOWAIT))
```

or alternatively:

```
(COS-CMD "DISPOSE,DN=MYFILE,DF=TR,TEXT='station-dependent text',NOWAIT.")
```

Parameters for the first form including lowercase letters or dots must be enclosed in string quotes.

With some of the above commands an interrupt AB025 will be caused if the staging command fails. In this case an "expected interrupt" occurs with no dump printed in reprieve processing. Only a short message will be printed.

1.7 Reprieve Mechanism

The PSL/COS reprieve mechanism is designed to keep PSL alive and well as long as possible. There are several causes for an interrupt and various situations in which the interrupt occurs, which have to be treated in different ways:

- (CAR -1) etc
- Interrupt while garbage-collector is running
- Interrupt while FORTRAN-code is running
- Terminal interrupt
- Timelimit in a batch-job
- Attention interrupt while I/O is running
- Abort by dataset staging routines (nasty, because there is an error output-parameter in the subroutine!)

The reprieve-processing is a nonkernel object with connected routines in the kernel (e.g. the reprieve address) and therefore it is extensible without kernel changes.

There are several indicators for reprieve processing:

*emsgp	tells whether an error message should be printed.
*dump	tells whether a register- and centerdump is to be printed
*cft-mode	tells whether or not CFT-code was running
*noreprieve	tells whether reprieve is possible, if non-NIL it is expected to be bound to a string containing reason for no reprieve.
*expected-interrupt	is for staging routines

Only *emsgp, *dump and *noreprieve should be set by the user !

Another problem arises when an interrupt occurs while an I/O operation takes place. In this case the COS IOLIB routine may recognize an CALL OUT OF SEQUENCE error when the system tries to reuse the file. This error may occur with time limit or attention interrupt, it is tried to resolve this problem by a continuation mechanism in reprieve processing, which allows special functions like I/O and garbage collection to get ready before error processing starts working. Any function forcing continuation is responsible for calling function Lisprpv when its operation is complete.

User interface for reprieve processing:

A user exit USER-ERROR-FUNCTION is called after a successful reprieve with one parameter (the abort-code) and preset by a dummy.

Processing is as follows :

If an interrupt occurs , COS gives control to the function COSRPV after filling the exchange package with the contents described in COS reference Manual. This function will decide now whether operation must continue or the LISP routine Lisprpv is called to start recovery. Lisprpv must reside in kernel, it calls nonkernel function LISP-reprieve immediately. Then:

```
*noreprieve is non-NIL : finish printing *noreprieve
*expected-interrupt is non-NIL : continue without dump
                                 reset indicators
*cft-mode  is non-NIL : if batch finish  (this is most
                                  probably a system-error)
                        else continue
timelimit : finish
otherwise : continue (try to)
```

"continue" stands for:

- reloading registers (e.g. reestablish NIL if damaged by CFT-code)
- print a message with abort-code and the abortcode explanation as described in the COS message manual
- try to analyse which function caused the error
- print a dump if *dump is non-NIL
- reset indicators (e.g. *cft-mode)
- set new reprieve address and
- call user-error-function.

After that reprieve processing is done, (THROW '$ERROR$ NIL) is called. All printing is done only if *emsgp is non NIL.

"finish" stands for:

- closing of all open files and
- (CALL ABORT). Abort will abort the job step so that an EXIT jcl statement can be used for processing afterwards. Sorry, we havnt yet found any possibility to suppress the COS backtrace.

1.8 I/O Functions for Word Addressable Files

Most of those functions simply pass arguments to CFT library routines. For error codes returned eventually, see COS Library Reference Manual. An exeption are the open and close routines. Their processing is described above for library files.

(WOpenread <string>) opens file for read , returns fid

(WOpenwrite <string>) opens file for write, returns fid

(WReadWord <fid> <index>) returns word with index <index>
if an error occurs, -1 is returned,
a message is printed.

(WReadBlock <fid> <index> <bufferaddress> <number of words>)
read number of words from file indexed by
<index> to buffer. returns 0 or an error code

(WWriteWord <fid> <index> <wordtowrite>)
writes 1 word to file indexed by <index>.
returns 0 or an errorcode.

(WWriteBlock <fid> <index> <bufferaddress> <numberofwords>)
writes numberofwords from buffer to file
indexed by <index>. returns 0 or an errorcode.

(WWClose <fid>) closes the file (the double W is no misspelling!).

If you are calling SAVESYSTEM with an open wordaddressable file your COS job is damaged afterwards.

1.9 Counting

A special compiler option called COUNTING can be loaded, that modifies the compiler to produce code that will count memory references to fluids and globals and external function calls. This option must be loaded when the compiler is already loaded. Counting uses a memory portion called SYMCNT (of size maxsymbols) as data area. The code produced is very expensive in runtime. The code produced by the compiler using the counting option is of course larger in size than the code without counting option, please regard this when you are doing memory allocation for compiling.

Functions involved with counting:

(allocate-symcnt) allocates symcnt vector
(beyond bndstk, for the curious)

(deallocate-symcnt) deallocates symcnt vector. Be sure that no code with counting option is still running.

(reset-counts) resets the counting fields to zeroes.

(list-counts) produces a list of counted memory references and called functions sorted in descending order. About one page (60 items) will be printed in each list.

Globals involved with counting option:

*counting	flags whether to produce counting code or not
*symcnt-allocated	flags whether memory is already allocated
alreadylisted*	page counts
numbertolist*	page limit

1.10 List of Available Utilities

General PSL utilities (only main entries listed):

```
COMMON                          these three modules belong
CLCOMP                          to an outmoded COMMON LISP
CLTRANS                         compatibility package
DEBUG
FAST-CHARS
FAST-EVECTORS
FAST-INT
FAST-STRINGS
FAST-VECTORS
IF-SYSTEM
INUM
MATHLIB
MERGE-SORT
MINI
BIG
PACKAGE
NSTRUCT
OBJECTS
PP
SLOW-STRING
SLOW-VECTORS
SYS-MACROS
STRINGS
STRINGX
USEFUL
UTIL
WSTACK
PROFILE
WRAPPERS
```

Note: RLISP is not supported. SYSLISP is only a dummy object for compatibility reasons.

COS specific utilities:

```
DISASSEMBLE
PRINT-ABCODE
```

2. Installation Guide

2.1 Distribution Tape

The PSL 3.4 distribution tape for CRAY / COS is written on the tape drive of the IO-Subsystem (800 bpi), because this is (we hope) a way to be independent of various combinations of station softwares and frontend systems. The distribution tape contains 17 files, binaries and update-pl's in Cray internal format.

NOTE: Although in most cases (non EMA machines) the absolute binary will run, we strongly recommend to reassemble the PSL kernel modules (files 12 and 13 using PSLTXT file 10 s.b.) and link them together with the precompiled FORTRAN and CAL modules (file 11 s.b.). For EMA machines this procedure is mandatory, otherwise the system is unable to use memory beyond the 4 M word boundary.

NOTE: A special ownership value is knwon to PSL3.4 as "standard-LISP-ownership". (LISP variable Owner-of-Psl*) All files on the distribution tape should be undumped to this ownership.
Therefore:
Choose your installation's standard-LISP-ownership.
The installation process should be performed using the account data of the standard-LISP-ownership, or the files must be copied to the appropriate ownership. The standard-LISP-ownership is used when the access to a file with job-standard-ownership fails. The actual ownership value must be installed with "savesystem".

The LISP variable cos-version* should be set to 115 (or higher) if you are running COS version 1.15 or higher, defaultvalue is 114.

See Job SAVESYSTEM in PSLINST.

When undumping the tape, please follow the installation procedure:

```
ACQUIRE,DN=FILE0,PDN=PSLINST,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:0:NR,PAM=R.
ACQUIRE,DN=FILE1,PDN=UTLIB,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:1:NR,PAM=R.
ACQUIRE,DN=FILE2,PDN=BAREPSL34,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:2:NR,PAM=R.
ACQUIRE,DN=FILE3,PDN=PSL34MANUAL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:3:NR,PAM=R.
*
* If you want the executable system only, stop here
*
ACQUIRE,DN=FILE4,PDN=NONKERNELLIB,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:4:NR,PAM=R.
*
* LISP , CFT and CAL sources in Update PLs
*
ACQUIRE,DN=FILE5,PDN=NONKERNELPL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:5:NR,PAM=R.
ACQUIRE,DN=FILE6,PDN=UTILITYPL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:6:NR,PAM=R.
ACQUIRE,DN=FILE7,PDN=COMPILERPL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:7:NR,PAM=R.
ACQUIRE,DN=FILE8,PDN=KERNELPL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:8:NR,PAM=R.
ACQUIRE,DN=FILE9,PDN=CFTCALPL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:9:NR,PAM=R.
*
```

```
* binaries
*
ACQUIRE,DN=FILE10,PDN=PSLTXT,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:10:NR,PAM=R.
ACQUIRE,DN=FILE11,PDN=CFTCALBIN,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:11:NR,PAM=R.
*
* the sources of the kernel-modules (CAL)
*
ACQUIRE,DN=FILE12,PDN=KERNL,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:12:NR,PAM=R.
ACQUIRE,DN=FILE13,PDN=KERNLD,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:13:NR,PAM=R.
*
*** THE FOLLOWING 3 FILES SHOULD NOT BE NEEDED !!!!
*  finally the dumped contents of UTLIB and NONKERNELLIB
*  and the program to reinstall them.(hope you never will have to use
*  them)
*
ACQUIRE,DN=FILE14,PDN=UTLIBDMP,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:14:NR,PAM=R.
ACQUIRE,DN=FILE15,PDN=NKLIBDMP,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:15:NR,PAM=R.
ACQUIRE,DN=FILE16,PDN=PSLLRST,ID=PSL,MF=AP,DC=MT,TEXT=PSL34:16:NR,PAM=R.
```

The PSL34 Manual (file 3) is ready for printing with ASCII control characters, 11 inches of paper per page assumed.

Now, after undumping the tape you may test the installation with following data:

```
FILE3.
/EOF                or type on terminal
(LOAD BIG)
(EXPT 2 100)
(QUIT)
```

The job described above tests the access and function of PSL34 and the UTLIB library.

Some messages will be printed:

- Dayfile message "WA-FILE OPENED....." (with COS 1.14 or below)
- A welcome banner " Portable Standard Lisp ..."
- the result of the computation 2**100 (full precision)
- "WA-FILE closed ..."
- and "STOP in COSQUIT"

If any of these messages fail to occur, please check the installation procedure. Hopefully an error message will be printed indicating the type of error.

If, unfortunately, an error-loop occurs, please drop the job TWICE, second break shortly after the first one, because of PSL34s reprieve processing.

If anything looks fine now, please look at job SAVESYSTEM in PSLINST (next chapter) and run it with the appropriate substitution. After that the installation is done.

2.2 Description of the File PSLINST

The file PSLINST is a poor indirect file. It contains several specimens of useful jobs to maintain PSL34. The dataset contains many files separated by a line *** xxx ***, where xxx stands for the main purpose of the job resp. list.

Please fill in the appropriate accounting data into the jobs.

Jobs for rebuilding the kernel , nonkernel and utilities (e.g. the compiler), saving the workspace "savesystem" and lists of Update decks of source files are supplied.

Short Description of Jobs:

The whole bootstrap sequence for a new PSL:

0. Get the sources you intend to change from PL's and read them carefully.
   NOTE Use NS parameter in UPDATE, otherwise the identifiers will cause compilation errors.

1. Save a running compiler and build a cross-compiler by altering the compiler and make a savesystem with the cross-compiler already loaded.

2. Produce a new NONKERNELLIB (specimen job NONKERN).

3. Compile the CFT and CAL modules (Job CFTCAL).

4. Compile the Kernel modules (Job KERNEL).

5. Link the results of step 2 and 3.(Job KERNEL)

6. Run the new kernel. If system comes up, do a savesytem (Job NEWSYSTEM).

7. Check utilities for compiler dependencies. Utilities are sometimes sensative to special compiler constructs. A dependency known is contained in the big utility. If you have to recompile a utility you can use job UTILITY.

8. Good luck.

Of course you need not run the whole sequence to make little changes in any cases. The whole sequence is used when the compiler is changed in some basic features such as alloc-sequence.

2.3 Presetting System Variables and Memory Sizes

The job SAVESYSTEM must be used after the tape has been undumped, because useful presetting can be done, specially variables like owner-of-psl* can be set or memory sizes for heap, bps, stack and bndstk can be adapted to the needs of the installation. Defaults for the file access are ID = PSL and OWN = PSL34 when system is undumped. Change them by (cos-set-own "oooo") resp. (cos-set-id "iiii").
If you are running COS 1.15 or higher, please set the LISP variable COS-version* to 115 or higher

2.4 Lists of Modules

The following chapter is for those users who want to recompile parts of the PSL34 system.

Not all utilities or compiler modules are compiled to UTLIB, mostly because they have not been needed up to now, you will get a list of compiled load modules by (GET 'UTLIB 'LIB-MEMBS). If you changed the name UTLIB, please use your name instead.

Unfortunately, there is no way to keep the bootstrapping jobs frontend independent .

A problem that will arise at recompiling is the naming convention of PSL. The names of PSL load modules are derived from VAX where 15 chars (including lowercase chars) plus an appendix of 3 chars are allowed. Other frontend systems (e.g. the MVS, where this version was build) or UPDATE dont allow so many chars in a filename or membername of an indirect file. So a mapping has to be done from PSL names to membernames (or UpdatePl names). The lists describing this mapping are supplied also.

These lists have the following structure (e.g.):

```
'PASS1DEC'  'pass-1-decls'        needs dskin of other files
    ...
```

this means: the deck PASS1DEC (max 8 chars) must be compiled to pass-1-decls (note the lowercase letters, they are needed!). A suffix ".b" is appended by PSL3.4 automatically!! The compilation of PASS1DEC needs two other files present on CRAY.

If you compile to datasets rather than to libraries, make sure that your computing-center is able to handle dataset names containing lowercase letters and dots.

All decknames are contained in:	ALLCOMPDECKS resp. ALLUTILDECKS
the already useable objects in the above form in:	COMPDECKS resp. UTILDECKS
Nonkernelobjects have all been compiled. A list of them is in	NONKERNDECKS

NOTE Use NS parameter in UPDATE, otherwise the identifiers will cause compilation errors.

3. Implementation Details

3.1 COS Specific Features

Additional to the PSL documentation the following variables and functions are present in Cray PSL and are not described explicitly in other parts of this document:

Switch variables:

*DUMP	T: print octal dump in error case
SHOW-NEW-IDS	T: print names of new interned ids

Other variables:

*BATCH	T if PSL has started noninteractively
COS-Version*	the actual COS version (as integer , default 114)

Dump of memory:

(CENTER-DUMP adr)
dump the memory around location adr

3.2 Mapping of PSL to the Cray X-MP Architecture

The basic element of PSL is the ITEM. An item is a 64 bit quantity or a word. The 64 bits are divided into parts

Bit 1-5	tag
Bit 6-27	gc
Bit 28-64	inf

The tag describes how the inf has to be interpreted. If situated in memory some tags describe how the following words are to be interpreted. The special tags 0 (no bit set) and 31 (all bits set) describe the whole word to be a positive or negative integer value; in these cases tagged information and normal machine representation are compatible, an important fact for efficiency. The tags 1 to 10 say, that inf holds a pointer and give information on the data type pointed to. The tags 23 to 30 are used to describe memory areas and special value references (e.g. identifiers, unbound values).

The gc field is used by the garbage collector. The garbage collector has the task to compact the heap in order to eliminate unused areas. Therefore in a first phase all items in use have to be marked: a special mark is placed into the gc field. In a second phase a relocation takes place: a calculated offset is placed in the gc field of every item in the heap. During the last phase the items are compacted and the gc field is cleared (in the case of negative integers to all ones) again.

Because of the usage of the gc field the size of direct integer values is restricted to 37 bits. Bigger integers are represented by boxed values: a tagged pointer to a two word quantity is held; the two words (in memory) contain a tagged length field and one word with the value itself. In the same manner float values are stored. Vectors and strings are stored with a leading length word too and pairs in memory are two word quantities with two items.

Interned identifiers are represented by a small integer number tagged by 30. The 128 ASCII characters are represented by their tagged numeric equivalent 0 to 127 and the NIL is represented by a tagged 128 (that is in octal representation a 8#200). Due to the tag technique type tests are very fast in PSL: only the tag has to be inspected.

Four cells are associated to each identifier: a pointer to the print name (string), a pointer to the property list, a cell for the value if the identifier is used as non local variable and a cell with a jump instruction to the code for the evaluation of the identifier as function name. PSL uses a shallow binding scheme: A variable value is stored in the value cell directly and by recursive usage the old value is pushed to the binding stack (s. b.).

The total number of identifiers is limited to 8000.

3.3 Memory Layout

The memory contains

- the kernel (assembled and linked basic parts of PSL) including the identifier cells
- the BPS (binary program space) which holds loaded and compiled programs
- the heap as major working area
- the stack
- the binding stack
- the catch stack

The "stack" controls recursion. During function evaluation the living top of the stack (the actual "frame") is interfaced by a section of the T-registers. By alteration of the recursion level this block of T-registers is written to / read from the memory allocated to the stack.

The "binding stack" is used for the recursive redefinition of special (PSL: "fluid" or "global") variables in LAMBDA or PROG lists.

The "catch stack" controls the non local exits CATCH-THROW or ERRORSET-ERROR. It is fixed in size (400 words).

The following memory portions can be altered in size at runtime. They are initialized to the values (sequence as allocated in memory)

BPS	80000	Words
Heap	50000	Words
Stack	5000	Words
Binding stack	500	Words

Note, that an important part of BPS is used by PSL itself: some parts of PSL are implemented as loadable modules and they are preloaded into BPS at installation time of PSL. Additional parts of BPS are used if the compiler or other autoloading features (e.g. like trace) are used or if additional parts of PSL or private applications are loaded explicitly.

On EMA machines it is possible to enlarge heapsize above 4 million words. A special garbage collector is loaded automatically in this case.

3.4 I/O-operations

The PSL3.4 SYSIO uses FORTRAN-calls to do I/O-Operations. Because the buffering is done in "LISP-buffers" except for emergency printouts the foreigncall overhead is acceptable.

There are three types of I/O operations:

- character I/O for textfiles including I/O to a terminal or to $IN/$OUT. This is done via READC/WRITEC-calls (full record mode , see COS Library Reference Manual). For $OUT and terminal-out the line is shifted by 1 blank.

- binary I/O for loadfiles. This is done via READ/WRITE-calls (see COS Library Reference Manual).

- binary I/O for PSL-libraries using WOPEN, WCLOSE, PUTWA and GETWA calls to wordaddressable files (see COS Library Reference Manual). The functions for management of word addressable files are open for users(see above).

3.5 Batch Processing

Because batch jobs are standard at most CRAY/COS installations, batch processing in PSL has been improved for the COS version.

The LISP variable *batch indicates the mode of operation.

The function (batch?) return T if in batch mode and NIL otherwise. As a sideeffect the function cos-waitio is called.

Counting of parens simplifies the search for errors in parenthesis structure (toggle with ON/OFF parens).

3.6 Arithmetic

The arithmetic of PSL is implemented with the objective of high speed execution. Some typical restrictions are involved with that.

3.6.1 Integers

PSL knows three types of integer representations:

INUMS (up to 37 Bit , the whole information residing in the item,
the rest of word must be either all 0 or 1)
FIXNUMS (up to 64 bits, or 1 word)

BIGNUMS (arbitrary length, PSL internal representation)

Of course inums are the easiest and fastest integer type. Because the 37 bit reside in the inf field of the LISP-item, no extra load has to be done.

As you certainly know, there is a curious situation with integers on Cray machines. There is no hardware to do 64 bit integer multiplication

or division; there is a way to multiply integers in address units when they dont exceed the 24 bit range, and you can do integer multiplication and division via floating point units if the operands and the result dont exceed 46 bits (as CFTs FASTMD).

Standard LISP integer arithmetic uses instructions for 46 bit represented numbers. If the low level arithmetic functions are to be used, the 24 bit integer artihmetic is available as well:

function	WTIMES2	uses 24-bit arithmetic
	LONGTIMES2	uses 46-bit arithmetic via float
	WQUOTIENT	uses 24-bit arithmetic
	LONGQUOTIENT	uses 46-bit arithmetic via float
	WREMAINDER	uses LONGTIMES2 and LONGQUOTIENT, because wremainder is merely used in hash function and has to operate 46-bit integers.

All functions above expect inums as arguments, no typechecking is done. These functions should not be invoked by the user directly. This can damage the system badly, if one operand turns out to be not an inum. Generic arithmetic including the big arithmetic uses LONG... operations. No 64 bit integer arithmetic is implemented because of the enormous overhead.

Consequences are the following:

First of all : use generic arithmetic!

Fixnums cannot be multiplied or divided, and therefore not printed or read correctly, if the result or the operands exceeds 46 bits (use: big load-module).

If big arithmetic must be avoided, numeric values exceeding 46 bits must be partitioned by logical operations before they can be manipulated by standard LISP operations. (So does the CRAYASM module). Take care that no "inum" greater than 37 bits gets into heap.

Note: Inums are recognized by the system if the item has an inum tag and not by the size as in generic arithmetic.

3.6.2 Floats

PSL3.4 (CRAY) uses normal 64 bit float arithmetic, double precision is not implemented.

3.6.3 Complex

No complex arithmetic

3.7 Cray Specific Compiler Features

Cray specific compiler features have been build to assist the PSL user, optimize the system resp. to make CFT linkage possible (foreignlink).

3.7.1 LISP Variables in Bregs and Tregs

Beside the stack frame (containing the local variables of a LISP function) which resides in T-Registers T26 to T70 resp T71 to T76, the room left in Tregs and Bregs has been filled up almost completely.

Many LISP variables often used are located in Bregs or Tregs, because there is a much faster access. This is done automatically for FLUIDS if the register allocation is set in the property list of a LISP-id.

The mapping of variables to registers is done by

(put '<LISP-id> 'registername "xxx") where xxx stands for Bnn or Tnn
and
(put '<LISP-id> 'breg mm) resp. (put '<LISP-id> 'treg mm)

where mm is a numerical value in the range of 0 - 63. The actual setting is done in the compiler modules Cmacs3 and Crforms.

and
(flag '(<LISP-id>) '*user-variable)

This last flag is used for security, because many fluid system variables should not be destroyed. This flag is not necessary if *sysLISP is not NIL (not recommended!), because then all fluid to register conversions will take place.

(Please take care that mm=nn, the value of registername is used by CRAYASM the one of breg or treg property by CRAYLAP).

Caution:
The way described above does not ensure that these register values are updated by garbage collector or when portions of memory are moved. There are special flags:

*warray-pointer, *user-variable (for correct setting from interpreted code), and the *known-to-gc property

which must be set in an appropriate way(look at in Crforms).
Any new register allocation needs a recompilation of (at least) the nonkernel objects BINDING, COMPACTING-GC and SYMBOL-VALUES.

Please remember that values residing in a Breg are untagged in the PSL sense. So Bregs are good for variables which are "hard" addresses (e.g. kernel structures) or variables which are always positve integers (e.g. in* or out*). Pointers to warrays are also put into Bregs, but they must be flagged.

3.7.2 Foreignlink

CFT functions can be called from the kernel modules only, because there is no dynamic linkage in COS. Therefore any new FORTRAN call needs a recompilation of kernel modules.

CFT functions are restricted to a maximum of 4 parameters.

CFT functions are called in a quite normal way with Register A6 pointing to the parameter block. This parameter block contains pointers to LISP items. Therefore only integers can be passed directly to a CFT subroutine, other types must be declared as POINTER in CFT code.

The value of a CFT function is the value placed into the first parameter.

Example:

```
(flag '(mycosfnc) 'foreignfunction)

(de myveryspecial (x y) (mycosfnc x y))

(myveryspecial "a string" 7)

% returns the value 15 if :
                        SUBROUTINE MYCOSFNC(IPTR,INUM)
                        POINTER(IPRT,ISTRING)
                        DIMENSION ISTRING(5)
                C
                C inum is 7. iptr points to a block with the length-1
                C of string in the first word and the chars in the
                C second(...) word.
                        ....
                        IPTR=15
                        END
```

Registers A3 and A4 all Bregs and Tregs are saved. Especially the Breg and Treg saving is expensive, but without that you have no chance to recover from an error where CFT code is involved, because various LISP variables reside in Bregs and Tregs.

A little grain of salt: the backtrace information for CFT is not setup correctly, so COS-backtrace will give up when invoked (which should not happen anyway).

3.7.3 Counting

Cray computers with pipeline architecture are slowed down a lot by too many jump instructions (especially if the instruction stack must be loaded). Therefore it is a good optimization effort to do small but often used functions (such as cons, putbyte and byte) as inline functions. (See compiler module opencoded-fns for examples.)

References to memory are very slow in comparison with register operations. So memory references will force the instruction issue unit to wait until operands are ready. A first effort was taken to improve the code by an final pass called lapopt of the compiler, but there is a simple way to put often used variables into registers (see 2.) This can be done for fluid variables but of course not for indirect memory operations such as Car or Cdr of a fluid variable.

In general memory references will slow down the overall performance because "memory is not quiet" and block transfer (in alloc sequence) must wait.

For the recognition of very often called functions and often used fluid variables it is useful to count references automatically. This can be done by the load module COUNTING.

Counting must be loaded after the compiler modules and modifies the compiler to produce counting code. For details see the description of counting. The running system was optimized using results of counting.

3.7.4 cons cmacro

The compiler will generate a call to *cons cmacro from a call to cons function. In this way a call to cons wont produce a *link cmacro and a function calling only cons may be alloc-free. There is a problem with the usage of *cons cmacro because the heap may get exhausted. Then the garbage collector must be invoked (without a *link cmacro) saving the complete environment of the function for continuation after garbage collection. A very special function Cons-reclaim was coded to ensure this. This function is invoked by a *reclaim cmacro (or LISP function !%!%reclaim).

3.7.5 Vector Instructions

A first effort is done towards the usage of vector registers in LISP functions (e.g. the LISP functions list7 - list15). See the special documentation to appear.

3.7.6 Miscellaneous

Several compiler functions and optimizations have been altered to produce a better code for Cray's instruction set. Most of them are contained in PASS3CR compiler module.

The compiler produces calls for functions list1 - list15 instead of list1 - list5. This reduces overhead for longer lists. List1 up to list15 have no external function calls.

3.8 Test Assistance

Some useful functions have been written during the PSL migration. They helped us to analyse some strange situations, for instance in which the LISP I/O is unable to operate. These functions may be important for a PSL user too.

3.8.1 Dump Routines

A dump of (scalar-)register contents and code area is done automatically if an interrupt occurs and the variable *dump is non NIL. This can be invoked by /ATT input in interactive mode. The B and T registers are used to hold values of many often used variables. The names of these variables are found in Crforms in the compilerpl for a check whether any register contents is destroyed. The garbage collector also dumps some heap contents if an illegal item is found and *dump is non NIL.

3.8.2 Emergency I/O

In some situations the LISP I/O is unable to operate, e.g. an illegal item has been produced, or the I/O is not yet initialized. An independent Mini-I/O has been written to do printouts in such situations. This I/O may be called by following functions:

```
console-print-string <string>  :  10 ; prints a string and flushes
                                     the buffer
console-newline <>: 10               ; flushes the buffer

terminalwritechar<ignore char> : char; puts a char into buffer

putoct   <item>   :   0              ; prints any item in octal
                                     ; representation with 22 digits
                                     ; into buffer

putint <number> : 0                  ; prints an integer into buffer
```

Note that this I/O uses independent buffers, so that the printout may be intermixed with LISP printouts.

3.8.3 Load map

In some situations , especially when an interrupt has occured or an undefined function has been called, a function mapping binary addresses to names of LISP functions is useful. This can be invoked by:

```
(map-at <word-address>)
```

which prints a string containing the load module involved and the names of external LISP functions located next to the given word address. Because of internal functions which are unknown at runtime the mapping cant be precise.

Bibliography

/1/ M. L. Griss, A. C. Hearn, A Portable LISP Compiler, Software Practice and Experience, Vol 11, 1981

/2/ The Utah Symbolic Computation Group: The Portable Standard LISP Users Manual. Department of Computer Science, University of Utah, Version 3.2: March 1984

/3/ Guy L. Steele et al.: COMMON LISP: The Language. Digital Press, 1984

/4/ Cray-1 Computer Systems, M Series Mainframe Reference Manual, Cray Research Inc, Mendota Heights, 1983