$LispWorks^*\ for\ UNIX\ and\ the\ Windows^*\ Operating\ System$

Delivery User Guide

Version 4.1



Copyright and Trademarks

LispWorks Delivery User Guide

Version 4.1

November 1998

Part number: 3LBDT2K15LJ

Copyright © 1994-1998 by Harlequin Group plc.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks is a registered trademark of Harlequin Group plc. Harlequin, Delivery, Transducer/PC, The Authoring Book, ClassWorks, and KnowledgeWorks are all trademarks of Harlequin Group plc.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.

The code for walker.lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Texas Instruments Incorporated, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 Texas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

US Government Use

The LispWorks Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be 'unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800 fax +44 1223 873 873 North America:

Harlequin Incorporated One Cambridge Center Cambridge, MA 02142 USA telephone +1 617 374 2400

fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd. Level 12 12 Moore Street Canberra, ACT 2601 Australia

telephone +61 2 6206 5522 fax +61 2 6206 5525

Contents

Introduction 1

1

3

	What does Delivery do? 1	
	What do you get with Delivery? 2	
	Conventions and terminology used in this manual	2
	A breakdown of the delivery process 2 Runtime licensing on UNIX 5	
2	A Short Delivery Example 9	
	Developing the program 9	

Basic considerations when coding for delivery 13 Efficiency considerations when coding for delivery 14

4 Delivering your Application 19

Writing Code Suitable for Delivery

The delivery function: DELIVER 19
Using the delivery tools effectively 20
How to deliver a standalone version of your application 21
How to deliver a smaller and faster application 22
How Delivery makes an image smaller 24
Additional steps after delivery level 5 25

5 Keywords to the Delivery Function 27

Topic-based list of DELIVER keywords 27 Alphabetical list of DELIVER keywords 33

6 Delivery and Internal Systems 65

Delivery and CLOS 65
Editors for delivered applications 68
Error handling in delivered applications 70
Foreign Language Interface templates. 74
Modules 75
Symbol and package issues during delivery 76
Throwing symbols and packages out of the application 77
Keeping packages and symbols in the application 79
Coping with INTERN and FIND-SYMBOL at runtime 80

7 Troubleshooting 83

Debugging errors in the delivery image 83
Problems with undefined functions or variables 84
Failure to find a class 84
Possible explanations for a frozen image 85
Errors when finalizing classes 85
Warnings about combinations and templates 85
VALID TYPE SPECIFIER errors 86
Stack frames with the name NIL in simple backtraces 86
Blank or obscure lines in simple backtraces 86
NIL IS NOT OF TYPE HASH-TABLE errors 86
Trying to construct a constructor 87
Reducing the size of the delivered application 87
Interrogate-Symbols 87

8 User Actions in Delivery 91

Generat strategy for reducing the image size 91 User interface to the delivery process 92

9 Delivering CAPI Othello 99

Preparing for delivery 99
Delivering a standalone image 103

Making a smaller delivered image 101

Index 103

1

Introduction

1.1 What does Delivery do?

Delivery allows you to take applications developed in LispWorks, and turn them into smaller, standalone applications. This process is called *delivery*.

The principle behind application delivery is quite simple: an application does not often use everything in the LispWorks development environment when it is running, so there is no need for those unused parts of LispWorks to be in the image. Delivery can discard the unnecessary code and create a single executable image file that contains just what is needed to run the application.

Because the delivered application is smaller, it can reduce virtual memory paging and thereby run faster than it did under LispWorks. Delivery can also actively speed code up by, for example, converting single-method generic functions into ordinary functions. Packing it all into a single executable file means it is simple to start up and can be run without the need for a copy of LispWorks.

You can use Delivery to deliver commercial applications. In fact, Harlequin is using Delivery itself in successful commercial products.

1.2 What do you get with Delivery?

Delivery consists of an extended routine that is called once all the code that your application requires has been loaded in to LispWorks.

To deliver your application, you run giving it an init file (using the command line option -init) which does all the required preparations (normally just loading the application) then call the function deliver.

1.2.1 Programming libraries and facility support code

LispWorks also provides sets of programming libraries and code supporting various other facilities that you may want to use in your application. Some of these facilities are available in the basic LispWorks image, while others are provided as modules and need to be loaded explicitly using require.

See the *LispWorks User Guide* for further details.

1.3 Conventions and terminology used in this manual

This section discusses the conventions and terminology that are used throughout this manual.

1.3.1 Common Lisp reference text

The Common Lisp reference text for Delivery and LispWorks is the ANSI Common Lisp standard. An HTML version of this standard is installed with LispWorks and can be viewed by choosing Help > Manuals from the LispWorks podium and selecting "ANSI Common Lisp Standard". This is referred to as "the ANSI standard" throughout.

1.4 A breakdown of the delivery process

The process of developing and delivering a LispWorks application can typically be broken down as follows:

- 1. Develop and fully compile your application.
- 2. Load the application into the LispWorks image and deliver a standalone image.

- **3.** If the delivered version of the image is broken, go back to step 2 and adjust the delivery parameters.
- **4.** If performance problems remain, go back to step 1 and refine your code.

1.4.1 Developing your application

Develop your application using LispWorks. Applications may be developed in pure Common Lisp, or if you wish to build a graphical user interface (GUI) into your final application, also using CAPI and Graphics Ports (GP) or CLIM.

Application development is covered in detail in Chapter 3, "Writing Code Suitable for Delivery". If you use CLOS in your application, you should also read Chapter 6, "Delivery and Internal Systems".

1.4.2 Managing and compiling your application

You can use the defsystem macro to help organize your sources during development and use functions such as load-system and compile-system to work with them as a whole.

1.4.3 Debugging, profiling and tuning facilities

You may discover performance bottlenecks in your application, before or after delivery. LispWorks provides tools to help eliminate these sorts of problems. A profiler is available in LispWorks, in order to help you make critical code more efficient.

You can also tune the behavior of the garbage collector. See the *LispWorks User's Guide* for details.

There is a TTY-based debugger available to help debug applications broken by severe delivery parameters. You can deliver this debugger in the application so that you can debug it on-line if something goes wrong.

See the LispWorks User Guide for more information about these facilities.

1.4.4 Delivering your compiled application

Once your application is ready, you can deliver it by loading it and then calling deliver. Note that this has to be done in a script, as described in "Delivering the program" on page 10.

You can also make LispWorks throw unused code out of the image at this point, in order to reduce the image size and thereby improve performance.

There is usually some trial-and-error work involved in delivering an application. You will almost certainly need to attempt delivery several times in order to find the best set of delivery parameters. This trial-and-error work is necessary because it is not possible for Delivery to work out for itself precisely what can and cannot be thrown out of an application. Because of this, you should allot time to delivery itself when planning your application's development.

A good set of delivery parameters should not be too lenient, leaving too much unused code in the delivered application. Nor should it be too severe, throwing necessary code out and thereby breaking the delivered application.

Delivery is covered in Chapter 4, "Delivering your Application".

Chapter 5, "Keywords to the Delivery Function", describes the keywords you can pass to the delivery function, deliver, that permit fine control over the delivery process.

1.4.5 Security issues

Runtime files that are created using Delivery with the LispWorks for the Windows Operating System Professional Edition do not require a runtime license. Files created with the Enterprise Edition require a LispWorks runtime license.

Runtime files generated by UNIX editions of LispWorks also require a Lisp-Works runtime license. See Section 1.5 Runtime licensing on UNIX for more information.

1.4.6 Libraries

You need to load all the libraries that your application needs into the LispWorks image before attempting to deliver your application. If you fail to do this, the loader must be retained within your delivered application, and all the required library files must be distributed as additional fasl files that need to be loaded when your application is started.

1.4.7 Error handling

Delivered applications can deal with errors using the Common Lisp Condition System and error handling facilities if so desired. But if you cannot keep the full Common Lisp Condition System because it is too large, you can still use some basic facilities provided for handling errors.

See Chapter 9, "Error Handling in Delivered Applications" for more details.

1.4.8 Troubleshooting

Chapter 7, "Troubleshooting", presents a number of explanations and workarounds for problems you might have when delivering your application.

1.4.9 Examples

There are a number of examples in the manual which help to illustrate the delivery process.

Chapter 2, "A Short Delivery Example", shows how to deliver a very small application.

Chapter 9, "Delivering CAPI Othello", shows how an example CAPI program can be delivered.

1.5 Runtime licensing on UNIX

1.5.1 Protection of the delivery product on UNIX

[UNIX only] When you start up LispWorks and call (require "delivery"), a check is made that you are licensed to run LispWorks Delivery. If this check fails, the require does not succeed.

1.5.2 Protection of the delivered image on UNIX

[UNIX only] In general, the delivered application is also protected by the keyfile and network licensing mechanism. Unless action is taken to retarget the image, the end-users of your application will require a LispWorks Delivery key (but no other key).

Retargeting the image means: substituting the image's requirement for a Delivery key with the requirement for a run-time key. This substitution is controlled by the product code which LispWorks Support at Harlequin will supply to your organization. (See Reporting bugs in the LispWorks Release Installation Notes for information on contacting LispWorks Support.) You should use the same code for retargetting all of your products. You may wish to make your own security arrangements in additions to those required by Harlequin.

Unless you have made arrangements to the contrary, run-time licenses will be generated by the LispWorks Support desk. Run-time licenses will be issued only to you (the application developer) and not to the end-user. We will need to know the machine identifier of the host target machine in the usual way. Note that undated run-time keys are only transferrable from one machine to another upon payment of an administration charge.

All Harlequin keys are specific to the major version of LispWorks for which they are issued. The current release is Release 4. If you re-issue your application to your end-users and base it on a different release of LispWorks, then all existing keys will need replacement. This re-issue of keys for existing platforms will not attract the above administration fee.

While you are working on the delivery of your application there is no need to retarget it as you can run trial versions with your Delivery keys.

1.5.3 Unprotected run-time applications on UNIX

[UNIX only] It is possible to remove all keyfile protection from the delivered application by specifying :product-code :none. If you do this, a check is made during the delivery process to ensure that you have in addition to a LispWorks Delivery key, a key for LispWorks Delivery PLUS. If you do not have this key then your image will exit immediately when the check fails.

Therefore you should only specify :none as your product code if you have made a prior arrangement with Harlequin to do so.

You may wish to make your own security arrangements or you may choose to leave the run-time image totally unprotected. Although an unprotected runtime application will not require any keys (even for any layered products which were loaded into it before delivery), it may still be subject to time-expiration.

1.5.4 Expiration of unprotected run-time applications on UNIX

[UNIX only] If any of the keys required by the delivery image (whether upon startup, for loading layered products, or for authorising the product code :none) are dated, then the earliest expiration date of all such keys will be hardwired into the run-time application when it is delivered. When you obtain undated keys for any Harlequin product, it is therefore advisable to either delete or comment out any corresponding dated keys from that keyfile.

1 Introduction

A Short Delivery Example

This chapter presents a simple example of Delivery in use. It shows a small, pre-written program being delivered.

There are usually four stages to application delivery: coding, compiling, delivering, and debugging. The example is broken up into these stages and the discussion in each case points to more detailed material later in the manual.

If you would like to try this example delivery out while following the text, you can find a copy of the program in the Delivery distribution. To locate the pathname of the file, evaluate the following form in a listener:

```
(sys:example-file "delivery/capi/hello/hello-world")
```

2.1 Developing the program

The program we use in the example is:

```
(in-package "CL-USER")
(defun hello-world ()
  (capi:display-message "Hello World!"))
```

Put this in a file, compile it and test it by calling (hello-world).

2.2 Delivering the program

Having compiled the program, the next step is to attempt delivery using the compiled file.

Programs are delivered with the function deliver. This function takes three mandatory arguments. There are also many optional keyword arguments to help Delivery make the smallest image possible.

You can read more about this function in Chapter 4, "Delivering your Application". Chapter 5, "Keywords to the Delivery Function", describes all the optional keywords available.

In this example, we do not use any of them, but just provide the mandatory arguments. They are:

- The name of a startup *function*. This is the first function called when the application is run.
- A *pathname* saying where to put the delivered image.
- A delivery level.

This is an integer in the range 0 to 5. It controls how much work is done to make the image smaller during delivery. At level 0, little effort is put into making a smaller image, while at level 5 a variety of strategies are employed.

Continuing with the example:

 Write a delivery script file (deliver.lisp) that loads the program and then call deliver:

```
(load "hello")
(deliver 'hello-world "hello" 5 :interface :capi)
(quit)
```

Run the lisp image using the script as an init file. For example, on Windows open a DOS window and type:

```
lispworks-4100.exe -init deliver.lisp
On UNIX, type the following into a shell:
unix% lispworks -init deliver.lisp
```

- If you want to see the output, you can redirect the output with > to a file or use |, if it works on your system.
- 3. Run the application, which is saved in hello on UNIX and hello.exe on the PC.

2.2.1 Further example

There is another more detailed example at the end of the manual. This is in Chapter 9, and shows how to deliver a small CAPI application. The application is an implementation of the board game Othello.

Writing Code Suitable for Delivery

How successfully you can deliver your application depends to a large extent upon how you wrote it in the first place. Delivery reduces the size of some symbols and constructs more than others, so a knowledge of what sort of code leads to the best delivered images is useful.

This chapter explains what sorts of considerations you might make when coding your application.

3.1 Basic considerations when coding for delivery

The main consideration to make when developing an application that is to be delivered is efficiency.

Efficiency Does one implementation technique tend to produce

smaller and/or faster delivered images than another?

Can you avoid using large modules?

This extra consideration probably means that it takes longer to develop the application in the first place. But the time is usually well spent: choosing the right techniques and facilities at the development stage avoids costly rewrites after delivery reveals that the application code as it stands cannot be delivered within the required size.

3.2 Efficiency considerations when coding for delivery

There are numerous efficiency considerations when coding for delivery. They are detailed below.

3.2.1 Use of modules

Can you avoid using a large module and still get the functionality you need? modules are saved in the image, and even after Delivery has gone through them to throw things out, they may still have a noticeable effect on the size of the delivered image. The fewer modules you use, the smaller the delivered size of your application.

Warning: Some modules are built on top of others. If you load such a module into the image the others are loaded too. Pay close attention to these "hidden" contributions to image size by following the loader messages in the Listener.

3.2.2 Loading code at runtime

You may retain the loader in a delivered application, and use it to load compiled code or any of the supplied modules at runtime. This is useful if your application's users need to load their own code into it.

However, we do not recommend using this as a means of deferring the addition of module code to your image. It is far better to deliver your application with all the modules it needs. The first benefit is that the module itself is delivered — if you load it at runtime you cannot do this. Second, you avoid slowing your application to a halt while it loads the module. Finally, if you leave the option open of loading arbitrary code into the image, you may need to keep the entire common-lisp package, which adds greatly to the size of the delivered image.

3.2.3 Use of symbols, functions, and classes

Bear in mind that symbols, functions, and classes contribute significantly to the size of a delivered application. While it is not worth letting this interfere greatly with good design and maintainability, efforts to minimize their use in your application may pay off. **Note:** Symbols, functions and classes interact. If a symbol is retained, any function or class bound to it is also retained in the delivered application, even if it is never funcalled or instantiated. Delivery cannot be sure that the symbol is not ever used to do these things, and so errs on the side of safety, at the expense of image size.

3.2.4 Making references to packages

Certain Common Lisp functions and macros make explicit reference to packages. If you use any of these on particular packages, you may need to keep those packages in the application. This can contribute greatly to the size of the delivered application image. For more details, see Section 6.7.5 on page 78.

3.2.5 Declaring the types of variables used in function calls

You can minimize, or even eliminate, runtime decisions about the types of function arguments by making them instances of a known type. This gives the compiler a chance to inline appropriate code or perform other optimizations.

3.2.6 Use of TYPEP, TYPECASE and SUBTYPEP for type discrimination

If you use typep, typecase, and subtypep to discriminate between types, Delivery is forced to keep additional information in the delivered image. This is because these operations are open-ended and require the presence of information about all possible types. Large functions and tables containing this information must be retained in order for them to work.

Built-in predicates, such as consp and floatp, can, on the other hand, be retained with little cost in image size. The same applies to structure predicates. (For example, the predicate foo-p for (defstruct foo) would not be costly.)

If you retain CLOS, we recommend that you use methods for type discrimination instead of open-ended type operations, as an unused method has a greater chance of being deleted. For example,

```
(typecase x
  (integer (process-an-integer x))
  (string (process-a-string x))
```

becomes

```
(cond ((integerp x) (process-an-integer x))
           ((stringp x) (process-a-string x)))
or
     (defmethod process-anything ((x integer)) ...)
     (defmethod process-anything ((x string)) ...)
     (process-anything x)
```

3.2.7 Use of CLOS

If your application uses CLOS, Delivery finds it harder to make those parts of the image that make use of CLOS smaller. Its use of introspection and the potential for dynamic definition means that little of can be thrown out. Some classes must be retained, meaning that certain object slots cannot be deleted. However, the delivery code is written assuming that almost all applications use CLOS, and is designed to deal with it efficiently.

3.2.8 Use of the INTERN and FIND-SYMBOL functions

These functions allow a running program to locate arbitrary symbols. If your application uses them you may need to keep many symbols in the image, along with any associated definitions. See "Coping with INTERN and FIND-SYMBOL at runtime" on page 80.

Note: The read function typically calls intern, thus causing the same problems.

3.2.9 Use of the EVAL function and the invocation of uncompiled functions

Applications using eval or invoking uncompiled functions in other ways need the entire Common Lisp interpreter available to them. Delivery therefore keeps it in the delivered image, adding significantly to its size.

3.2.10 User-defined and built-in packages

Try to develop your application using a well-defined set of packages. Particularly, try not to intern symbols in built-in packages. You may find at delivery

time that a particular built-in package is suitable for throwing out, and therefore have to go back and take your symbol out of it in order to do so safely.

Warning: When you use built-in packages in your own packages (via defpackage), take care when naming symbols, since they may accidentally tie up with external function or class definitions in the built-in package and cause them to be retained unnecessarily. (This retention occurs because Delivery does not throw out unused definitions if they are referred to by some other symbol in the application — See "Use of symbols, functions, and classes" on page 14.)

3 Writing Code Suitable for Delivery

4

Delivering your Application

This chapter describes the process of delivering a completed application.

The first part of the delivery process is to make a standalone version of your application, that runs without assistance from LispWorks. After that, you may want to look into making your program smaller and more efficient.

Delivering a standalone application, and much of the work in making it smaller and faster, is extremely simple and can be accomplished by entering a simple form. However, fine-tuning the delivery process to make the application as small and as fast as possible is a more involved process that usually requires trial-and-error work. You should therefore allot time to a delivery phase when planning the development of your application.

A call to the function deliver starts the delivery process. A variety of arguments control the effects of delivery. A few of the keywords are introduced in this chapter, but all are documented fully in Chapter 5, "Keywords to the Delivery Function".

4.1 The delivery function: DELIVER

The function deliver is the main interface to the delivery tools. Its basic syntax is shown below:

deliver function file level &rest keywords

Function

The following three arguments are required:

function The name of the function that starts the application.

file A string or pathname naming the file in which the

delivered application should be saved.

The file extension .exe or .dll is appended to applica-

tions delivered on Windows.

level An integer representing the delivery level of the delivery.

This is a measure of how much work Delivery does to reduce the size of the image. It must be an integer in the range 0 to 5. Level 5 is the most severe, while the least

work on image reduction is done at level 0.

In addition, a variety of keywords can be passed to deliver in order to control aspects of delivery explicitly. These keywords are documented in Chapter 5, "Keywords to the Delivery Function".

4.2 Using the delivery tools effectively

This section gives some useful tips that should speed the delivery process up and make mistakes less likely.

4.2.1 Saving the image before attempting delivery

Since you must almost certainly make several delivery attempts before finding the optimal set of delivery parameters, the time spent starting LispWorks and loading application and library code soon adds up.

You can cut down on this startup time by saving a copy of the image when the compiled application and library code has been loaded. Use save-image (see the *LispWorks Reference Manual*) to do this. You then have an image that is "ready to go" for delivery as soon as it is started up.

Note: Before and after saving the image, it is a good idea to check that the application still works exactly as it did running on top of the LispWorks development environment.

4.2.2 Delivering the application in memory

You can save time when experimenting with delivery parameters by delivering the application in memory rather than saving it to disk.

If the deliver keyword :in-memory-delivery is non-nil, the delivered image is not saved to disk, but instead starts up automatically after the delivery operations are complete.

For example, a good early test is

Note: The image exits as soon as the application terminates.

4.3 How to deliver a standalone version of your application

There are usually two considerations when delivering an application.

- Making the application run standalone.
 That is, turn the application into a single executable file that needs no assistance from LispWorks in order to run.
- Make the application smaller.That is, make the application smaller than the development environment plus application code.

We recommend delivering a standalone application first, with no attempt to make the image smaller. Do this by delivering at delivery level 0, which removes very little from the image. You can then look into making the image smaller if you need to.

If you try to do both of these in the first attempt and the delivered application does not work, it is not clear whether the wrong thing was removed from the

image, or the application would not have delivered properly even if no image reduction work was done.

Once you have developed and compiled your application, you are ready to deliver it as a standalone application Delivering a standalone version is done by calling deliver with level 0, which does not try to make the image smaller, but removes the LispWorks environment. To do this modify your deliver, lisp script from "Delivering the program" on page 10 as appropriate to your application:

```
(load-my-application)
;;; unless you have it already loaded as suggested in
;;; "Saving the image before attempting delivery" on page 20
(deliver 'my-function "my-program" 0 :interface :capi)
(quit)
```

This is assuming your application uses CAPI. If it does not, you can eliminate interface :capi. In this case, if your application requires multiprocessing, you to need to ass :multiprocessing t:

```
(deliver 'my-function "my-program" 0 :multiprocessing t)
```

Then call LispWorks with deliver.lisp as an init file.

On Windows, open a DOS window and type:

```
lispworks-4100.exe -init deliver.lisp
On UNIX, type to a shell:
unix% lispworks -init deliver.lisp
```

This creates an executable in my-program.exe on the PC, or my-program on UNIX. When this executable starts, it calls my-function without arguments.

4.4 How to deliver a smaller and faster application

Saving your application standalone is only the first step towards delivering a satisfactory image. The next step is to try and make it smaller.

An entire Common Lisp system, and other supporting code, remains in a standalone image delivered at delivery level 0. A good deal of this can usually be thrown away.

What can be thrown away depends on the needs of the application. Few applications use all the facilities in the basic image. For instance, if the application does not use any complex numbers, all the code in the image for working with complex numbers can be deleted.

4.4.1 Making the image smaller

You can specify that the image be made smaller in two complimentary ways:

- **1.** By increasing the delivery level.
 - This is the simplest way to make the image smaller. As you increase the delivery level, Delivery employs different and increasingly severe delivery strategies.
- 2. By specifying what to remove and what to keep, using keyword arguments to deliver.

This is a more complicated way to control image size, and should only be resorted to if there are problems or not enough savings can be gotten with increasing the delivery level. These keywords are documented in Chapter 5, "Keywords to the Delivery Function".

These two approaches are based upon the same mechanism: delivery levels are in fact nothing more than different combinations of keyword parameters. But when you specify a delivery level and at the same time pass keyword values, the values you pass override any settings forced by the delivery level.

As an example of how explicit directions to Delivery can be necessary for effective delivery, consider the general addition function, +. The internal representation of the function contains references to functions that carry out complex number arithmetic, since + has to use them if it is given complex arguments. If you know your application does not ever pass complex arguments to +, you should probably remove those functions from the delivered image.

Delivery cannot decide for itself that you do not pass + any complex arguments, and so does not delete the complex number functions. You can tell Delivery to do so explicitly, by passing :keep-complex-numbers nil to deliver. (See page 44 for a discussion of this keyword.)

4.5 How Delivery makes an image smaller

Delivery makes an image smaller in two ways.

- **1.** By garbage collecting the image. This is done automatically.
- 2. By "shaking" the image with the treeshaker. This is done automatically from delivery level 2 upward.

4.5.1 Garbage collecting the image

The image is garbage collected during delivery. The garbage collector locates any unreferenced objects and frees the space they occupy. Then Delivery compacts the remaining memory so that the saved image is smaller.

Garbage collection is a generally good method of trimming the image size at delivery time. However, it is general too conservative, and so it has no effect on a significant portion of the Common Lisp system and your application: Interned symbols, class definitions, and methods discriminating on classes. Such objects must be dealt with by the treeshaker.

4.5.2 Shaking the image

From delivery level 2 upward, the image is "shaken" by default during delivery with the treeshaker. You can also invoke the treeshaker directly with the deliver keyword: shake-shake-shake, discussed on page 57.

As discussed above, the garbage collector does not delete any interned symbols, class definitions, or methods discriminating on classes from the image, even when they are unused. This is because it is designed to keep any object for which a reference exists.

There are always references to interned symbols, class definitions, and methods discriminating on classes. Interned symbols, naturally, are referred to by their package. Class definitions are always pointed to by their superclasses (the root class, t, has no superclass but is protected from garbage collection), and a method discriminating on a class is always pointed to by the class.

Thus we have a special class of objects that cannot be removed under the normal garbage collection scheme. Using the treeshaker, however, we can do so. The treeshaker does the following to overcome the default links between these objects:

- Record the default links.
- **2.** Break the links.
- **3.** Garbage collect the image.
- 4. Reinstate the links.

Step 2 renders the objects the same as all others in the image. They are now only protected from garbage collection if there are links to them elsewhere in the image — that is, if they are actually used in the application.

The term "treeshaker" is derived from the notion that the routine picks up, by its root, a tree comprising the objects in the image and the links between them, and then shakes it until everything that is not somehow connected to the root falls off, and only the important objects remain. (An image would usually be better characterized as a directed graph than a tree, but the metaphor has persisted in the Lisp community.)

4.6 Additional steps after delivery level 5

There are several suggestions for additional steps that may be taken to reduce the size and speed of the delivered application image after the maximum delivery level has been reached.

- User packages may be pushed onto the list of packages to be deleted by using the keyword argument :delete-packages to the function deliver (see page 36). This means that all unreferenced symbols are shaken from the pushed packages.
- 2. Return to compiling the application code, and experiment with optimization. Type declarations and safety declarations can both speed the application up and make it smaller. Compile code for delivery with the compiler option (space 3). However, you should ensure that low-safety inlining does not bloat the image.

- 3. Parts of the numeric code can be deleted by using the override list to specify that, for example, the code to handle complex numbers should be deleted. This can be done by passing :keep-complex nil to the function deliver.
- **4.** Study the output from a typical run, and delete unused functions with the deliver keyword :functions-to-remove, or do so manually with fmakunbound.
- 5. If the application does not use CLOS, remove it with :keep-clos nil.
- 6. Delete the format code by passing the :format nil to the function deliver.

Keywords to the Delivery Function

This chapter describes the keywords to the delivery function, deliver.

The keyword descriptions are given in alphabetical order. Before the alphabetical section, there is a topic-based list of keyword names which should be of value if you are looking for a keyword to perform a particular task for you, but do not know what it is called or do not know if it exists.

The list of keywords can be printed by calling deliver-keywords, which is documented in Section 8.2 on page 92.

Caution: Many keywords interact with one another, causing apparent values to change. It is a good idea to check how keywords interact and also what happens to their defaults at the different severity levels.

5.1 Topic-based list of DELIVER keywords

This section provides a topic-based index to the descriptions of deliver keywords. Use the topic headings to find a keyword related to a particular kind of delivery task, then look it up on the page given to see how to use it.

5.1.1 Controlling the behavior of the delivered application

The following keywords control aspects of the delivered application's behavior. There are keywords for specifying startup banners, application icons, image security, and so on.

- :action-on-failure-to-open-display
- :analyse
- :console
- :dll-exports
- :icon-file
- :image-type
- :interface
- :interrupt-function
- :interrogate-symbols
- :keep-gc-cursor
- :multiprocessing
- :product-code
- :product-name
- :query-end-session-handler
- :quit-when-no-windows
- :startup-bitmap-file
- :versioninfo

5.1.2 Testing and debugging during delivery

The following keywords can be used to help test and debug the application either during delivery or at runtime. There are keywords for encoding test routines into the delivered application, for ensuring that features such as the debugger and the read-eval-print loop are kept in the image, for performing delivery without writing the image out to disk, and so on.

- :analyse
- :call-count
- :clos-info
- :condition-deletion-action
- :diagnostics-file
- :error-on-interpreted-functions
- :exit-after-delivery
- :in-memory-delivery
- :interrogate-symbols
- :keep-conditions
- :keep-debug-mode
- :keep-modules
- :keep-stub-functions
- :keep-symbol-names
- :keep-top-level
- :kill-dspec-table
- :run-it
- :symbol-names-action

5.1.3 Delivering the application

The keywords listed in this section control the main part of the delivery process, involved in keeping things in and deleting things from the image. Most of the deliver keywords are in this general category, so it has been split up into a number of subcategories.

5.1.3.1 Directing the behavior of the treeshaker and garbage collector

The following keywords control the invocation of the treeshaker and garbage collector during delivery:

- :compact
- :shake-shake-shake
- :clean-down
- :redefine-compiler-p

5.1.3.2 Classes and structures

The following keywords are for examining, for keeping and for removing data information in the image about structured data: structures, classes and so on.

- :classes-to-keep-effective-slots
- :classes-to-keep-effective-slots
- :generic-function-collapse
- :gf-collapse-output-file
- :gf-collapse-tty-output
- :keep-all-classes
- :keep-clos
- :keep-clos-object-printing
- :keep-structure-info
- :metaclasses-to-keep-effective-slots
- :shake-class-accessors
- :shake-class-direct-methods
- :structure-packages-to-keep
- :warn-on-missing-templates

5.1.3.3 Symbols, functions, and packages

The following keywords are for examining, for keeping and for removing symbols, functions, and entire packages from the image.

:delete-packages

- :exports
- :functions-to-remove
- :keep-documentation
- :keep-function-name
- :keep-load-function
- :keep-package-manipulation
- :keep-symbols
- :macro-packages-to-keep
- :never-shake-packages
- :no-symbol-function-usage
- :packages-to-keep
- :packages-to-keep-symbol-names
- :redefine-compiler-p
- :remove-setf-function-name
- :shake-externals
- :smash-packages
- :smash-packages-symbols
- :symbol-names-action

5.1.3.4 LispWorks environment

Keywords for keeping and for removing editor commands and LispWorks environment tools:

- :editor-commands-to-delete
- :editor-commands-to-keep
- :keep-editor
- :keep-walker

5.1.3.5 CLOS metaclass compression

- :classes-to-keep-effective-slots
- :metaclasses-to-keep-effective-slots

5.1.3.6 Input and output

The following keywords are for keeping and for removing code loading facilities, fasl dumping facilities, special printing code, and so on, from the image.

- :format
- :keep-fasl-dump
- :keep-lisp-reader
- :keep-load-function
- :print-circle

5.1.3.7 Dynamic code

The following keywords are for keeping and for removing code facilitating dynamic runtime activities, such as macroexpansion, evaluation, use of the Common Lisp reader and the lexer, and so on, from the image.

- :keep-eval
- :keep-lexer
- :keep-macros
- :macro-packages-to-keep
- :remove-setf-function-name

5.1.3.8 Numbers

The following keywords are for keeping and for removing code from the image that can handle various numerical types: complex numbers, floating point numbers, ratio numbers, bignums, and transcendental numbers:

- :keep-bignum-numbers
- :keep-complex-numbers

- :keep-float-numbers
- :keep-ratio-numbers
- :numeric

5.1.3.9 Conditions deletion

The following keywords are for controlling the preserving or deletion of conditions.

- :condition-deletion-action
- :keep-character-encoding
- :keep-conditions
- :keep-foreign-symbols
- :package-to-keep-conditions
- :redefine-compiler-p

5.2 Alphabetical list of DELIVER keywords

This section describes each of the deliver keywords. They are presented in alphabetical order.

```
:action-on-failure-to-open-display
```

Keyword

Default: print and exit

If the application uses the X11 code or CAPI, it may fail to run if it cannot open the display. In this case, it calls this function with one argument, the display name. The default is to print a message and exit.

:analyse Keyword

Default: nil

When non-nil, the delivery code arranges to generate an analysis of what there is in the image before running the application. If the value of :analyse is a string or a pathname, it writes the analysis to this file, otherwise it writes to *standard-output*.

:call-count Keyword

Default value: nil

This keyword can be used to produce reports about what is left in the image when delivery is over. It is useful when determining which remaining parts of the system are not needed. When <code>nil</code>, no reports are generated.

Possible values of :call-count are:

size After running the application, the image is scanned,

and the size of each object, in bytes, is printed out. This produces a lot of output, comparable in size to the delivered image itself, so make sure you have plenty of

disk space first.

:all After running the application, the image is scanned,

and the name of each symbol found is printed out. A + sign is printed next to the symbol if it is non-nil. If the symbol is fbounds, the call count (that is, the number of times it was called while the application ran) is printed

too.

Delivery sets the call counter for all symbols to 0 before the saving the delivered image.

Interpreted functions do not maintain a call counter.

t This has the same effect as :all, but only symbols with

function definitions that were *not* called are printed.

The output is *always* written to a file. You can specify its name with :diagnostics-file. On Windows, the default is dvout.txt. On UNIX, the default is nil.

:classes-to-keep-effective-slots

Keyword

Value: classes

Classes on this list retain their effective-slot-definitions.

:classes-to-remove Keyword

Default value: nil

This keyword accepts a list naming the classes to be deleted from image during delivery.

Note: Their subclasses are also deleted, because they have lost their connection to the root class.

:clos-info Keyword

Default value: '()

With this keyword you can make the delivered image print a list of the remaining classes, methods, or both, after execution terminates.

Possible values of :clos-info are:

:classes print remaining classes only

:methods print remaining methods only

:classes-and-methods

print remaining classes and methods

The output is written to a file. You can specify its name with :diagnos-tics-file, page 36. The default is stdout.

:compact Keyword

Default value: (>= severity 5)

If this is non-nil, the heap is compacted just before the delivered image is saved. This usually gives the greatest gain in delivery, but takes a long time to complete. It is therefore worth leaving this until the final delivery.

:console Keyword

Windows only. This is the same as the :console keyword to hcl:save-image. See the *LispWorks Reference Manual* for details.

Keyword :delete-packages

Default value: ()

This keyword takes a list of packages, in addition to those in the variable *delete-packages*, that should be deleted during delivery. The Common Lisp function delete-package is used to do this.

When a package is deleted, all of its symbols are uninterned, and the package's name and nicknames cease to be recognized as package names.

After the package is deleted, its symbols continue to exist, but because they are no longer interned in a package they become eligible for removal at the next garbage collection. They survive only if there are references to them elsewhere in the application.

Note: Invoking the treeshaker has much the same effect on packages as deleting them. However, by deleting a package you regain some extra space taken up by hash tables.

Affected by: :packages-to-keep

:diagnostics-file

Keyword

Default value: nil

The string passed with this keyword specifies a file to which output generated by :call-count and :clos-info is written (in that order). nil means write to *standard-output*.

Keyword :dll-exports

Default: nil

This keyword is applicable to the Windows version only. Changes the image type created by save-image or deliver to :dll, which means that the saved image is a DLL file rather than an executable. The value of dll-exports should be a list of strings naming the exports of the DLL. Each external name must be defined as a Lisp function by using fli:define-foreign-callable.

For example, the script below (with lispworks-4100.exe -init hello.lisp) creates hello.dll, which can be run using "rundll32 hello.dll,Hello" from the command line. To see the dialog, you may need to click on the LispWorks splashscreen first.

You can use LoadLibrary from the main application to load the DLL and GetProcAddress to find the address of the external names. Lisp multiprocessing is started when the DLL is loaded, so any initialization operations can be done by adding process specifications to mp:*initial-processes* before creating the DLL.

For example, if you have a function like this:

```
(defun my-server ()
  (let ((s (establish-a-socket)))
      (loop (accept-connection s))))
You need to do:
(pushnew '("My server" () my-server) mp:*initial-processes*
      :test 'equalp)
before saving/delivering.
```

mp:*initial-processes* is a list of lists. Each list is used by the system as a set of arguments to mp:process-run-function. During initializing multiprocessing, the system does this:

```
(dolist (x mp:*initial-processes*)
 (apply 'mp:process-run-function x))
```

:editor-commands-to-delete

Keyword

Default value:

:all-groups

When the Editor is loaded, you can delete some of its commands by passing a list of them with this keyword. Note that, by default, most Editor commands are retained. See Chapter 8, "Editors for Delivered Applications" for more details.

Affected by: :keep-debug-mode

:editor-commands-to-keep

Keyword

Default value: nil

When the Editor is loaded, you can keep some of its commands by passing a list of them with this keyword. Note that, by default, most Editor commands are retained. See Chapter 8, "Editors for Delivered Applications" for more details.

:error-on-interpreted-functions

Keyword

Default value: nil

If this is non-nil, an error is signalled during delivery if the interpreter is removed (with:keep-eval nil) while interpreted functions remain in the image.

:exit-after-delivery

Keyword

Default value: t

If this is non-nil, the LispWorks exits immediately after writing the delivered application image to disk, by calling quit. If nil, quit is not called, and deliver returns nil.

Warning: If you pass mil to this keyword, remember to keep listener code in the application or the image may simply hang.

:exports Keyword

Default value: nil

This keyword takes a list of symbols that should be exported from their home packages before any delivery work takes place.

:format Keyword

Default value: t

If this is nil, part of the functionality of format is removed. The format directives deleted are:

BCEGOPR? < / WS\$

:functions-to-remove

Keyword

Default value: nil

This keyword takes a list of symbols to be fmakunbound during delivery.

:generic-function-collapse

Keyword

Default value: (>= severity 3)

If this is non-nil, generic functions with single methods and simple arguments are collapsed — that is, replaced by ordinary functions.

Note: Methods cannot be added to collapsed generic functions, since after their collapse to ordinary functions the generic functions definitions are deleted. If you want collapsing to be performed in general, but with the exception of a few generic functions, see :gf-collapse-exceptions-file, page 40.

A formatted report detailing the actions performed during the collapse is output to the file specified by :gf-collapse-output-file. The default is "gfclps.txt".

```
:gf-collapse-exceptions-file
```

Keyword

Default value: nil

This keyword takes a string naming a file containing information about generic functions that should not be collapsed to ordinary functions.

You only need to specify this file if generic function collapsing is on (with :generic-function-collapse, see page 39) and want to specify some generic functions that should be treated as exceptions and not collapsed. You might want to do this if there are generic functions to which you would like to add methods at runtime.

If this is nil, no exceptions file is consulted.

Exceptions are specified with some special forms in the exceptions file. You can either specify a particular generic function to except from the process, or name a whole package in which all generic functions should be protected from collapse.

The first of these is done with an entry in the file as follows:

```
(:do-not-collapse < generic function name>)
```

where *egeneric function name* is a symbol naming a generic function to be excluded from the collapse.

Exclusion of a whole package's generic functions can be forced with the entry:

```
(:protect-pkg <package name>)
```

where *package name>* is a string naming a package in which no generic functions should be collapsed.

For example, the forms

```
(:do-not-collapse my-pkg::my-generic-function)
(:protect-pkg "MY-OTHER-PKG")
```

specify that the generic function named my-pkg::my-generic-function and all generic functions in the package my-other-pkg should not be collapsed.

:gf-collapse-output-file

Keyword

Default value: "gfclps.txt"

The string passed with this keyword is the name of the file in which the report of the generic function collapse is written.

:gf-collapse-tty-output

Keyword

Default value: nil

If true, send the output to the console about generic function collapsing.

:icon-file Keyword

Default value: :default for windowing applications and mil for console applications.

The name of a file containing the icon to use, in Windows .ico format, or mil (meaning no icon -- not recommended except for console applications) or :default (which uses the standard LispWorks icon).

:image-type Keyword

Default: (if (eq (delivery-value :dll-exports) :no) :exe :dll)

On the PC, passed to save-image to define if the image is to be a DLL or an executable.

:in-memory-delivery

Keyword

Default value: nil

If this is non-nil, the delivered application is not saved, but run in memory instead.

This can be useful while still deciding on the best delivery parameters for your application. Writing the delivered image to disk takes a lot of

time, and is not really necessary until you have finished work on delivering it.

Note: When using this keyword, the deliver function still demands that you pass it a filename. However, the filename you give is ignored. You can use nil.

Keyword :interface

Default value: nil

If set to :capi, ensures that if the delivery function (that is, the first argument to deliver) returns, then a message loop runs to catch future events. This is useful because a typical CAPI application function only needs to create and display its interfaces; :interface ensures that the user's events are processed.

Typically, you should set this keyword in all CAPI applications.

:interrogate-symbols

Keyword

Default: nil

Does two things:

First it loads the reverse-pointers-code module. This can be used to check what things to keep in the image. See separate documenatation for reverse-pointer-code.

Then it sets it up such that calling the application with argument interrogate-symbols, before starting the application, allows you to interrogate-symbols. See separate documentation.

:interrupt-function

Keyword

Deffault: t.

A function to call when an interrupt occurs. When it is t, it is calling quit.

:keep-all-classes

Keyword

Default value: (<= severity 3)

If this is nil, the treeshaker considers class definitions when looking for things to delete from the image. If the symbol pointers to the class are gone, and there are no other pointers to it, it is deleted.

Affected by: :keep-clos

:keep-bignum-numbers

Keyword

Default value: numeric

If the value of this keyword is non-nil, all the numeric functions in the associated list, *bignum-symbols*, that is all the numeric functions which use bignums, are kept. If the keyword is nil, all the functions in the list are deleted.

Affected by: numeric

:keep-character-encoding

Keyword

A list of character encodings that must be kept. The character encodings that the image knows about are shaken, this is, if the symbol that names a character encoding is not referenced otherwise, the character encoding is deleted.

:keep-clos Keyword

Default value:

```
(if (= severity 0)
    t
    :no-dynamic-definition)
```

If this is nil, the clos package is smashed. It can no longer be used by the application.

If it is :no-dynamic-definition, then the functions for dynamic class and method definition are deleted — functions like defmethod, defclass and so on, but the rest of the clos package is retained.

If it is t, nothing is smashed or deleted.

Note: MOP programmers should note that the direct-slots and directmethods of all classes are emptied during delivery. To prevent this, set :keep-clos to :no-empty (which keeps dynamic definition) or :noempty-no-dd (which does not keep dynamic definition).

Affected by: :keep-walker

:keep-clos-object-printing

Keyword

Default value: (<= severity 2)

If nil, the generic function print-object is redefined to be the following ordinary function:

```
(defun x-print-object (object stream)
  (print-unreadable-object
   (object stream :identity t)
   (if (and (fboundp 'find-class)
            (find-class 'undefined-function nil)
            (ignore-errors
             (typep object 'undefined-function)))
         (write-string "Undefined function " stream)
         (prin1 (cell-error-name object) stream))
       (princ (ignore-errors
                (type-of object))
              stream))))
```

You may redefine this function.

Affected by: :keep-debug-mode

:keep-complex-numbers

Keyword

Default value: t

If this is non-nil, all numeric functions that can handle complex numbers are retained.

:keep-debug-mode

Keyword

Default value: nil

If this is non-nil, Delivery retains the full TTY debugger, so it can be used when debugging delivered applications.

This is done by setting the keyword :compact to nil and the following keywords to t:

```
:keep-clos-object-printing
:keep-eval
:keep-function-name
:keep-lexer
:keep-lisp-reader
:keep-load-function
:keep-modules
:keep-structure-info
:keep-top-level
:keep-walker
:top-level-listener-p
```

On both platforms, if :keep-debug-mode is set to :keep-packages, the appropriate keywords are set, and all packages are retained as well, so that they can be used for debugging purposes.

:keep-documentation

Keyword

Default value: (= severity 0)

If non-nil, documentation is preserved.

:keep-editor Keyword

Default: nil

Keep the editor intact. By default some parts of the editor (mainly those that deal with Lisp definitions) are explicitly eliminated. When this keyword is true, nothing is removed.

:keep-eval Keyword

Default value: (< severity 4)

If this is non-nil, the evaluator is preserved.

Keyword :keep-fasl-dump

Default value: (= severity 0)

If this is non-nil, the internal functions needed to dump fasl files are preserved.

:keep-float-numbers

Keyword

Default value: numeric

If the value of this keyword is non-nil, all the numeric functions in the associated list *float-symbols*, that is all the numeric functions which use floating point numbers, are kept. If the keyword is nil, all the functions in the list are deleted.

The value of :keep-float-numbers is overwritten to t if CLX is used, i.e. if: windowing has any value other than nil or: mp-only, because CLX uses floats.

Affected by: keep-clx

> numeric windowing

:keep-foreign-symbols

Keyword

Unless this is true, the code and information that is required for dynamic loading of foreign code is eliminated from the image.

:keep-function-name

Keyword

Default value: nil

This keyword controls the retention of names for functions.

If the keyword :compact is non-nil, :keep-function-name is set to nil. However, if :call-count is either t or :all, then :keep-function-name is set to t.

On either platform, when :keep-debug-mode is set, :keep-functionname is set to t automatically.

Affected by: :keep-debug-mode

:keep-gc-cursor Keyword

Default value: nil

Windows only. This keyword is applicable to the PC only. If this is non-nil, the mouse pointer turns into a distinctive 'GC' cursor during the garbage collection of generations 1 and above. (Even if the cursor is kept, generation 0 collections are never indicated, because they occur frequently and do not cause a noticeable delay in operation.)

:keep-keyword-names

Keyword

Default: t

If non-nil, keep symbol names of keywords.

:keep-lexer Keyword

Default value: (< severity 3)

If this is non-nil, the lexer is preserved.

You should keep the lexer if your application requires macroexpansion or uses special forms. For example, if you delete the lexer you cannot evaluate defun or enclose forms at runtime.

:keep-lisp-reader

Keyword

Default value: (<= severity 5)

If the keyword is mil, the functions and values used to read Lisp expressions are deleted. This means that the listener no longer works.

The :keep-lisp-reader keyword is set to t automatically if :keep-debug-mode is t.

:keep-load-function

Keyword

Default value: (<= severity 2)

If this is nil, the load function is deleted. Runtime loading is no longer possible when this is done, whether or not require is being used.

It can take two non-nil values:

Keeps the loading code required to load data files. t

:full

Keeps the code as for t, plus those internal functions that are required for loading Lisp code. Note that if the Lisp code uses functions that are shaken, these functions must be explicitly kept.

Note: In most cases you need to keep the common-LISP (CL) package if files might be loaded into your application, and probably some other packages too. (See :packages-to-keep.)

Keyword :keep-macros

Default value: (< severity 2)

If this is nil, the functions macroexpand, macroexpand-1 and macrofunction are deleted, and all macro functions and special forms are undefined.

Note: This has no effect on compiled code, unless it explicitly calls macroexpand.

:keep-modules Keyword

Default value: (< severity 1)

If non-nil, the mechanism for loading modules is preserved.

:keep-package-manipulation

Keyword

Default value: (< severity 5)

If this is non-nil, the following package manipulation functions are pre-Served: shadowing-import, shadow, unexport, unuse-package, deletepackage, rename-package, import, export, make-package, use-package, unintern.

:keep-ratio-numbers

Keyword

Default: (delivery-value :numeric)

When mil, eliminate ratio numbers functionality.

:keep-structure-info

Keyword

Default value:

This keyword controls the extent to which structure internals are shaken out of the image.

If nil, all references from structure-objects to their conc-names, (BOA) constructors, copiers, slot names, printers and documentation are removed. See also :structure-packages-to-keep.

To retain slot name information (necessary if either the #s() reader syntax or CLOS slot-value are to be used for structure-objects) set :keep-structure-info to :slots.

To retain slot names and the default structure printer, set :keep-structure-info to :print.

Note: Any functions (constructors, copiers or printers) referenced in the application are retained, just as any other code would be. It is therefore not normally necessary to set this keyword.

Affected-by: :keep-debug-mode

:keep-stub-functions

Keyword

Default value: t

When this is non-nil, all functions deleted by the treeshaker are replaced by small stub functions. When a deleted function is called by the application, its stub prints a message telling you that the function has been deleted and how it can be reinstated. These stubs can take up a

lot of space if you smash large packages, but are invaluable while refining delivery parameters.

For instance, if your application calls complexp after delivery with :keep-complex-numbers set to nil, a message like the following is printed:

```
Attempt to invoke function COMPLEXP on arguments (10).
  COMPLEXP was removed by Delivery keyword : KEEP-COMPLEX-NUMBERS
  Try : KEEP-COMPLEX-NUMBERS T.
```

:keep-symbols

Default value: ()

This keyword takes a list of symbols that are retained in the delivered image. A pointer to this list is kept throughout the delivery process, protecting them from garbage collection.

Compare with :save-symbols.

:keep-symbol-names

Keyword

Keyword

Default: nil

A list of symbols that must retain their symbol names.

:keep-top-level Keyword

Default value: (< severity 5)

If this is nil, functions for handling the top level read-eval-print loop are deleted. Note that this means that if the line based debugger is invoked, there is no way to communicate with it

If this is non-nil, the top level history is also cleared when the severity level is higher than 0.:keep-symbol-names

Affected by: :keep-debug-mode

:keep-walker Keyword

Default value: nil

If this is nil, the walker is deleted.

If CLOS dynamic definitions are kept, by setting :keep-clos to t, then the walker will be kept.

If :keep-debug-mode is t, :keep-walker will automatically be set to t.

:kill-dspec-table

Keyword

Default value: (< severity 2)

The dspec table is an internal table used for tracking redefinitions by defadvice, trace and so on. If this keyword is non-nil it does am implicit call to untrace, and previous uses of trace and defadvice are discarded.

:macro-packages-to-keep

Keyword

A list of package names. Symbols in these packages that have a macro definition are not made funbound when the delivery process delete macros from the image (when :keep-macros is nil). Note that if these symbols are not referenced, they may be shaken anyway. When :keep-macros is nil, this keyword has no effect.

:metaclasses-to-keep-effective-slots

Keyword

Value: metaclasses

Classes on this list of metaclasses retain their effective-slot-definitions.

:multiprocessing

Keyword

Default value: nil

If set, starts multiprocessing with the delivery function (that is, with the first argument to deliver) running in an "initial delivery process".

This keyword is automatically t when :interface is :capi, so you only need to set it if CAPI is not being used.

:never-shake-packages

Keyword

Default: delivery::*never-shake-packages*

A list of package-names that will not be shaken. These packages and all their symbols are preserved.

:no-symbol-function-usage

Keyword

Default: (not (delivery-value :keep-debug-mode))

On the PC, eliminates symbols that are used only for function calls.

:numeric

Keyword

Default: t.

Keep all numeric operations, unless overriden by the specific keywords :keep-float-numbers, :keep-ratio-numbers, :keep-bignum-numbers. :keep-complex-numbers.

:packages-to-keep

Keyword

Default value: ()

This keyword takes a list of packages to be retained. All packages in the list are kept in the delivered image, regardless of the state of the :smashpackages and :delete-packages keywords.

If :packages-to-keep is :all, then the two variables above are set to nil. See also "Coping with INTERN and FIND-SYMBOL at runtime" on page 80.

Note: Other keywords push packages onto the :packages-to-keep list.

:packages-to-keep-externals

Keyword

A list of packages that should retain their external symbols, even when :shake-externals is t (the default). When :shake-externals is nil, this keyword has no effect.

:packages-to-keep-symbol-names

Keyword

Default value: :

A list of packages that should keep their symbol names. The names of symbols in these packages are not modified, irrespective of the value of :symbol-names-action.

:packages-to-shake-externals

Keyword

A list of package names for which their external symbols should be shaken when the value of :shake-externals is nil. When the value of :shake-externals is t (the default), this keyword has no effect

:print-circle

Keyword

Default:

```
(or (= *delivery-level* 0)
  (delivery-value :interrogate-symbols))
```

When this is mil, the mechanism for printing circular structures is eliminated.

:product-code

Keyword

Default value: nil

UNIX only. Used to re-target the licensing requirements of the delivery image to those of the delivered application. :product-code is a fixnum supplied by Harlequin. If the :product-code is :none, the application will have no keyfile protection. You should not use the product code :none without a prior arrangement with Harlequin. If :product-code is not supplied then the image is not re-targeted and will require a "Lisp-Works Delivery" key to restart. Note that this should not be a problem while developing an application.

Keyword :product-name

Default value: nil

UNIX only. Used to re-target the licensing requirements of the delivery image to those of the delivered application. :product-name is used only in keyfile error messages to identify a product whose key is incorrect. If it is not supplied then this defaults to \"Anonymous Application\".

:query-end-session-handler

Keyword

Default value: nil

Useful if the CAPI environment is used. It takes the name of a function that is called before exiting the environment.

:quit-button Keyword

Default value: nil

This value is passed to :start-application.

If it is a string, :start-application opens an additional window consisting of a single button. You can quit the application by clicking on this button. If the value is not a string, no button window appears.

:quit-when-no-windows

Keyword

Default value: :default

If t, or if :default and one of :deliv-call-application or :startapplication has been used to start the application, then whenever the application is waiting for input, a routine is run to check whether any of its main (Toolkit or CAPI or CLIM) windows are still open. If there are no open windows, the application exits.

When the image exits, the string given in :exports is printed.

You can suppress the check for active windows temporarily, by setting the variable : *windows-on* to nil. Returning it to a non-nil value restarts the checking.

This facility only works if windowing is started in a normal manner, that is, if one of the following functions was called:

```
:start-application
:deliv-call-application
```

:redefine-compiler-p

Keyword

Delivery-level: 2

When this is true, the compiler is eliminated from the image.

:remove-plist-indicators

Keyword

Default value: '()

This keyword takes a list of plist indicators to be deleted.

:remove-setf-function-name

Keyword

Default: (not (delivery-value :keep-macros)

When t, the direct pointer from a symbol to its setf expansion is removed. That means that macroexpansion of setf is not reliable anymore. Normally, that is not a problem for the application.

:run-it Keyword

Default value: t

If this is t, the *function* argument to deliver is used as the application startup function.

If this is nil, no application startup function is called when the delivered image is started up.

The image exits immediately upon startup when :run-it is nil. Any :call-count report requested is still generated on exit.

This keyword can be useful if you want to look at the symbols in the image (with the keyword :call-count) but cannot you actually run the application — for example because the application links up to a data-

base, but the database has not been started up. In such cases, set it to nil.

:save-symbols Keyword

Default value: nil

This keyword takes a list of symbols to be retained in the delivered image. The symbols are migrated to and exported from the SAVED-STUFF package. See also "Coping with INTERN and FIND-SYMBOL at runtime" on page 80.

Compare with :keep-symbols.

:shake-class-accessors

Keyword

Default value:

```
(cond ((>= severity 4) :remove)
      ((>= severity 3) t)
      (t nil)))
```

This keyword controls whether class accessor functions are kept in their slot-definition objects. Removing them allows unreferenced functions to be deleted.

If it is nil it ensures all accessors are kept.

If it is non-nil, class accessors which are never referenced are deleted.

If it is :remove, all class accessor functions are removed from their slot descriptions.

In general, accessors may be safely removed. However, if your application needs to examine the slots of class instances, you need to retain them.

:shake-class-direct-methods

Keyword

Default value: (>= severity 3)

This keyword controls whether class-direct methods are deleted.

Note: A method is not deleted if it specializes on a class that remains in the delivered image.

:shake-externals Keyword

Default value: t

If this is nil, all external symbols are preserved.

If this is non-nil, external symbols are also made eligible for garbage collection when the treeshaker is invoked. See also :packages-to-shake-externals, page 53.

:shake-shake Keyword

Default value: (>= severity 4)

If this is non-nil, the treeshaker is invoked during delivery. The treeshaker attempts to get rid of unreferenced symbols from the delivered image.

It uninterns every package's internal symbols. (In the case of the KEY-WORD package, the external symbols.) A garbage collection is then carried out, after which any remaining symbols are reinterned in the package from which they came. A similar procedure for class definitions and methods discriminating on classes is also performed.

If you require that certain internal symbols be kept, and know they will not be kept because they are not referenced in the image, you can export them explicitly. See :exports. Doing so prevents them from being deleted.

External symbols are shaken by default.. See :shake-externals, page 57,.

:smash-packages Keyword

Default value: '()

This keyword takes a list of packages that should be smashed during delivery.

When a package is smashed, all of its symbols are uninterned, and the package structure is deleted. Also, its function definitions, property lists, classes, values, and structure definitions are deleted or set to nil.

See "Smashing packages" on page 77 for more details.

CAUTION: Smashing destroys a whole package and all information within its symbols. you are advised to avoid using it if possible. A better alternative, if you cannot deal individual with symbols, is :smashpackage-symbols.

Affected by: :keep-clos, :packages-to-keep, :keep-debug-mode

:smash-package-symbols

Keyword

Default value ()

Takes a list of packages as for :smash-packages but only the symbols in each specified package are smashed. The package is left, making it easier to see which symbols in the specified packages are pointed to by other packages.

:startup-bitmap-file

Keyword

Default value: nil (no bitmap is displayed)

The name of a file containing a bitmap (in Windows .bmp format) to be displayed when the application starts.

:structure-packages-to-keep

Keygord

Delivery-level: 1

A list of packages. Symbols in these packages that have a structure definition keeps all the information in this structure definition, whatever the value of :keep-structure-info is.

:symbol-names-action

Keyword

Default value: t

Defines what to with symbol names. When it is nil, or :packages-to-keep-symbol-names is :all, all symbol names are kept. When it is t, symbol names (except those which are kept by :keep-symbol-names, :keep-keyword-names or :packages-to-keep-symbol-names) are either changed to the same string Dummy Symbol Name (on UNIX), or shortened to a three characters unique code (on the PC).

Removing symbol names makes it very difficult to debug the application, and it is assumed that it is done after the application is essentially error free. However, some applications may make use of symbol names as strings, which may causes errors to appear only when the symbol names are removed. In some cases, the easiest solution is not to get rid of symbol names, but this has an effect on the size of the application. To aid debugging this situation, <code>:symbol-names-action</code> can have other values.

On the PC, :symbol-names-action can be :dump, in which case it dumps a lookup table of the codes of the shortened names into a file with the same name as the delivered application but with type sym. Note the gain in reduced size is still there, so the final release of the application may be done with :symbol-names-action :dump, On UNIX, :dump is equivalent to t.

On UNIX, :symbol-names-action can be one of :spell-error, :reverse, :reverse-case and :plist. In the case of :spell-error (which is,probablythe most useful), the last alphabethic characters in the first 6 characters of the symbol name are rotated by one, that is, A become B, g become h, and Z become A. This leaves the symbol names quite readable, but any function that relies on symbol names fails. A more drastic effect is achieved by :reverse, which reverses the symbol name. :reverse-case just change the case of every alphabethic character to the other case. This is more readable than :spell-error, but if the application relies on symbol-names but does not care about case, the errors do not appear. :plist causes the symbol names to be set to the dummy name, but the old string is being put on the plist of the symbol (get symbol 'sys::real-symbol-name). The simple backtracer uses the property when it exists to get the symbol name.

Note that in the UNIX case, all the debugging keywords mean that there is no reduction in size, so the final release of the product should be done with :symbol-names-action t to reduce the size of the application.

If the debugging shows that some symbols must retain their symbol name for the application to work, this must be flagged to deliver by either: keep-symbol-names Or: packages-to-keep-symbol-names.

:symbols-to-keep-structure-info

Keyword

A list of symbols of which the structure info should be kept, in addition to the symbols in the packages in :structure-packages-to-keep.

:symbols-to-precache-symbol-names

Keyword

On the PC, a list of symbols whose names to precache after shaking.

:top-level-listener-p

Keyword

Default value: (< severity 5)

This keyword controls the kind of top-level loop used in the listener.

When non-mil, the listener is started with a Lisp top level. If mil, the listener is inhibited until a Lisp top level is run for some other reason, such as entering the debugger. See also :keep-top-level, page 50.

:versioninfo Keyword

Default value: nil (no version information)

This keyword is applicable to the PC only. A plist containing version information to be placed in the executable file. If :versioninfo is not nil, it should be a plist of the following keywords. All strings should be in a form suitable for presentation to the user. Some of the keywords discussed below are mandatory, and some are optional.

Mandatory keywords:

:binary-version :binary-file-version :binary-product-version

You must specify either: binary-version or both binary-file-version and binary-product-version.

The file version relates to this file only; the product version relates to the product of which this file forms a part.

If :binary-version is specified, it is used as both the file and product version.

The binary version numbers are 64-bit integers; conventionally, this quantity is split into 16-bit subfields, denoting, for example, major version, minor version and build number. For example, version 1.10 build 15 might be denoted #x0001000A000000F.

Note: There is no requirement to follow this convention; the only requirement is that later versions have larger binary version values.

:version-string :file-version-string :product-version-string

You must specify *either* :version-string *or* both :file-version-string and :product-version-string.

The file version relates to this file only; the product version relates to the product of which this file forms a part.

If :version-string is specified, it is used as both the file and product version.

The version strings specify the file and product versions as strings, suitable for presentation to the user. There are no restrictions on the format.

:company-name The name of the company producing the product.

:product-name The name of the product of which this file forms a part.

:file-description

A (brief) description of this file.

Optional keywords:

:private-build Indicates that this is a private build. The value should be a string identifying the private build (for example, who the build was produced for).

:special-build Indicates that this is a special build, and the file is a variation of the normal build with the same version number. The value should be a string identifying how this build differs from the standard build.

A non-mil value indicates that this is a debugging ver-:debugp sion.

A non-nil value indicates that this file has been :patchedp patched; that is, it is not identical to the original version with the same version number. It should normally be nil for original files.

:prereleasep A non-nil value indicates that this is a prerelease version.

Indicates the OS for which this file is intended. The :file-os default value is :windows32. (:nt :windows32) may be specified instead, to indicate that this application is intended for Windows NT.

A string value, which allows additional comments to be :comments specified, in a form suitable to presentation to the user.

:original-filename

This specifies the filename (excluding drive and directory) of this file. Normally it is defaulted based on the filename argument to deliver.

:internal-name This the internal name of this file. Normally it is defaulted to the value of original-filename, with the extension stripped.

:legal-copyright

A string containing copyright messages.

:legal-trademarks

A string containing trademark information.

:language

The language for which this version of the file is intended.

This can be either a numeric Windows language identifier, or one of the keywords listed below. The default is :us-english.

:arabic :bulgarian :catalan :traditional-chinese :czech :danish
:german :greek :us-english :castilian-spanish :finish :french
:hebrew :hungarian :icelandic :italian :japanese :korean :dutch
:norwegian-bokmal :polish :bralilian-portuguese :rhaeto-romanic
:romanian :russian :croatio-serbian-latin :slovak :albanian
:swedish :thai :turkish :urdu :bahasa :simplified-chinese
:swiss-german :uk-english :mexican-spanish :belgian-french
:swiss-italian :belgian-dutch :norwegian-nynorsk :portuguese
:serbo-croatian-cyrillic :canadian-french :swiss-french

5 Keywords to the Delivery Function

Delivery and Internal Systems

6.1 Delivery and CLOS

Most applications using CLOS can be delivered without difficulty. However, there are a few potential exceptions to this rule. Code dynamically redefining classes and methods, and with certain method combinations, needs some extra work.

6.1.1 Applications defining classes or methods dynamically

Set the dv:deliver keyword :keep-clos to t to keep the code needed for dynamic redefinition in the image.

6.1.2 User-defined method combinations

The LispWorks CLOS implementation achieves fast method dispatch by producing a compiled function for all method combinations used in the application. Since necessary method combinations can often only be determined by seeing what arguments a generic function is called with, these functions can often end up being generated and compiled at runtime.

Because delivered application should not contain the compiler, method combination functions generated at runtime cannot be compiled on the fly. A solution to this problem is to use closure-based code to represent the

combinations. In addition, the image can make use of many "template" method combination functions, already compiled, from which it can choose when deciding upon the right method combination for a particular call. The correct methods can be plugged into these template functions at runtime. The templates provided can deal with:

- 1. A simple method combination, with the operator naming a function (or generic function) — not a macro or special form.
- 2. A more complicated method combination, constructing a form which should effectively be a tree of progn, multiple-value-prog1 and callmethod forms.

The templates included should cover nearly all cases of user-defined method combinations. In those cases where they do not, however, Delivery substitutes closure-based code to do the job.

In most cases the effect on method-dispatch time of using this substitute technique is negligible. Pathological cases might, however, cause a slowdown of 10–20% over compiled method combination functions. Fortunately, you can find out what the correct template would be, return to LispWorks, and precompile it.

6.1.2.1 Finding the necessary method combination templates

Even though it cannot compile method combination templates at runtime, delivery can generate the forms for them. The necessary method combination templates can be found by using the keyword: warn-on-missing-templates, default nil. If this keyword is non-nil, a warning is issued whenever a missing template is detected. The value of this keyword can be either a string or a pathname, in which case it is a file to put the warning in, or t, in which case the warning goes to *terminal-io*. The warning takes this form::

```
;>>> Add this combination to the template file <<<
(PRE-COMPILE-COMBINED-METHODS
 ((1 NIL) NIL (_CALL-METHOD_)))
. ****
```

You can take this method combination template, place it in an ordinary lisp file, return to LispWorks, and compile it. This compiled file should be loaded into the image before delivery. See "Incorporating the templates into the application" on page 67.

Most missing templates can be found statically, and if :warn-on-missing-templates has been set, they are output at the time of saving the delivery image.

It is also possible to find the required templates without having to save an image, by calling

```
(clos::force-method-combination t)
```

after the application code has been loaded into the delivery image.

In either case, an attempt is made to find all missing templates. However, because method combinations are dependent on the actual arguments to generic functions, it is not always possible to find every missing template. The application must be run to be sure of finding all the missing templates.

Note: Valid combinations may be generated or seen in warnings even if they are never used. Delivery can only tell you what combinations the application could potentially use.

6.1.2.2 Incorporating the templates into the application

A typical measure is to put all the templates generated into a file. You can add new ones to it as you work through the delivery process. The templates must be compiled and loaded into the application before delivery. To do this:

Collect into one template file all the method combination template forms
that have been output. Start the template file with the form (in-package
"CLOS"), so that it looks something like this:

```
(define-pre-templates
  demand-caching-dcode-miss-function (6 nil (4)))
```

No matter how many times the template form is printed, it only needs to be included in the template file once.

- **2.** In the LispWorks image, compile the template file.
- **3.** Load the compiled template file into the image (along with the application and library files) before delivery.

6.1.3 Compression of CLOS metaobjects

To reduce the size of the delivered image, the delivery process compress the representation of CLOS metaobjects (classes generic-functions, methods). This includes:

- 1. nullifying the class-direct-slots of the class.
- 2. Changing the effective-slots to a function that is used in the initialization of the instance. This is controlled by :metaclasses-to-keep-effective-slots and :classes-to-keep-effective-slots.
- 3. Compressing the representation of method objects. This is controlled by :keep-clos. If :keep-clos is t, the representation of method objects is not compressed. There is also no compression if you add a method to method-qualifiers, method-specializers or closmethod-function
- **4.** Compressing the representation of generic functions (PC only, at the moment). This is not done if :keep-clos is t, or if you add methods to any of the accessors of generic functions.

6.1.4 Classes, methods, and delivery

See "Shaking the image" on page 24 for a discussion of how unused class definitions and methods are treated by delivery process.

6.2 Editors for delivered applications

This section contains information on how to include the Common LispWorks editor in your delivered applications.

6.2.1 Editor command groups

If any part of the editor is present in the image, every editor command that has been loaded will be kept in the delivered image. To get rid of editor commands, use the keyword argument :editor-commands-to-delete to deliver.

Deleting a command does not automatically delete the associated function. For example, the function editor:do-something-command could be called by the application even if the command "Do Something" has been deleted.

The function itself is only deleted if it is not referenced elsewhere in the application or if removed explicitly. Therefore, an application which uses the editor in a non-interactive or limited interactive manner can delete all or most of the editor commands. Note also that key bindings associate key sequences with commands and not functions, so if a command is deleted any sequences bound to it will no longer work. For consistency, the delivery process unbinds them too.

The keyword :editor-commands-to-delete is processed in different ways depending on the sort of value passed:

List value Process each element of the list. (Thus the list is tra-

versed recursively.)

String value The corresponding editor command is deleted.

Symbol value Taken to specify a Command Group.

The available command groups are:

ing text files, including regions, buffers and windows, movement, insertion and removal commands, key bindings, the echo area and extended commands (such as Alt+x), file handling commands, filling and indenting, and undo.

:full-editor

The full editor has all the facilities of the simple editor, and adds handling for Lisp forms, auto-save help and other documentation commands searching, including the system based search commands, tags support, and support for interactive modes.

:extended-editor

The extended editor adds Lisp introspection to those features: arglists, evaluate, trace, walk-form, symbol completion, dspecs, callers and callees, buffer changes, and hooks into the inspector and class, generic function, and system browsers.

:demand-loaded Commands present in the standard LispWorks image only if they are demand loaded.

:tools Commands supporting tools which must be explicitly

loaded on top of the editor, for example the listener.

:exclude Commands always deleted by the delivery process, for

example, compilation commands.

6.3 Error handling in delivered applications

The error handling facilities ordinarily provided by the Common Lisp Condition System are not present by default in delivered applications. If you choose not to retain the full Condition System, you can make use of the more limited, but smaller, error systems available with Delivery.

Simplified error handling is still possible in Windows applications without the Condition System. They can only trap "conditions" of type error or warning. If an application signals any condition other than warning or simple-warning, the condition is categorized (and therefore trappable) as one of type error.

6.3.1 Making the application handle errors

There are two classes of error an application is likely to need to handle: errors generated by the application, and errors generated by the Lisp system.

6.3.1.1 Handling errors generated by the application

Error conditions that can occur in your application domain can be handled easily enough if you define your own error handling or validation functions to trap them. For instance, you might ordinarily have the following code, which manages an error condition and makes a call to error:

You can easily define a version of generate-error that does all the work:

6.3.1.2 Handling errors generated by the Lisp system

Errors generated by the Lisp system, rather than the application domain, are a little harder to deal with.

Suppose your application performs an operation upon a file. The application calls a system function to complete this operation, so when there is no error system, any errors it generates must be caught by be the application itself.

The best solution to this problem is to wrap an abort restart around the operation. For example:

```
(defun load-knowledge-base (name pathname)
  (restart-case
     (internal-load-knowledge-base name pathname)
     (abort ()
            (capi:display-message
             "Failed to load knowledge base ~a from file ~a"
             name (namestring pathname))
             nil)))
```

Another solution would be to use a handler, as in the example below:

```
(defun my-handler (type &rest args)
  (if (symbolp type)
      (apply 'capi:display-message
             "An error of type ~A occured, args ~A"
             type args)
      (apply 'capi:display-message args)))
```

The disadvantage of this approach is that the message is unclear.

In general, the application should not cause Lisp errors. Because it is difficult to ensure that these never happen, it is a good idea for the application to wrap an error handler around all its code. For example:

```
(handler-bind ((error 'application-handler-error))
 (loop
    (catch 'application-error
        (setup-various-things)
        (do-various-things))))
(defun application-handler-error (condition)
  (when *application-catch-errors*
      (progn (give-some-indication-of-error)
             (do-some-cleanup)
             (throw 'application-error nil))))
```

(when *application-catch-errors* is nil, this just returns and then the debugger is invoked).

In addition, the areas that are more prone to errors should be dealt with specifically. For example, file access is prone to error, so it should wrapped with error handling.

6.3.2 Deletiing of condition classes.

Condition types are classes like any othe class, so may be shaken out. However the code may contain many references to condition types through error calls that are never going to happen in the application. Therefore, there is special conditions deletion action, which is controlled by three keywords.

:condition-deletion-action

Keyword

Value: one of:

nil Don't delete any condition class.

:delete (The default). Delete unwanted conditions. If an error

for a deleted condition is signaled, it is signalled as a simple error condition, with the arguments in the for-

mat-arguments slot.

:redirect Redirect unwanted conditions to the first parent in their

hierarhy which is not deleted.

:keep-conditions Keyword

Default: nil

Values: one of:

:none Eliminate all conditions

:minimal Keep only the conditions that are in the class-prece-

dence-list of simple-error. (simple-error, simple-condition error, and serious-condition condition). This is useful for applications that use only ignore-

errors. It is equivalent to

:keep-conditions '(simple-error) :packages-to-remove-conditions

'("common-lisp")

:all Keep all conditions.

A list A list of conditions to keep. For each condition, all the precedence list is kept.

:packages-to-remove-conditions

Keyword

Default: nil

A list of packages whose conditions are removed (that is where the symbol-package of the name of the condition is one of the packages). The system automatically adds the internal packages to this list. Conditions that are in these packages but are also in the :keep-conditions list or its precdence list are kept. The defaults cause all the conditions that are defined by the system and are not standard to be deleted. To keep all the conditions, you should do :keep-conditions :all (or :conditiondeletion-action nil). To eliminate all conditions, you should do :keep-conditions :none.

When a condition is deleted (that is when :condition-deletion-action is :delete), trying to signal it returns a simple-error, which means that it got the wrong type. On the other hand, it has all the information in the formatarguments slot. If the conditions are redirected (that is, when :conditiondeletion-action is :redirect), a stricter type is returned, but some of the information may be lost, because the condition that it redirects to has fewer slots.

User defined conditions are kept, unless:

- You add packages to:packages-to-remove-conditions.
- 2. You set: keep-conditions to: none, in which case all the conditions are eliminated, or :minimal, in which case all the user conditions are deleted.

6.4 Foreign Language Interface templates.

The Foreign Language Interface requires pieces of compiled code to convert foreign objects to lisp objects and backwards, and it is difficult to know in advance which pieces of code are needed. Most of these pieces are already avalilable in the image, and most applications do not need anymore templates, so you need to deal with it only if your application gives an error of missing templates after delivery.

When a new template is required, it is compiled. In a delivered image, where the compiler has been removed, this causes an error. To solve this you need to find which templates your application uses that are not already avaliable, compile them, and load them before delivering.

To find which templates your application needs, do the following:

- Starts the full image with your application (without delivering).
- **2.** call

```
(FLI:START-COLLECTING-TEMPLATE-INFO)
```

- **3.** exercise the application.
- 4. Call

```
(FLI:PRINT-COLLECTED-TEMPLATE-INFO)
```

This prints all the templates that are dynamically generated by using your application. These have to be put into a file and compiled. FLI:PRINT-COL-LECTED-TEMPLATE-INFO takes a keyword :OUTPUT-STREAM to make this easier, for example:

Once you have compiled the file containing the templates, it should be loaded as part of your application.

6.5 Modules

Part of the system is implemented using load on demand modules that are loaded automatically when a function is called. Most of these modules are only useful during development, so are not needed in the application. However, in some cases the application may need some module.

The list of loaded modules can be obtained by entering

```
:bug-form nil
```

to a listener. Among other things, this prints the list of loaded modules. This list includes modules that the application itself does not need (bug-people will always be there, because it is used by :bug-form). To get a more accurate list, do:

- 1. Start a fresh LispWorks image, making sure it does not load any irrelevant code (for example, in your .lispworks init file).
- **2.** Load the application and run it.
- 3. Enter :bug-form nil. The list of loaded modules should include only modules that your application needs, with the exception of bug-people.

Once you know a module is required in your application, you have to require it before delivering, by calling require:

(require module-name)

6.6 Symbol and package issues during delivery

Symbols and packages usually have the most significant effect on the size of a delivered application, so it is worth paying attention to them during delivery.

The basic principle of delivery is to garbage collect the image, freeing anything the application does not refer to in order to make the image smaller. This strategy works well enough for most objects, but not for symbols within packages: since all such symbols are referred to by their package, none of them can be deleted.

You can overcome this problem in the following ways:

- 1. By shaking the image.
- **2.** By *deleting* packages.
- 3. By smashing packages.

Deleting and smashing packages are not recommended. Deleting and smashing are explained in the next section. They are both ways of removing symbols from the application, one being more extreme than the other. You should note, however, that it is possible to handle specific symbols individually. This is preferred.

By default, Delivery deletes all of the system's packages, and smashes some of them. This following section also explains how to prevent this when necessary.

6.7 Throwing symbols and packages out of the application

This section discusses the circumstances in which you might want to throw symbols and packages out of the application, by *deleting* or *smashing* them.

6.7.1 Deleting packages

When you *delete* a package, the following happens:

- 1. All the package's symbols are uninterned.
- 2. The package name is deleted.

After the package is deleted, its symbols continue to exist, but because they are no longer interned in a package they become eligible for collection at the next garbage collection. They survive only if there are useful references to them elsewhere in the application.

Note: Invoking the treeshaker has much the same effect on packages as deleting them. However, by deleting a package you regain some extra space taken up by hash tables.

6.7.2 How to delete packages

You can pass deliver a list of packages to delete with the keyword :delete-packages (page 36).

6.7.3 Smashing packages

When you smash a package, the following happens:

- 1. All the package's symbols are uninterned.
- 2. The package structure is deleted.
- **3.** Its symbols' function definitions, property lists, classes, values, and structure definitions are deleted or set to nil.

After the package is smashed, the symbols continue to exist, but all the information they contained is gone. By being uninterned they become eligible for garbage collection. Also, the chances of any objects they referred to being collected are increased.

CAUTION: Smashing destroys a whole package and all information within its symbols. use it carefully.

Note: Any symbol whose home package is to be smashed can be retained by being uninterned before delivery commences.

6.7.4 How to smash packages

You can pass deliver a list of packages to smash with the keyword :smashpackages or : smash-package-symbols (pages 57 and 58)

6.7.5 When to delete and smash packages

Note: In general, you are advised against deleting or smashing packages unless it is absolutely necessary. Always try to reduce the image size as much as possible by treeshaking first.

If an application does one of the following things, packages are involved and you must consider keeping them in the application:

- 1. Makes an explicit reference to a package by some of the package functions, for example, intern, find-symbol and so on.
- **2.** Uses the reader, with read or any of the other reader functions.
 - These functions make reference to a package (either *package* or one given explicitly) whenever they read a symbol.
- Printing a symbol with the format directive ~s.
 - The format function prints the symbol with a package prefix if the symbol is part of a package.
- **4.** Loading a file, whether compiled or interpreted.
- Using the function symbol-package.

Fortunately, most applications are unlikely to do these things to more than a small number of packages. You should, therefore, be able to delete most packages without breaking the application. When you know that none of the symbols belonging to a package are used, you can go one step further and smash it.

Smashing a package guarantees space savings where deleting it would not. Even in a case where a symbol is referenced but unused, because it has been smashed you still regain space taken up by objects hanging from slots for function definition, value, property list and so on.

You do not usually gain much by smashing your own packages that you would not gain by just deleting them — you are after all unlikely to have included an entire package of symbols in your final application if you know it is not going to use them. The real benefits of smashing can be seen when it is performed on the *system's* packages, some of which may be entirely irrelevant to your application. In addition, you are unlikely to gain very much by deleting a package that you would not gain by treeshaking. In general, you should try to avoid either deleting or smashing packages explicitly.

However, if symbols in your packages are referenced through complex data structures, making it difficult to track references down, smashing may still prove useful.

6.8 Keeping packages and symbols in the application

This section explains how to keep packages and symbols in the application when Delivery would otherwise remove them.

6.8.1 Making sure that packages are kept

Your application may rely upon certain system packages that Delivery deletes or smashes by default.

You can protect these packages with :packages-to-keep (page 52). All packages in the list passed with this keyword are kept in the delivered image, regardless of the state of the :smash-packages (page 57) and :delete-packages (page 36) keywords. If you pass :packages-to-keep :all, then the two variables are set to nil.

Note: COMMON-LISP is the package your application is most likely to rely on, and it is also very large. Keeping it has a very noticeable effect on the size of

the application. However, if your application uses read or load, it invites the possibility of reading arbitrary code, and so common-lisp must be kept.

See also "Coping with INTERN and FIND-SYMBOL at runtime" on page 80.

6.8.2 Making sure that symbols are kept

Internal symbols in packages you have kept may still be shaken out. If any such symbol must be kept in the application, it can be retained by force in the following three ways:

- 1. With the :keep-symbols keyword. This is the recommended solution. See :keep-symbols (page 50).
- **2.** Export the symbol from the package.
 - External symbols are *always* shaken during delivery.
 - You can override this behavior by passing :shake-externals nil to deliver. See :shake-externals (page 57). You can also specify :packages-to-shake-externals and :packages-to-keep-externals, pages 52 and 53.
- Make explicit reference to the symbol with another object that you know will not be deleted.
 - A reference from the object to the symbol ensures that the garbage collector passes over it during delivery.

See also "Coping with INTERN and FIND-SYMBOL at runtime" on page 80.

6.9 Coping with INTERN and FIND-SYMBOL at runtime

If you want to delete or smash a package, but discover that a symbol is created in it at runtime with intern, or found in it with intern or find-symbol, you have two choices: either change the source to create or manipulate the symbol in another package, or keep the package after all.

If you cannot or do not want to change the source, and the package is large, you face the annoying prospect of having to keep a lot of code in the image for the sake of one symbol created or manipulated at runtime. Fortunately, there are ways to get around this.

The method is to migrate the symbols by hand into new or smaller, "dummy" packages. This is the only working method if at compile time you do not know the names of the symbols to be saved.

Create a special package or packages for the symbols mentioned in these calls, and delete the original packages. When this package is created (with make-package or defpackage), it should use as few of the other packages in the application as possible. Typically, :use nil suffices. For example:

```
(rename-package "XYZ" "XXX")
(push "XXX" *delete-packages*) ;; discard pkg
(make-package "XYZ" :use nil) ;; new pkg to reference
```

This allows the real package xxz to be deleted without breaking a call to intern such as the following:

```
(intern "FISH" "XYZ")
```

6 Delivery and Internal Systems

Troubleshooting

This chapter provides solutions to common delivery problems.

7.1 Debugging errors in the delivery image

In general, it is worth avoiding debugging an image that has been delivered at a high severity level. If you discover a bug:

- 1. First check if the same error occurs, in the original (undelivered) image. If it does, debug the problem in original image.
- 2. If it is not reproducible in the original image, check if it is reproducible with the delivered image at a lowered delivery-image (level 4). If it is, read the error message and backtrace carefully. In most of the cases, this is enough to figure out what went wrong.
- 3. If the problem occurs only in the delivered image and not in the original image, and it is not clear what the problem is, contact Harlequin Support immediately. Send all the output of the delivery process and the output from running of the application. This situation is regarded by Harlequin as a bug that should be fixed.

7.2 Problems with undefined functions or variables

A function or variable can be undefined for any of the following reasons:

- 1. It was never defined.
 - Check the image to see if it was defined before calling deliver again.
- **2.** It belongs to a package that was smashed.
 - Check whether its package is in the list of smashed packages printed by deliver. Use symbol-package to find out its home package.
- **3.** It was interned in the wrong package.
 - This would probably be because its real package was deleted. Check if the symbol that was called is one that was interned after delivering the image — that is, while the application was running.
- 4. It has been deleted explicitly.
 - For example, load, complex number functions, and so on. Check in Chapter 5 that there is no deliver keyword with a default setting that throws it out.
- 5. It is an internal symbol and was shaken out.
 - If a symbol that is printed is uninterned and you cannot work out its home package from its name, try using find-all-symbols or apropos in the image after loading the application, but before the call to deliver, to find the possible symbols.
- **6.** It belongs to a load-on-demand module. See Section 6.5 on page 75.

See "Symbol and package issues during delivery" on page 76 for the explanation and suggestions in cases 2, 3 and 5 above.

7.3 Failure to find a class

This situation can be resolved by much the same procedure as that described in "Problems with undefined functions or variables" on page 84.

If a lot of classes go missing, the deliver keyword :keep-all-classes can be used to keep them all.

7.4 Possible explanations for a frozen image

The image may die or hang up without issuing any useful message, either at runtime or possibly during delivery. Some possible remedies follow:

- Deliver the application at a lower severity level.
 If things work after this, try the same severity, but override the changed keywords one by one.
- Retain more packages, with the keyword :packages-to-keep
 For example:

The common-list package normally should not be deleted or smashed, so it is unlikely to cause problems, but listworks and the packages defined in the application itself are worth investigating.

If this gets the image working again, try to find out why the package is required and see if you can eliminate this need. See "Symbol and package issues during delivery" on page 76 for more information on keeping and throwing away packages.

7.5 Errors when finalizing classes

If an error occurs when finalizing a class, it usually means that a superclass is missing.

7.6 Warnings about combinations and templates

Warning messages such as the following:

```
;*****
;>>> Add this combination to the template file <<<
(PRE-COMPILE-COMBINED-METHODS
  ((1 NIL) NIL (_CALL-METHOD_)));
*****</pre>
```

occur when a method combination required by a particular function call is not available. You can eliminate these warnings either by compiling the method combination template forms output in the message and loading them into the image before delivery, or by using the keyword :warn-on-missing-templates. See "Finding the necessary method combination templates" on page 66, "Incorporating the templates into the application" on page 67.

7.7 VALID TYPE SPECIFIER errors

You may occasionally see an error of the form "symbol is not a valid type specifier". This usually means that a class named *symbol* is missing.

7.8 Stack frames with the name NIL in simple backtraces

Such frames probably correspond to methods. Use the deliver keyword :keep-function-name, page 46 to get the names back.

7.9 Blank or obscure lines in simple backtraces

These are usually stack frames named by the empty string. The keyword :packages-to-keep-symbol-names, page 53 may fix this. This technique can also be used on any symbol which prints as #: ||.

7.10 NIL IS NOT OF TYPE HASH-TABLE errors

This error is typically caused by evaluating special forms when the deliver keyword: keep-macros has been set to nil.

Beware of this when interacting with the debugger at severity levels 2 and higher. The absence of the special forms quote and function can cause difficulty. You may find the functions find-symbol, symbol-function and funcall useful here. It may also help to keep the common-lisp package (and perhaps also the system package), or specific symbols (with the :keepsymbols keyword).

7.11 Trying to construct a constructor

And error starting with "Trying to construct a constructor ..." is probably a result of missing foreign language interface templates. See Section 6.4 *Foreign Language Interface templates*. for instructions.

7.12 Reducing the size of the delivered application

If your application does not contain very large data structures, the greatest factor in its size when delivered is usually the number of symbols left in it.

This is because function definitions (which are large) are usually associated with symbols. Only when these symbols are deleted can the associated function definitions be deleted. Until that happens, the garbage collector passes over them during delivery.

You should look for symbols that are left in the image, which do not need to be there. You can do this by starting the delivered image in level 4 (or with :keep-debug-mode) with the argument -listener. The image starts by interacting with the user. You can then check which packages and symbols are left.

list-all-packages is one function you can use. Using the :call-count keyword is another possibility.

7.13 Interrogate-Symbols

interrogate-symbols is designed to find why symbols are left in the image even though they should not be. Since keeping information in the image would itself keep symbols, the facility has as little functionality as possible. The result is a non-intuitive interface, and you should be ready for this. You are encouraged to try other methods first. In particular, you might consider-contacting Lispworks support first.

To get interrogate-symbols to work, the keyword :interrogate-symbols should be give to deliver, with the value t. This loads the interrogate symbol facility. and in addition set the image so that on startup it checks if the command line contains -interrogate-symbols. If it does, the image first does symbol interrogation, and then goes on to the application.

Symbol interrogation starts by building an internal table of reverse pointers, during which the image prints some messgaes about its progress. When it finishes, it prompts:

```
Enter Symbol >
```

The input is read one line at a time. Each line is interpreted as a single string, as follows:

1. If the string does not contain the character #\:, and does not starts with #\+, it is a symbol name. The string is used as the argument to findsymbol (in the current package). Note the string is as it is, so it must not contain escape characters or leading or trailing spaces, and must be in the right case. for example, the symbol that is printed

```
SETF::\"USER\" \"WHATEVER\"
must be entered:
SETF:: "USER" "WHATEVER"
```

[omitting the escape characters #\\] and to find the symbol CAR, you must enter CAR, and not car. #\: characters after the first one (or the first pair) are taken as part of the symbol.

If the symbol is found, the image prints a list, when the car is the symbol, the second elemnt is a list of *interesting* symbols that points to that symbol (possibly through *uninteretsing* symbols), and the third element is a list of symbols that points to the symbol directly. A symbol B points to symbol B directly when there is a chain of pointers from A to B which does not go thorugh another symbol.

An *interesting* symbol is a symbol in another package, or a symbol from the same package which is pointed to by a symbol from another package. The idea is that the interesting symbols are the symbols that are most likely to be worth a further investigation.

Both the second and the third element may be the symbol :MANY rather than a list, if there are more the sys::*maximum-interrogate-return* (default 30) of them.

- 2. If the string contain a #\: character or a pair of #\: characters, and there characters after it, it a package name followed by a symbol name. The characters up to the first #\: are used to search for the package. If it is found, it skips the #\:, and if the following characters are #\: it skips them, too. The rest of the string is then used as a symbol name. Like in 1. above, both the package name and the symbol name must match exactly the actual package and symbol name. The output is the same as in 1.
- 3. If the string starts with #\+ followed by a string as in 1. or 2., then the symbol is found as in 1. or 2. Instead of looking for symbols that point to it, the image builds a tree of rereverse pointers starting from the symbol, going to depth sys::*check-symbol-depth*. In the tree, the car is an object and the cdr is a list of pointers to it. Each pointer may be a single object (if it has reached the depth limit, or found an object that is already in the tree), or a recursive tree. The tree may be quite extensive
- 4. If the first #\: character (or pair) is the last character in the string, than the line specifies a package name. If the string does not start with a #\+, the image prints each symbol from _other_ package that points (as defined in(1) above) to symbols in the package, followed by a list of the symbol in the package that it points to. To construct is this it has to check the reverse pointers from all the symbols in the package, which may take a long time if the package contain many symbols.
 - This option is specially useful, in conjunction with the :smash-pack-ages-symbols keyword to deliver, to find why a package that should have gone is left in the image.
- 5. If the string ends with #\: as in (4) above, but starts with #\+, then the rest of the string is treated as in (4), but the image simply prints for each the symbol in the package the same information that (1) prints for a single symbol.

7 Troubleshooting

User Actions in Delivery

8.1 Generat strategy for reducing the image size

In many cases, the size of the image can be reduced if part of the user code or data is eliminated, for example, when this code/data is there for debugging purposes. The system, however, cannot tell which part of the code/data can be eliminated, so you have to do it yourself.

That can be done in either of two ways:

- You can eliminated the code/data explictly before calling deliver, by using fmakunbound, makunbound, remhash etc. The adavantage of this approach is that it does not require you to know anything about delivery. The disadvantage of this is that it has to be explicitly put in the delivery script..
- 2. The image come with an action list (see entry for action-lists), "Delivery actions", which you can add actions to. This action list is executed when the delivery process starts, before any system action. For example, if *my-hash-able* contains entries that are not required in the delivered application, The you may write:

```
(defun clear-my-hash-table()
  (maphash #'(lambda (x y)
              (unless (required-in-the-application-p x y)
                (remhash x *my-hash-able*)))
         *my-hash-able*))
(define-action "delivery actions" "Clear my hash table"
              'clear-my-hash-table)
```

Using the action list has two advantages:

- 1. It does not have to be part of the deliver script, so it can be written near the rest of the code that is using *my-hash-able*, which makes it easier to maintain the code.
- 2. It can use the user interface of the delivery process. This is done via the function delivery-value and (setf delivery-value).

8.2 User interface to the delivery process

delivery-value Function

Signature: delivery-value deliver-keyword

(setf delivery-value) assigns a new-value to deliver-keyword

These must be called after deliver is called. deliver-keyword must be one of the legal keywords to deliver (which are listed in Section 5.2 on page 33, or can be displayed by calling deliver-keywords). delivery-value returns the value associated with this keyword. When deliver is called, the values associated with each keyword are initialized from the arguments to deliver or using their default values (which are listed by deliver-keywords), or set to nil. They can be modified later by user actions that were added to the "Delivery actions" action-list, and then by the system. Before starting the shaking operations, the values of the keywords are reset, and delivery-value cannot be called after the shaking.

(setf delivery-value) can be used to set the value of a keyword. Since the user actions are done before the system ones, the system actions (which also use delivery-value to access the keywords value) will see any change that the user actions did.

deliver-keywords Function

Lists the legal keywords to deliver. If the keyword default is non-nil, it is printed on the same line. The default is a form that is evaluated if the keyword was not passed to deliver, in the order that deliver-keywords prints. deliver-keywords also prints a short documentation string for each keyword.

delivery-shaker-cleanup

Function

Signature: delivery-shaker-cleanup object function

Used to define a cleanup function that is called after the shaking operation. delivery-shaker-cleanup stores a pointer to *function* and a weak pointer to *object*. After the shaking, the shaker goes through all the object/function pairs, and for each object that is still alive, calls this function with the object as argument. This is used to perform operations that are dependent on the results of the shaking operation.

If the cleanup function has to be called unconditionally, the object should be t. The cleanup function should be a symbol or compiled function/closure, unless the evaluator is kept via :keep-eval. The shaker performs another round of shaking after calling the cleanup functions, so unless something points to them, they are shaken away before the delivered image is saved. This also means that objects (including symbols) that survived the shaking until the cleanup function is called, but become garbage as a result of the cleanup function, are shaken away as well.

The cleanup function *cannot* use delivery-value. If the value of one of the keywords to deliver is needed in the cleanup function, it has to be stored somewhere (for example, as a value of a symbol, or closed over). It *cannot* be bound dynamically around the call to deliver, because the cleanup function is executed outside the dynamic context in which deliver is called.

An example:

Suppose the symbol P:X is referred to by objects that are not shaken, but its values are used in function P:Y, which may or may not be shaken. We want to get rid of the value of P:X if the symbol P:Y has been shaken, and set the value

of P:X to T if :keep-debug-mode is passed to deliver and is non-nil, or nil otherwise.

```
(defun setup-eliminate-x ()
  (let ((new-value (if (delivery-value :keep-debug-mode) t nil)))
    (delivery-shaker-cleanup
    t
     #'(lambda()
         (unless (find-symbol "Y" "P")
           (let ((sym (find-symbol "X" "P")))
             (when sym
               (set sym new-value))))))))
(define-action "Delivery actions" "Eliminate X"
  'setup-eliminate-X)
```

This sets up the lambda to be called after the shaking operation. It will set the value of P:X if the symbol P:Y has been shaken. Notes about the cleanup function:

- 1. It does not call delivery-value itself. Instead, it closes over the value.
- It does not contain pointers to P:X or P:Y. In this case, it is specially important not to keep a pointer to P:Y, because otheriwse it is never shaken.
- **3.** It does not assume that P:X will survive the shaking.

[The code as it is assumes the package "P" is not deleted or smashed]

The cleanup functions are called *after* the operation of delivery-shakerweak-pointer is complete, and are useful for cleaning up the operations of delivery-shaker-weak-pointer.

delivery-shaker-weak-pointer

Function

Signature: delivery-shaker-weak-pointer pointing accessor & key setter remover dead-value pointed

Used to make a pointer from one object to another weak object during the shaking operation. The operations of delivery-shaker-weakpointer are:

1. At the time it is called it computes the *setter* and *remover* if these are not given, and stores all its arguments for the shaker.

- Before the shaker starts, the shaker finds the value of the *pointed* object (if this is not given) using the *accessor*, and stores weak pointers to the *pointing* object and the *pointed* object. It then uses the *remover* to remove the pointer from the *pointing* object.
- **3.** After the main shaking operation, for each pair of *pointing/pointed* objects it checks if both have survived the shaking. If they did, it stores a pointer to the *pointed* object in *pointing* using the *setter*.

Arguments:

pointing

The pointing object. Because of the way deliveryshaker-weak-pointer is defined, you are free to use your own notion of pointing, for example, it may be the key in a hash-table.

accessor

The accessor that is called with the pointing object. It returns the pointed object. The *accessor* is used for two purposes:

- getting the pointed object if it is not given.
- 2. computing the setter if it is not given.

If both :pointed and :setter are passed to deliveryshaker-weak-pointer, the accesor is not used. The accessor can be one of:

A symbol. This defines a functions that is called with the pointing object as its argument.

A list starting with a symbol. In this case the car of the list is called with pointing object as its first argument, and the cdr forming the rest of the arguments, that is:

```
(apply (car accessor) pointing (cdr accessor)).
```

For example if the accessor is (slot-value name), the call is (slot-value *pointing* name), and

```
(aref 1 2) => (aref pointing 1 2).
```

setter If the setter is not given, it is computed by the system

using the accessor and the same expansion that setf would use. If it is given, it has the same properties as the accessor, except that in the call the *pointed* object is inserted before all the argument. That is, if the setter is (set-something name), the call is (set-something pointed pointing name). In addition, where the accessor accept a symbol, the setter also accept a function object.

remover Default t, which means use the setter. This is used to

remove the pointer from the pointing object. It is called exactly like the *setter*, except that the first argument is

dead-value, rather than pointed.

pointed This gives the value of the pointed object. If it is not

given, the accessor is used to get the pointed object.

dead-value (Default nil). This the value that is stored by the remover

in the pointing value before starting the shaking. Note that if the pointed object is shaken, the pointing object

is left with the dead-value.

Note that between the calls to the remover and the setter (steps 2 and 3 above), the pointing object points to the wrong thing (the *dead-value*). This may cause problems if the object is used by the system during the shaking (does not happen unless you access objects you should not), or if you define more than one delivery-shaker-weak-pointer on the same object, and one of these uses a slot that has been defined by the other. Thus you have to make sure that you do not cause this situtaion.

Examples One:

Suppose the keys of *my-hash-table* are conses of an object and a number, and it is desired to remove from *my-hash-table* those entries where the car is not pointed anywhere else. This can be done by something like this:

```
;;;; This will eliminate all the entries where the car is nil
```

```
(defun clean-my-hash-table (table)
  (maphash (lambda (x y) (declare (ignore y))
             (unless (car x) (remhash x table)))
         table))
(defun shake-my-hash-table ()
;;; this will cause the car of any entry where the car is not
;;; pointed to from another object to change to nil
 (maphash #'(lambda (x y) (declare (ignore y))
              (delivery-shaker-weak-pointer x 'car))
          *my-hash-table*)
     ;;; this will cause clean-my-hash-table to be called later
     ;;; in the shaking, provided *my-hash-table* is still alive.
(delivery-shaker-cleanup *my-hash-table* 'clean-my-hash-table))
(define-action "delivery actions" "shake my hash table"
            'shake-my-hash-table)
            ;; call this function at delivery time
;;;;-----
```

If the car can be nil, the code above removes some entries it should not. In this case the appropriate lines should be changed to:

```
(delivery-shaker-weak-pointer x 'car :dead-value 'my-dead-value))
and
  (when (eq(car x) 'my-dead-value) (remhash x table))
```

[Assuming there are no entries where the car is my-dead-value.]

Note that the cleanup function is not going to be called unless the hash table actually survives the shaking operation.

Example Two:

The value of *aaa* is a list of objects of type a-struct, which has a slot called name, which points to a symbol. We want to get rid of any of these structures if the symbol is not pointed to by some other object.

Implementation a:

Make the pointers from the structures to the names weak, and have the cleanup function throw away any structure where the name becomes nil.

```
(defun clean-*aaa* ()
  (loop for a on *aaa*)
(delivery-shaker-weak-pointer a 'a-struct-name))
  (delivery-shaker-cleanup
   '*aaa*
   #'(lambda(symbol)
       (set symbol
             (remove-if-not 'a-struct-name
               (symbol-value symbol ) )))))
(define-action "Delivery Actions" "Clean aaa" 'clean-*aaa*)
```

Implemenation b:

Make a pointer from the symbol to the structure, and make *aaa* point weakly to the names, and set *aaa* to nil. The remover and accessor do nothing, and the setter is defined to restore *aaa*. This implementation does not use the cleanup function.

```
(defun clean-*aaa* ()
  (let ((setter #'(lambda (name symbol)
                    (set symbol (ncons
                                (symbol-value symbol)
                                  (list(get name 'a-struct))) )
                                (remprop name 'a-struct))))
    (dolist (x *aaa* ()
      (let ((name (a-struct-name x)))
        (setf (get name 'a-struct) x)
        (delivery-shaker-weak-pointer '*aaa* nil
                                      :remover nil
                                      :pointed name
                                      :setter setter)))
    (setq *aaa* nil)))
(define-action "Delivery actions" "Clean aaa" 'clean-*aaa*)
```

Delivering CAPI Othello

This short example demonstrates how to deliver a small graphical application: an implementation of the board game Othello, with the graphical portion of it written using the CAPI library.

You can find the location of the code for this application in your LispWorks installation by evaluating the following form:

```
(sys:example-file "delivery/capi/othello/othello.lisp")
```

9.1 Preparing for delivery

With our ready-written application we can move straight to delivery. But first, try the application out in an ordinary image so that you can see what it does.

To do this:

- Create a directory called othello and copy the example file into it.
- **2.** Start up LispWorks and its environment.
- **3.** Compile and load the example file.

```
CL-USER 1 > (compile-file "othello.lisp" :load t)
[compilation messages elided]
```

4. Start up the application with the following form:

```
CL-USER 2 > (play-othello)
```

5. Play Othello!

Once you are familiar with this implementation of Othello, you can move on to delivery preparations.

9.1.1 Writing a delivery script

The next task is to create a delivery script. This is a Lisp file that, when loaded into the image, loads your compiled application code into the image, then calls the delivery function deliver to produce a standalone image.

The first delivery should be at delivery level 0. A successful delivery at this level proves that the code is suitable for delivery as a standalone application. After assuring yourself of this, you can look into removing code from the image to make it smaller.

If the delivered image is small enough for your purposes, there is no need to pursue a smaller image. An application delivered at level 0 contains a lot more in the way of debugging information and aids, and so is in some ways preferable to a leaner image.

The startup function in the Othello game is cL-user:play-othello. The initial delivery script therefore looks like this:

```
(in-package "CL-USER")
;; Load the compiled file othello. Should be in the same
;; directory as this script.
(load "othello")
;; Now deliver the application itself to create the image othello
(deliver 'play-othello "othello" 0 :interface :capi)
(quit)
```

Save this script in the newly created othello directory as script.lsp.

9.2 Delivering a standalone image

We now have a delivery script, enabling us to deliver the application as conveniently as possible. We can now try to deliver a simple, standalone image (with the delivery script having been set up to deliver at delivery level 0) to verify that the application can function standalone, before trying to make it smaller.

1. Run the image with the script as an initialization file:

```
lispworks-4-1-0 -init script.lsp
```

The script runs for a while, and as delivery proceeds a number of messages are printed. When it is finished, the image exits and there is an executable file called othello.exe in your current working folder on Windows, and othello in your working directory on UNIX.

2. Execute the othello file.

This should be a working, standalone Othello game.

See "How to deliver a standalone version of your application" on page 21 for a more detailed discussion of this part of the delivery process.

9.3 Making a smaller delivered image

Having delivered a standalone image successfully, we can look into delivering a smaller one. To do this we adjust the parameters passed to deliver in the delivery script. The typical approach is to experiment with parameters until you find a set that produces the smallest possible working image from your application.

There are many ways to make the image smaller, but the simplest is to increase the delivery level specified to the deliver function. See "How to deliver a smaller and faster application" on page 22 for more details.

9.3.1 Increasing the delivery level

Applications that do not use any of Common Lisp's more dynamic features (creating classes at runtime, evaluating arbitrary code) can usually be delivered all the way up to the maximum level of 5 without breaking. Our Othello game is one such application.

Try re-delivering the Othello game at different levels. Do this by editing your delivery script, changing the third argument to deliver to a number between 0 and 5 inclusive.

Index

A	and keeping 44
applications	console keyword 35
coding for efficient delivery 13–17	
name of delivered image file 20	D
standalone delivery 21–??	debugging and testing
automatic memory management. See gar-	checking an image without running it 55
bage collection.	in a delivered image 45
5	stub definitions for deleted functions 49
C	*delete-packages* list 36
call counting	:delete-packages keyword 36,77,79
all symbols in application 34	deleting and keeping
recording results of 34, 36	class accessors 56
setting up 34	classes 35
call-count keyword 34	classes, all 43
classes	CLOS 43
accessors 56	complex number representation 44
deleting and keeping 35, 43	debugger 45
delivery issues 24	documentation 45
dynamic definition 65	dspec table 51
keeping all 43	editor commands 38
printing information about 35	eval function 16
:classes-to-remove keyword 34	evaluator 45
CLOS 65-67	external symbols 57
deleting and keeping 43	fasl dumper 46
diagnostics 35	find-symbol function 16, 80-??
dynamic definition 65	format directives 39
method dispatch efficiency 65-68	function names
object printing code 44	functions 39
templates for method combinations 67	history of forms entered (Windows
clos-info keyword 35	only) 50
coding applications for efficient	listener top level 50 load function 48
delivery 13–17	macros 48
Common Lisp Object System. See CLOS.	methods, class-direct 56
:compact keyword 35	module facility 48
complex number representation, deleting	module facility 40

packages 25, 36, 76–77	deleting and keeping commands 38, 69-
packages, all 45	70
plist indicators 55	efficiency 13
structure internals 49	runtime code loading 14
stub definitions for deleted	See also size of the application.
functions 49	error handling ??-72
symbols in SAVED-STUFF package 56	application-generated errors 70, 71
walker 51	system-generated errors 70–71, 72
deliv-call-application	:error-on-interpreted-functions
function 54	keyword 38
deliver function 10, 20	eval function
delivered image	deleting and keeping 45
debugger 45	effects on size of application 16
module facility, deleting and	exceptions file. See generic functions.
keeping 48	:exit-after-delivery keyword 38
deliver-keywords function 27	: exit-message keyword 39
delivery 10, 19–26	exporting symbols from packages 39, 57
class issues 14, 24	exports keyword 39
diagnostics for all symbols 34	external symbols and delivery 57
function issues 14	₽
keywords for controlling 33–??	F
library dependencies, and 14	fasl dumper, deleting and keeping 46
Lisp interface to 10, 20, 33–??	file for call-count output 36
methods, and 24	find-symbol function
package issues 16, 17, 39, 48, 76	effects on application size 16, 80-??
preparation for 20–??	:format keyword 39
severity level 20, 23	function names, deleting and keeping
stages of 2, 22	functions
standalone applications 21–??	deleting and keeping 39
stub definitions for deleted	deliv-call-application 54
functions 49	deliver-keywords 27
symbol issues 14, 24, 76	names, deleting and keeping
system packages 77	save-image 35
treeshaking 24, 57	start-application 54
without running the application 55	stub definitions for deleted functions 49
without writing to disk 41	
diagnostics	:functions-to-remove keyword 39
all delivered symbols 34	G
CLOS usage 35	
:diagnostics-file keyword 36	garbage collection 3, 25
DLL delivery	delivery, and 24
:dll-exports keyword 36	heap compaction before delivery 35
documentation, deleting and keeping 45	See also treeshaking.
dspec table, deleting and keeping 51	generic functions
:dump-symbol-names keyword 38	class-direct methods 56
·ddiip symbol hames keyword so	collapsing into ordinary functions 39–41
E	excepting functions from collapse 41
_	method combination. See methods, dis-
:editor-commands-to-delete	patch efficiency.
keyword 38, 69	:generic-function-collapse
:editor-commands-to-keep	keyword 39
keyword 38	:gf-collapse-exceptions-file
editors	- -

keyword 40	:console 35
:gf-collapse-output-file	:delete-packages 36,77,79
keyword 41	:diagnostics-file 36
	:dump-symbol-names 38
H	:editor-commands-to-delete 38,
heap compaction before delivery 35	69
history list of forms entered	:editor-commands-to-keep 38
deleting and keeping (Windows	:error-on-interpreted-
only) 50	functions 38
3,	exit-after-delivery 38:
I	exit-message 39
:icon-file keyword 41	exports 39
:image-type keyword 41	:format 39
:in-memory-delivery keyword 41	:functions-to-remove 39
:interface keyword 42	:generic-function-collapse 39
intern function and application size 16,	:gf-collapse-exceptions-
24, 80	file 40
internal symbols and application size 57	:gf-collapse-output-file 41
Tr	:icon-file 41
K	:image-type 41
	:in-memory-delivery 41
<pre>:keep-bignum-numbers keyword 43 :keep-clos keyword 43,65</pre>	:interface 42
:keep-clos-object-printing	:keep-bignum-numbers 43
keyword 44	:keep-clos 43,65
:keep-complex-numbers keyword 44	:keep-clos-object-printing 44
:keep-debug-mode keyword 44	:keep-complex-numbers 44 :keep-debug-mode 44
:keep-documentation keyword 45	:keep-debug-mode 44 :keep-documentation 45
:keep-eval keyword 45	:keep-eval 45
:keep-fasl-dump keyword 46	:keep-fasl-dump 46
:keep-float-numbers keyword 46	:keep-float-numbers
:keep-function-name keyword 46,86	46
:keep-gc-cursor keyword 47	:keep-function-name 46,86
keeping. See deleting and keeping.	:keep-gc-cursor 47
:keep-lexer keyword 47	:keep-lexer 47
:keep-lisp-reader keyword 47	:keep-lisp-reader 47
:keep-load-function keyword 47	:keep-load-function 47
:keep-macros keyword 48,86	:keep-macros 48,86
:keep-modules keyword 48	:keep-modules 48
:keep-package-manipulation	:keep-package-manipulation 48
keyword 48	:keep-structure-info 49
:keep-structure-info keyword 49	:keep-stub-functions 49
:keep-stub-functions keyword 49	:keep-symbol-names 50
:keep-symbol-names keyword 50	: keep-symbols $50, 80, 86$
:keep-symbols keyword 50, 80, 86	:keep-top-level 50
:keep-top-level keyword 50	:keep-walker 51
:keep-walker keyword 51	:kill-dspec-table 51
keywords	:macro-packages-to-keep 51
call-count 34	:multiprocessing 51
:classes-to-remove 34	:packages-to-keep 52,79
clos-info 35	:packages-to-keep-symbol-
:compact 35	names 53

<pre>:print-circle 53 :product-code 53 :product-name 54 :query-end-session- handler 54 :quit-button 54 :quit-when-no-windows 54</pre>	class-direct, deleting and keeping 56 discriminating on classes 24 dispatch efficiency 65–68 dynamic definition 65 printing information about 35 MOP. See Meta-Object Protocol. :multiprocessing keyword 51
<pre>:redefine-compiler-p 55 :remove-plist-indicators 55</pre>	P
<pre>:remove-setf-function- name 55 :run-it 55 :save-symbols 56 :shake-class-accessors 56 :shake-class-direct-</pre>	package manipulation, deleting and keeping 48 packages deleting and keeping 25, 36, 76–77 deleting versus smashing 77–78 delivery 16, 17
methods 56 :shake-externals 57,80 :shake-shake-shake 24,57 :smash-packages 57,78,79 :startup-bitmap-file 58 :structure-packages-to-	exporting symbols from 39, 57 keeping 52, 79 keeping all 45 smashing 57, 76–77 :packages-to-keep keyword 52, 79
keep 58 symbol-names-action 58 symbols-to-keep-structure- info 60 symbols-to-precache-symbol- names 60	<pre>:packages-to-keep-symbol-names</pre>
:top-level-listener-p 60	Q
keywords for controlling delivery 33-?? severity level, and 23	<pre>:query-end-session-handler</pre>
:kill-dspec-table keyword 51	quit-when-no-windows keyword 54
L	R
libraries 2 dependencies between 14 effects on application size 14 listener top level 60	<pre>:redefine-compiler-p keyword 55 :remove-plist-indicators</pre>
deleting and keeping 50 load function, deleting and keeping 48 loading code at runtime 14 restrictions upon 48	keywords 55 :run-it keyword 55
M :macro-packages-to-keep keyword 51 macros, deleting and keeping 48 memory management. See garbage collection. Meta-Object Protocol 44 methods	S SAVED-STUFF package 56 save-image function 35 :save-symbols keyword 56 severity level of the delivery 20, 23 keyword parameters, and 23 :shake-class-accessors keyword 56 :shake-class-direct-methods keyword 56 :shake-externals keyword 57, 80

```
:shake-shake-shake keyword 24, 57
                                                        variable 67
shaking. See treeshaking.
                                                *windows-on* variable 54
size of the application
  intern function, and 16,80
  internal symbols, and 57
  interned symbols, and 24
  packages, and 17
smashing packages 57, 76–77
:smash-packages keyword 57, 78, 79
standalone applications. See delivery; secu-
        rity, separately licensed applica-
        tions; applications, standalone
        delivery.
start-application function 54
startup and shutdown
  shutdown when all windows closed 54
  startup function 20
  startup function, ignoring 55
:startup-bitmap-file keyword 58
structure internals, deleting and
        keeping 49
:structure-packages-to-keep
        keyword 58
stub definitions for deleted functions 49
:symbol-names-action keyword 58
symbols
  deleting and keeping 56, 79–80
:symbols-to-keep-structure-info
        keyword 60
:symbols-to-precache-symbol-
        names keyword 60
system packages and delivery 77
Т
:top-level-listener-p keyword 60
treeshaking 25
  garbage collection, and 24
  interned symbols, classes, functions,
      and 24
  Lisp interface to 57
type declaration and discrimination 15
\mathbf{V}
variables
  *warn-on-missing-templates* 67
  *windows-on* 54
:versioninfo keyword 60
W
walker, deleting and keeping 51
```

warn-on-missing-templates