CHAPTER 11 PACKAGES

Standard Packages

The following standard packages are included in Xerox Lisp:

LISP	contains all symbols (other than keywords) defined in *Common Lisp: the Language*. Some of these symbols are shared with the Interlisp package (by importing them into INTERLISP), in the cases where the semantics of the symbols are identical (e.g., CAR).
KEYWORD	contains all Common Lisp keywords.
SYSTEM	contains system internals.
USER	is the default package in a standard Common Lisp Exec. It uses the LISP package.
INTERLISP	contains (or imports) all Interlisp symbols. All symbols in this package are external, reflecting Interlisp-D's flat symbol name space.
XCL	contains symbols of the Xerox Common Lisp extensions. Many symbols in this package are also shared with Interlisp.
XCL-USER	is the default package in a Xerox Common Lisp Exec. It uses both LISP and XCL; thus, extensions to Common Lisp are accessible in this package. Most users will prefer this package to USER.

Extensions to Standard Packages

`*package*` [*Variable*]

This symbol is bound in each exec.

`xcl:*total-packages-limit*` [*Constant*]

An inclusive limit to the total number of packages in the system. Currently this is 255 but will increase in the next release.

`do-symbols` [*Macro*]

`do-external-symbols` [*Macro*]

`do-all-symbols` [*Macro*]

These macros are as specified in *Common Lisp: the Language*, except note that symbols may be iterated over more than once.

`xcl:do-internal-symbols` [*Macro*]

Maps over only the internal symbols of a package, not those that are external (exported).

`xcl:do-local-symbols` [*Macro*]

Maps over the symbols interned in a package, internal and external (exported) symbols, not bothering to map those that are merely accessible in it (as by inheritance).

`xcl:delete-package` *package* [*Function*]

Uninterns all of the symbols interned in *package* and then removes the package structure itself. All of *package*'s symbols become uninterned and will then print out preceded by "hash colon," e.g., `#:foo`. This should obviously be used with caution.

`make-package` *name* `&key :prefix-name :internal-symbols :external-symbols :external-only` [*Function*]

There are several additional keywords for make-package:

`:prefix-name` *name*

The symbol printer uses *name* to prefix symbols that need to be qualified, instead of the package's full name.

`:internal-symbols` *positive-integer*

The number of internal symbols this package should expect to accommodate.

`:external-symbols` *positive-integer*

The number of external symbols this package should expect to accommodate.

`:external-only` *truth-value*

If this keyword is present the package will have only external symbols; i.e., interning a symbol in this package implicitly exports it.

`rename-package` *package* *new-name* `&optional` *new-nicknames* *prefix-name* [*Function*]

The function `rename-package` has been extended with a second optional argument *prefix-name*, with is the name the symbol printer will use to qualify symbols of this package when needed.

`defpackage` *name* `&rest` *option-clauses* [*NLambda Function*]

Define a package named *name*. If no such package already exists, create it using *option-clauses*. If one does exist, try to make it match the description, or produce an error if that's not possible. Arguments are unevaluated (it is an Interlisp NLambda). Each of *option-clauses* is a list whose car is a keyword from those described below.

This function 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. It is somewhat similar to the Symbolics `defpackage`.

The following option clauses are implemented:

(`:use` *name1 name2* ...)

Causes the package to use the named package(s).

(`:nicknames` *name1 name2* ...)

Adds the nickname(s) to the package.

(`:prefix-name` *name*)

The symbol printer will use *name* to prefix symbols that need to be qualified, rather than the package's full name.

(`:internal-symbols` *positive-integer*)

The number of internal symbols this package should expect to accommodate (only noticed if the package needs to be created).

(`:external-symbols` *positive-integer*)

The number of external-symbols this package should expect to accommodate (only noticed if the package needs to be created).

(`:external-only` *truth-value*)

If this keyword is present, the package will have only external symbols.

(`:shadow` *symbol1 symbol2* ...)

Shadow the given symbol(s) in this package.

(`:export` *symbol1 symbol2* ...)

Export (make external) the given symbol (s) from this package. Note: This option can only be used in a `defpackage` for an already-defined package, because otherwise the arguments to this option clause (internal symbols in the package) can't exist yet. (Of course, you can still export inherited symbols this way, but this is not a very interesting case.)

(`:import` *symbol1 symbol2* ...)

Import (make internal and accessible) the given symbol(s) in this package.

(`:shadowing-import` *symbol1 symbol2* ...)

Import (make internal and accessible) the given symbol(s) in this package, shadowing any conflicts.

Modules

`require` *module-name* `&optional` *pathname* [*Function*]

The implementation-specific way in which Xerox Lisp searches for a *module-name* when no pathname is provided is to first merge *module-name* with `*default-pathname-defaults*` and then with each of the contents of the variable `il:directories`.

Error Conditions Raised by the Package System

There are a number of situations in which the package system will raise error conditions, which can be caught and handled by the user from within the debugger. These situations are described below. For details on how to handle these conditions and invoke these proceed cases, see the error system documentation in Chapter 24 of this manual.

While in the reader:

The conditions listed in this section are all subtypes of the `xcl:read-error` condition.

`xcl:symbol-colon-error` *name* [*Condition*]

Indicates that the reader has found a name with too many colons in it. *name* is a string containing all of the characters of the invalid symbol.

`xcl:escape-colons-proceed` [*Proceed case*]

Returns a symbol made in the current package, with the colons quoted.

`xcl:missing-external-symbol` *name package* [*Condition*]

This indicates that a name, qualified as external in a package, has not been found in a package. *name* is a string. *package* is a package.

`xcl:make-external-proceed` [*Proceed case*]

Creates and returns an external symbol.

`xcl:make-internal-proceed` [*Proceed case*]

Creates and returns an internal symbol.

`xcl:missing-package` *package-name symbol-name* [*Condition*]

This indicates that a package named *package-name*, referred to in a qualified symbol name, has not been found. Both *package-name* and *symbol-name* are strings.

`xcl:new-package-proceed` [*Proceed case*]

Creates a new package named *package-name* and interns the symbol there. The package is created with default attributes. If the symbol was qualified external it is exported from the new package.

`xcl:ugly-symbol-proceed` [*Proceed case*]

Creates a new internal symbol, in the current package, with a name composed of the package name, an appropriate number of colons, and the symbol name.

That is, it creates the symbol that would have resulted had the colon(s) in the name been escaped. This is handy when an old Interlisp symbol like `DECLARE:` has been typed, which should (in a Common Lisp readtable) be typed `DECLARE\:`.

`xcl:read-conflict` *name packages* [*Condition*]

Indicates that the reader compatibility feature has found a name whose package it cannot determine. This is described in detail below under "Koto Reader Compatibility Feature".

Package System Supertype Conditions:

`xcl:package-error` *package* [*Condition*]

This indicates an error has occurred in a call to package code that attempts to alter *package*. It is a subtype of the `error` condition. All the conditions described in this chapter, except for the reader errors listed above, are a subtype of this one. This error will almost never be signaled directly; most package conditions are actually of the type or types described below. The slot *package* is inherited by all subtype conditions of `xcl:package-error`.

`xcl:symbol-conflict` *symbols* [*Condition*]

This condition is a subtype of the `xcl:package-error` condition. It indicates that, during a package system operation, a set of symbol names has been found to conflict (the list of symbols in *symbols*). This error will almost never be signaled directly; most package system conditions are subtypes of this type, since it is the most common error. The slot *symbols* is inherited by all subtype conditions of `xcl:symbol-conflict`.

While calling use-package:

`xcl:use-conflict` *used-package* [*Condition*]

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that during a use-package operation the conflicting *symbols* exported by the *used-package* have names that conflict with symbols already accessible in the *package*. *symbols* (inherited from `xcl:symbol-conflict`) is a list of the symbols.

`xcl:shadow-use-conflicts-proceed` [*Proceed case*]

Shadow conflicting symbols in the "using" package (*package*). This is the the safest way to proceed from this condition, but remember that references to any of the shadowed names will now refer to a local symbol, not the one that you might have been expecting to inherit.

`xcl:unintern-user-proceed` [*Proceed case*]

Unintern conflicting symbols from the "using" package (*package*). This is useful if you have inadvertantly interned the conflicting symbols by typing them to an executive before calling `use-package`. However, unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:unintern-usee-proceed` [*Proceed case*]

Unintern conflicting symbols from the package being used (*used-package*). Unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:abort` [*Proceed case*]

Abort the `use-package` operation.

While calling export:

`xcl:export-conflict` *exported-symbols packages* [*Condition*]

A subtype of the `xcl:symbol-conflict` condition. This condition indicates that exporting *exported-symbols* from *package* results in name conflicts with *symbols* in *packages*.

`xcl:unintern-proceed` [*Proceed case*]

Unintern conflicting symbols in *package*. Unless you are very sure of the use of the symbols being uninterned this operation may make those symbols permanently unavailable. This is a dangerous option; use it with caution.

`xcl:abort` *[Proceed case]*

Abort exporting from *package*.

`xcl:export-missing` *symbols* *[Condition]*

A subtype of the `xcl:package-error` condition. This condition indicates that the *symbols* are not available in *package* to be exported.

`xcl:import-proceed` *[Proceed Case]*

Import these symbols into *package* before exporting them.

`xcl:abort` *[Proceed Case]*

Abort export from *package*.

While calling import:

`xcl:import-conflict` *[Condition]*

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that importing the *symbols* into *package* causes a name conflict with symbols already accessible in *package*.

`xcl:shadowing-import-proceed` *[Proceed case]*

Import *symbols* with `shadowing-import`.

`xcl:abort` *[Proceed case]*

Abort import into *package*.

While calling unintern:

`xcl:unintern-conflict` *symbol* *[Condition]*

This is a subtype of the `xcl:symbol-conflict` condition. It indicates that uninterning *symbol* from *package* causes name conflicts among the symbols on *symbols*.

`xcl:shadowing-import-proceed` *[Proceed case]*

Shadowing-import a new symbol into *package* to hide *symbols*.

`xcl:abort` *[Proceed case]*

Abort unintern of *symbol* from *package*.

Koto Reader Compatibility Feature

For the benefit of Koto users of the CML Library module, the Lyric release contains a "reader compatibility feature" to aid in reading Koto CML files into Lyric. If you do not have any such files, you can ignore this section.

The Koto release did not have an implementation of packages, so the CML module used a syntactic convention in which symbols containing colons were used to denote keywords and those Common Lisp symbols whose names conflicted with Interlisp symbols. Of course, the vast majority of Common Lisp names do not conflict, and those symbols were written with no package prefix. Ordinarily, if you were to load such a file into Lyric, all the symbols would be read as Interlisp symbols, and the colons would be treated as any other alphabetic character, consistent with the syntactic conventions of Interlisp in releases prior to Lyric. For example, the character sequence "CL:UNLESS" would read as the symbol `il:cl\:unless`; the sequence "FIND-PACKAGE" would read as the symbol `il:find-package`.

Enabling the reader compatibility feature causes the reader to attempt to resolve all symbols into the appropriate package. The feature is enabled when

(a) `il:litatom-package-conversion-enabled` (a special variable) is true, and

(b) the read table being used (the value of `*readtable*`) is either `il:filerdtbl` or `il:coderdtbl`.

Condition (b) is met when loading files produced by the File Manager and the compiler prior to Lyric and is (usually) not true for files produced in Lyric. You should only enable the compatibility feature when loading Koto CML files, as it may cause other files to be read incorrectly.

The reader feature handles two cases: strings containing an explicit "package prefix", and unqualified strings that name a symbol in the LISP package. When enabled, the reader follows the following procedure when it encounters a string of characters to be interpreted as a symbol:

1. If the string contains an explicit package prefix, such as a leading colon, or "CL:", the string is interned in the package indicated by the prefix.

2. If the string does not name a symbol in the LISP package, then no conversion is needed—the string is interned in the INTERLISP package.

3. If the string names a symbol in the LISP package and there is not already a symbol by the same name in the INTERLISP package, the reader returns the LISP symbol.

4. At this point, the string names symbols in both LISP and INTERLISP. If the LISP symbol is not an external one, then the conflict is with a private LISP symbol and hence accidental; the reader returns the INTERLISP symbol.

5. If exactly one of the symbols is on the preferred reading list (see below), the reader returns that symbol.

6. Otherwise, there is a conflict that cannot be automatically resolved. This will in general happen for any symbol of Common Lisp for which there happens to already exist an Interlisp symbol that was not "shadowed" in CML.

In case 6, a debugger window appears with the message

Symbols named *name* exist in packages Lisp and Interlisp.

Several proceed cases are available under the "PROCEED" option in the debugger menu. These are:

Return Lisp symbol, make it preferred

This returns the symbol from the Lisp package and also puts it on the global list `xcl:*preferred-reading-symbols*`, removing the Interlisp symbol if it was there. This is useful in cases where you have accidentally interned an uninteresting symbol in Interlisp by typing a name without a CL: qualifier. This usually results in an error, such as undefined function, but in the meantime you have created the symbol in the Interlisp package, making it difficult for the compatibility feature to decide what to do. From the moment this symbol is made preferred, you will no longer receive warnings and it will *always* be read as a Lisp symbol.

Just return Lisp symbol

This is a conservative version of the above choice.

Return Interlisp symbol, make it preferred

In some cases you may want to prefer the reading of an Interlisp symbol to that of a similarly named Common Lisp one. The symbol is made preferred by placing it on the global list `xcl:*preferred-reading-symbols*`, removing the Lisp symbol if it was there. The next time it is encountered, the reader feature will use it instead of the Lisp symbol.

Just return Interlisp symbol

Again, this is a conservative version of the above choice, one which does not make the Interlisp symbol preferred.

Reader Compatibility Feature: Making Symbols Preferred

`xcl:*preferred-reading-symbols*` [*Global variable*]

This global list contains symbols (not namestrings) whose reading is preferred. If both Interlisp and Lisp symbols appear on the list the name will still be considered ambiguous. Be careful about placing symbols on this list. You should be very sure that they will never be referred to on a file in such a way that the other meaning is desired. When a symbol from one package is marked preferred (via a proceed option in the debugger), the other one is removed, if it was present. This list initially contains a set of Interlisp symbols corresponding to Lisp symbols "shadowed" in CML (by symbols beginning with "CL:") or not implemented in CML.

Reader Compatibility Feature: Enabling and Disabling

`il:litatom-package-conversion-enabled` [*Variable*]

Set or bind this flag true to enable the compatibility feature. Note that reader performance drops considerably when the compatibility feature is enabled. This flag should only be turned on while reading files written using the Koto Common Lisp library module.

Reader Compatibility Feature: Format of the Conversion Table

`il:litatom-package-conversion-table` [*Global variable*]

This table is a list of clauses specifying the "package prefixes" to check when reading a symbol while the compatibility feature is enabled. The clauses are searched linearly. Each clause has the form:

(prefix-string exception-list package-name where-keyword)

The initial contents of this table are suitable for converting files produced using the Koto CML library module. Such clauses are:

```
(":" NIL "KEYWORD" :external)

("CL:" ("CL:FLG") "LISP" :external)
```

You need only alter the table if you are trying to convert files that contained additional user "pseudo-packages."

prefix-string is a string which is matched to the first characters of a name. If the name matches the **prefix-string** this clause is "activated."

exception-list is a list of strings. If the name, including its prefix, matches any of these strings it is not converted and the conversion is aborted.

package-name is a string containing the name of the package in which the symbol name (without its prefix) will be interned.

where-keyword is one of the keywords `:internal` or `:external`, indicating whether the symbol is to be interned or interned and immediately exported.

Note that since the clauses are tested sequentially, longer prefixes must go earlier in the list. If, for example, you wanted to convert "CL::" prefixed names to be internal in "LISP" then you would have to place a clause before the one starting "CL:". This avoids the "CL:" clause being activated for symbols named "CL::FOO" and the like.

Reader Compatibility Feature: Conditions

The reader compatibility feature uses the following condition and proceed cases in its interaction:

`xcl:read-conflict` *name packages* [*Condition*]

This condition indicates that the reader compatibility feature (see below) has found a name whose package it cannot determine. It is a subtype of the `xcl:read-error` condition. *name* is a string. *packages* contains a list of the packages in which the name was found, and between whom the reader feature cannot decide.

`xcl:prefer-clsym-proceed` [*Proceed case*]

Return LISP symbol, make it preferred.

`xcl:return-clsym-proceed` [*Proceed case*]

Just return LISP symbol.

`xcl:prefer-ilsym-proceed` [*Proceed case*]

Return INTERLISP symbol, make it preferred.

`xcl:return-ilsym-proceed` [*Proceed case*]

Just return INTERLISP symbol.

Moving Existing Code into a New Package

Now that Xerox Lisp supports Common Lisp packages, users may wish to take advantage of package modularity by moving existing code modules into their own packages. For Common Lisp code being maintained in purely text form, *Common Lisp: the Language* tells you much of what you need to know. However, there are several additional considerations for code maintained by the Xerox Lisp File Manager; this section addresses some of these considerations.

The Lyric release contains no tools for completely automating the conversion to another package, nor does it supply tools for supporting very complex packages. The discussion that follows points out some of the mechanisms that may help for creating relatively simply user packages. It assumes you have a file or set of files produced by the File Manager in Lyric. Files written with the Koto CML module should first be converted to Lyric as described above.

How to specify the makefile-environment

In order to have a file written in your own package, it must have a makefile-environment property, which takes the form of a list `(:readtable` *tbl* `:package` *package*`)`. The *Xerox Lisp Release Notes* on the File Manager discuss how this property is used. The discussion here is confined to the form in which the *package* is described.

If you want to write a file in one of the standard packages, such as XCL-USER or INTERLISP, you need only specify the package name, preferably as a string (make sure it is upper-case). If you want to use your own package, you *must* supply an expression whose evaluation will return the package, creating it if necessary. The expression must not assume that any package, other than the standard ones, already exists; in particular, the expression cannot contain any symbols that are in your new package. It should also be self-contained; e.g., if it calls `in-package`, it must be sure to bind `*package*`, in order not to side-effect whatever code is loading or otherwise using the expression to produce a reader environment. And finally, it should not assume that the file it is on is actually being loaded; makefile environments are examined and evaluated by various system utilities that manipulate files (e.g., `il:loadfns`), not just the loader.

For most packages, the simplest expression to use is `defpackage`. It creates the package if it does not yet exist, and returns the package's name in any case, so it is well suited as a *package* expression. For example, to specify a package that inherits both LISP and XCL (as the pre-supplied XCL-USER package does) and imports the Interlisp window system symbols `createw` and `windowprop`, you could write

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:nicknames "MH")
     (:import il:createw il:windowprop))
```

The major complication arises if you want to export any symbols (any package that presents a programmer's interface surely does). You can't put the exported symbols in the defpackage expression, because the package doesn't yet exist in which to type them. There are two principal ways to do the exporting: export the symbols later (in the body of the file), or write a more complex expression.

In the former case, you write a minimal defpackage expression for the makefile-environment, then write a more complete one in the body of the file (e.g., in a P command, or in an initialization function). The minimal defpackage is responsible for creating enough of the package so that expressions on the file can be read. This means it has to specify inheritance, imported symbols and any shadows. For example, you might write

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:import il:createw il:windowprop))
```

as the minimal expression, then in the body of the file write the "full" expression, which can rely on the package already having been created:

```
(defpackage "MYHACK"
     (:use "LISP" "XCL")
     (:nicknames "MH")
     (:export make-hack-window save-hack))
```

This method requires some discipline on the part of package users, since the package as created by the simple expression lacks external symbols. In this state, forms on the file can still be read correctly (though when printed from any other package context the not yet exported symbols will appear with two colons in their name). However, the package cannot be properly inherited by any other package, since references from such a package to the not yet exported symbols will instead create internal symbols in the other package, which will (a) be the wrong symbols and (b) create a package conflict when the full package definition is evaluated. Thus, users of the package *must* ensure that the file containing the full package definition is loaded before attempting to use-package it or refer to its symbols.

The alternative is to write a single very careful expression to define the whole package, e.g.,

```
(let (*package*)
   (in-package "MYHACK" "MH" '("LISP" "XCL"))
   (import '(il:createw il:windowprop))
   (export (mapcar #'intern
           '("MAKE-HACK-WINDOW" "SAVE-HACK"))))
```

If you have a very complex package, or one that is used on many files, it may be preferable just to create a file whose sole purpose is to define the package, then require that file. For example, the package expression might simply be

```
(progn (require "MYHACKDEFS") "MYHACK")
```

Changing the Package of Existing Code

Once you've decided how to define the package, you still have to arrange for the symbols currently in some old package to be moved into your new package. Much of this task can be done by specifying an explicit package to the loader, either as the :package keyword to cl:load, or the fourth argument to il:load. The package you specify overrides whatever package is specified in the file's makefile environment.

In order for this to work, the new package must have fundamentally the same inheritance structure as the package in which the file was written. For example, if the file was written in the XCL-USER package, your new package must inherit LISP and XCL. If the file was written in the INTERLISP package, your new package must inherit INTERLISP (but read the cautions below). When you load the file, the reader will then do the "right" thing whenever it encounters a symbol with no package qualifier—if the symbol was inherited by the old package, it will also be inherited by the new package, so the exact same symbol is read; if the symbol was local to the old package, it will not be inherited, but will be read as a local symbol in the new package.

After loading the file and doing whatever touchups seem appropriate (e.g., there may be local symbols in the old package that really should have been references to the old package), give the file a new makefile-environment property as described above, then call `il:makefile` to write out the new file.

Changing Package Inheritance

It may be the case that you want your new package to have a different inheritance structure than the old. For example, you have a file written in the INTERLISP package that you want to move into a Common Lisp package. In this case, you should temporarily define your new package to have the same inheritance as the old package, and load the file as above. At this point, all the important symbols have been read correctly. Then change the new package's inheritance structure to be as desired, for example:

```
(cl:in-package "RAPT")
(cl:unuse-package "INTERLISP")
(cl:use-package '("LISP" "XCL"))
```

At this point, your new package is defined the way you want, but you may still have unexpected references to other packages, typically in the form of lexical variables in functions. For example, in the case of moving from INTERLISP to a non-INTERLISP package, many of your module's lexical variables happened to coincide with symbols already extant in INTERLISP, so were still read as INTERLISP symbols. If you view the definition of a function in your new module, you may see such things as

```
(let ((il:x (car il:top))
      il:a il:b)
      ...)
```

You'll likely want to rename those variables to be locals in your new package. It is fairly easy to write an SEdit mutator function that searches for symbols not accessible in the current package and replaces them (with user approval) with symbols of the same name interned in the current package.

Caution about referring to other packages

If you use symbols from other packages (other than the standard ones), you must, of course, make sure that the other package is defined. There are some subtleties here when the File Manager is involved.

If you want your package to inherit another non-standard package, you must ensure that the module defining the other package is loaded before defining your package. For example, your package expression might look like:

```
(progn (require "MYHACKDEFS")
       (defpackage "MOREHAX"
           (:use "LISP" "MYHACK")))
```

Similarly, if you simply want to refer to symbols of another package, you must ensure that its module is loaded first. The same procedure is recommended, even if you are not defining your own package. For example, on an Interlisp file, you might give as package expression:

```
(progn (require "MYHACKDEFS")
       "INTERLISP")
```

You might be tempted to do this instead by including one or more `files` commands in the file's coms, e.g., the command

```
  (il:files 'myhackdefs).
```

However, there are two problems with this method. You can't refer to any of the variables in the coms, for example,

```
  (vars (myhack:default-size 37)
        (myhack:default-speed :fast))
```

because at the time the coms expression itself is read, the `files` command contained in it has not yet been executed, so you can get a missing package error when the reader encounters `myhack:default-size`.

Even if you solve that problem, for example by hiding all the variable settings in an initialization function, system utilities that attempt to read only part of the file (e.g., `il:loadfns`) may fail when they encounter the other symbols. The utility will have read the makefile environment (which is why the `require` works there), but will not necessarily have read, much less evaluated, the commands in the body of the file that cause the other file to be loaded.

Special considerations for the IL package

It is perfectly permissible to define a package that inherits from the INTERLISP package. However, when you do so, you must keep in mind a couple of things that are special about it: INTERLISP is external only, and the INTERLISP package's external symbols (i.e., the entire package) are not all defined in the standard sysout—loading Library and LispUsers modules generally adds new symbols to the INTERLISP package. Following are some of the pitfalls to watch for.

The File Manager currently requires that certain symbols be in the INTERLISP package, regardless of the package of the file's contents: the symbol naming the file's coms (whose value is a list describing the file's contents), and the file's rootname (whose property list includes the file's makefile environment, file type and other File Manager properties). However, INTERLISP is external only, so those symbols are automatically inherited by any package using INTERLISP. Thus, for example, when the File Manager writes the symbol `il:foocoms` on the file `foo`, whose environment is the package `bar` inheriting `il`, the printer does not qualify the symbol with its package name, printing simply `"foocoms"`. If you subsequently load `foo` into a standard sysout, the reader encounters the token `"foocoms"` and, since `il:foocoms` does not exist, reads it as `bar::foocoms`.

To avoid this problem, you must ensure that the symbols `il:foocoms` and `il:foo` exist when the file is loaded. The simplest way is to include them in the makefile environment's package expression:

```
(progn '(il:foocoms il:foo)
       (defpackage "BAR"
            (:use "INTERLISP")))
```

For essentially the same reason, you must be careful that your file loads in advance any modules that define INTERLISP symbols it needs to refer to. Otherwise, since those symbols are external in INTERLISP, references to them are not qualified, and thus will be read instead as new internal symbols in your package. See the discussion above about referring to symbols from other packages.

When you follow the procedure outlined above for loading an Interlisp file into an Interlisp-inheriting package, be sure to load the file into a pristine sysout, or at least one in which your file has never been read. Otherwise, all the symbols on the file will already have been interned in INTERLISP, and thus would be read as the same INTERLISP symbols, rather than symbols of your new package.

[This page intentionally left blank]