HOTLINE!

Bulletin 5
October 16, 1987

HOTLINE! is published periodically by the Customer Support group of Xerox Artificial Intelligence Systems to assist its customers in using the Xerox Lisp environment. Topics covered include answers to questions that are most frequently asked of Customer Support, suggestions to help you work in the Xerox Artificial Intelligence Environment (XAIE) as well as announcements of known problems that may be encountered.

Feel free to make copies of individual bulletin pages and insert them in the appropriate place(s) in your Interlisp Reference Manual, Lisp Library Modules manual or other relevant manual. The documentation reference at the end of each topic can be used as a filing guide.

For more information on the questions or problems addressed in this or other bulletins please call us toll-free in the Continental United States 1-800-228-5325 (or in California 1-800-824-6449). Customer Support can also be reached via the ArpaNet by sending mail to AISUPPORT.PASA@Xerox.com, or by writing to:

Xerox AIS Customer Support
250 North Halstead Street
P.O. Box7018
Pasadena, CA 91109-7018
M/S 5910-432

In this issue

In response to user requests this issue of *HOTLINE!* answers many of the questions we have received related to Packages in the Lyric Release. The following topics are covered:

- Creating and interning symbols
- Accessing symbols in packages
- Packages and Readtables
- Difference between MAKE-PACKAGE, IN-PACKAGE and DEFPACKAGE
- Exporting symbols using DEFPACKAGE
- Building a file that exports symbols on loading
- Creating and interning symbols
- Package prefix for symbols and their values
- Exporting symbols in name-conflict
- Importing symbols that have name-conflict
- Deleting a package

Terminology

Terminology used in this *HOTLINE!* bulletin:

CLtL	-	Common Lisp: the Language, by Guy Steele, Jr.
AR	-	Action Request, a Xerox problem tracking number (e.g. AR 8321)
IRM	-	Interlisp Reference Manual

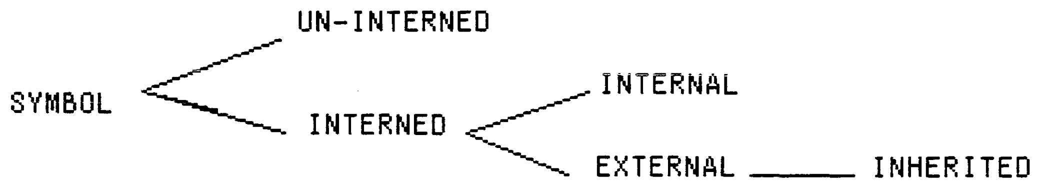
Creating and interning symbols

Release Lyric

Keywords Packages, Symbols, Interned, Creating Symbols

Question What are the different categories of symbols used by the package system? In particular, what is an interned symbol? When and how is a symbol interned?

Answer In the context of packages, symbols can be classified into the following types:



UNINTERNED:

An uninterned symbol is a symbol that is not owned by any package; its package cell does not point to any existing package. Uninterned symbols are generally only used as data. An uninterned symbol with print-name SYM can be created with the make-symbol command in the XCL exec:

```
(make-symbol  "SYM")
```

and this symbol SYM is printed as: #:SYM

Note that if you type the symbol twice, it's two different symbols because #:SYM is not EQ to #:SYM, i.e.,

```
(eq '#:SYM '#:SYM)  returns  NIL.
```

INTERNED:

A symbol that is accessible in a package (say PKG-1) and "owned" by PKG-1 or any other package is said to be interned. A package is said to "own" a symbol if the symbol resides in the package's symbol table. A symbol is said to be "accessible" in a package if it can be referenced without a package qualifier prefix. If a symbol is previously unowned, then the package it is being interned in becomes its owner (home package); but if the symbol was previously owned by another package, that other package continues to own the symbol. There are two types of interned symbols: internal and external. The third type, i.e., inherited symbols, are analogous to external symbols: only the symbols specified as external in a package can be inherited by another package that uses the first package

External: An external symbol is a symbol for public use, declared exportable with:

```
(export 'symbolname)
```

and can be referenced (accessed) in any other package using:

```
owner-package-name:symbolname
```

if the package doing the access doesn't import the symbol, or:

symbolname

if the package doing the access imports "symbolname".

All documented symbols, as well as all Interlisp (IL) symbols, are external.

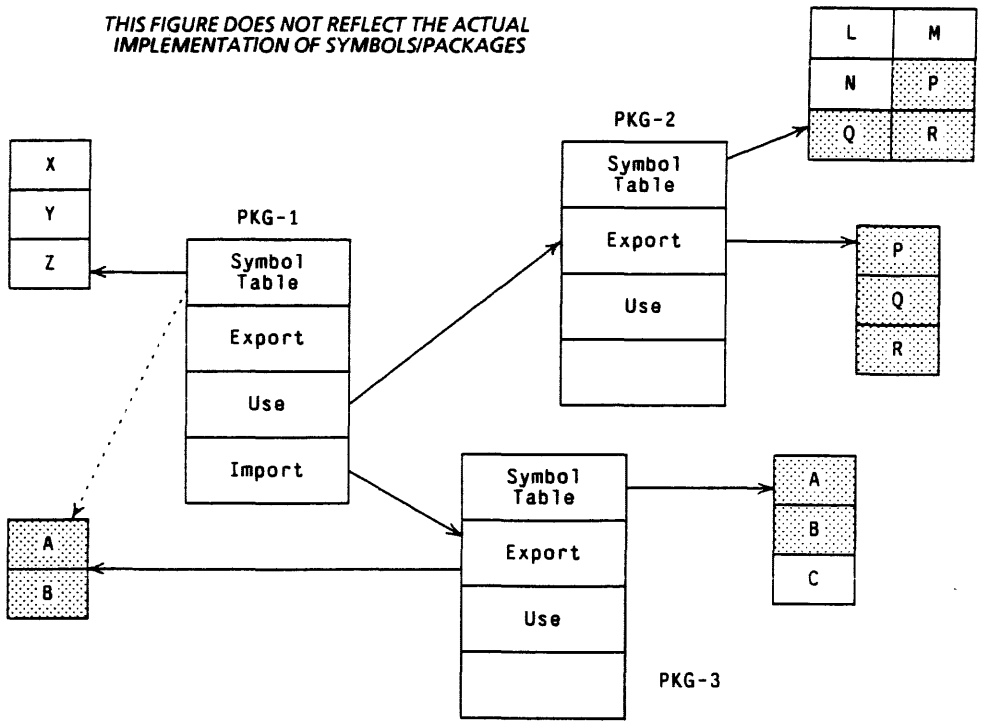
Internal: An internal symbol is a symbol that has not been declared exportable, and is hidden from other packages. However, an internal symbol can be accessed in a package other than the owner package by the owner-package prefix with a double colon as follows:

owner-package-name::symbolname

All system variables are internal (hidden from users).

Inherited: An inherited symbol (say L) is a symbol that is accessible in a package (say PKG-1) by virtue of the fact that PKG-1 uses the package (say PKG-2) that has declared its interned symbol L to be exported. If PKG-2 is deleted (using delete-package) or removed from the use-list of PKG-1 (using unuse-package), the inherited symbol L no longer exists in PKG-1, because inherited symbols (unlike imported symbols) are not interned to the inheriting package

THIS FIGURE DOES NOT REFLECT THE ACTUAL IMPLEMENTATION OF SYMBOLS/PACKAGES



For example, in the diagram above, symbols X, Y, and Z are accessible only in PKG-1, and are owned by package PKG-1. Thus, symbols X, Y, and Z are interned in package PKG-1 only. Similarly, symbols L, M, N, P, Q and R are interned in package PKG-2. However, package PKG-1 uses PKG-2 and therefore the exported symbols P, Q, and R interned in PKG-2 are inherited by PKG-1. Symbols L, M, and N of PKG-2 are only available to PKG-1 as internal symbols of PKG-2. All of the symbols of PKG-2 are unavailable to PKG-1 if PKG-2 is deleted, or PKG-1 unuses PKG-2.

PKG-1 imports the external symbols A and B interned to PKG-3. Thus, A and B are interned in both packages PKG-1 and PKG-3, whereas C is interned in package PKG-3 only. Deleting PKG-3 leaves A and B interned in PKG-1. Symbol C, an internal symbol of PKG-3, is lost forever.

An unprefixed symbol (i.e., a symbol without a package qualifier) is interned in the default (current) package whenever it is encountered by the Lisp Reader, for example when a symbol is typed in, or when a symbol is referenced in any manner (such as INSPECT foo). So anytime we type in or create a symbol, it gets interned to the package we are in. If we want to get rid of this symbol, for example in a case where we may have made a typing error, we must do an unintern.

Note that package names are case-sensitive. Thus, the package "MM1" is different from package "mm1".

References Common Lisp: the Language, by Guy Steele, Jr., pages 168, 172, 176-77.
Xerox Common Lisp Implementation Notes, pages 18-20.

Accessing symbols in packages

Release Lyric

Keywords Packages, Symbol, Access

Question What are the rules for accessing symbols in packages?

Answer For the discussion below, let P2 be the package that owns the symbol SYM, and P1 be the package that is trying to access the symbol SYM from package P2. Then:

Symbols can be accessed in the package in which they exist simply by their name, without any prefix. This is the case if the symbol is interned in the package, by either of these two ways:

- the package owns the symbol, i.e., the package is the symbol's home package, or
- the symbol is exported by the home package P2, and imported by the using package P1.

Symbols can also be accessed by their name, without any prefix, if the symbol is inherited in the package from some other package (by using use-package), i.e., P1 has declared (USE-PACKAGE 'P2).

If the symbol SYM has been exported from P2, but has not been imported into P1, then we must use P2:SYM to access the symbol SYM in package P1.

If the symbol SYM has not been exported from P2, then SYM is an internal symbol of package P2 and we must use P2::SYM to access it in package P1.

These statements are summarized in the following table:

Accessing a symbol SYM owned by package P2 in a package P1

CASE		Relationship of symbol SYM with P1/P2	ACCESS for SYM in P1
P2	P1		
Exports SYM	Uses P2	SYM is an external interned symbol in P2. SYM is inherited in P1.	SYM
Exports SYM	Imports SYM	SYM is an external interned symbol in P2. SYM is interned in P1.	SYM
Exports SYM	Doesn't import SYM	SYM is an external interned symbol in P2. SYM is non-existent in P1.	P2:SYM
Doesn't export SYM	Not Applicable	SYM is an internal interned symbol in P2. SYM is non-existent in P1.	P2::SYM

References Common Lisp: the Language, by Guy Steele, Jr., pages 174-176.
Xerox Common Lisp Implementation Notes, pages 18-20.

Packages and Readtables

Release Lyric

Keyworks Packages, Readtable

Question What are the default packages and readtables available in the Execs?

Background An Exec is simply an extension of the basic read-eval-print loop of the Lisp language. Packages are a mechanism for providing modularity in large systems by permitting multiple name spaces (symbol tables). Each package has its own name space. Readtable is a data structure that is used to control the reader and contains information about the syntax of each character. Packages govern the interpretation of symbols, whereas readtables govern the interpretation of characters.

Answer Each Exec has a local binding of *readtable* and *package* as follows:

Exec	*package*	*readtable*
XCL	XCL-USER	XCL
CL	USER	LISP
IL	INTERLISP	INTERLISP
OLD-IL	INTERLISP	OLD-INTERLISP-T

When typing to the system, you must be aware of the environment in force. The two most important sets of bindings governing the environment are the *package* and *readtable*. The readtable and package for an Exec can be changed without getting a new Exec.

References Xerox Common Lisp Implementation Notes, page 17.
Common Lisp: the Language, by Guy Steele, Jr., pages 183, 360-61.
Xerox Lisp Release Notes - Integration of Languages: File Package, pages 23-35.

Difference between MAKE-PACKAGE, IN-PACKAGE, and DEFPACKAGE

Release Lyric

Keywords Packages, Make-package, In-package, Defpackage

Question What are the differences among MAKE-PACKAGE, IN-PACKAGE, and DEFPACKAGE ?

Answer MAKE-PACKAGE and IN-PACKAGE are standard Common Lisp functions (described in CLtL), whereas DEFPACKAGE is a Xerox Common Lisp function (not part of the CL standard). Thus, it is more appropriate to refer to these as CL:MAKE-PACKAGE, CL:IN-PACKAGE, and XCL:DEFPACKAGE. XCL provides the function DEFPACKAGE to enable easy interaction with the File Manager. Because most of the file-related operations are done in the IL exec, DEFPACKAGE also exists in the Interlisp package (i.e., you can refer to DEFPACKAGE in the IL package without any package prefix).

MAKE-PACKAGE creates and returns a new package with the specified package name. If the package already exists, a correctable error is signalled. MAKE-PACKAGE does not change the default package:

```
make-package package-name &key :nicknames :use
:prefix-name :internal-symbols :external-symbols
:external-only
```

Note that MAKE-PACKAGE does not allow selective import or export of symbols: all this must be done explicitly after the package has been defined. However, if :external-only is set to T (true), then all symbols interned in the package will be exported; all symbols from all package(s) in the :use are imported.

IN-PACKAGE is the function to change the default package:

```
in-package package-name &key :nicknames :use
```

IN-PACKAGE may be placed in a file containing a subsystem that is to be loaded into some package other than the default package. If the package referred to by IN-PACKAGE does not already exist, the function IN-PACKAGE is similar to MAKE-PACKAGE and DEFPACKAGE, except that after the new package is created, *package* is set to it. This binding remains in effect until changed by the user or until the *package* variable reverts to its old value at the completion of a LOAD operation. If the package referred to by IN-PACKAGE already exists, it is assumed that the user is re-loading after making some changes. The existing package is augmented to reflect any new nicknames or new packages in the :USE list, and *package* is then set to this package.

DEFPACKAGE defines a package. If the package does not already exist, DEFPACKAGE creates it; if the package does exist, DEFPACKAGE tries to match its description, producing an error if a match is not found. DEFPACKAGE does not change the default package. However, when a file is loaded, the loader temporarily sets *package* to the package declared by DEFPACKAGE in the DEFINE-FILE-INFO of the file, reverting back to the old value of *package* at the termination of the LOAD.

DEFPACKAGE extends the capabilities of MAKE-PACKAGE by providing four additional keywords for importing and exporting symbols, :shadow, :export , :import , and :shadowing-import. Note that a symbol must be interned in the package doing the export before it can be exported:

```
defpackage package-name &key :nicknames :use :prefix-
name :internal-symbols :external-symbols :external-only
:shadow :export :import :shadowing-import
```

DEFPACKAGE has been provided to enable easy interaction with the File Manager. DEFPACKAGE can be used in a file's IL:MAKEFILE-ENVIRONMENT property to define the package in which the file is to be read and written. For example, typing the following in the IL exec:

```
(PUTPROP 'foo 'MAKEFILE-ENVIRONMENT '(:PACKAGE
(DEFPACKAGE "MYPACKAGE" (:USE "XCL" "USER"))
:READTABLE "XCL "))
```

and then using:

```
(FILES?)
```

to save this property (as well as any other functions and variables) to the variable FOOCOMS. Instead of doing (FILES?), one can edit the FOOCOMS and put:

```
(PROP MAKEFILE-ENVIRONMENT FOO)
```

in the FOOCOMS for saving the MAKEFILE-ENVIRONMENT property.

References Xerox Common Lisp Implementation Notes, pages 18-20, 30-32.
Common Lisp: the Language, by Guy Steele, Jr., page 183.
Xerox Lisp Release Notes - Integration of Languages: File Package, pages 23-35.

Exporting symbols using DEFPACKAGE

Release Lyric

Keywords Packages, Export, Defpackage

Question I can't seem to export symbols using DEFPACKAGE at the top level. Why?

Background DEFPACKAGE defines a package, but does not bind *package* to this defined package. Thus, any symbols declared in the :EXPORT option in DEFPACKAGE are not interned to the package defined by DEFPACKAGE, but to the current package (usually INTERLISP, XCL-USER, or USER) in which DEFPACKAGE is called. In order to do the export, the symbols must be interned to the package doing the export. Therefore, DEFPACKAGE cannot be used directly to do the export, unless we are in the package being defined by DEFPACKAGE.

Answer Most likely, DEFPACKAGE failed because you were not in the correct package. At the top level (in an EXEC window), DEFPACKAGE can be employed to do an EXPORT by setting the *package* variable to the package being defined by DEFPACKAGE. In order to do this binding, you can use an IN-PACKAGE command. As an example, consider the following:

In the XCL package, evaluate:

```
(defpackage "MM1" (:use "LISP" "XCL") (:export  m-sym1
m-sym2))
```

you will get the error message:

```
"These symbols aren't in package MM1; can't export them
from it:
M-SYM1    M-SYM2"
```

even though the package "MM1" already exists. If you evaluate the above defpackage expression when MM1 is the current package, the symbols will be exported:

```
(in-package  "MM1")
(defpackage "MM1" (:use "LISP" "XCL") (:export  m-sym1
m-sym2))
```

returns:

```
"MM1"
```

signalling that the export has been done.

Reference Xerox Common Lisp Implementation Notes, pages 18-20.

Building a file that exports on loading

Release Lyric

Keywords Packages, Export, Defpackage

Question How do I build a file that exports symbols on loading?

Background Assuming you have created a package (say MM1) and defined some functions, you want to build a file so that it will export symbols upon loading in a fresh sysout. In the example below, when the file is loaded, the loader binds *package* to the package "MM1" being declared by DEFPACKAGE in the file's DEFINE-FILE-INFO; at the end of the LOAD, *package* is reset back to its original value. A second DEFPACKAGE statement in the body of the filecoms, relying on the first statement having bound *package* to "MM1", can now do the EXPORT as follows:

```
(DEFPACKAGE "MM1" (:USE "LISP" "XCL") (:EXPORT
M-SYM1   M-SYM2))
```

Instead of the second DEFPACKAGE statement, one can simply use the EXPORT command inside a P statement in the file as follows:

```
(P      (EXPORT '(M-SYM1 M-SYM2)  "MM1")   )
```

Putting the package-name "MM1" in the EXPORT command ensures that the correct symbol is exported.

DEFPACKAGE is preferable because it allows other functionalities to be put in a single statement, preserving consistency in the file and package environments.

Answer The following sequence of steps can be followed in the IL exec in order to build a file that exports symbols on loading:

1. Make sure that the exporting package will exist at load time. Most commonly, a DEFPACKAGE is put in the file's MAKEFILE-ENVIRONMENT property. This is typically done by typing the following in the IL exec:

   ```
   (PUTPROP 'filename 'MAKEFILE-ENVIRONMENT
   '(:PACKAGE (DEFPACKAGE "MM1" (:USE "LISP"
   "XCL")) :READTABLE "XCL"))
   ```

   On the subsequent MAKEFILE (see Step 4), a DEFINE-FILE-INFO expression will be written into the file which will result in the package MM1 being created when the file is loaded in a fresh sysout.

2. Save this property (as well as any other functions and variables that you have created) to the file by doing a (FILES?). Actually all this does is add the symbols to the variable filenamecoms. For example, if in response to (FILES?) you start saving symbols on the file FOO, these are added to the variable FOOCOMS. Call DV filenamecoms (for example, DV

FOOCOMS) so that you can see and verify that the property as well as all other symbols are there.

Instead of (FILES?), you can directly edit the filecoms and put:

```
(PROP  MAKEFILE-ENVIRONMENT  filename)
```

in the filenamecoms for saving the MAKEFILE-ENVIRONMENT property. Note that the coms of a file (i.e., the FOOCOMS for a file FOO) must be in the IL package, though the file may not be.

Note that we are assuming that the package MM1 exists. However, if it doesn't exist, one can create the package MM1 using CL:MAKE-PACKAGE, DEFPACKAGE, or CL:IN-PACKAGE at the top level (in an exec window), or using (MAKEFILE 'filename) in the IL exec.

3. Call the editor (such as SEdit) from the IL exec for editing the filenamecoms. Put the second DEFPACKAGE statement in a P (file) statement in the filenamecoms as follows:

```
(P      (DEFPACKAGE "MM1" (:USE "LISP" "XCL")
(:EXPORT  MM1::M-SYM1  MM1::M-SYM2))  )
```

   Note that the symbols to be exported in the second DEFPACKAGE statement, "M-SYM1" and "M-SYM2", must be typed in the editor window as:

```
MM1::M-SYM1   and    MM1::M-SYM2
```

   respectively. Otherwise "M-SYM1" and "M-SYM2" will be interned to the default value of *package* which will be INTERLISP for the IL exec from which the editor has been called. Look at the title bar of the editor window to see and verify the package you are in.

   Instead of putting the package prefix in the symbols to be exported, one can, alternatively, set the default package to be the package "MM1" by using the SET-PACKAGE command in the SEDIT window. In this case, however, P, DEFPACKAGE, and keywords such as EXPORT, etc. must be preceded by their respective package qualifiers. Either way, the key idea is to intern the symbols to the correct package.

4. In the IL exec, do:

```
(MAKEFILE 'filename)
```

   to create and save the file filename with package MM1 set up for exporting symbols when the file is loaded.

References Xerox Common Lisp Implementation Notes, pages 18-20, 30-32.
Xerox Lisp Release Notes - Integration of Languages: File Package, pages 23-35.

Creating and interning symbols

Release Lyric

Keywords Packages, Symbol, Interned, Creating Symbols

Question By mistake, I typed in CREATEW in the XCL Exec. I then tried to import IL:CREATEW from the IL package into the XCL package, and got an error message which says that "Importing this symbol into package XEROX-COMMON-LISP causes a name conflict". I never really created CREATEW in the XCL exec. Then why did this happen?

Answer Anytime we type in or create a symbol, it gets interned to the package we are in. When CREATEW is typed in in the XCL exec, a symbol XCL-USER:CREATEW is created in the XCL-USER package. Subsequent attempts to import IL:CREATEW will result in a symbol name conflict error, because the erroneously typed-in symbol CREATEW continues to exist in the XCL-USER package.

To get rid of the erroneously typed-in CREATEW and use the IL:CREATEW symbol, first unintern CREATEW from the XCL-USER package by typing in the following in the XCL exec:

```
(UNINTERN  'CREATEW)
```

Then import IL:CREATEW from the IL package into the XCL-USER package, as follows:

```
(IMPORT  'IL:CREATEW)
```

Reference Common Lisp: the Language, by Guy Steele, Jr., pages 168, 172, Questions 1 and 2.

Package prefix for symbols and their values

Release Lyric

Keywords Packages, Access symbols, Access values, Prefix

Question I created a symbol in the IL package. I now want to use it in an XCL exec. How do I strip off the package prefix from my symbol?

Answer Symbols do not have prefixes. Prefixes are used to identify symbols with identical print-names in different packages. Note that such symbols are not EQ, i.e., IL:NAME is not EQ to XCL-USER:NAME.

Whenever a function or a symbol is created in a given package, it is interned to that package. In order to access these symbols/functions across packages, it is necessary to put the package prefix so that the symbol/function is looked up in the correct location. If this is not done, then the default behavior in Common Lisp is to create this symbol/function in the package that is trying to access it (other than the package to which the symbol/function is interned).

Following is an example of a situation that can arise from falsely equating two symbols.

Example In the IL exec type:

```
(SETQ LIST1 '(NAME))
```

Then in the XCL exec type:

```
(setq   a   (make-hash-table))
(setf  (gethash  'name  a)  'TOM)
```

Then in the XCL exec, typing:

```
(gethash  'name  a)       returns TOM.
(car  IL:LIST1)       returns  IL:NAME,
```

whereas:

```
(gethash  (car  IL:LIST1)  a)       returns NIL.
```

This is because the value TOM is associated with the hash key XCL-USER:NAME and not IL:NAME. Note again that IL:NAME is not EQ to XCL-USER:NAME.

Now, if we do the following in the XCL exec window:

```
(setf  (gethash  'il:name  a)  'DUM)        then:
(gethash  (car  IL:LIST1)  a)      returns DUM.
```

Further:

```
(gethash   'IL:NAME   a)      returns DUM, while:
(gethash   'NAME   a)      returns TOM.
```

Reference Common Lisp: the Language, by Guy Steele, Jr., pages 174-176, Question 2.

Exporting symbols in name-conflict

Release Lyric

Keywords Packages, Export, Use-package, Unuse-package, Name-conflicts, Unintern

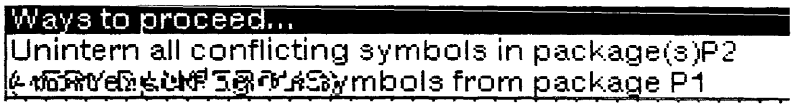
Question How do I export a symbol (say X) from a package (say P1) in a case where it is in name-conflict with another package (say P2) that uses the package P1?

Background If you try to load a module that exports a symbol that is in name conflict with another package you will get a break:

```
In IL:RESOLVE-EXPORT-CONFLICT:
Exporting these symbols from the P1 package:
X
results in name conflicts with package(s):
P2


112(debug)
```

The PROCEED menu in the BREAK window for the conflicting - symbols error under EXPORT comes out to be garbaged, as shown below:



```
Ways to proceed...
Unintern all conflicting symbols in package(s)P2
[illegible]symbols from package P1
```

One of the choices offered, to UNINTERN all the conflicting symbols in P2, doesn't have the desired result when picked: it gets you out of the BREAK, but it does not unintern any symbol and it does not do the export.

Answer In the XCL Exec window, do:

(UNUSE-PACKAGE 'P1)

Now do the export:

(EXPORT 'X)

Then call USE-PACKAGE P1 again:

(USE-PACKAGE 'P1)

When you do the USE-PACKAGE P1, you get the same error, but this time it will be under USE-PACKAGE. Also, the PROCEED menu in the BREAK window is not garbaged, and picking the UNINTERN-FROM-P2 choice works and the USE-PACKAGE succeeds. Notice that the USE-PACKAGE menu has warnings like "VERY DANGEROUS" attached to the UNINTERN options.

Reference AR #9029

Importing symbols that have name-conflict

Release Lyric

Keywords Packages, Import, Name-conflict

Question How do I import symbols that have name-conflict?

Answer Importing a symbol that causes a name conflict will result in a break. In the break window, select PROCEED, which brings up a two-item menu:

> Import symbols with shadowing-import instead
> Abort import into package "package-name"

In this menu, select:

> "Import symbols with shadowing-import instead" to enable the import.

Shadowing-import makes the imported symbol shadow (or hide) the symbol with the same name already present in the package, thereby resolving the name conflict. The symbol being imported is put on the shadowing-symbols list of the package importing the symbol with the shadowing-import command. Because this is a destructive operation on the symbol being shadowed out, it must be used with caution.

Note that selecting OK from the break window does not change the bindings in the computation, and will not give the desired result.

References Xerox Common Lisp Implementation Notes, pages 20, 23, 24.
Common Lisp: the Language, by Guy Steele, Jr., pages 179, 186.

Deleting a package

Release Lyric

Keywords Packages, Unuse-package, Delete-package

Question What happens when I delete a package?

Answer When a package (say p2) is deleted using the XCL:DELETE-PACKAGE command, all symbols interned in p2 are uninterned and then the package structure itself is removed. Further, all packages that use this package p2 unuse it before it is destroyed. For example, in the XCL exec, do the following:

```
(make-package 'p1)
(make-package 'p2)
(make-package 'p3)
```

Then, let packages p1 and p3 use package p2:

```
(use-package 'p1 (find-package 'p2))
(use-package 'p3 (find-package 'p2))
```

Now, delete package p2:

```
(delete-package (find-package 'p2))
```

Then, look at the package-used-by-list for packages p1 and p3:

```
(package-used-by-list (find-package 'p1))     returns NIL
(package-used-by-list (find-package 'p3))     returns NIL
```

implying that p2 has been unused by packages p1 and p3 before being deleted.

If we have a USE "chain" between packages, i.e., p1 uses p2, p2 uses p3, then if we delete package p2:

```
(package-used-by-list (find-package 'p1))     returns NIL
package p2    does not exist anymore.
package p3    exists as before.
```

If we have a circular USE "chain" between packages, i.e., p1 uses p2, p2 uses p3, p3 uses p1, then if we delete package p2:

```
(package-used-by-list (find-package 'p1))     returns NIL
(package-used-by-list (find-package 'p3))     returns  p1
package p2    does not exist anymore.
```

The internal symbols of a deleted package are lost forever. Symbols that are inherited from the deleted package (say p2) into a package (say p1) by virtue of the fact that p1 uses p2 (ie, the external symbols of p2) are also lost. However, the external symbols of p2, if imported into p1, continue to exist as interned symbols in the package p1.

Note that the function XCL:DELETE-PACKAGE is not a part of the Common Lisp standard: it is a Xerox extension to Common Lisp.

Reference Xerox Common Lisp Implementation Notes, page 18.