Delivery User Guide

Version 6.0

Copyright and Trademarks

LispWorks Delivery User Guide

Version 6.0

November 2009

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839 (toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

1

Introduction

1.1 What does Delivery do?

Delivery allows you to take programs developed in LispWorks, and turn them into smaller, standalone applications, as executables or dynamic libraries. This process is called *delivery*.

The principle behind application delivery is quite simple: an application does not 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 (sometimes called a *runtime*) 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 file means it is simple to start up and can be run without running LispWorks as well.

1.2 What do you get with Delivery?

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

To deliver your application, you use the Application Builder tool in the LispWorks IDE, or run LispWorks directly with your build file which does all the necessary preparations (normally just loading patches and the application code) and then calls 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 and Reference Manual* for further details.

1.2.2 Functionality removed by delivery

The following general Lisp development functionality is forcibly removed by delivery:

- **compile-file**
- **save-image**
- **deliver**
- The graphical LispWorks IDE

Contact Lisp Sales if you want to build an application which uses these features.

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. A 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.

1.3.2 Platform-specific keywords

Some of the delivery parameters do not apply to all platforms. This is indicated where applicable:

Windows	means all supported Microsoft Windows operating systems.
Linux	means all supported Linux and FreeBSD operating systems.
x86/x64 Solaris	means all supported Solaris operating systems running on x86 or x64 hardware. It does not include SPARC hardware.
UNIX	means supported UNIX operating systems including SPARC Solaris and HP-UX, but not including Linux, FreeBSD, x86/x64 Solaris or Mac OS X.
DLL	means a Microsoft Windows dynamic link library.
Dynamic library	means a loadable dynamic shared library on any platform, including Windows DLLs.

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".

Read Chapter 6, "Delivery on Mac OS X", Chapter 7, "Delivery on Microsoft Windows" or Chapter 8, "Delivery on Linux, FreeBSD and Unix", as appropriate according to your target platform(s).

If you use CLOS, the FLI or the LispWorks editor in your application, you should also read Chapter 9, "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 Guide and Reference Manual* 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 and Reference Manual* 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 12.

deliver takes many keyword arguments for fine-tuning, but it is intended to work well with a minimal number of keywords. You should start by delivering with no more than the following keywords if required: **:interface :capi**, or **:multiprocessing t**. Only add other keywords when you find that they are needed.

You can also make LispWorks discard unused code, in order to reduce the delivered image size and thereby improve performance. You should not do this until your delivered application is working, though, because discarding certain code impedes debugging.

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. It is expected that you will need to use no more than a few delivery keywords: if you find that you use more than 6 delivery keywords, please contact Lisp Support with details.

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 Licensing issues

Runtime files that are created using Delivery with LispWorks for Windows, LispWorks for Linux, LispWorks for FreeBSD, LispWorks for x86/x64 Solaris

and LispWorks for Macintosh Professional and Enterprise Editions do not require a runtime license.

Runtime files generated by LispWorks (32-bit) for UNIX do require a LispWorks runtime license. See “Runtime licensing on UNIX” on page 7 for more information.

1.4.6 Modules

You should load all the Lisp modules that your application needs into the LispWorks image before attempting to deliver your application. Do this by calling `require` with each module name in your delivery script.

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 Section 9.4 on page 96 for more details.

1.4.8 Troubleshooting

Chapter 10, “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 12, “Delivering CAPI Othello”, shows how an example CAPI program can be delivered.

1.5 Runtime licensing on UNIX

This section applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

1.5.1 Protection of the delivery product on UNIX

This section applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

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

This section applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

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 runtime key. This substitution is controlled by the product code which Lisp Support will supply to your organization. (See *Reporting bugs* in the *LispWorks Release Notes and Installation Guide* for information on contacting Lisp 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 Lisp Support.

Unless you have made arrangements to the contrary, runtime licenses will be generated by the Lisp Support desk. Runtime 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 runtime keys are only transferrable from one machine to another upon payment of an administration charge.

All keys are specific to the major version of LispWorks for which they are issued. The current release is LispWorks 6.0. If you re-issue your application to

your end-users and base it on a different major version 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 runtime applications on UNIX

This section applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

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 Lisp Support to do so.

You may wish to make your own security arrangements or you may choose to leave the runtime 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 runtime applications on UNIX

This section applies only to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD, or x86/x64 Solaris).

Dated license keys used at delivery time when delivering an unprotected runtime affect the expiration date of that delivered runtime image.

Specifically, if any of

- the license key used by the delivery image upon startup, or
- the keys used when loading layered products

are dated, then the earliest expiration date of all such keys will be hard-wired into the runtime image. However, the LispWorks Delivery PLUS key itself does not affect the expiration date.

When you obtain undated keys for LispWorks or any layered product, it is therefore advisable to either delete or comment out any corresponding dated keys from that keyfile.

2

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 LispWorks distribution. To locate the pathname of the source file, evaluate the following form in a listener:

```
(example-file "delivery/hello/hello.lisp")
```

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!"))
```

Compile the file to a writable location, load it and test the program by calling `(hello-world)`.

2.2 Delivering the program

Having compiled the program, the next step is to attempt delivery, where you will load the compiled file in a fresh LispWorks session.

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 the **deliver** function in Chapter 4, "Delivering your Application".

Chapter 5, "Keywords to the Delivery Function" describes all the optional keyword arguments available.

In this example, we use just one of the optional keyword arguments, and of course we provide the mandatory arguments. These are:

- The name of a startup function. This is the first function called when the application is run.
- A pathname specifying where to write 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.

You can deliver and run the application in two ways: either use the LispWorks IDE, or use a command shell. This means a DOS command window (on Microsoft Windows), Terminal.app (Mac OS X) or a shell (Unix/Linux etc).

2.2.1 Delivering the program using the LispWorks IDE

You can use the Application Builder tool in the LispWorks IDE to deliver your application. This performs the same steps as described in "Delivering the program using a command shell" on page 13, but provides a windowing interface which is easier to use.

To start, you will need a script which loads your compiled application code. This can be as simple as

```
(in-package "CL-USER")
(example-compile-file "delivery/hello/hello" :load t)
```

but you can also start with a complete delivery script such as that shown in “Delivering the program using a command shell” on page 13.

For full instructions on using the Application Builder tool, see the *LispWorks IDE User Guide*.

2.2.2 Delivering the program using a command shell

Continuing with the example:

1. Write a delivery script file (`deliver.lisp`) that compiles and loads the program, and then calls `deliver`:

   ```
   (in-package "CL-USER")
   (load-all-patches)
   (example-compile-file "delivery/hello/hello" :load t)
   #+:cocoa
   (example-compile-file
     "configuration/macos-application-bundle" :load t)
   (deliver 'hello-world
            #+:cocoa
            (write-macos-application-bundle
             "~/Desktop/Hello.app"
             ;; Do not copy Lisp Source File
             ;;  association from LispWorks.app
             :document-types nil)
            #-:cocoa "~/hello"
            0
            :interface :capi)
   ```

2. Run the lisp image passing your file as the build script. For example, on Microsoft Windows open a DOS window. Ensure you are in the folder containing the LispWorks image and type:

   ```
   lispworks-6-0-0-x86-win32.exe -build deliver.lisp
   ```

   On UNIX, Linux or FreeBSD type the following into a shell:

   ```
   % lispworks-6-0-0-x86-linux -build deliver.lisp
   ```

   Note: the image name varies between the supported platforms.

On Mac OS X, use Terminal.app. Ensure you're in the directory of the image first:

```
% cd "/Applications/LispWorks 6.0/LispWorks.app/Contents/MacOS"
% ./lispworks-6-0-0-macos-universal -build 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.exe`** on Microsoft Windows, **`hello`** on UNIX/Linux/FreeBSD, and **`Hello.app`** on Mac OS X.

4. Now generate a smaller executable by discarding unused code while delivering. Do this by editing your file **`deliver.lisp`** to specify a higher level argument in the call to **`deliver`**. Try changing it to 5 for the largest effect.

2.2.3 Further example

There is another more detailed example at the end of the manual. This is in Chapter 12, and shows how to deliver a small CAPI application. The application is an implementation of the board game Othello.

3

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. Ask yourself: does one implementation technique tend to produce smaller and/or faster delivered images than another? Can I 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.

Note: 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 9.8.5 on page 103.

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 Avoid referencing type names

Referencing the name of a type (that is, a symbol) in code means that delivery cannot remove that type even if it is not used anywhere else. This is often seen in code using **`typep`**, **`typecase`** or **`subtypep`** to discriminate between types.

For example, if you have code like this:

```
(defun foo (x)
  (cond ((typep x 'class1) ...)
        ((typep x 'class2) ...)
        ...
        ((subtypep x 'class1000) ...)))
```

then delivery would keep all of the classes **`class1`**,**`...`**,**`class1000`** even if nothing else references these classes.

Possible solutions are described in “Referencing types via methods” on page 18 and “Referencing types via predicates” on page 18.

3.2.6.1 Referencing types via methods

Code can reference type names either directly as shown in "Avoid referencing type names" on page 17 or via **type-of** in code like this:

```
(defun foo (x)
  (let ((type (type-of x)))
    (cond ((eq type 'class1) ...)
          ((eq type 'class2) ...)
          ...
          ((eq type 'class1000) ...))))
```

Instead, you could express the conditional clauses as methods specialized for each class:

```
(defmethod foo ((x class1)) ...)
(defmethod foo ((x class2)) ...)
...
(defmethod foo ((x class1000)) ...)
```

This would allow any unused classes to be removed by delivery, because each method is a separate function.

3.2.6.2 Referencing types via predicates

If you do not wish to retain CLOS, and are referencing types that have built-in predicates, or structure types, you could use these predicates instead of the type names to allow delivery to remove unused types. For example this code:

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

could be rewritten as:

```
(cond ((integerp x) (process-an-integer x))
      ((stringp x) (process-a-string x))
      ((a-struct-p x) (process-a-struct x)))
```

3.2.7 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 106.

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

3.2.8 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.

3.2.9 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.

Note: 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 16.)

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*

Signature: **deliver** *function file level* **&rest** *keywords*

The following three arguments are required:

function The name of the function that starts an executable application.

file A string or pathname naming the file in which the delivered image should be saved.

The file extension **.exe** or **.dll** is appended to executables or DLLs delivered on Microsoft Windows.

If the delivery keyword **:split** is true then a second file containing the Lisp heap is created.

On Mac OS X, you may wish to create an application bundle containing your delivered image. For an example showing how to do this, see "Creating a Mac OS X application bundle" on page 129.

level An integer specifying the *delivery level*.

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.

The most important *keywords* arguments are **:interface** and **:multiprocessing**. If your application uses the CAPI, you must pass **:interface :capi**. If your application does not use the CAPI, but does use multiprocessing, then you must pass **:multiprocessing t**. Your first attempt to deliver your application should use no more than these keywords.

In addition, a variety of other keywords can be passed to **deliver**. These are for fine-tuning by controlling aspects of delivery explicitly. Add more keywords only when you find that you need them.

All the **deliver** keywords are documented in Chapter 5, "Keywords to the Delivery Function". Additionally, they can been seen in the LispWorks image by calling:

```
(require "delivery")
(deliver-keywords)
```

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 User Guide and 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

```
(deliver 'run
         "the-application"
         0
         :in-memory-delivery t)
```

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

4.3 Delivering a standalone application executable

There are usually two considerations when delivering an application.

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

We recommend delivering a standalone executable 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 does remove the LispWorks development tools as described in “Functionality removed by delivery” on page 2. To do this modify your **`deliver.lisp`** script from “Delivering the program” on page 12 as appropriate to your application:

```
(load-all-patches)
(load-my-application)
;;; unless you have it already loaded as suggested in
;;;“Saving the image before attempting delivery” on page 23
(deliver 'my-function "my-program" 0 :interface :capi)
```

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 pass **`:multiprocessing t`**:

```
(deliver ‘my-function "my-program" 0 :multiprocessing t)
```

Then run LispWorks with **`deliver.lisp`** as a build script. You can do this using the graphical Application Builder tool (see “Delivering the program using the LispWorks IDE” on page 12) or in a command window, like this:

- On Microsoft Windows, open a DOS window and enter:

```
MS-DOS> lispworks-6-0-0-x86-win32.exe -build deliver.lisp
```

- On UNIX, Linux and FreeBSD systems, enter a command line like this in a shell:

```
% lispworks-6-0-0-x86-linux -build deliver.lisp
```

  Note: the image name varies between the supported platforms.

- On Mac OS X, use Terminal.app:

```
% ./lispworks-6-0-0-macos-universal -build deliver.lisp
```

This creates an executable in **my-program.exe** on Microsoft Windows, or **my-program** on UNIX/Linux/FreeBSD/Mac OS X. When this executable starts, it calls **my-function** without arguments.

4.4 Delivering a dynamic library

Depending on how your application needs to interoperate with other software, you may want to build it as a DLL (also referred to as a dynamic library) rather than an executable.

4.4.1 Simple delivery of a dynamic library

Supply the names of your library's exports in a list value for the **deliver** keyword **:dll-exports**. Each name in *dll-exports* should be a string naming a Lisp function defined by **fli:define-foreign-callable**.

The **deliver** *function* argument should be **nil**, because a dynamic library does not have a startup function.

Supply the file type of the delivered image in the **deliver** *file* argument if necessary.

As when delivering a LispWorks executable, start at **deliver** *level* 0. Increase the delivery level, if desired, after you have debugged your library. Whenever possible, debug your code running in the LispWorks development image. If the problem only occurs when your code runs inside a dynamic library, you may be able to debug it on your development machine in a dynamic library created by **save-image** rather than **deliver**.

4.4.2 Using the dynamic library

A Microsoft Windows application should use **LoadLibrary** to load the DLL and **GetProcAddress** to find the address of the exported names. On other platforms the application should use **dlopen** and **dlsym**.

For more information about the behavior of LispWorks dynamic libraries see the chapter "LispWorks as a dynamic library" in the *LispWorks User Guide and Reference Manual*.

4.4.3 Simple Windows example

The script below creates **hello.dll**.

```
-------------------- hello.lisp -------------------------
(in-package "CL-USER")
(load-all-patches)
;; The signature of this function is suitable for use with
;; rundll32.exe.
(fli:define-foreign-callable ("Hello"
                              :calling-convention :stdcall)
    ((hwnd w:hwnd)
     (hinst w:hinstance)
     (string :pointer)
     (cmd-show :int))
  (capi:display-message "Hello world")
  ;; quit when library's job is done
  (dll-quit))

(deliver nil "hello" 0 :dll-exports '("Hello") :interface :capi)
---------------------------------------------------------
```

You can build the DLL with this command line:

```
lispworks-6-0-0-x86-win32.exe -build hello.lisp
```

and you can test it with this command line:

```
rundll32 hello.dll,Hello
```

4.4.3.1 Using the Application Builder

The Application Builder tool provides another way to build and test **hello.dll**:

1. In the LispWorks for Windows IDE do **Works > Tools > Application Builder**
2. Set the Build script to be your file `hello.lisp` and do **Works > Build > Build** to build the DLL.
3. Do **Works > Build > Run With Arguments**. Enter `rundll32` in the Execute pane, enter `hello.dll,Hello` in the Arguments pane, and press **OK** to test the library.

4.4.4 Further example

This example builds a dynamic library which in principle could be loaded by any application and called to calculate square numbers.

For illustrative purposes, we show how to load the dynamic library into the LispWorks development image. This illustrates some platform-specific initialization. Then we use the library, ensure it exits cleanly, and finally delete the dynamic library file.

Note that on Linux / Macintosh / FreeBSD, to deliver a dynamic library, the build machine must have a C compiler installed.

For convenience the code is presented without external files. To run it, copy each form in turn and enter it at the Listener prompt.

1. Define a path for the dynamic library:

```
(defvar *dynamic-library-path*
  (merge-pathnames (make-pathname :name "CalculateSquareExample"
                                  :type scm::*object-file-suffix*)
                   (get-temp-directory)))
```

2. Define a function to create the dynamic library:

```
(defun save-dynamic-library ()
  (let* ((file (make-temp-file t "lisp"))
         (ns (namestring file)))
    (format file
      "
        (fli:define-foreign-callable (calculate-square :result-
type :int)
            ((arg :int))
          (* arg arg))
    (deliver nil ~s 5 :dll-exports '(\"calculate_square\"))"
      (namestring *dynamic-library-path*))
    (close file)
    (sys:call-system-showing-output (list (lisp-image-name)
                                          "-build"
                                         ns ))
    (delete-file file nil)))
```

3. Create the dynamic library:

```
(save-dynamic-library)
```

4. Define functions to use the dynamic library:

```
(fli:define-foreign-function (my-quit-lispworks "QuitLispWorks")
    ((force :int)
     (milli-timeout :int))
  :result-type :int
  ;; specifying :module ensures the foreign function finds
  ;;  the function in our module
  :module 'my-dynamic-library)
(fli:define-foreign-function (my-init-lispworks "InitLispWorks")
    ((milli-timeout :int)
     (base-address (:pointer-integer :int))
     (reserve-size (:pointer-integer :int)) ; really size_t
     )
  :result-type :int
  :module 'my-dynamic-library)
(fli:define-foreign-function calculate-square
    ((arg :int))
  :result-type :int
  :module 'my-dynamic-library)
```

5. Define a function to load the dynamic library, use it, and then unload it:

```
(defun run-the-dynamic-library ()
  (fli:register-module 'my-dynamic-library
                       :connection-style :immediate
                       :file-name *dynamic-library-path*)
;; Windows and Mac OS X can detect and resolve memory clashes.
  ;; On other platforms, tell the library to load at different
  ;;  address (that is, relocate) because otherwise it will use
  ;;  the same address as the running LispWorks development image.
  ;; Relocation may be needed when loading a LispWorks dynamic
  ;; library in other applications.
  #-(or mswindows darwin)
  (my-init-lispworks 0
                     #+lispworks-64bit #x5000000000
                     #+lispworks-32bit #x50000000
                     0)
  (dotimes (x 4)
    (format t "square of ~d = ~d~%" x
            (calculate-square x)))
  (my-quit-lispworks 0 1000)
  (fli:disconnect-module 'my-dynamic-library))
```

6. Use the dynamic library:

   ```
   (run-the-dynamic-library)
   ```

   Check the output to see that it computed square numbers.

7. (optional) Delete the dynamic library file:

   ```
   (delete-file *dynamic-library-path* nil)
   ```

4.4.5 More about building dynamic libraries

On Macintosh / Linux / FreeBSD / Unix you can supply files to be included in the library via the **deliver** keyword argument **:dll-added-files**. This is useful if you need to write wrappers around calls into the library.

You can specify whether your LispWorks dynamic library initializes itself automatically on loading with the **deliver** keyword argument **:automatic-init**. For more information see "Initialization of the dynamic library" in the *LispWorks User Guide and Reference Manual*.

4.5 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 removed.

What can be removed 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.

4.5.1 Making the image smaller

You can specify that the image be made smaller in two complementary 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 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 achieved by simply 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 argu-

ments 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 52 for a discussion of this keyword.)

4.6 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.

4.6.1 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 generally 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.

4.6.2 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 68.

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:

1. 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.)

5

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 11.2 on page 120.

Note: Delivery is designed to work well with a small number of delivery keywords only. Start attempting delivery by passing no keywords, or **:interface :capi**, or **:multiprocessing t**, as required. Only add other keywords when you find that you need them. If you are passing more than 6 delivery keywords, please contact Lisp Support with details.

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 delivery levels. In the descriptions of the default values of deliver keywords in “Alphabetical list of deliver keywords” on page 39, the level appears as the symbol ***delivery-level***.

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`**
- **`:clean-for-dump-type`**
- **`:console`**
- **`:dll-exports`**
- **`:editor-style`**
- **`:icon-file`**
- **`:image-type`**
- **`:interface`**
- **`:interrogate-symbols`**
- **`:interrupt-function`**
- **`:keep-gc-cursor`**
- **`:license-info`**
- **`:multiprocessing`**
- **`:product-code`**
- **`:product-name`**
- **`:quit-when-no-windows`**
- **`:redefine-compiler-p`**
- **`:registry-path`**
- **`:split`**
- **`: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`
- `:post-delivery-function`
- `: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`
- `:warn-on-missing-templates`

5.1.3 Behavior of the delivery process

The following keywords control the behavior of the delivery process itself. They do not affect the delivered application's behavior or the debugging information generated.

- `:display-progress-bar`

5.1.4 Retaining or removing functionality

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.

5.1.4.1 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`

5.1.4.2 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`
- `:generic-function-collapse`
- `:gf-collapse-output-file`
- `:gf-collapse-tty-output`
- `: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`

5.1.4.3 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-foreign-symbols**
- **: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**

5.1.4.4 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**

5.1.4.5 CLOS metaclass compression

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

5.1.4.6 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`

5.1.4.7 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-macros`
- `:macro-packages-to-keep`
- `:remove-setf-function-name`

5.1.4.8 Numbers

The following keywords are for keeping and for removing code from the image that can handle certain numerical types:

- `:keep-complex-numbers`
- `:numeric`

5.1.4.9 Conditions deletion

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

- **:condition-deletion-action**
- **:keep-conditions**
- **:packages-to-remove-conditions**

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 value: **nil**

GTK and Motif applications only: if the application uses the X11 code or CAPI, it may fail to run if it cannot open the X display.

In this case, if the value is a function it calls this function with one argument, the display name. The default value of **nil** means that a message is printed and Lisp quits.

: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***.

:automatic-init *Keyword*

Default value: **t** on Microsoft Windows, **nil** on other platforms

automatic-init specifies whether a LispWorks dynamic library should initialize automatically on loading. Automatic initialization is useful when the dynamic library does not communicate by function calls but prevents you from relocating the library if necessary or doing other initialization.

To deliver a dynamic library on Linux / Macintosh / FreeBSD, the build machine must have a C compiler installed. This is typically **gcc** (which is available on the Macintosh by installing Xcode).

deliver uses *automatic-init* just like **save-image**. See **save-image** in the *LispWorks User Guide and Reference Manual* for more details.

For more information about the behavior of LispWorks DLLs (dynamic libraries) and in particular a discussion of automatic and explicit initialization, see the chapter "LispWorks as a dynamic library" in the *LispWorks User Guide and Reference Manual*.

: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 **nil**, 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 **fboundp**, 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 written to a file or the standard output. You can specify its name with **:diagnostics-file**.

:classes-to-keep-effective-slots *Keyword*

Default value: **nil**

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.

:clean-down *Keyword*

Default value: **t**

If true, call **clean-down** before saving the image.

:clean-for-dump-type *Keyword*

Default value: **:user**

Related to the **:type** argument of **save-image**. This is for expert use only - please consult Lisp Support before using.

:clos-info *Keyword*

Default value: **nil**

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 the file given by **:diagnostics-file**.

:compact *Keyword*

Default value:

```
(and (not (delivery-value :keep-debug-mode))
     (not (delivery-value :interrogate-symbols))
     (eq (delivery-value :dll-exports) :no))
```

x86 platforms only: If this is non-nil, the heap is compacted just before the delivered image is saved, with all functions being made static. This usually gives the greatest size reduction in delivery. You may want to leave this until the final delivery if you are using a slow machine on which this operation takes some time.

:condition-deletion-action *Keyword*

Default value: **(when (> *delivery-level* 0) :delete)**

The value is one of:

nil Do not delete any condition class. This is the default at delivery level 0.

:delete 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 **format-arguments** slot. This is the default at delivery level > 0.

:redirect Redirect unwanted conditions to the first parent in their hierachy which is not deleted.

See “Deleting of condition classes” on page 98.

:console *Keyword*

Default value: **:default**

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

:delete-packages *Keyword*

Default value: **nil**

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). The value **nil** means write to ***standard-output***.

Compatibility Note: In LispWorks 4.4 and previous on Windows and Linux platforms, the default value of **:diagnostics-file** was "**dvout.txt**". The default value is now **nil** on all platforms.

:display-progress-bar *Keyword*

Default value: **t**

Windows only: by default a progress bar is displayed during the delivery process. If the value of the **:display-progress-bar** keyword is false, it does not display a progress bar.

Compatibility Note: In LispWorks for Windows 4.4 and previous, there was no way to prevent the display of the progress bar.

:dll-added-files *Keyword*

Default value: **nil**

Unix/Linux/FreeBSD and Macintosh only: A list value means that the saved image is a dynamic library file rather than an executable. The build machine must have a C compiler installed.

If non-nil, *dll-added-files* should be a list of filenames and then a dynamic library containing each named file is saved. Each file must be of a format that the default C compiler can incorporate into a shared library and must not contain exports that clash with predefined exports in the LispWorks shared library. The added files are useful to write wrappers around calls into the LispWorks dynamic library.

deliver uses *dll-added-files* just like **save-image**. See **save-image** in the *LispWorks User Guide and Reference Manual* for more details.

For more information about the behavior of LispWorks DLLs (dynamic libraries) see the chapter "LispWorks as a dynamic library" in the *LispWorks User Guide and Reference Manual*.

:dll-exports *Keyword*

Default value: **:default**

dll-exports is implemented only on Windows, Linux, x86/x64 Solaris, Macintosh and FreeBSD. It controls whether the image saved is an executable or a dynamic library (DLL). If *dll-exports* is **:default**, the delivered image is an executable. The value **:com** is supported on Microsoft Windows only (see below). Otherwise *dll-exports* should be list (potentially **nil**). In this case a dynamic library is saved, and each string in **dll-exports** names a function which becomes an export of the dynamic library and should be defined as a Lisp function using **fli:define-for-**

eign-callable . Each exported name can be found by **GetProcAddress** (on Windows) or **dlsym** (on other platforms). The exported symbol is actually a stub which ensures that the LispWorks dynamic library has finished initializing, and then enters the Lisp code.

On Microsoft Windows *dll-exports* can also contain the keyword **:com**, or *dll-exports* can simply be the keyword **:com**, both of which mean that the DLL is intended to be used as a COM server. See the *LispWorks COM/ Automation User Guide and Reference Manual* for details.

To deliver a dynamic library on Linux / Macintosh / FreeBSD, the build machine must have a C compiler installed. This is typically **gcc** (which is available on the Macintosh by installing Xcode).

On Mac OS X the default behavior is to generate an object of type "Mach-O dynamically linked shared library" with file type dylib. See *image-type* below for information about creating another type of library on Mac OS X.

ON Microsoft Windows you can use **LoadLibrary** from the main application to load the DLL and **GetProcAddress** to find the address of the external names.

There is an example DLL delivery script in “Delivering a dynamic library” on page 25.

For more information about the behavior of LispWorks DLLs (dynamic libraries) see the chapter "LispWorks as a dynamic library" in the *LispWorks User Guide and Reference Manual*.

: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 “Editors for delivered applications” on page 93 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 “Editors for delivered applications” on page 93 for more details.

:editor-style *Keyword*

Default value **:default**

This controls the editor emulation style used in **capi:editor-pane** (and subclasses) in the delivered image.

The value should be one of:

:emacs	Use Emacs emulation.
:pc	Use Microsoft Windows emulation on Windows, and KDE/Gnome style keys on GTK and Motif.
:mac	Use Mac OS X editor emulation.
:default	Use the default emulation style for the current platform. That is, use **:pc** on Microsoft Windows, **:mac** on Mac OS X/Cocoa and **:emacs** on GTK and Motif.
nil	Use the default setting on the current machine.

Note that not all emulation styles are supported on all platforms. See the the "Emulation" chapter of the *LispWorks Editor User Guide* for details about the different emulation styles.

:error-handler *Keyword*

Default value: **nil**

The value **:btrace** changes error handling, so that a simple backtrace is generated whenever **error** is called.

: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.

:exe-file *Keyword*

On Microsoft Windows, used as the basis for the new executable. This is for expert use only - please consult Lisp Support before using.

: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:

~ | R P O G E C B ? < / W $

The value can also be a list of directives to keep. The elements of the list should be Lisp characters corresponding to (some of) the format directives above.

: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:

```
(and (>= *delivery-level* 3)
     (not (member (delivery-value :keep-clos)
                  '(t
                    :full-dynamic-definition
                    :method-dynamic-definition))))
```

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.

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-output-file *Keyword*

Default value: **nil**

If the value is a string, it 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 report of generic function collapsing to the console.

:icon-file *Keyword*

Default value: **(if (eq (delivery-value :console) t) nil :default)**

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

Note: to achieve the same effect on Mac OS X, do not pass **:icon-file,** but put your delivered image in a suitable application bundle which contains the application icon. See the examples in the LispWorks library directory **examples/delivery/macos/**.

:image-type *Keyword*

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

image-type defines whether the image is to be an executable or a dynamic library. The value can be **:exe**, **:dll** or **:bundle**. It defaults to **:exe** or **:dll** according to the value of *dll-exports* and therefore you do not normally need to supply *image-type*.

image-type **:bundle** is used only when saving a dynamic library. On Mac OS X it generates an object of type "Mach-O bundle" and is used for creating shared libraries that will be used by applications that cannot load dylibs (FileMaker for example). It also does not force the filename extension to be **dylib**. On other Unix-like systems *image-type* merely has the effect of not forcing the filename extension of the delivered image, and the format of the delivered image is the same as the default. On Microsoft Windows *image-type* **:bundle** is ignored.

On Linux / Macintosh / FreeBSD *image-type* **:bundle** requires that the build machine has a C compiler installed. This is typically **gcc** (available by installing Xcode on the Macintosh).

Note: *image-type* **:bundle** is completely unrelated to the Mac OS X notion of an application bundle.

: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**.

:interface *Keyword*

Default value: **nil**

Set this to **:capi** for applications that use the CAPI.

Because the CAPI uses multiprocessing, **:interface :capi** also sets the **deliver** keyword **:multiprocessing** to **t**.

:interrogate-symbols *Keyword*

Default value: **nil**

When non-nil this does two things:

First it loads the **reverse-pointers-code** module. This can be used to check what things to keep in the image. If you need documentation for **reverse-pointers-code**, please contact Lisp Support.

Secondly it sets the image up such that calling the application with command line argument **-interrogate-symbols**, before starting the application, allows you to **interrogate-symbols**. See “Interrogate-Symbols” on page 115.

:interrupt-function *Keyword*

Default value: **t**

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

:keep-clos *Keyword*

Default value:

```
(if (= *delivery-level* 0)
    :full-dynamic-definition
  (if (= *delivery-level* 1)
      :method-dynamic-definition
    :no-dynamic-definition))
```

If this is **:no-dynamic-definition**, then the functions for dynamic class and method definition are deleted -- **defmethod** , **defclass** and so on and the direct slots and direct methods slots all classes are set to **nil**.

If the value of the **:keep-clos deliver** keyword is **nil**, then it is treated as **:no-dynamic-definition**.

If it is **:meta-object-slots**, then the direct slots and direct methods of all classes are retained, and the dynamic definition functionality is deleted.

If it is **:method-dynamic-definition**, nothing is smashed or deleted, though the direct slots and direct methods of all classes are emptied. With this setting, methods can be defined dynamically but not classes.

If it is **:full-dynamic-definition** or **t**, then all dynamic class and method definition is allowed.

Compatibility Note: In LispWorks 4.3 and previous versions the values **:no-empty** and **:no-empty-no-dd** were documented for the **:keep-clos deliver** keyword. These values are still accepted in LispWorks 6.0, but you should not rely on this. Change to one of the new values described above.

Note: CLOS **make-instance** initarg checking in the delivered application may be controlled by **:make-instance-keyword-check**.

:keep-clos-object-printing *Keyword*

Default value:

```
(or (delivery-value :keep-debug-mode)
    (<= *delivery-level* 2))
```

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

```
(defun x-print-object (object stream)
  (t-print-object object stream))
```

```
(defun t-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)))
        (progn
          (write-string "Undefined function " stream)
          (prin1 (cell-error-name object) stream))
      (progn
        (princ (or (ignore-errors (type-of object))
                   "<Unknown type>")
               stream)
        (ignore-errors
          (when-let (namer (find-symbol "NAME" "CLOS"))
            (when-let (name (and (slot-exists-p object namer)
                                 (slot-boundp object namer)
                                 (slot-value object namer)))
              (format stream " ~a" name))))))))
```

You may redefine **x-print-object**.

Affected by: **:keep-debug-mode**

:keep-complex-numbers *Keyword*

Default value: **(delivery-value :numeric)**

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

Compatibility Note: This keyword has an effect on all platforms in LispWorks 5.0 and later. It has no effect in LispWorks 4.4 and previous on Windows and Linux platforms.

Affected by: **:numeric**

:keep-conditions *Keyword*

Default value: **nil**

The value should be one of:

:none Eliminate all conditions.

:minimal	Keep only the conditions that are in the class-precedence-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.

See “Deleting of condition classes” on page 98.

:keep-debug-mode *Keyword*

Default value: **(> 5 *delivery-level*)**

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

On Unix, if the value is **:all**, all debug information is kept

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

The value of **:keep-debug-mode** affects the default value of the following keywords to:

```
:compact
:keep-clos-object-printing
:keep-eval
:keep-function-name
:keep-lisp-reader
:keep-load-function
:keep-structure-info
:keep-top-level
:make-instance-keyword-check
:no-symbol-function-usage
:packages-to-keep-symbol-names
```

:keep-documentation *Keyword*

Default value: `(= *delivery-level* 0)`

If non-nil, documentation strings in the image are 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:

```
(or (delivery-value :keep-debug-mode)
    (< *delivery-level* 4))
```

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

:keep-fasl-dump *Keyword*

Default value: `nil`

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

:keep-foreign-symbols *Keyword*

Default value: **nil**

UNIX only: If this is **nil**, the code and information that is required for dynamic loading of foreign code is eliminated from the image.

The value can be a list of strings which are the foreign symbols to keep. The value can also be **t**, meaning keep all foreign symbols.

:keep-function-name *Keyword*

Default value:

```
(if (delivery-value :shake-shake-shake)
    (if (delivery-value :keep-debug-mode) t nil)
   :all)
```

This keyword controls the retention of names for functions. The following values are accepted:

nil	Do not keep names
:minimal	Keep names as strings, but keep no other debug information
t	Keep names as strings and retain argument information.
:all	Do not modify function names

On x86 platforms, if **:call-count** is either **t** or **:all**, then **:keep-function-name** is set to **t** automatically.

When **:keep-debug-mode** is non-nil, **:keep-function-name** is set to **t** automatically.

Affected by: **:keep-debug-mode**, **:shake-shake-shake**

Compatibility Note: In LispWorks 4.4 and previous on Windows and Linux platforms, if the keyword **:compact** is non-nil, function names are eliminated. This is not true in LispWorks 5.0 and later versions.

:keep-gc-cursor *Keyword*

Default value: **nil**

Windows 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-lisp-reader *Keyword*

Default value:

```
(or (delivery-value :keep-debug-mode)
    (< *delivery-level* 5))
```

If the value is **nil**, 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:

```
(when (or (delivery-value :keep-debug-mode)
          (delivery-value :keep-modules)
          (<= *delivery-level* 2))
  :full)
```

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:

t Keeps the loading code required to load data files.

: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**.)

:keep-macros *Keyword*

Default value: **(< *delivery-level* 2)**

If this is **nil**, the functions **macroexpand**, **macroexpand-1** and **macro-function** 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: **(< *delivery-level* 1)**

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

:keep-package-manipulation *Keyword*

Default value: **(< *delivery-level* 2)**

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

:keep-pretty-printer *Keyword*

Default value: **nil**

If **nil** the **pprint** functionality is eliminated.

:keep-structure-info *Keyword*

Default value:

```
(or (delivery-value :keep-debug-mode)
    (case *delivery-level*
      ((0 1) t)
      (2 :print)
      (otherwise nil)))
```

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
  NIL.
  Try :KEEP-COMPLEX-NUMBERS T.
```

:keep-symbol-names *Keyword*

Default: **nil**

A list of symbols that must retain their symbol names.

:keep-symbols *Keyword*

Default value: **nil**

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.

:keep-top-level *Keyword*

Default value:

```
(or (< *delivery-level* 5) (delivery-value :keep-debug-mode))
```

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

Note: the top level history is cleared, regardless of the value of the **:keep-top-level** argument.

Affected by: **:keep-debug-mode**

:keep-trans-numbers *Keyword*

Default value: **(delivery-value :numeric)**

If this is **nil**, eliminate transcendental functions (for example **sin**).

Compatibility Note: This keyword has an effect on all platforms in LispWorks 5.0 and later. It has no effect in LispWorks 4.4 and previous on Windows and Linux platforms.

Affected by: **:numeric**

:keep-walker *Keyword*

Default value: **nil**

If this is **nil**, the walker is deleted.

:kill-dspec-table *Keyword*

Default value: **(> *delivery-level* 0)**

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 an implicit call to **untrace**, and previous uses of **trace** and **defadvice** are discarded.

:license-info *Keyword*

Default value: **nil**

This keyword is obselete. Was previously used to pass license information for products on certain platforms.

:macro-packages-to-keep *Keyword*

Default value: **nil**

A list of package names. Symbols in these packages that have a macro definition are not **fmakunbound** when the delivery process deletes 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.

:make-instance-keyword-check *Keyword*

Default value: **(if (delivery-value :keep-debug-mode) :default nil)**

The value of the **:make-instance-keyword-check** keyword controls whether **make-instance** checks its initargs in the delivered application.

If the value is **nil**, then **make-instance** checks are switched off. If the value is **t**, then **make-instance** checks are switched on.

If the value is **:default**, the **make-instance** checks are not affected by the delivery process. See the function **clos:set-make-instance-argument-checking** for instructions on controlling **make-instance** checks in this situation.

Compatibility note: In LispWorks 5.1 and previous versions, the value **t** of **:keep-clos** overrides the effect of **:make-instance-keyword-check**. In LispWorks 6.0 **:make-instance-keyword-check** always affects the behavior in the delivered application, regardless of **:keep-clos**.

Compatibility note: In LispWorks 5.1 and previous versions, a true value of **:keep-debug-mode** would always switch the checks on. In LispWorks 6.0 **:keep-debug-mode** retains the current setting of **make-instance** checks, rather than forcing the checks to be switched on.

Affected by: **:keep-debug-mode**

:manifest-file *Keyword*

Default value: **nil**

Windows only. Overrides the default application manifest, which can affect whether an executable application is themed.

If the value is a string it must name a file that is a legal application manifest containing the "dependency" element for Microsoft.VC80.CRT. If the value is the keyword **:no-common-controls-6** a manifest without the element for common controls is used. If the value is **nil**, then the LispWorks manifest is used.

See “Application Manifests” on page 82 for more information about Windows application manifests in LispWorks applications.

:metaclasses-to-keep-effective-slots *Keyword*

Default value:

```
(when (member (delivery-value :keep-clos)
              '(t :full-dynamic-definition))
  :all)
```

If the value is a list, the elements are metaclasses whose classes retain their effective-slot-definitions. Value **:all** means all metaclasses.

:multiprocessing *Keyword*

Default value: **nil**

If set to **t**, starts multiprocessing with the delivery function (that is, the first argument to **deliver**) running in a process created specially for it.

If set to **:manual**, allows multiprocessing to be started by the delivery function, which should call **mp:initialize-multiprocessing**.

If set to **nil**, multiprocessing cannot be used in the delivered application.

The value of this keyword argument is automatically **t** when **:interface** is **:capi**, so you only need to supply 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 value: **(not (delivery-value :keep-debug-mode))**

x86 platforms only: eliminates symbols that are used only for function calls.

See “Debugging with :no-symbol-function-usage” on page 115 for information about debugging an image where these symbols have been eliminated.

:numeric *Keyword*

Default: **t**

Keep all numeric operations, unless overridden by **:keep-complex-numbers**.

Compatibility Note: This keyword has an effect on all platforms in LispWorks 5.0 and later. It has no effect in LispWorks 4.4 and previous on Windows and Linux platforms.

:packages-to-keep *Keyword*

Default value: **nil**

This keyword takes a list of packages to be retained. All packages in the list are kept in the delivered image, regardless of the values of the **:smash-packages** 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 106.

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

Note: When you keep a package by **:packages-to-keep**, this does not cause that package's symbols to be kept. To retain symbols, see "Ensuring that symbols are kept" on page 105.

:packages-to-keep-externals *Keyword*

Default value: **nil**

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.

The externals of the **setf** package are always retained, regardless of the value of **:packages-to-keep-externals**.

:packages-to-keep-symbol-names *Keyword*

Default value:

```
(if (or
     (delivery-value :keep-debug-mode)
     (< *delivery-level* 5))
    :all
  nil)
```

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**.

The value can also be **:all**, meaning all packages.

:packages-to-remove-conditions *Keyword*

Default value: **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 pass **:keep-conditions :all** (or **:condition-deletion-action nil**). To eliminate all conditions, you should do **:keep-conditions :none**.

See "Deleting of condition classes" on page 98.

:packages-to-shake-externals *Keyword*

Default value: **nil**

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.

The externals of the **keyword** package are always shaken, regardless of the value of **:packages-to-shake-externals**.

:post-delivery-function *Keyword*

Default value: **nil**

When non-nil, the value *post-delivery-function* should be a function designator for a function of one argument:

post-delivery-function *successp*

The system calls *post-delivery-function* after delivery. *successp* is true if delivery was successful and false otherwise.

Note: during the delivery process, the Lisp system can be in an unstable state, so it is not always possible to recover when delivery is not successful.

:print-circle *Keyword*

Default value:

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

When this is **nil**, 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 Lisp Support. 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 Lisp Support. If **:product-code** is not supplied then the image is not re-targeted and will require a "LispWorks Delivery" key to restart. Note that this should not be a problem while developing an application.

:product-name *Keyword*

Default value: **nil**

On UNIX only the value *product-name* is used in keyfile error messages to identify a product whose key is incorrect. If it is not supplied then *product-name* defaults to **"Anonymous Application"**.

On Microsoft Windows only *product-name* has an entirely different interpretation: it provides the name that is used in CAPI dialogs which have no specific title or owner.

On other platforms, *product-name* is ignored.

:quit-when-no-windows *Keyword*

Default value: **t**

If **t**, then after the application has opened at least one CAPI window, whenever the application is waiting for input, a routine is run to check whether any of its CAPI windows are still open. If there are no open windows, the application exits.

Note: a multiprocessing LispWorks executable will stop multiprocessing when there is no process other than the Idle Process. So if your application simply displays a window, which is closed, then multiprocessing will stop. This is independent of *quit-when-no-windows*.

:redefine-compiler-p *Keyword*

Default value: **(>= *delivery-level* 1)**

When this is true, the function **compile** is eliminated from the image.

Note: the function **compile-file** is always removed by delivery, regardless of **:redefine-compiler-p**.

:registry-path *Keyword*

Path for storing user preferences.

On Microsoft Windows this is relative to HKEY_CURRENT_USER.

On UNIX / Linux / FreeBSD / Mac OS X, this relative to the user home directory.

Note: see “Delivery and CAPI” on page 95 for information on a possible problem with delivered applications that record window geometries in the registry.

:remove-plist-indicators *Keyword*

Default value: **nil**

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

:remove-setf-function-name *Keyword*

Default value: **(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 database, but the database has not been started up. In such cases, set it to **nil**.

:shake-class-accessors *Keyword*

Default value:

```
(cond ((>= *delivery-level* 4) :remove)
      ((>= *delivery-level* 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: `(>= *delivery-level* 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-classes *Keyword*

Default value: `(>= *delivery-level* 2)`

This keyword controls whether classes are shaken.

: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**.

:shake-shake-shake *Keyword*

Default value: `(>= *delivery-level* 2)`

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 special case of the **KEYWORD** package, it uninterns 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**.

:smash-packages *Keyword*

Default value: **nil**

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 103 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 individually with symbols, is **:smash-packages-symbols**.

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

:smash-packages-symbols *Keyword*

Default value: **nil**

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.

:split *Keyword*

Default value: **nil**

When true, causes the Lisp heap and the executable to be saved in two separate files.

If *split* is **nil** (the default), then the saved image is written as a single executable file containing the Lisp heap. If *split* is **t**, then the saved Lisp heap is split into a separate file, named by adding **.lwheap** to the name of the executable (as specified by the argument *file*). When the executable runs, it reloads the Lisp heap from the **.lwheap** file automatically.

In addition, when saving LispWorks as an application bundle on the Macintosh (for example by using **create-macos-application-bundle**), *split* can be the symbol **:resources**. This places the Lisp heap file in the **Resources** directory of the bundle, rather than in the **Contents/MacOS** directory alongside the executable, which allows the heap to be included in the signature of the bundle.

The main use of *split* is to allow third-party code signing to be applied to the executable, which is often not possible when saving an image with the Lisp heap included in a single file.

:startup-bitmap-file *Keyword*

Default value: **nil**

A string naming a file containing an image to be displayed when the application starts.

On Microsoft Windows, the image needs to be in Windows Bitmap format and must be Indexed Color rather than RGB color.

On Cocoa, GTK and Motif, the image can be in any format supported by Graphics Ports, and the file will be read as if by **gp:read-external-image**. See the "Working with images" section in the *LispWorks CAPI User Guide* for details.

On Windows the user can dismiss the startup screen by clicking on it. It can be dismissed programmatically by calling **win32:dismiss-splash-screen** - see the *LispWorks User Guide and Reference Manual* for details.

The value **nil** means no bitmap is displayed.

:structure-packages-to-keep *Keyword*

Default value: **nil**

A list of packages. For symbols in these packages that have a structure definition, delivery keeps all the information in this structure definition, regardless of the value of **:keep-structure-info**.

:symbol-names-action *Keyword*

Default value: **(>= *delivery-level* 5)**

Defines what to with symbol names. When it is **nil**, or when **: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 changed to the same string **"Dummy Symbol Name"**.

Compatibility Note: in LispWorks 4.4 and previous on Windows and Linux platforms, **:symbol-names-action t** shortens symbol names to a three-character unique code. This has changed, as described above, in LispWorks 5.0 and later.

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 cause errors to appear only when the symbol names are removed. In some cases the easiest solution is to retain symbol names. This will result in a larger executable, though the size increase is usually small.

If you do want to remove symbol names and need to debug your application, **:symbol-names-action** takes these other values **:spell-error**, **:reverse**, **:invert** and **:plist**. In the case of **:spell-error** (which is probably the most useful), the last alphabetic characters in the first 6 characters of the symbol name are rotated by one, that is, A becomes B, g becomes h, and Z becomes A. This leaves the symbol names quite readable, but any function that relies on symbol names fails. A more drastic effect is achieved by the value **:reverse**, which reverses the symbol name. The value **:invert** just changes the case of every alphabetic 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. The value **: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.

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**.

After debugging your delivered application using **:spell-error**, **:reverse**, **:invert** or **:plist**, you may want the production build to be done with **:symbol-names-action t** to remove symbol names and achieve a small reduction in size.

Compatibility Note: in LispWorks 4.4 and previous on Windows and Linux platforms, **:symbol-names-action** allows the value **:dump**. This is no longer supported.

:symbols-to-keep-structure-info *Keyword*

Default value: **nil**

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

:versioninfo *Keyword*

Default value: **nil**

Windows only. A plist containing version information to be placed in the delivered file.

If **:versioninfo** is **nil**, no version information is supplied. Otherwise **:versioninfo** 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 **#x0001000A0000000F**.

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.

:debugp A non-nil value indicates that this is a debugging version.

:patchedp A non-nil value indicates that this file has been 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.

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

:comments A string value, which allows additional comments to be 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
```

:warn-on-missing-templates *Keyword*

Default value: **nil**

Controls whether to warn about missing CLOS templates, which should be pre-compiled. See “Finding the necessary templates” on page 90 for details.

6

Delivery on Mac OS X

This chapter describes several issues relevant to delivery with LispWorks for Macintosh.

6.1 Universal binaries

LispWorks (32-bit) for Macintosh is a universal binary, containing both PowerPC and Intel architectures. To deliver a universal binary application from LispWorks (32-bit) for Macintosh, you will need an Intel-based Macintosh computer. You can specify a universal binary build in the Application Builder tool (see the *LispWorks IDE User Guide*) or call **`save-universal-from-script`** directly (see the *LispWorks User Guide and Reference Manual*).

LispWorks (64-bit) for Macintosh is also a universal binary, containing both PowerPC and Intel architectures. To create a universal binary application from LispWorks (64-bit) for Macintosh, you will need to deliver separately on both a PowerPC Macintosh and an Intel-based Macintosh computer. Contact Lisp Support if you need advice on creating a 64-bit universal binary after that.

6.2 Application bundles

deliver creates a single executable file. However graphical Macintosh applications consist of an application bundle, which is a folder **Foo.app** with several subfolders containing the main executable and other resources.

LispWorks for Macintosh is supplied with example code that constructs an application bundle. It is convenient to use this code - or your variant of it - at delivery time, so that your delivered executable is ready to run in its application bundle in the usual Mac OS X way. See “Creating a Mac OS X application bundle” on page 129 for an illustration of this.

There is another example in **examples/configuration/save-macos-application.lisp**. This code is actually a **save-image** script (rather than **deliver**) but it shows how to avoid writing the application bundle twice when saving a universal binary application.

6.3 Cocoa and GTK images

LispWorks for Macintosh is supplied with two images. One supports the Cocoa GUI, the other supports the GTK+ GUI (and can load the Motif GUI). You cannot build a Cocoa application using the GTK LispWorks image, and vice versa.

You should use the appropriate image to deliver your application.

For GTK and Motif applications delivered with LispWorks for Macintosh, the issues described in Chapter 8, *Delivery on Linux, FreeBSD and Unix* will be relevant.

6.4 Terminal windows and message logs

6.4.1 Controlling use of a terminal window

A graphical Macintosh application does not usually have a console / terminal window.

You can achieve this by supplying the keyword argument *console* **:input** when delivering your application.

6.4.2 Logging debugging messages

Output to **`*terminal-io*`** from an application without a console/terminal window will not ordinarily be visible to the user, so debugging messages should be written to a log file.

Log files are recommended for any complex application as they make it easier for you to get information back from your users.

You can use **`dbg:log-bug-form`** for logging errors. See the *LispWorks User Guide and Reference Manual* for details.

6.5 File associations for a Macintosh application

To create an assocation between your LispWorks for Macintosh application and files with a specified type (file extension):

1. Create the appropriate entries for the file type in the CFBundleDocumentTypes array within the Info.plist file of the delivered application.
2. Define a subclass of **`capi:cocoa-default-application-interface`** with a *message-callback*.
3. Implement the **`:open-file`** message in the *message-callback* function.
4. Set the application interface on startup.

Also see the examples in

`examples/delivery/macos/simple-application.lisp` and **`examples/delivery/macos/full-application.lisp`**.

6.6 Editor emulation

If your application uses **`capi:editor-pane`** or its subclasses, your should consider the input style. The editor in the delivered application can emulate Emacs or Mac OS X style editing. The **`deliver`** keyword **`:editor-style`** controls which emulation is used.

6.7 Standard Edit keyboard gestures

To implement the standard gestures **Command+X**, **Command+C** and **Command+V** in your CAPI/Cocoa runtime application, you must include an **Edit** menu explicitly in your **capi:interface** definition.

Note: The LispWorks IDE adds a minimal **Edit** menu to all CAPI interfaces automatically, in order to make these standard gestures work in the LispWorks IDE, but this does not persist after delivery.

6.8 Quitting a CAPI/Cocoa application

The application menu's quit callback (that is, the callback normally invoked by **Command+Q**) should simply call **capi:destroy** with the application interface and should not call **lw:quit** directly.

For an example see the **Quit My Application Full** menu item in **examples/capi/cocoa-application.lisp**.

6.9 Platforms supporting dynamic library delivery

You can deliver a dynamic library using 32-bit LispWorks or 64-bit LispWorks on Intel-based Macintosh machines.

However you cannot deliver a dynamic library using any LispWorks product on a PowerPC Macintosh.

LispWorks does not support dynamic libraries for the PowerPC architecture. Therefore you cannot build a universal binary dylib.

7

Delivery on Microsoft Windows

This chapter describes several issues relevant to delivery with LispWorks for Windows.

7.1 Runtime library requirement

Applications that you build with LispWorks for Windows require the Microsoft Visual Studio runtime library **msvcr80.dll**, so you must ensure it is available on target machines. It is part of Windows Vista, but for earlier Windows operating systems you should use the Microsoft redistributable mentioned below.

At the time of writing, the redistributable **vcredist_x86.exe** for use with for LispWorks (32-bit) applications is freely available at

```
http://www.microsoft.com/downloads/
details.aspx?familyid=32BC1BEE-A3F9-4C13-9C99-
220B62A191EE&displaylang=en
```

The redistributable **vcredist_x64.exe** for use with LispWorks (64-bit) applications is freely available at

```
http://www.microsoft.com/downloads/
details.aspx?FamilyID=90548130-4468-4bbc-9673-
d6acabd5d13b&DisplayLang=en
```

Run the redistributable from your application's installer, or tell your users to run it directly themselves before running your application.

7.2 Application Manifests

LispWorks for Windows is supplied with an embedded application manifest. This default manifest tells the Operating System:

- which **msvcr80.dll** to use, and
- to use Common Controls 6

You can change the manifest in your delivered image by passing the keyword argument **:manifest-file** to **deliver**. The value must be the name of a file that is a legal application manifest, which is is used as the manifest. The manifest must contain at least the "dependency" element for **Microsoft.VC80.CRT** (without it, your application will fail to start with error messages "Failed to find msvcr80.dll" or "The application configuration is incorrect"). If the manifest does not contain the "dependency" element for **Microsoft.Windows.common-controls** your application will use Common Controls 5, and therefore will not be a "Themed" application.

The value of **:manifest-file** can also be the special value **:no-common-controls-6**, in which case a default manifest without the element for Common Controls is used.

The default manifests that LispWorks uses are provided by way of documentation in the **lib/6-0-0-0/config** directory. If desired, you can base your application manifests as supplied via **:manifest-file** on these files:

	32-bit LispWorks	64-bit LispWorks
With Common Controls 6	**winlisp32.manifest**	**winlisp64.manifest**
Without Common Controls 6	**lisp32.manifest**	**lisp64.manifest**

Table 7.1 The default manifests used by LispWorks

Note: the above only applies when LispWorks is an executable. If LispWorks is a DLL, then it will be themed if the executable that loads it contains the Common Controls 6 manifest

7.3 DOS windows and message logs

7.3.1 Controlling use of a DOS window

A graphical Windows application does not usually have a console (or "DOS window").

You can achieve this by supplying the keyword argument *console* **:input** when delivering your application.

7.3.2 Logging debugging messages

Output to ***terminal-io*** from an application without a console will not ordinarily be visible to the user, so debugging messages should be written to a log file.

Log files are recommended for any complex application as they make it easier for you to get information back from your users.

You can use **dbg:log-bug-form** for logging errors. See the *LispWorks User Guide and Reference Manual* for details.

7.4 File associations for a Windows application

To create an assocation between your LispWorks for Windows application and files with a specified type (file extension), create a DDE server in Lisp and register the file types in Windows.

There is an example of this (for the LispWorks IDE) in **examples/dde/lispworks-ide.lisp**, but the technique is the same for any file extension.

7.5 Editor emulation

If your application uses **capi:editor-pane** or its subclasses, your should consider the input style. The editor in the delivered application can emulate Emacs or Microsoft Windows style editing. The **deliver** keyword **:editor-style** controls which emulation is used.

7.6 ActiveX controls

If your library `foo` is a Windows ActiveX control (that is, it uses `capi:ole-control-component` and `capi:define-ole-control-component`) you may choose to specify file `"foo.ocx"` as the *file* argument to `deliver`. The file type defaults to `"dll"`.

The file extension does not alter functionality - the system simply loads the file referenced in the Windows registry.

8

Delivery on Linux, FreeBSD and Unix

This chapter describes issues relevant to delivery with LispWorks for Linux, LispWorks for FreeBSD, LispWorks for x86/x64 Solaris, and LispWorks for Unix.

8.1 GTK+ considerations

The section describes issues relevant to delivery of CAPI applications running on GTK+.

8.1.1 GTK+ libraries on the target machine

A suitable version of the GTK+ libraries must be installed on the target machine for your CAPI/GTK application to run. The version requirements are as for LispWorks itself, as mentioned in the *LispWorks Release Notes and Installation Guide*.

8.1.2 Fallback resources

If your CAPI/GTK application needs fallback resources then it should pass the `:application-class` and `:fallback-resources` arguments when calling `capi:display` and/or `capi:convert-to-screen`.

See **capi:convert-to-screen** in the *LispWorks CAPI Reference Manual* for a full description of these arguments.

You could use the LispWorks resources as a starting point when constructing your application's resources. You can see the LispWorks fallback resources (these are for application class **Lispworks**) as described under "Using X resources" in the *LispWorks CAPI User Guide*.

You can override the default resource name using the **capi:element** initarg **:widget-name** or the accessor **(setf capi:element-widget-name)**. There is an example in **examples/capi/elements/gtk-resources.lisp**.

8.2 X11/Motif considerations

The section describes issues relevant to delivery of CAPI applications running on X11/Motif.

Note that the X11/Motif GUI is deprecated on Linux, FreeBSD, x86/x64 Solaris and Mac OS X, because the alternative GTK+ GUI library is now supported.

8.2.1 Loading Motif

On LispWorks platforms supporting pthreads, the supplied image contains the GTK GUI only, and therefore GTK is the default graphical library for applications. To build a Motif application on these platforms you need to include

```
(require "capi-motif")
```

in your delivery script.

You may wish to consider building a GTK version of your application too.

8.2.2 Motif on the target machine

A suitable version of the OpenMotif library must be installed on the target machine for your CAPI/Motif application to run. The version requirements are as for LispWorks itself, as mentioned in the *LispWorks Release Notes and Installation Guide*.

8.2.3 Fallback resources

If your CAPI/Motif application needs fallback resources then it should pass the **:application-class** and **:fallback-resources** arguments when calling **capi:display** and/or **capi:convert-to-screen**.

See **capi:convert-to-screen** in the *LispWorks CAPI Reference Manual* for a full description of these arguments.

You could use the LispWorks resources as a starting point when constructing your application's resources. You can see the LispWorks fallback resources (these are for application class **Lispworks**) as described under "Using X resources" in the *LispWorks CAPI User Guide*.

You can override the default resource name using the **capi:element** initarg **:widget-name** or the accessor **(setf capi:element-widget-name)**.

8.2.4 X resource names use Lisp symbol names

The default color and other attributes for each CAPI pane on X11/Motif is computed as an X resource using the symbol name of the pane's class. Therefore obtaining the correct X resources depends on the application containing these symbol names.

Symbol names are removed at delivery level 5, but you can retain specific names in the delivered image by passing a list of the class names to **deliver** as the value of the keyword argument **:keep-symbol-names**.

8.3 Logging debugging messages

Log files are recommended for any complex application as they make it easier for you to get information back from your users. The log should contain any debugging messages, and can also contain information from your program.

You can use **dbg:log-bug-form** for logging errors. See the *LispWorks User Guide and Reference Manual* for details.

8.4 Editor emulation

If your application uses **capi:editor-pane** or its subclasses, your should consider the input style. The editor in the delivered application can emulate

Emacs or KDE/Gnome style editing. The **deliver** keyword **:editor-style** controls which emulation is used.

8.5 Products supporting dynamic library delivery

You can deliver a dynamic library using LispWorks on any supported x86 hardware and also using any 64-bit LispWorks product.

However you cannot deliver a dynamic library using LispWorks (32-bit) for SPARC Solaris or LispWorks (32-bit) for HP-UX.

9

Delivery and Internal Systems

9.1 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.

9.1.1 Applications defining classes or methods dynamically

Set the **`deliver`** keyword **`:keep-clos`** to **`t`** or **`:full-dynamic-definition`** to keep the code needed for dynamic definition in the image.

9.1.2 Special dispatch functions and templates for them

The LispWorks CLOS implementation achieves fast method dispatch by producing special functions to perform discrimination and method dispatch. Since the required operation 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.

If the compiler has been removed in a delivered application, then these special runtime-generated functions cannot be compiled on the fly.

There are two ways in which the delivery system deals with this problem.

The first is to have a set of pre-compiled "template" constructors which can construct an appropriate function. LispWorks comes with extensive set of such constructors, which should cover most of cases. The programmer can add her own, as explained below.

The other mechanism is to construct generic closures to do the work. The code that generates the closures can cope 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 `call-method` forms.

In most cases the effect on method dispatch time of using the generic technique is negligible. Pathological cases might, however, cause a slowdown of 10-20% over compiled special functions. In this case, as well as for cases of user-defined complex method combinations which the generic mechanism cannot cope with, the delivered image must have precompiled "template" constructors, and if they are not already there the user needs to add them, as described next.

9.1.2.1 Finding the necessary templates

Even though it cannot compile the functions 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`. This defaults to `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 <<<

(CLOS::PRE-COMPILE-COMBINED-METHODS

  ((1 COMMON-LISP:NIL) COMMON-LISP:NIL (CLOS::_CALL-METHOD_)))

; *****
```

You can take this 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 91.

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. 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.

9.1.2.2 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:

1. Collect into one template file all the method combination template forms that have been output, so that it looks something like this:

```
(CLOS::PRE-COMPILE-COMBINED-METHODS ((1 COMMON-LISP:NIL) COMMON-
LISP:NIL

    (COMMON-LISP:MULTIPLE-VALUE-PROG1 (CLOS::_CALL-METHOD_)

                          (CLOS::_CALL-METHOD_)

                          (CLOS::_CALL-METHOD_))))

(CLOS::DEFINE-PRE-TEMPLATES

  CLOS::DEMAND-CACHING-DCODE-MISS-FUNCTION (5 COMMON-LISP:NIL
(4)))

(CLOS::DEFINE-PRE-TEMPLATES

  CLOS::DEMAND-CACHING-DCODE-MISS-FUNCTION (6 COMMON-LISP: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.

9.1.3 Delivery and the MOP

MOP programmers should note that, by default, the direct slots and direct methods of all classes are emptied at delivery level 1 and above. To prevent this, set the **`deliver`** keyword **`:keep-clos`** to **`t`**, **`:full-dynamic-definition`** or **`:meta-object-slots`** as required.

9.1.4 Compression of CLOS metaobjects

To reduce the size of the delivered image, the delivery process compresses the representation of CLOS metaobjects (classes, generic functions and 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 **method-function**.
4. Compressing the representation of generic functions. This is not done if **:keep-clos** is **t**, or if you add methods to any of the accessors of generic functions.

9.1.5 Classes, methods, and delivery

See “Shaking the image” on page 31 for a discussion of how unused class definitions and methods are treated by delivery process.

9.1.6 Delivery and make-instance initarg checking

By default **make-instance** checks for valid initargs in LispWorks, signalling an error on an invalid call. However, in a delivered application this behavior may not be useful.

Initarg checking in the delivered application is controlled by the **deliver** keyword **:make-instance-keyword-check**.

For more information about **make-instance** initarg checking, see the *LispWorks User Guide and Reference Manual*.

9.2 Editors for delivered applications

This section contains information on how to include the LispWorks editor in your delivered applications and how to control its behavior.

9.2.1 Form parsing and delivery

If the delivered image is used to edit LISP code, the parsing of forms will still not work properly. The **deliver** keyword **:keep-editor** can be used to keep the code for parsing forms in the editor.

9.2.2 Emulation and delivery

The editor in the delivered application can emulate Emacs style, and Microsoft Windows or Mac OS X style editing (depending on the platform). The **deliver** keyword **:editor-style** controls which emulation is used.

9.2.3 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. Two **deliver** keywords allow you to specify which commands to keep and which commands to delete:

:editor-commands-to-keep (default **nil**)

:editor-commands-to-delete (default **:all-groups**)

The effect of these default values is that all the commands are deleted. If a command is both these lists, it is kept.

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 it is 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 traversed recursively.)
String value	The corresponding editor command is deleted.

Symbol value Taken to specify a Command Group.

The available Command Groups are:

:simple-editor The simple editor contains basic mechanisms for editing 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.

9.3 Delivery and CAPI

This section describes platform-independent issues in delivered applications which use CAPI. See also Chapter 6, “Delivery on Mac OS X”, Chapter 7, “Delivery on Microsoft Windows”, and Chapter 8, “Delivery on Linux, FreeBSD and Unix” for issues specific to each supported windowing system.

See the *LispWorks CAPI Reference Manual* for details of the CAPI symbols mentioned.

9.3.1 Interface geometry depends on Lisp symbol names

The function **capi:top-level-interface-geometry-key** depends on symbol names and hence will break at delivery level 5 unless the relevant symbol names are retained. Use the **deliver** keyword **:keep-symbols** to keep the class name of your top level interface.

9.4 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 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**.

9.4.1 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.

9.4.1.1 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**:

```
.....
(let ((res (call-something)))
  (when res
    (generate-error res))
.....
```

```
(defun generate-error(res)
  (error 'application-error
         :error-number res))
```

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

```
(defun generate-error (res)
  (let ((action
         (capi:prompt-with-list
          '(("Abort Operation" . abort)
            ("Retry Operation" . retry)
            ("Ignore Error")
            ("Quit" . stop-application)
            ("Do Something Else" . do-something-else))
          (find-error-string res)
          :print-function 'first
          :value-function 'rest)))
    (case action
      ((abort retry) (invoke-restart action))
      ((nil))
      (t (funcall action)))))
```

9.4.1.2 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 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.

9.4.1.3 Providing a fallback handler for uncaught errors

The variable **cl:*debugger-hook*** can be used to handle errors that are not caught by other handlers.

9.4.2 Deleting of condition classes

Condition types are classes like any other 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 a special deletion action for conditions, which is controlled by the **deliver** key-

words **:condition-deletion-action**, **:keep-conditions** and **:packages-to-remove-conditions**.

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 **format-arguments** slot. If the conditions are redirected (that is, when **:condition-deletion-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:

1. 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.

9.5 Delivery and the FLI

This section describes particular issues relevant to a delivered image containing Foreign Language Interface (FLI) code.

9.5.1 Foreign Language Interface templates

The Foreign Language Interface requires compiled code (known as FLI templates) to convert between foreign objects and Lisp objects. Most of these FLI templates are already available in the image, and most applications do not need extra templates.

However it is difficult to know in advance exactly which FLI templates will be needed. When a new template is actually required, it is compiled. In a delivered image where the compiler has been removed, this causes an error like this:

```
FLI template needs to be compiled
(see 'Foreign Language Interface templates' in the LispWorks
Delivery User Guide):
  (FLI::DEFINE-PRECOMPILED-FOREIGN-OBJECT-SETTER-FUNCTIONS
((:FLOAT :SIZE 4)))
```

To solve this you need to find which templates your application uses that are not already available, compile them, and load them before delivering.

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

1. Start the undelivered application image (that is, LispWorks with your application code loaded).
2. Call

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

3. Exercise the application.
4. Call

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

This prints all the templates that were generated while exercising your application. These FLI template forms should be put in a file which is compiled and loaded as part of your application . **FLI:PRINT-COLLECTED-TEMPLATE-INFO** takes a keyword **:OUTPUT-STREAM** to make this easier, for example:

```
(with-open-file (stream "fli-templates.lisp" :direction :output)
                (FLI:PRINT-COLLECTED-TEMPLATE-INFO
                  :OUTPUT-STREAM stream))
```

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

9.5.2 Foreign callable names

In most cases foreign callable names are passed to **deliver** in the value of the **:dll-exports** keyword argument, and each of these foreign callables will be retained automatically in the delivered image.

However other foreign callables defined with a string *foreign-name* are liable to be shaken from the delivered image. The best approach is to use a symbol to name such foreign callables, as described under **fli:define-foreign-callable** in the *LispWorks Foreign Language Interface User Guide and Reference Manual*.

9.6 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.

You can obtain the list of loaded modules by entering

```
:bug-form nil
```

in a Listener. This prints, inter alia, the list of loaded modules.

To obtain a minimal list of modules, follow these steps:

1. Start a fresh LispWorks image, making sure it does not load any irrelevant code (for example in your `.lispworks` init file):

   ```
   C:\Program Files\LispWorks> lispworks-6-0-0-x86-win32.exe -
   init -
   ```

2. Load the application and run it.
3. Exercise the application, to ensure that any entry points for load on demand modules are called.
4. Enter `:bug-form nil` in a Listener. The list of loaded modules should include only modules that your application needs.

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

```
(require module-name)
```

Add the call to `require` to your delivery script.

Note: `require` is case-sensitive, and generally ***module-name*** is lowercase for LispWorks modules.

9.7 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.

9.8 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.

9.8.1 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.

9.8.2 How to delete packages

You can pass `deliver` a list of packages to delete with the keyword `:delete-packages`.

9.8.3 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.

9.8.4 How to smash packages

You can pass `deliver` a list of packages to smash with the keyword `:smash-packages` or `:smash-packages-symbols`.

9.8.5 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.
3. 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.
5. 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.

9.9 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.

9.9.1 Ensuring 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**. All packages in the list passed with this keyword are kept in the delivered image, regardless of the state of the **:smash-packages** and **:delete-packages** 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 106.

9.9.2 Ensuring 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, retain it force in one of the following four ways:

1. With the **:keep-symbols** keyword.

   This is the recommended solution. See **:keep-symbols**.

2. With the **:never-shake-packages** keyword.

   This solution is suitable when all the symbols to keep are in one package, **FOO-PKG** say. Pass **:never-shake-packages (list "FOO-PKG")**. See **:never-shake-packages**.

3. 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**.

You can also specify **:packages-to-shake-externals** and **:packages-to-keep-externals**.

4. 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 106.

9.10 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 **XYZ** to be deleted without breaking a call to **intern** such as the following:

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

9.11 Symbol-name comparison

In a non-delivered LispWorks image, the form

```
(eq (symbol-name 'foo) (symbol-name 'foo))
```

evaluates to `t`. This behaviour is due to the way symbol names are cached. There is no requirement or guarantee that the results of successive calls to `symbol-name` be the same (`eq`) object.

After delivery, LispWorks symbol names are implemented differently such that the `eq` test above fails. Take care that your application does not rely on identity of symbol names.

Note: `eq` is not a reliable comparison of strings in general. Use `equal` for reliable string comparison.

10

Troubleshooting

This chapter provides solutions to common delivery problems.

10.1 Debugging errors in the delivery image

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

1. First check if the same error occurs in the original (undelivered) development image. If it does, debug the problem in this image.
2. If the error is not reproducible in the development image, check if it is reproducible in an image delivered at a lower delivery level (try 0, then 1 etc). If it is, read the error message and backtrace carefully. In most cases, this is enough to debug the problem.
3. Make sure you can see messages printed by the application (the *runtime output*), which may contain useful information. In the case of a graphical application on Microsoft Windows or Macintosh these messages may not normally be visible but can be captured by redirecting the runtime output to a file.

To redirect the runtime output, run the application in a command shell. This means a DOS command window (on Microsoft Windows), Terminal.app (Mac OS X) or a shell (Unix/Linux etc). Enter the application executable filename followed by > followed by the output filename, for example,

on Windows:

```
C:\Program Files\MyApp> myapp.exe > C:\temp\myapp-output
```

on Macintosh:

```
mymac:/Applications/MyApp/MyApp.app/Contents/MacOS 2 % ./myapp >
/tmp/myapp-ouput
```

4. Consider the possibility that you are trying to use functionality that was removed by delivery. You may need to keep the functionality explicitly, by using one of the **deliver** keywords described in "Retaining or removing functionality" on page 36.

5. If the problem occurs only in the delivered image and not in the original image, and it is still not clear what the problem is, please contact Lisp Support immediately. Send us your deliver script, all the output of the delivery process and the runtime output of the application itself. This situation is regarded by Lisp Support as a bug that should be fixed.

10.2 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 9.6 on page 101.

See “Symbol and package issues during delivery” on page 101 for the explanation and suggestions in cases 2, 3 and 5 above.

10.3 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 110.

10.4 REQUIRE was called after delivery time with module ...

This error message means that a loadable module was omitted from the application build, and the program now tries and fails to load that module. The solution is described in “Modules” on page 101.

10.5 Failed to reserve... error in compacted image

Loading a compacted LispWorks (32-bit) for Windows DLL might result in an error message like this:

```
Failed to reserve 14024705 bytes of memory (preferred address
0x20000000)
Error 487: Attempt to access invalid address.
```

LispWorks normally relocates its heap if the default address 0x20000000 is already in use (for example, by another DLL) but this is not possible if the DLL is compacted.

The solution is to build a non-compacted DLL:

```
(deliver nil  "foo" 5 :dll-exports '("Foo") :compact nil)
```

10.6 Memory clashes with other software

LispWorks executables and dynamic libraries have a default startup location which may clash with other software already mapped at that location. Also, a LispWorks image may grow up to an address where other software is already mapped. Where possible LispWorks attempts to avoid such clashes automatically.

If LispWorks fails to use other memory as it grows, the effect will be to limit the size of the Lisp heap, possibly leading to messages

```
failed to enlarge memory
```

at the console. On some platforms LispWorks can fail to detect a clash safely, which will lead to unpredictable behavior if it overwrites other code.

The behavior is specific to the particular platform and LispWorks implementation. There is a discussion of these issues (with the platform-specific details) and a description of how you can avoid memory clashes under "Startup relocation" in the *LispWorks User Guide and Reference Manual*.

10.7 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 delivery level.

  If things work after this, try the same level, but override the changed keywords one by one.

- Retain more packages, with the keyword `:packages-to-keep`

  For example:

```
(deliver 'application-entry
           "application"
           5
           :packages-to-keep '("LISPWORKS"))
```

The **COMMON-LISP** package normally should not be deleted or smashed, so it is unlikely to cause problems , but **LISPWORKS** 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 101 for more information on keeping and throwing away packages.

10.8 Errors when finalizing classes

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

10.9 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_))) ;
*****
```

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 templates" on page 90, "Incorporating the templates into the application" on page 91.

10.10 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.

10.11 Stack frames with the name NIL in simple backtraces

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

10.12 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 63 may fix this. This technique can also be used on any symbol which prints as **#:||**.

10.13 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 delivery 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 **:keep-symbols** keyword).

10.14 FLI template needs to be compiled

An error starting with

```
"FLI template needs to be compiled"
```

is probably a result of missing Foreign Language Interface templates. See “Foreign Language Interface templates” on page 99 for instructions.

10.15 Failure to lookup X resources

X resource names use Lisp symbol names in CAPI/Motif, which might be removed from the delivered image. This issue and the solution is described on page 87.

10.16 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.

10.17 Debugging with :no-symbol-function-usage

When *no-symbol-function-usage* is true while delivering an image **"foo"** on x86 platforms, delivery writes a file named **"foo.zaps"** (the "zaps file") containing debug information about the symbols that were eliminated.

If an error occurs in the delivered image, the backtrace will contain a line of the form.

```
("SYMBOL-FUNCTION-VECTOR" nnn)
```

where *nnn* is an integer. The actual function name can be recovered from the zaps file by doing this in the LispWorks development image:

```
(require "delivery")
(dv::recover-zapped-symbol-from-file "foo.zaps" nnn)
```

The numbers are unique to each image, so take care to use the zaps file that was produced at the same time as the delivered image.

10.18 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 Lisp Support first.

To use **interrogate-symbols** pass **:interrogate-symbols t** to **deliver**. This loads the interrogate symbol facility. and causes the delivered image to check for the command line argument **-interrogate-symbols** on startup. If this command line argument appears, the image first does symbol interrogation, and then proceeds to run the application as normal.

Symbol interrogation starts by building an internal table of reverse pointers, during which the image prints some messages 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 begin with **#\+**, it is a symbol name. The string is used as the argument to **find-symbol** (in the current package).

   Note the string is used as-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 first element is the symbol, the second element is a list of *interesting* symbols that point to that symbol (possibly through *uninteresting* symbols), and the third element is a list of symbols that point 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 via 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 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 value **30**) of them.

2. If the string contains a **#\:** character or a pair of **#\:** characters, and there are characters after it, it is 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 packages that points (as defined in 1.above) to symbols in the package, followed by a list of the symbols 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 contains many symbols.

This option is especially useful in conjunction with the **:smash-packages-symbols** keyword to **deliver**, to find why a package that should have gone remains 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.

11

User Actions in Delivery

11.1 General 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 or data is present only for debugging purposes. The system, however, cannot tell which part of the code or data can be eliminated, so you have to do it yourself.

That can be done in either of two ways:

1. You can eliminate the code or data explictly before calling **`deliver`**, by using **`fmakunbound`**, **`makunbound`**, **`remhash`** and so on. The advantage of this approach is that it does not require you to know anything about Delivery. The disadvantage of this is that these calls must be put explicitly in the delivery script.

2. The LispWorks image contains an action list called "Delivery actions", which you can add actions to. See the *LispWorks User Guide and Reference Manual* for information about action lists.

   The "Delivery actions" action list is executed when the delivery process starts, before any system action. For example, if **`*my-hash-table*`** contains entries that are not required in the delivered application, then you may write:

```
(defun clear-my-hash-table()
  (maphash #'(lambda (x y)
               (unless (required-in-the-application-p x y)
                 (remhash x *my-hash-table*)))
           *my-hash-table*))
(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 code that uses ***my-hash-table***. This makes it easier to maintain that code.
2. It can access the user interface of the delivery process. This is done via the function **delivery-value** and **(setf delivery-value)**.

11.2 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 39, 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 beused 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.
2. 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 otherwise it is never shaken.
3. It does not assume that P:X will survive the shaking.

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

The cleanup functions are called *after* the operation of **delivery-shaker-weak-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-weak-pointer** 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.
2. 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 **delivery-shaker-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:

1. 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 **delivery-shaker-weak-pointer**, the accessor is not used. The *accessor* can be one of:

A symbol. This specifies a function 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 the *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 arguments. That is, if the setter is **(set-something *name*)**, the call is **(set-something *pointed pointing name*)**. In addition, where the *accessor* accepts a symbol, the *setter* also accepts a function object.

remover — Default value **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 value **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 (this does not happen unless you access objects which you should not access), 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 situation.

Example 1:

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 to from 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))

;; this will cause the car of any entry where the car is not
;; pointed to from another object to change to nil
(defun shake-my-hash-table ()
  (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))

;; call this function at delivery time
(define-action "Delivery Actions" "shake my hash table"
             'shake-my-hash-table)

;;;;-------------------------------------------------------------
```

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))
```

[This assumes 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 2:

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 be 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*)
```

Implementation 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 (nconc
                                 (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*)
```

12

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:

```
(example-file "capi/applications/othello.lisp")
```

12.1 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:

1. 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.

12.1.1 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-all-patches)
;; Load the compiled file othello. Should be in the same
;; directory as this script.
(load (current-pathname "othello" nil))
;; Now deliver the application itself to create the image othello
(deliver 'play-othello "othello" 0 :interface :capi)
```

Save this script in the newly created **othello** directory as **script.lisp**.

Note: Alternatively you can create a delivery script using the Application Builder tool in the LispWorks IDE. The Application Builder is a windowing interface offering another way to performs the steps described the following sections. For full instructions on using the Application Builder tool, see the *LispWorks IDE User Guide*.

The remainder of this section shows you how to complete delivery of the othello application using a command shell.

12.2 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 like this:

   ```
   lispworks-6-0-0 -build script.lisp
   ```

   See “Delivering the program” on page 12 for details of how to run the image with a script on your platform. The LispWorks image name will differ from the above according to the platform.

   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 Microsoft Windows, and `othello` in your working directory on UNIX / Linux / FreeBSD / Mac OS X.

2. Execute the `othello` file.

   This should be a working, standalone Othello game.

   Note: On Mac OS X / Cocoa you will also need to create an application bundle to run GUI applications properly. See “Creating a Mac OS X application bundle” on page 129 for details.

See “Delivering a standalone application executable” on page 24 for a more detailed discussion of this part of the delivery process.

12.3 Creating a Mac OS X application bundle

The section applies only to LispWorks for Macintosh with the native Cocoa GUI.

You should not simply run a Mac OS X / Cocoa GUI application from the command line in Terminal.app. Instead you should put the image in a suitable Application Bundle and run it using the Finder. The example delivery scripts in this manual automatically create the Application Bundle, using the code described below.

Your LispWorks Library contains example code which constructs a suitable Mac OS X application bundle for your delivered image. The function **write-macos-application-bundle** does several things:

- creates the folders comprising an Application Bundle
- adds the resources from a supplied template bundle (or **LispWorks.app**) to the Application Bundle
- writes a suitable **Info.plist** file in the Application Bundle
- returns the path of the executable within the Application Bundle

12.3.1 Example application bundle delivery script

Note how this script calls **deliver** with the executable path returned by **write-macos-application-bundle**:

```
(in-package "CL-USER")
(load-all-patches)
;; Compile and load othello example code
(compile-file (example-file "capi/applications/othello")
              :output-file :temp
              :load t)
;; Compile and load Application Bundle example code
#+:cocoa
(compile-file
 (example-file "configuration/macos-application-bundle")
 :output-file :temp
 :load t)
;; Now deliver the application itself and create the
;; application Othello.app
(deliver 'play-othello
         #+:cocoa (write-macos-application-bundle
                   "~/Desktop/Othello.app"
                   ;; Do not copy Lisp Source File
                   ;;  associations from LispWorks.app
                   :document-types nil)
         #-:cocoa "~/othello"
         0 :interface :capi)
```

In the session below **script.lisp** is in the user's home directory. Here is the start and end of the session output in Terminal.app:

```
mymac:/Applications/LispWorks 5.1/LispWorks.app/Contents/MacOS 2
% ./lispworks-5-1-0-macos-universal -build ~/script.lisp
LispWorks(R): The Common Lisp Programming Environment
Copyright (C) 1987-2007 LispWorks Ltd.  All rights reserved.
Version 5.1.0
Saved by LispWorks as lispworks-5-1-0-x86-darwin, at 18 Oct 2007
1:41
User dubya on mymac
; Loading text file /u/dubya/deliver-othello.lisp
;  Loading text file /Applications/LispWorks 5.1/Library/lib/5-1-
0-0/private-patches/load.lisp
;;; Compiling file /Applications/LispWorks 5.1/Library/lib/5-1-0-
0/examples/capi/applications/othello ...
;;; Safety = 3, Speed = 1, Space = 1, Float = 1, Interruptible = 0

[... full compilation and delivery output not shown...]

Shaking stage : Saving image
Build saving image: /u/dubya/Desktop/Othello.app/Contents/MacOS/
Othello
Build saved image: /u/ldisk/dubya/Desktop/Othello.app/Contents/
MacOS/Othello

Delivery successful - /u/dubya/Desktop/Othello.app/Contents/
MacOS/Othello
```

The last line of the **deliver** output shows the full path to the executable, but you should run the application bundle **Othello.app** via the Finder.

12.3.2 Further Mac OS X delivery examples

These can be found in your LispWorks library directory **examples/delivery/macos/**.

12.4 Command line applications

If you need to deliver a non-GUI application for Mac OS X, change the delivery script to remove the code (conditionalized in the examples under **#+cocoa**) that constructs the Application Bundle.

On all platforms, delivering a non-GUI application will not need the **:interface :capi** keyword argument.

Your delivery script to build a command line application will look something like this:

```
(in-package "CL-USER")
(load-all-patches)
(load "non-gui-code")
(deliver 'dont-start-the-gui
         "non-gui-app"
         5)
```

12.5 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 30 for more details.

12.5.1 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

L

M

N

O

P

Q

R

S

T

U

V

W

X