The
Connection Machine
System

*Lisp Release Notes

Version 5.2
October 1989

These release notes
do not replace those for
Version 5.1

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1989

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1214
(617) 876–1111

Contents

Tables

About These Release Notes

Objectives

The **Lisp Release Notes Version 5.2* are published to inform *Lisp programmers about all new and changed *Lisp features introduced with the Connection Machine System Software Version 5.2

Intended Audience

The reader is assumed to have a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and of *Lisp, as described in the *Lisp documentation for Versions 5.0 and 5.1. The reader is also assumed to have a general understanding of the Connection Machine system.

Revision Information

These release notes are new with *Lisp Version 5.2. They do not replace **Lisp Release Notes* Version 5.1, nor do they replace any other manual in the *Lisp documentation for Versions 5.0 or 5.1.

Organization of These Release Notes

1 **About Version 5.2**
Identifies *Lisp and Version 5.2.

2 **Porting Code to Version 5.2,**
Explains what to do to ensure that 5.1 *Lisp code runs under 5.2.

3 ***Lisp Language Version 5.2**
Describes language features that are new and enhanced in Version 5.2.

4 ***Lisp Compiler Version 5.2**
Describes compiler features that are new and enhanced in Version 5.2.

5 ***Lisp Interpreter Version 5.2**
Describes interpreter features that are new and enhanced in Version 5.2.

6 ***Lisp Simulator Version 5.2**
Describes simulator features that are new and enhanced in Version 5.2.

Related Manuals

- **Lisp Release Notes, Version 5.0*

  The Version 5.0 release notes provide a succinct overview of the many new features introduced in Version 5.0 and of the changes made to *Lisp between the release of Version 4.3 and the release of Version 5.0. These are essential reading.

- **Lisp Release Notes, Version 5.1*

  The Version 5.1 release notes provide a succinct overview of the many new features introduced in Version 5.1 and of the changes made to *Lisp between the release of Version 5.0 and the release of Version 5.1. These are essential reading.

- *Supplement to the *Lisp Reference Manual, Version 5.0*

  This manual updates the **Lisp Reference Manual*, adding descriptions of all features new with the release of *Lisp Version 5.0.

- **Lisp Compiler Guide, Version 5.0*

  This manual describes the current implementation of the *Lisp compiler.

- *Connection Machine Front-End Subsystems*

  The manuals in this volume should be read before the **Lisp Reference Manual*. It explains the configuration of the Connection Machine system and how to access the Connection Machine from a front-end computer.

- *Connection Machine Parallel Instruction Set*

  The **Lisp Reference Manual* explains how to call Paris from *Lisp. Users who wish to do so should also refer to the Paris manual.

- *Common Lisp: The Language*, by Guy L. Steele Jr. Burlington, Mass.: Digital Press, 1984.

  This book defines the de facto industry standard for the Common Lisp language.

Notation Conventions

The notation conventions used in these release notes are the same as those used in all current *Lisp documentation.

Convention	Meaning
boldface	*Lisp language elements, such as keywords, operators, and function names, when they appear embedded in text.
italics	Parameter names and placeholders in function formats.
`typewriter`	Code examples and code fragments.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1214
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 876–1111

For Symbolics users only:

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl–M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

```
To:  bug-connection-machine@think.com
```

Please supplement the automatic report with any further pertinent information.

1 About Version 5.2

The *Lisp language is a parallel extension of Common Lisp for programming the Connection Machine system. Programs using *Lisp typically include both Common Lisp and *Lisp constructs.

Version 5.2 is an incremental *Lisp release. All Version 5.2 CM System Software components, including *Lisp, now support two new CM hardware options: double-precision floating-point accelerators and larger processor memories. In addition, *Lisp Version 5.2 provides enhanced compilation and corrects a number of implementation errors.

1.1 Summary of Enhancements

The following categories of *Lisp enhancements distinguish *Lisp Version 5.2 from previous versions.

- **Double-Precision Floating-Point Accelerators Supported.** Previously, the Connection Machine model CM–2 included one 32-bit floating-point accelerator (fpu) for every 32 physical processors. Now, these may be replaced by 64-bit fpu's. *Lisp Version 5.2 supports this optional hardware upgrade. See the *Paris Release Notes*, Version 5.2, for a discussion of the performance impact of 64-bit fpu's.

- **Larger Processor Memories Supported.** Whereas CM-2 configurations previously included 64K bits of memory per physical processor, an optional hardware upgrade to 256K bits of memory per physical processor is now available. *Lisp Version 5.2 supports this hardware option. See the *Paris Release Notes*, Version 5.2, for a discussion of the performance impact of larger memories.

- ***Lisp Compiler Code Walker.** The *Lisp compiler has been significantly enhanced by the addition of a code walker.

- ***Graphics.** A *Lisp interface to the CM Graphic Programming tools is now available.

- ***Lisp Library.** A library of new, experimental *Lisp features is now available.

- **Reimplemented pref!!.** The **pref!!** macro has been reimplemented to provide more robust execution.
- **Structure pvar Pretty Printing.** A new debugging aid, **ppp-struct**, has been added to *Lisp. This feature pretty-prints structure pvars.

2 Porting Code to Version 5.2

*Lisp code written under Version 5.1 runs under Version 5.2 unchanged with the following exceptions:

- Lucid Common Lisp Version 2.5 is required for VAX front ends.
- The porting instructions described in the **Lisp Release Notes* Version 5.1 apply to Version 5.2 also.
- Obsolete functions (those documented as such) are not guaranteed to work under Version 5.2.

2.1 Lucid Common Lisp Versions

*Lisp currently requires different versions of Lucid Common Lisp on VAX and on Sun-4 front ends.

*Lisp Version 5.2 on a VAX front end requires Lucid Common Lisp 2.5. (*Lisp Version 5.1 on a VAX front end required Lucid Common Lisp Version 2.1.) On Sun-4 front ends, both Version 5.1 and Version 5.2 of *Lisp require Lucid Common Lisp Version 3.0.

The following chart shows which Lucid versions are required by *Lisp versions 5.*x*.:

Table 1. Correspondence of *Lisp and Lucid Common Lisp Versions

CMSS Release Front End	**5.0**	**5.1A**	**5.1**	**5.2A**	**5.2**
Sun-4	2.1	2.1	3.0	3.0	3.0
VAX	2.1	2.1	2.1	2.1	2.5

In each case, the version correspondence applies equally to the *Lisp interpreter, compiler, and simulator.

*Lisp programmers are strongly advised to obtain Lucid Common Lisp documentation appropriate to their front-end environment.

2.1.1 Lucid Common Lisp Version 2.5 on VAX Front Ends

The release notes for Lucid Common Lisp Version 2.5 completely detail how Version 2.5 differs from Version 2.1. The Lucid changes that most affect *Lisp programs are noted below.

(1) **Use of the change-memory-management function discouraged**

The **change-memory-management** function is not recommended by Lucid for Version 2.5. If it is nonetheless used, the **:growth-limit** keyword value should be much less than previously recommended by Thinking Machines Corporation. Failing to reduce the **:growth-limit** value before using **change-memory-management** causes problems with the garbage collection mechanism.

Here is an acceptable **change-memory-management** call used by some Thinking Machines Corporation developers in their **lisp-init.lisp** files:

```
(change-memory-management
   :expand 64
   #+sun :growth-limit #+sun 768
   )
```

(2) **Foreign function interface changed**

The Lucid foreign function interface has changed. Please consult the Lucid Common Lisp Version 2.5 documentation for details.

(3) **Lucid ephemeral garbage collector**

In previous Lucid releases, garbage collection occurred frequently and took significant amounts of time. Lucid Common Lisp Version 2.5 includes an ephemeral garbage collector. Consequently, full garbage collection is neither as frequent nor as noticeable.

3 *Lisp Language Version 5.2

Version 5.2 of the *Lisp language is substantially the same as Version 5.1. Changes that distinguish 5.2 from its predecessor are described below.

3.1 Reimplemented pref!!

The **pref!!** macro has been reimplemented to ensure that it does not cause an out-of-memory condition if the optional *collision–mode* argument is set to **:many–collisions**.

3.2 Structure pvar Pretty Printing

The following dictionary pages document the new **ppp–struct** function:

ppp–struct *[Function]*

Prints the contents of the supplied structure pvar in a readable format.

Arguments

ppp–struct *pvar per–line* **&key :start :end :print–array :stream :width :title**

pvar	Structure pvar. Pvar to print in readable format.
per–line	Positive integer. Number of values to display per line.
:start	Send address of processor at which to start printing. Defaults to 0.
:end	Send address of processor at which to stop printing. Defaults to ***number–of–processors–limit***.
:print–array	Boolean. Determines whether arrays are printed out in full. Defaults to **t**.
:stream	Stream object or **t**. If supplied, output is written to the specified stream. Defaults to **t**, sending output to ***standard–output***.
:width	Integer. Width, in characters, of each value displayed. Defaults to 8 characters.
:title	String or **nil**. Text to display as title line, or **nil** for no title. Defaults to name of *pvar*'s structure type.

Returned Value

nil	Evaluated for side-effect.

Side Effects

The contents of *pvar* from processor *start* up to processor *end* is written to *stream* in a readable format.

Description

This function is new with *Lisp Version 5.2.

The function **ppp–struct** attempts to print out the structure pvar *pvar* in readable format, with processor values for each slot being shown left to right, one line per slot. The number of values displayed per line is determined by *per–line*.

The keyword arguments **:start**, **:end**, **:print–array**, and **:stream** control the amount, format, and destination of the output exactly as with **ppp**.

The argument **:width** determines the printed width of each slot value, and defaults to 8 characters.

The argument **:title** defaults to **t**, which specifies that the title printed out is the name of the ***defstruct** of which *pvar* is an instance. If **:title** is **nil**, no title is printed out. If it is a string, then that string is used as the title.

Examples

```
(*defstruct person
  (ssn 0 :type (unsigned-byte 32))
  (age 0 :type (unsigned-byte 16))
  (height 0.0 :type single-float)
  (weight 0.0 :type single-float)
  )

  (ppp-struct a-person 8 :end 16 :width 10)

*DEFSTRUCT PERSON

SSN:    219101296  545417079  833166928  508389095
945762998  685245194  687147484  442455228
AGE:    43         76         9          96         63
31         59         82
HEIGHT: 0.7566829  6.0384245  6.8458276  2.9526687  6.920122
2.5360777  0.65423644 0.16378379
WEIGHT: 52.873016  11.53174   29.510529  223.5896   244.6509
130.44492  24.180532  214.51915
```

```
SSN:    604959766  822929695  445946453  856011938  68420622
724449217  967664808  640359065
AGE:    27         28         88         68         98
66         61         31
HEIGHT: 2.01059    5.2301087  6.1360407  1.8808416  6.919573
5.686286   5.1784062  4.504147
WEIGHT: 82.76129   200.76877  165.2837   48.37853   154.9278
84.00104   16.700924  232.88974


NIL
```

Notes

Not implemented in F16 simulator

The **ppp-struct** operation is not implemented in the simulator version F16, which corresponds to *Lisp Version 5.2.

3.3 *Lisp Language Restrictions Update

Most previously reported *Lisp language implementation errors and restrictions have been corrected for the release of *Lisp Version 5.2. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* may therefore be discarded.

3.3.1 Known Errors Corrected

The following implementation errors reported in *In Parallel* Vol. II, No. 1, August 1989, are fixed in *Lisp Version 5.2:

grid-from-vp-cube-adr-bug
list-of-active-processors-bug
ppp-css-truncates-proc-addresses
pvar-print-functions-ignored
ranking-zero-breaks
star-defstruct-symbolics-bug

3.4 Known Errors and Restrictions

All known unintentional language restrictions for Version 5.2 *Lisp operation are reported here in alphabetical order by bug report ID. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. III.

ID initial–coldboot–geometry–def–fails

Environment

*Lisp (interpreter and compiler) Version 5.1, Sun OS 4.0

Description

The first ***cold–boot** operation fails if it specifies an *initial–geometry–definition* form. Subsequent calls to ***cold–boot** do not fail.

Reproduce By

```
> (cm:attach)
;;; Loading source file "/cm/configuration/configura-
tion.lisp"
16384
> (in-package '*lisp)
#<Package "*LISP" 1A2542E>
>   (*cold-boot :initial-geometry-definition
         (create-geometry :dimensions '(1024 1024)
                :on-chip-bits '(0 4)
                :off-chip-bits '(10 0)))

>>Error: The symbol CMI::*ALL-GEOMETRIES* has no global val-
ue.

SYMBOL-VALUE:
   Required arg 0 (S): CMI::*ALL-GEOMETRIES*
:C  0: Try evaluating CMI::*ALL-GEOMETRIES* again
:A  1: Abort to Lisp Top Level
-> :a
Abort to Lisp Top Level
Back to Lisp Top Level
```

```
> (*cold-boot)
16384
(128 128)
>   (*cold-boot :initial-geometry-definition
         (create-geometry :dimensions '(1024 1024)
                :on-chip-bits '(0 4)
                :off-chip-bits '(10 0)))

16384
(1024 1024)
>
```

Notice that the second call to ***cold-boot** succeeded.

Workaround

There are two workarounds:

1. Don't use this method of specifying the VP set.
2. Invoke ***cold-boot** twice at the beginning of a session: first, without the *initial-geometry-definition* specified; and then again, with the *initial-geometry-definition* specified.

Status

Outstanding.

ID star-pset-or-pref-struct-bug

Environment

*Lisp (interpreter compiler) Version 5.1; any hardware configuration.

Description

In some cases, ***pset** and **pref!!** do not work when used across VP sets.

Specifically, if any pvar argument to either function is a structure pvar with a variable-length slot whose size depends on the size of the current VP set, then execution will enter the debugger.

Reproduce By

In the following example, slot **a** of the **bugbug** parallel structure varies in size based on the value of ***current-send-address-length*** and therefore causes an error.

```
;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Pack-
age: *LISP -*-
(in-package '*lisp)

(*defstruct bugbug
 (a 0 :type fixnum
      :cm-type (field-pvar *current-send-address-length*))
 (b 0 :type (unsigned-byte 32))
  )


(def-vp-set fred (list *minimum-size-for-vp-set*))
(def-vp-set wilma (list (* 2 *minimum-size-for-vp-set*)))


(defun bug ()
  (*with-vp-set fred
     (*let (dest)
       (declare (type (pvar bugbug) dest))
   (*with-vp-set wilma
     (*let (source)
       (declare (type (pvar bugbug) source))
       (*when (<!! (self-address!!)
                  (!! (the fixnum *minimum-size-for-vp-set*)))
             (*pset :no-collisions source dest (self-address!!)
         )))))))


  (*warm-boot)
NIL
  (bug)

Error: Trying to acess off of the end of field 65536.  The
passed field has a length of 41, and the length passed to this
instruction is 42.

CM:MOVE-ALWAYS
   Arg 0 (CMI::DESTINATION): 589824
   Arg 1 (CMI::SOURCE): 65536
   Arg 2 (LENGTH): 42
s-A, :    Return to Breakpoint ZMACS in Editor Typeout Window 13
s-B:           Editor Top Level
s-C:           Restart process Zmacs Windows 6
```

Workaround

Do not use varying length slots in ***defstruct** forms when defining parallel structures that will be passed across VP sets.

Status

This is a permanent restriction.

4 *Lisp Compiler Version 5.2

Version 5.2 of the *Lisp compiler offers substantial improvements over Version 5.1. The following changes distinguish 5.2 from 5.1:

- There is now a code walker for the *Lisp compiler.
- The *all construct now generates Paris **–always** instructions.
- The **pref!!** macro, when called with a non-simple source expression, compiles if the vp–set argument is either unspecified or given as *current–vp–set*.

These enhancements are further described below, along with a hint about providing correct declarations.

4.1 *Lisp Code Walker

In Version 5.2, the *Lisp compiler has been significantly enhanced by the addition of a code walker. The *Lisp code walker is an extension of the CommonLoops code walker developed at Xerox Palo Alto Research Center. CommonLoops, including its codewalker, is generously made available by Xerox Corporation to the Common Lisp community for the preparation of derivative works.

The code walker can be enabled and disabled by the user and is disabled by default.

4.1.1 Increased Compilation Scope

The code walker is an extension of the *Lisp compiler that "walks" through all the individual forms of a piece of *Lisp code. It records all declarations it encounters and compiles each *Lisp form it finds.

The code walker allows the *Lisp compiler to:

- Find declarations it would otherwise ignore.
- Generate *compiled code for *Lisp expressions that would not otherwise be *compiled.

4.1.1.1 Enhancement of Code Compilation.

As of Version 5.2, the *Lisp compiler has these new capabilities when the code walker is enabled:

- *Lisp declarations are recognized in all locations where Common Lisp allows declaration forms. In particular, the *Lisp compiler can now recognize declarations within **defun**, **let**, and **let*** forms without the need to use the ***locally** construct.

- All properly declared *Lisp forms are compiled, not only those within the scope of a *Lisp macro operator such as ***set**.

4.1.1.2 Enabling the Code Walker

The code walker can be enabled and disabled by the user. It is disabled by default. To enable the code walker, do either of the following:

1. Type

   ```
   (compiler-options :class :all)
   ```

   to display a menu of compiler options. At the bottom of the menu is an item that enables/disables the code walker

2. Set the variable **slc::*use–code–walker*** to **t**. For example,

   ```
   (setq slc::*use-code-walker* t)
   ```

   enables the code walker for all *Lisp code that is compiled. Setting **slc::*use–code–walker*** to **nil** disables the code walker.

4.1.1.3 Simpler Code Now Compiles: An Example

Previously, if one wanted to write a function that *compiled, one would need to write it like this:

```
(defun sum-of-squares!! (x y)
  (*locally          ;;; *locally to declare arguments x and y
     (declare (type single-float-pvar x y))
     (*let (result)  ;;; declaration of result within *let form
       (declare (type single-float-pvar result))
       (*set result (+!! (*!! x x) (*!! y y)))
       result)))
```

With the code walker enabled, the *Lisp compiler now recognizes declarations in all the places Common Lisp permits declarations—without the need for ***locally**. In particular, the *Lisp compiler now recognizes declarations within **defun**, **let**, and **let*** forms. A list of all special forms within which Common Lisp permits declarations may be found in *Common Lisp the Language*, pp. 153–54.

Thus, one can now write the **sum-of-squares** definition as

```
(defun sum-of-squares!! (x y)
  (declare (type single-float-pvar x y))
  (*let (result)
    (declare (type single-float-pvar result))
    (*set result (+!! (*!! x x) (*!! y y)))
    result))
```

In addition, the *Lisp compiler now *compiles all properly declared *Lisp forms, not just those within the scope of a *Lisp macro operator such as ***set**. Because of this change, the **sum-of-squares** definition may be condensed even further, producing

```
(defun sum-of-squares!! (x y)
  (declare (type single-float-pvar x y))
  (+!! (*!! x x) (*!! y y)))   ;;; *let and local variable result
                               ;;;  are no longer needed
```

which *compiles into Paris code just as well as the original function definition.

4.1.1.4 Minor Feature — Automatic Declaration of Loop Indices

A minor additional feature of the code walker is that it automatically declares iteration variables as integers, eliminating the need for separate declaration of these variables. For example,

```
(dotimes (j 100)
  (*set x (*!! x (!! j))))
```

would not previously *compile unless **(!! (the fixnum j))** were used instead of **(!! j)**. With the code walker enabled, special declarations are no longer necessary and this code will *compile.

4.2 Changes to Compilation of *all and pref!!

As of the release of Version 5.1, the *Lisp compiler generates unconditional (**-always**) Paris instructions from ***all** forms if the *Lisp compiler option variable ***use-always-instruction*** is set to **t**. Previously, the ***use-always-instruction*** variable controlled only whether the *Lisp compiler used unconditional instructions for temporary stack operations.

As of the release of Version 5.1, the *Lisp compiler can compile a **pref!!** form that specifies a non-simple *pvar-expression* source argument, if the **:vp-set** keyword argument is either unspecified or given as ***current-vp-set***. (Note: This was originally reported in *In Parallel* Vol. II, No. 1, August 1989, under the ID **star-all-now-compiles-to-always.**

4.3 Declaration Hint

The following hint is reprinted from *In Parallel* Vol. II, No. 1, August 1989.

ID declaration-hint

Environment

*Lisp Version 5.1, 5.2; any hardware configuration.

Description

Never declare the result of a *Lisp function that the *Lisp compiler is documented to handle. At best such a declaration is superfluous and makes the code less readable. At worst, such a declaration confuses the *Lisp compiler and results in inefficient compiler output.

Reproduce By

For example, suppose we declare the result of a **load-byte!!** expression, thus:

```
(*set (the (field-pvar 16) u16)
   (the (field-pvar 16) (load-byte!! x (!! 0) (!! 16))))
```

The *Lisp compiler renders this as:

```
(let* ((slc::stack-field (cm:allocate-stack-field 0))
       (*lisp-i::*temp-pvar-list* *lisp-i::*temp-pvar-list*))
  ;; Move (coerce) source to destination - *set.

  (cm:move (pvar-location u16)
           (pvar-location (load-byte!! x (!! 0) (!! 16))) 16)
  (cm:deallocate-upto-stack-field slc::stack-field)
  nil)
```

Notice that although the *Lisp compiler normally compiles **load-byte!!** forms, this **load-byte!!** function is not compiled.

In this example, the *Lisp compiler must know the type of **x** in order to compile the **load-byte!!**.

Workaround

If, instead, we write:

```
(*set (the (field-pvar 16) u16)
   (load-byte!! (the (field-pvar 32) x) (!! 0) (!! 16)))
```

The *Lisp compiler generates the following:

```
(progn ;; Load con-
stant size byte out of middle of a pvar -
          load-byte!!.
       (cm:move (pvar-location u16) (pvar-location x) 16)
       nil)
```

This is clearly more efficient.

Forms that the *Lisp compiler does not recognize are exceptions to this rule. Declaring the type of an expression that the *Lisp compiler cannot compile is reasonable.

Status

This is a permanent compiler restriction.

4.4 *Lisp Compiler Restrictions Update

Most previously reported *Lisp compiler implementation errors and restrictions have been corrected for the release of *Lisp Version 5.2. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* may therefore be discarded.

4.4.1 Known Errors Corrected

The following compiler implementation errors reported in *In Parallel* Vol. II, No. 1, August 1989, are fixed in *Lisp Version 5.2:

dsf–compiler–bug
reduce–and–spread–high–safety–err
star–pset–var–len–dest

4.4.2 Known Errors and Restrictions

All known unintentional compiler restrictions for Version 5.2 *Lisp operation are reported here in alphabetical order by bug report ID. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. III.

ID ash–bang–bang–compiler–restriction

Environment

*Lisp compiler Version 5.1; any hardware configuration.

Description

The *Lisp compiler requires specification of the length of the *count–pvar* argument to the **ash!!** function. In declarations, do not use non-specific types, such as **fixnum**. Instead, use more specific types.

Reproduce By

For example, the *Lisp compiler will compile the following call to **ash!!** without complaint.

```
(ash!! y (!! (the (unsigned-byte 4) n)))
```

However, only after issuing a warning and only after assuming that a more specific declaration was given, will the *Lisp compiler compile the call below.

```
(ash!! y (!! (the fixnum n)))
```

A call such as the one above signals the following warning:

```
Warning: While compiling (!! (THE FIXNUM N)): Ash!!
returns a result that depends on the size of the
count argument.  Using your declaration, this
expression would take up to 2147483679 bits of
stack space.
```

This warning is signaled because a **fixnum** declaration assumes a 32-byte argument; shifting a pvar by 32 bytes yields a pvar that is at least 32 bytes long, which is too large to represent. The *Lisp compiler assumes a reasonable declaration and compiles the expression. Nonetheless, the programmer should change the code to avoid such warnings.

Workaround

To avoid the warning, change the declaration of the *count-pvar* variable to something smaller, such as **(signed-byte 5)**. Again: Do not use **fixnum**.

Status

This is a permanent compiler restriction.

ID compiler-restrictions-missed-in-5.1-doc

Documentation Error

**Lisp Release Notes* Version 5.1

Description

On page 34 of the **Lisp Release Notes* for Version 5.1, the following additional *Lisp compiler restrictions should have been noted:

dpb!! shares the compiler restrictions placed on **deposit–byte!!**. Specifically, the *value* and *into–value* arguments must be unsigned-byte pvars of definite length. The *position* and *size* arguments must be textually of the form **(!! x)**, where **x** must be an integer or a symbol that evaluates to an integer.

ldb!! shares the compiler restrictions placed on **load–byte!!**. Specifically, the *size–pvar* argument must be a constant such as **(!! x)**, where **x** must be an integer or a symbol that evaluates to an integer.

Status

These are permanent compiler restrictions.

5 *Lisp Simulator Version 5.2

Version F16 of the *Lisp simulator corresponds to *Lisp Version 5.1.

5.1 *Lisp Simulator Restrictions Update

Most previously reported *Lisp simulator implementation errors and restrictions have been corrected for the release of *Lisp Version 5.2. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* may therefore be discarded.

5.1.1 Known Errors Corrected

The following simulator implementation errors reported in *In Parallel* Vol. II, No. 1, August 1989, are fixed in *Lisp Version 5.2:

amap–bang–bang–sim–restrictions
divide–bang–bang–sim–bug
lucid–interpreted–the–sim–bug
missing–star–defuns
rank–bang–bang–sim–bug
star–pset–dest–uninitialized
star–setf–of–pref–sim–bug
vector–funs–sim–bug

5.1.2 Known Errors and Restrictions

All known unintentional simulator restrictions for Version 5.2 *Lisp operation are reported here in alphabetical order by bug report ID. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. III.

ID coerce–bang–bang–array–sim–bug

Environment

*Lisp simulator Version F16, which corresponds to CM System Software Version 5.1; any hardware configuration.

Description

In the simulator, the **coerce!!** function does not work when applied to array pvars.

Reproduce By

```
(coerce!! (!! #(1 1 1) '(vector-pvar single-float 3))
```

Workaround

Compose **amap!!** with a single-argument type conversion function such as **float!!** or **complex!!**. For example, the line of code above may be rewritten as:

```
(amap!! 'float!! (!! #(1 1 1)))
```

Status

Outstanding.

ID star–pset–void–with–array–or–struct–sim–bug

Environment

*Lisp simulator version F16, which corresponds to CM System Software Version 5.1; any hardware configuration.

Description

The ***pset** macro does not work properly if called with a void pvar as the destination and with an array or structure pvar as the source.

Reproduce By

```
> (*defvar foo)
FOO
> (*proclaim '(type (vector-pvar single-float 3) bar))
NIL
> (*defvar bar (!! #(1.0 1.0 1.0)))
BAR
> (*pset :no-collisions bar foo (self-address!!))
>>Error: You cannot *PSET a pvar of type
 (PVAR (ARRAY (DEFINED-FLOAT 23 8) (3)))
 into a pvar of type (PVAR *)
```

Workaround

Define the destination pvar as an array or structure pvar instead of as a void pvar.

Status

Outstanding.

6 *Lisp Library Version 5.2

As of Version 5.2, a new set of *Lisp functions and macros is available in the form of an on-line software library. Please note that all code included in the library is experimental. Users are welcome to make use of the library code at their own risk, with the understanding that some or all of these functions and macros may not be supported in future releases.

6.1 Accessing the *Lisp Library

The *Lisp library code is available in the directory

```
/cm/starlisp/library/f5201/*
```

On-line documentation for the library functions and macros is available in the file

```
/cm/starlisp/library/f5201/documentation.text
```

Ask your systems administrator to help you locate these files at your site.

All functions in the library are defined to autoload on demand. When any one function in a given interface file is autoloaded, the rest of the functions in that interface file are also autoloaded.

6.2 *Lisp Library Contents

The following interface files are included in the *Lisp library directory for Version 5.2:

• AREF32–SHARED	Lookup table interface
• FFT	CMSSL Fast Fourier Transform interface
• MATRIX–MULTIPLY	CMSSL matrix multiplication interface
• FAST–RNG	Fast random number generator
• ROW–AND–COLUMN–MAJOR	Row/column major address interface
• LET–ALIAS	Temporary storage reduction tool
• COLLECTED–MACROS	Useful macros

New and possibly incompatible interfaces for the CMSSL matrix multiplication and FFT routines will become available with the next version of the CMSSL.

7 *Graphics Version 5.2

With the release of Version 5.2, a facility known as *Graphics becomes available. *Graphics is a *Lisp interface to the CM Graphic Programming environment. Documentation for *Graphics can be found in the **Graphics Reference Manual*, included in the volume entitled *Connection Machine Graphics Programming*.