

# Delivering Applications

This document contains the following sections:

## [1.0 Delivery introduction](#)

## [2.0 Definitions](#)

## [3.0 Developing the application and preparing for delivery](#)

### [3.1 A summary of the delivery process](#)

### [3.2 Legal and licensing issues](#)

### [3.3 Deciding on necessary features](#)

### [3.4 Deciding on top-level \(user interaction\)](#)

### [3.5 Packaging the product](#)

### [3.6 Including all desired modules](#)

### [3.7 Defining the init functionality](#)

### [3.8 Specifying the initial value of \\*package\\*](#)

### [3.9 Setting up logical pathname translations](#)

### [3.10 Use of shared libraries \(foreign files\)](#)

### [3.11 CLOS training](#)

#### [3.11.1 Generic functions, method combination, and discrimination](#)

#### [3.11.2 Effective methods](#)

#### [3.11.3 Caches for fast dispatching](#)

#### [3.11.4 Constructor functions](#)

#### [3.11.5 How to do CLOS start up optimizations](#)

#### [3.11.6 make-instance optimization](#)

## [4.0 Creating the deliverable](#)

### [4.1 Resources](#)

### [4.2 Defsystem](#)

### [4.3 Tuning the application](#)

### [4.4 More on the development environment](#)

### [4.5 GC parameters and switches](#)

### [4.6 GC cursors](#)

### [4.7 Allegro Presto](#)

### [4.8 Allegro Runtime](#)

### [4.9 Windows specific information](#)

#### [4.9.1 The console window in applications](#)

#### [4.9.2 Icon files suitable as a value for icon-file](#)

#### [4.9.3 Creating a Console APP on Windows](#)

### [4.10 Installation of your application on Windows using the Install Wizard](#)

#### [4.10.1 Uninstalling an application on Windows](#)

### [4.11 Testing your application](#)

### [4.12 Expiration warnings](#)

## [5.0 generate-executable: a wrapper for generate-application](#)

## [6.0 Patching your application after delivery](#)

- [6.1 The Allegro CL patch naming scheme](#)
  - [6.2 Loading Allegro CL patches](#)
  - [6.3 Patches for your application](#)
  - [6.4 Creating patch files in general](#)
  - [6.5 Creating a patch file](#)
  - [6.6 What to do with patch files](#)
  - [6.7 Including application patches in an image at build time](#)
  - [6.8 Superseding a patch](#)
  - [6.9 Withdrawing a patch](#)
  - [6.10 Distributing patches](#)
  - [6.11 Loading patches](#)
- 
- 

## 1.0 Delivery introduction

This document describes the application generation utility in Allegro CL. The facility differs from the image creation utility (see *building-images.htm*) in that the end result is not a single image file but a directory of files in theory suitable to be delivered to customers after suitable packaging.

---



---

## 2.0 Definitions

- **Application:** an application is a set of files, including an executable image or shared library, and perhaps additional files, some or all of which are needed for the application to run successfully.
- **Executable:** an executable is a file that can be executed by the host operating system. This file is typically quite small and it is the program seen as running when the application is active. The memory usage of the program also depends on the *image* (see below). The executable is also called an *.exe* (from the file extension used in Microsoft Windows).
- **Shared library:** a shared library is an operating system entity that can be loaded and accessed from within Lisp. Allegro CL itself is partially implemented in a shared library.
- **Image:** an image file contains the Lisp heap portion of the application and necessary parts of Lisp. Images are created with **build-lisp-image** or **dumplisp**. Those created with **dumplisp** have a *parent image* (the image running when **dumplisp** is called). Characteristics of a parent image are inherited by a child image created with **dumplisp**. See also *original image*.
- **Original image:** an image which was part of the original Allegro CL installation. Any files referenced by the original image must be present when any descendent image is run.
- **Parent image:** if an image is created with **dumplisp**, the image running when **dumplisp** is called is the parent image. The child image, that created by **dumplisp**, inherits characteristics of the parent image.
- **Programmer:** a person who develops the application, perhaps part of a team. This person is also called a developer.
- **User:** a person who uses the application developed by the programmer.
- **VAR:** a value added reseller. In the context of this document, they are the user of Allegro CL (e.g., a *programmer* as defined above) who creates the application delivered to a *user* (also defined above).

- **Top-level:** the part of the Lisp that receives control when an image is loaded by the Lisp executable. The program that gets control can either be graphical (a Windows and X Windows program) or console based. If console based, it interactively receives input from a user and prints information to the user while the application is running. Command-line arguments and input files read by the application and output files written by the application are not part of the top-level.
  - **Allegro directory:** the directory in which the Lisp executable resides and the location where files needed while Allegro CL is running can be found. The `sys` logical pathname host translates to this directory. In some contexts, *Allegro directory* refers to the directory where Allegro CL was installed (since in typical practice they are the same directory). The Allegro directory is also sometimes called the *Allegro CL home directory* or *home location* (names used with earlier releases on Unix). In this document, *[home location]/foo*, *[Allegro directory]/foo* and *sys:foo* are the same. The name *home* has nothing to do with a user's home directory on Unix.
- 
- 

## 3.0 Developing the application and preparing for delivery

---

### 3.1 A summary of the delivery process

**generate-application** and **generate-executable** are the functional entry point for developers to deliver an application. (Note that **generate-application** and **generate-executable** are only available in Enterprise versions of Allegro CL.)

On Windows when you use the Integrated Development Environment, much of the file handling and such is done using projects. When you have a project defined, you build an application using the IDE menu option **File | Build Project Distribution** (which calls **generate-application** on a project). In this document, we describe **generate-application** rather than the **File | Build Project Distribution** menu command, but what we say about **generate-application** broadly applies to building applications using projects (thus most of the subsections of [Section 4.0 Creating the deliverable](#) apply). See [IDE User Guide, chapter 4: Projects](#) for discussion of projects.

On Unix or on Windows without the IDE, the task of **generate-application** is to build and assemble the files for the developer's application into a single directory. This directory has many purposes. As a developer, you might want to:

1. Test your application under non-development conditions.
2. Test your application on another machine, to shake out any environmental dependencies in your application.
3. Package your application for delivery to your customers.

For the purposes of this document, it does not matter what the motivation of the developer is, **generate-application** is used in the same way.

There are three distinct delivery types, as:

1. an `.exe` (Allegro CL is in control even though the user might not be able to tell this),
2. **[MS Windows only]** an OLE in-proc server (the application is started via OLE calls in another application), and
3. a shared library.

(1) is the most common case, by far. (3) is what is commonly called *Lisp as a subroutine*. (2) requires registration, either implicit or explicit. In the OLE samples there are examples of both.

It is important to understand that **generate-application** calls **build-lisp-image** and the latter starts a new (operating system) process to build the requested image. Any errors that occur during the building of the image will be handled in the other process. The development environment of the originating process (the one in which the call to **generate-application** was made) cannot be used to debug the problem in the image creating process.

See *Debugging an image build problem or failure* in *building-images.htm* for more information on debugging the image creating process.

## 3.2 Legal and licensing issues

If you want to distribute your applications outside your organization or inside your organization to users with machines not licensed to run Allegro CL, you must be licensed to do so by Franz Inc. One type of license that allows distribution is an Allegro Runtime license. See *runtime.htm*. Please contact your Franz Inc. account manager for information on licensing applications.

## 3.3 Deciding on necessary features

You need all the features necessary to run your application (whatever they may be). Certain items, like the debugger, the inspector, the tracer, will not be present in an image unless explicitly used or called for. If you are preparing a runtime delivery, be sure to check the Allegro CL runtime license to see what modules cannot be included in the image. See [Section 3.6 Including all desired modules](#) for information on insuring all needed and licensed functionality is included in the delivery image.

**generate-application** only builds runtime images (see *runtime.htm*). The value of the *runtime* keyword argument to **generate-application** (actually, it is a **build-lisp-image** argument accepted by **generate-application**) must be one of :standard (the default), :dynamic or :partners. If :standard, the compiler cannot be in the final image. You must either specify *include-compiler nil* or *include-compiler t* and *discard-compiler t*. The latter choice allows the compiler to be present during the build, which is sometimes useful. The compiler may be included in a Dynamic Allegro Runtime image (value of *runtime* :dynamic) and in a Partner's Runtime image (value of *runtime* :partners). Partner's Runtime is described in *Allegro CL Partner's Runtime* in *runtime.htm*.

## External formats and locale where the application is run

Allegro CL supports non-ASCII character sets and allows the specification of a locale that specifies the standard character set to be used. If the locale where the application being generated by **generate-application** is different from the locale on the machine where the application is generated, then the necessary code for switching external formats must be included in the image or available to the application. This is most easily done by specifying the *runtime-bundle* keyword argument true. See *Locales in applications* in *iacl.htm*.

## 3.4 Deciding on top-level (user interaction)

Are your users going to interact with your application through the Lisp top-level (so they will enter Lisp forms or at least one Lisp form), through a custom top-level of your own, or will users interact with your application via some graphical user interface? (Of course, some applications may have no top-level -- that is, little or no user interaction is necessary.)

If your application has a custom top-level, you must write its functionality and have it initiated when your application image starts. You do this by having the function that is the value of `*restart-app-function*` initiate your top-level. If your application has no top-level, then the value of `*restart-app-function*` should be the function that starts your application running.

If you want to use the standard Lisp top-level, leave the value of `*restart-app-function*` `nil`. Any initializations that are necessary can be done by the function that is the value of `*restart-init-function*`.

**Warning for users calling `generate-application` from the IDE:** the value of `*restart-app-function*` may be a function which starts the IDE. That value will be inherited if the `restart-app-function` keyword argument to **`generate-application`** (or **`build-lisp-image`**) is unspecified. Be sure to specify it `nil` if you want to use the standard Lisp top-level, rather than leaving it unspecified.

A minimal top-level is provided if you build an image with the `include-tpl` argument to **`build-lisp-image`** (or **`generate-application`**, which accepts **`build-lisp-image`** arguments). See *Minimal top levels* in *building-images.htm* for more information.

## 3.5 Packaging the product

In the past, it was possible to deliver an application as a single file (called a *standalone application*). This is no longer possible. We have, however, created a precise method to help you easily create and identify all the necessary parts of your application for it to be complete. **`generate-application`** is the entry point to creating your application. This function creates a directory containing all the files making up your application.

## 3.6 Including all desired modules

A standard Allegro CL image on startup does not contain all the system modules that a program may invoke. Instead, certain modules are left out (and contained in the bundle file, typically `sys:files.bu` but the name varies, or in a `fasl` file in `sys::code;`, the `code/` subdirectory of the Allegro directory). When a module is called for with **`require`**, it is looked for according to the `*require-search-list*`, which usually looks in `sys::code;` and, if it is not there, in the bundle file (which is essentially a collection of `fasl` files). Further, when an important function associated with the facility is called (e.g. **`trace`** for the trace module, **`def-foreign-call`** for the foreign module, etc.), the system detects that the module must be loaded and loads it automatically (a process called autoloading, see *Autoloading* in *implementation.htm*).

However, the bundle file supplied with the distribution, *files.bu* or whatever it is named, generally cannot be distributed with applications. (It is explicitly forbidden to distribute the bundle file, usually *files.bu* but the name varies, with a runtime image. If you have a VAR license, you may or may not be allowed to distribute the bundle file, depending on the terms of the license. In this document, we assume you are not permitted to distribute the bundle file. If you have any questions about licensing issues, please contact your Franz Inc. account manager. If you have an Allegro Runtime license, see *runtime.htm*)

There is, however, a bundle file specifically for a runtime application. Specify `:runtime-bundle t` in your call to **generate-application** and a *files.bu* will be created and placed in the application directory. This file may be distributed with your application.

If you are creating an application on Windows using the IDE project system, there is a tool for finding necessary modules. See the description of the **Find Required CG Modules** button on the **Project Manager Include Tab**

Or you can ensure that all desired modules are loaded into the image. The remainder of this section describes how to do that in case you do not want to use the runtime bundle option.

The file *sys:develenv.cl* contains a list of **require**'s that load optional functionality for the Lisp development environment. Some of the **require**'s in this file are explicitly forbidden by your Allegro CL Runtime or Allegro CL Dynamic Runtime license agreements, and those are identified by comments. Do not modify that file but rather copy it or the desired parts of it to your own file for use with your own application.

You should **require** the modules you need for your application to work properly. Check the *autoloads.out* file generated when the *autoload-warning* keyword argument to **generate-application** is specified true (the default when **generate-application** is called). That will tell you what modules might be autoloaded in your application and you can decide whether it is necessary to include the module. (Just because a module might be autoloaded, that does not mean that it will be autoloaded in your application. Modules are typically autoloaded when an important function associated with the module is called. If your application does not call a function that triggers an autoload, the autoload won't occur.)

*autoloads.out* actually identifies the files that will be autoloaded. A module name is the keyword symbol whose name is the filename. Thus, the name of the module associated with the file *acldns.fasl* is `:acldns`. The form `(require :acldns)` causes the file *acldns.fasl* to be loaded.

Note that the value of the *autoload-warning* argument can be a string naming a file (perhaps including directory information). If it is, the information will be written to that file rather than *autoloads.out*

If you determine a module which a function might autoload is not needed in your application, you should consider **fmakunbound**'ing the function, so calling it generates an undefined function error rather than a file not found error (in a runtime application with no bundle file, an error of some sort will be signaled when a function that triggers an autoload is called).

---

## 3.7 Defining the init functionality

Look at the file *sys:;src;aclstart.cl* (that is, the file *aclstart.cl* in the *src/* of the Allegro directory). It is the source for the startup routine used by Lisp (the function **start-lisp-execution**) along with the sources for some ancillary functionality.

(Note that the low-level initialization, including mapping of *.so/.dll* files built with the image -- systems libraries and others -- has already been performed when **start-lisp-execution** is called. As with any UNIX or Windows program, failure to find a needed shared object or Library file (*so/sl/dll*) during the low-level startup causes immediate program failure usually accompanied by a terse message identifying the unfound file. **list-all-foreign-libraries** can be used to identify dependencies on *.so*, *.sl* and *.dll* files.)

Examining the source for **start-lisp-execution** will tell you the exact sequence of operations -- when the command-line arguments are processed, when the init files are read, etc., so you can know in what order to do things. The startup sequence is also given in *startup.htm*.

There are several places where a programmer can intervene in the startup process. One relatively early place is the restart-actions list (the value of *\*restart-actions\**). Please note that this list may be used by Allegro CL or related functionality (such as CLIM). Therefore, you should add to the list but do not remove items from it or destroy it. Even earlier, *-e* command-line arguments are processed (assuming command-line arguments are not ignored by the image). Slightly later (but with a different interface), the *ACL\_STARTUP\_HOOK* environment variable is examined. If it has a value, that value is read by **read-from-string** and the result is evaluated.

The final two locations for programmer intervention in startup are the functions which are the values of *\*restart-init-function\** and *\*restart-app-function\**. As you may see from *aclstart.cl* (found in the *src/* subdirectory of the Allegro directory), if *\*restart-init-function\** is true, it is assumed to name a function with no arguments and that function is **funcall**'ed. The purpose of *excl:\*restart-init-function\** is to perform application initializations of any sort.

After *\*restart-init-function\** completes, either *\*restart-app-function\** is **funcall**'ed (if it is true) or a standard Lisp listener is started (but not both). If your application has its own top-level, it should be started with *\*restart-app-function\**. (Or if no top-level is needed, *\*restart-app-function\** would perform whatever your application does.) Note that the function that is the value of *\*restart-app-function\** must not return. The consequences are undefined if it does return.

It is entirely your choice whether you use your own top-level or use the Lisp top-level. Note too that if you are using the normal top level, that is the image was built with the *include-tpl* argument to **build-lisp-image** true, you can start a Lisp top-level at any time by evaluating the following form:

```
(tpl:start-interactive-top-level
 *terminal-io* 'tpl:top-level-read-eval-print-loop nil)
```

---

### 3.8 Specifying the initial value of *\*package\**

Programmers who use a Lisp listener as a top-level often want the current package (the value of *\*package\**) to be something other than the *common-lisp-user* package when the user sees the first prompt. Evaluating the following two forms as part of the function that is the value of *\*restart-init-function\** accomplishes this (replace *my-package* with the desired name, of course):

```
(tpl:setq-default *package* (find-package :my-package))
(rplacd (assoc 'tpl::*saved-package*
              tpl:*default-lisp-listener-bindings*))
```

```
'common-lisp:*package* )
```

The second form suppresses (on startup) the [changing package from ...] warning printed by Allegro CL when the value of `*package*` is changed by the system in a Lisp listener (usually printed when the debugger is entered).

---

## 3.9 Setting up logical pathname translations

All logical pathname translations (except the one for `sys:` set up in the low-level startup code) are cleared as one of the first actions of **start-lisp-execution** (indeed, the first form in the function definition):

```
(defun start-lisp-execution ()
  ...
  (flush-all-logical-pathname-translations)
  ...)
```

This is often not what an application developer wants. However, it is a conscious choice because the potential for mysterious bugs resulting from bogus translations being present in the image outweighed the cost of re-establishing the translations when needed.

If you want to use logical pathname translations in your application, then you will need to arrange for them to be present. You can do this in one of two ways:

1. Have `sys:hosts.cl` contain translations for additional hosts as required. Allegro CL, when it sees a logical host for which no translation is defined, will first look in `sys:hosts.cl` to see if a translation is defined there. This file may have to be configured at the customer site (since the directory structure is usually different in every system).
  2. Have another file where the translations are defined. Use **logical-pathname-translations-database-pathnames** prior to creating the image to tell the system about the additional translations file (and the system will look at that file in addition to `hosts.cl` when it encounters an unknown logical host). This solution is similar to the one just above but does not require modifying `hosts.cl` which may be inconvenient to modify.
- 

## 3.10 Use of shared libraries (foreign files)

**generate-application** causes all `.so/.dll/.sl` files loaded during image creation to be copied to the directory it creates. It also arranges, upon image restart, for these files to be reloaded from this directory (which is what will become `sys:`).

---

## 3.11 CLOS training

It is possible to significantly speed up the initialization of CLOS-based applications by gathering information about CLOS usage in the application and including that information in the application image. In this section, we discuss what information is useful for this purpose, how to collect it, and how to include it in an image.

**Note about the development environment:** Lisp users typically run in developer-environment mode. In that mode, the debugger, the code that runs the Emacs-Lisp interface, Allegro Composer (if ordered) etc. is loaded into Lisp. Often your application will not use those facilities and your application images will not want to include them. However, CLOS training will include information about those utilities (and then require you to have them available when you build your application image) unless you do one of the following:

- Train an image without the development environment.

This is the less-desirable solution. You can create an image without the development environment by specifying

```
:include-devel-env nil
```

in the call to **build-lisp-image**. You can then train with that image.

- Restrict training to specific packages.

This is the better choice. You will see forms like

```
(excl::preload-constructors ([packages]))
```

and

```
(excl::precache-generic-functions ([packages]))
```

below. If no packages are listed, the entire system (including development environment features) will be trained. If packages are listed, only things in those packages will be trained. Packages are named by keywords. Typically, packages should include `:user`, `:lisp`, and the packages of your application. Thus, if your application packages are `:foo` and `:bar`, the `preload-constructors` form would be

```
(excl::preload-constructors (:user :lisp :foo :bar))
```

The **precache-generic-functions** form would similarly list the packages.

### 3.11.1 Generic functions, method combination, and discrimination

A generic function examines its arguments to determine which method or methods are applicable. This is called discrimination. The symbol-function of a generic-function is a discriminator function. There are various kinds of discriminators and the discriminator for a particular generic function may change during the execution of a program.

When Lisp determines that a generic function needs a different kind of discriminator it checks to see if the one it needs has already been built and, if not, creates one. The creation of a discriminator is relatively expensive since it involves the Lisp compiler.

When a CLOS application is loaded the generic-functions all have a simple discriminator which will select the correct discriminator when the generic function is first called. Therefore when a CLOS application starts it will run very slowly

unless the discriminators it needs are already built. The only way to tell which discriminators your program needs is to run your program for a while and then look at the list of discriminators that exist. Allegro CL provides a mechanism for dumping out these discriminators and then loading them in with your program so that when the program starts all the discriminators it will need will already exist.

You can dump discriminator functions to a fasl file by compiling a source file that contains the following two lines after loading and running your application (note that the best optimization is achieved if you combine this with the caching optimization described below):

```
(in-package :excl)



---



```

### 3.11.2 Effective methods

With method-combination a call to a generic function can result in a sequence of methods being called. The code that calls the methods and processes the results of each call is called an effective method. In order to make effective methods fast, Lisp compiles them. In order to cut down on the compilation cost, Lisp actually creates effective-method templates which are functions closed over the particular methods to be called.

Thus many effective methods can share the same code. Just as in the case of discriminators above, it is expensive to start a CLOS application running if the effective methods it will need haven't been compiled already. And again Allegro CL provides a way of saving the effective methods that the application has used so that they can be defined before the application starts.

You can dump effective methods to a fasl file by compiling a source file that contains the following two lines after loading and running your application (this is the same as for discriminator functions):

```
(in-package :excl)



---



```

### 3.11.3 Caches for fast dispatching

Generic functions use caching to implement fast dispatching. When an application starts the caches are empty so initial performance is degraded by having to handle cache misses. Allegro CL provides a way to fill the caches when an image starts up.

You can dump caches to a fasl file by compiling a source file that contains the following two lines after loading and running your application (see [above](#) under the heading **Note about the development environment** for information on ([packages])) in the following form):

```
(in-package :clos)
(excl::precache-generic-functions ([packages]))
```

### 3.11.4 Constructor functions

As we describe briefly [below](#), calls to `make-instance` can be replaced with calls to some equivalent (but much faster) constructor functions. Allegro CL provides a way to preload compiled constructor functions.

You can dump constructors to a *fasl* file by compiling a source file that contains the following two lines after loading and running your application (see [above](#) under the heading **Note about the development environment** for information on ([packages]) in the following form):

```
(in-package :clos)
(excl::preload-constructors ([packages]))
```

Suppose those two lines are in a file named *myclosopt.cl*. Compile that file with a form like `(compile-file "myclosopt.cl")` to produce *myclosopt.fasl*. (You may use any filename, of course. We use a filename here so we can refer to the *fasl* file below.)

When you are generating an application with **generate-application** and you use this method to speed up constructor functions, you must include the module `:constructor` before the *fasl* file that contains the above call to **preload-constructors** (*myclosopt.fasl* in our example). These files and modules are typically in the list which is the value of the *input-files* argument to **generate-application** (the third required argument). Again, if the *fasl* file is named *myclosopt.fasl*, the value of *input-files* should be something like:

```
'(... :constructor ... myclosopt.fasl ...)
```

Of course, you may have some other way of specifying files and modules (so, for example, the *input-files* list contains one file which contains the loads and requires for your application. In that case as well, the `:constructor` must be required before the *myclosopt.fasl* is loaded.

### 3.11.5 How to do CLOS start up optimizations

The four possible start-up optimizations were just described. Conveniently, two (discrimination and effective methods) are achieved with the same utility. All depend on information being available. Therefore, the following is the first step for all optimizations:

1. Exercise your application. Load your application into a Lisp image and run it as you expect one of your users will run it. Doing this causes Lisp to gather experience about how CLOS is being used.

Once you have exercised your application sufficiently, you are ready to create the optimizing files. This is a standard *fasl* file created by compiling a special source file (described below).

2. Create the *clos* optimization *fasl* file. While Lisp is still running, create a Lisp source file (we call it *closopt.cl*) that contains the following four forms (see [above](#) under the heading **Note about the development environment** for information on ([packages]) in the following forms):

```
(in-package :excl)
(preload-forms)
(excl::preload-constructors ([packages]))
(excl::precache-generic-functions ([packages]))
```

Compile *closopt.cl* with **compile-file**. Lisp will put the discriminators, the effective methods, the constructors, and the contents of its CLOS caches into the resulting fasl file (*closopt.fasl*). If that file is built into the application binary image (it should be specified as one of the `:lisp-files` in a call to **build-lisp-image**, as an *input-file* in a call to **generate-application**, or required by another file specified in either location), then an application using CLOS will start up significantly faster.

Note that loading this file will not invoke the compiler so this can be loaded into a compilerless Lisp.

### 3.11.6 make-instance optimization

We have already discussed dumping **make-instance** constructor functions. Note that calls to **make-instance** where the value of the *class* argument is a quoted constant and each of the keywords is a constant are transformed by the compiler into calls to constructor functions. A constructor function is a piece of code that is equivalent to the **make-instance** call except that it is significantly (10 to 100 times) faster.

The optimization is automatic when the call to **make-instance** is formed in a particular way.

In order for an optimized constructor function to be used certain restrictions apply:

1. The set of keywords must be valid for the call.
2. Only certain methods must be applicable as defined by the following table:

Generic Function:	Condition for optimization:
<b>make-instance</b>	Only system-supplied methods are applicable
<b>initialize-instance</b>	Only system-supplied-standard methods and user-supplied <code>:after</code> methods are applicable
<b>shared-initialize</b>	Only system-supplied-standard methods and user-supplied <code>:after</code> methods are applicable

**Conditions for creation of constructor functions:** The calls to **make-instance** are replaced by calls to the constructor regardless of whether an optimized constructor can be used. The first time the constructor function is called, the system tests whether any of the restrictions apply. If none do, an optimized constructor is generated. When the restrictions are not obeyed, a non-optimized constructor function is created. It calls **make-instance**. Redefining a class or one of its superclasses or adding/removing a method to one of the generic functions mentioned above causes the constructor function to be recomputed.

## 4.0 Creating the deliverable

As stated above, **generate-application** assembles the files needed to deliver an application. **generate-application** both builds the application's image file and copies any other files needed to support this image.

Like many powerful Lisp functions, **generate-application** has many options and can do many things, but quite a lot can be done with simple calls. For example, the following call builds an application from *foo.fasl* into the directory *myappdir/* (relative to the current working directory of the running Lisp). You could then execute *myappdir/myapp* after the above form is evaluated.

```
(generate-application "myapp" "myappdir/" '("foo.fasl")
  :include-compiler nil)
```

**generate-executable** provides a simpler interface for generating applications.

### Important details of generate-application

**generate-application** is basically a wrapper around **build-lisp-image**. **generate-application** sets things up for producing a directory suitable for delivery and then calls **build-lisp-image**, which creates the actual deliverable image file. See *building-images.htm* for information on **build-lisp-image**.

**build-lisp-image**, and therefore **generate-application**, create a new image by starting a new Lisp process and having it load necessary files and then dump a final image. It is important to understand that the new Lisp process that does the work does not inherit anything from the Lisp process that spawns it. The arguments to **generate-application** and **build-lisp-image** specify details of the new image to be created, but outside the arguments, nothing is inherited (although some arguments do default to current values in the calling image). Thus, for example, if a user-defined function or package or variable or class is defined in the calling image, it will not be defined in the new image. Every aspect of the new image is defined by the arguments to **generate-application** and **build-lisp-image** or by the files loaded during the build process.

### The testapp example

There is an example showing how to create an application in *examples/testapp/*. It is more complex than the one above, but still relatively simple. See *readme.txt* in that directory for information on the example, which is designed to show how to package an application. The code in the *examples/testapp/* directory can be freely used. (Note that on Windows, you need the GNU **make** facility to use the example. The GNU **make** facility is (at the time of writing) available for free from <http://sources.redhat.com/cygwin/>.)

### More on generate-application

The definition of **generate-application** is:

```
(generate-application application-name
  destination-directory
  input-files
  &key allow-existing-directory
  application-administration)
```

```

application-files
(application-type :exe)
(autoload-warning t)
(copy-shared-libraries t)
(copy-file-function 'sys:copy-file)
debug
icon-file
demo
image-only
pure-files
purify
runtime-bundle
...build-lisp-image keyword args...
dumplisp keyword argument ignore-command-line-arguments...)

```

### The required arguments:

<i>application-name</i>	A string which is the name of the application (e.g., "myapp"). When coerced to a pathname, this name should not have a directory or type. It is used to create the name of the executable or .dll/.so/.sl file and ancillary files.
<i>destination-directory</i>	The name of a non-existent directory (the directory can exist if the <code>:allow-existing-directory</code> keyword argument is specified true). It is the directory used to create the output files. See <i>image-only</i> keyword argument below.
<i>input-files</i>	A list specifying files to be loaded into the application. The contents can be strings or pathnames naming Lisp ( <i>.fasl</i> or <i>.cl</i> ) files or keywords naming modules to be loaded with <b>require</b> . Note: this argument is passed to <b>build-lisp-image</b> as the value of the <code>:lisp-files</code> keyword argument to that function.

### The keyword arguments:

<i>:allow-existing-directory</i>	A boolean. If true, allows the <i>destination-directory</i> to exist. If the value of this keyword argument is <code>nil</code> (the default) and the directory exists, then an error is signaled.
<i>:application-files</i>	A list of files (strings or pathnames) which should merely be copied to the destination directory.

<b>:application-administration</b>	<p>This argument allows the specification of various application administrative tasks. The form of the value of this keyword argument is <code>(type-keyword ...)</code> or <code>((type-keyword ...) (type-keyword ...) ...)</code>. <code>(type-keyword ...)</code> can be: <code>(:resource-command-line "arg1" "arg2" "arg3" ...)</code></p> <p>On UNIX, this creates a <i>lisprc</i> in the destination directory which sets the default command line arguments to <code>"arg1" "arg2" "arg3" ....</code> See the <a href="#">Section 4.1 Resources</a> section below for more information. It also works on Windows using <b>win:set-default-command-line-arguments</b>.</p> <p>On Windows, the value can also be: <code>([:shortcut   :batch-file] filename ... command line arguments...)</code> [Windows only]</p> <p>This creates either a batch file or a shortcut named <i>filename</i> that will <i>initialize</i> the application. For a shortcut, the filename must have type "lnk" (the letter L, the letter N, the letter K). <i>command line arguments</i> are the arguments to <i>application-name.exe</i>. The <i>filename</i> argument should actually be a format control string given one argument, the name of the application. It is used like this: <code>(format nil filename application-name)</code>. This allows the customization of the generated filename based on information <b>generate-application</b> has already been given.</p> <p>One use of this is for OLE registration. For example:</p> <pre>(:shortcut "One-time registration of ~a" "-register")</pre> <p>If the given <i>application-name</i> was <code>"foo"</code> this would create a shortcut named <i>One-time registration of foo</i> with the following command line:</p> <pre>foo.exe -- -register</pre> <p>An error is signaled if <i>application-type</i> is not <code>:exe</code> in this case.</p>
<b>:application-type</b>	<p>A keyword specifying the application type. Valid values: <code>:exe</code>, <code>:ole-in-proc-server</code>, or <code>:dll</code>. If <code>:exe</code> is used, then <i>application-name.exe</i> is created. If <code>:ole-in-proc-server</code> or <code>:dll</code> is used <i>application-name.dll</i> is created.</p>
<b>autoload-warning</b>	<p>A boolean whose value can be string naming a file. When true, the file <i>autoloads.out</i> is created that contains the functions, macros and methods that could possibly be autoloaded. Defaults to <code>t</code> in calls to <b>generate-application</b>. Note that the default is <code>nil</code> in direct calls to <b>build-lisp-image</b>. The value can be a string naming a file (which can include directory information), in which case the information will be written to that file rather than <i>autoloads.out</i>.</p>
<b>:build-executable</b>	<p>This is a <b>build-lisp-image</b> keyword argument but is also used by <b>generate-application</b> if a value is supplied. The value must name a Lisp executable file (such as "mlisp" on Unix or "mlisp.exe" on Windows). It is used by <b>build-lisp-image</b> to start the Lisp process that builds the image. Unless <i>image-only</i> is true, <b>generate-application</b> copies a Lisp executable to the application directory. The executable specified as the value of this argument is the one copied.</p> <p>See the description of this argument in <i>building-images.htm</i> for information on when it is useful: a value is specified either when a custom executable has been built (see <i>main.htm</i>) or when you want a character size in the new image that is different from the character size in the running image. On this last point, see <i>The character size in the resulting image</i> in <i>building-images.htm</i>.</p>

<b><i>:copy-shared-libraries</i></b>	<p>A boolean. The value may be a lambda expression. If true, then copy shared objects/libraries that have been loaded with the Common Lisp function <b>load</b>, with the <i>system-library</i> keyword argument <code>nil</code>, by the time the image is dumped. (See <i>Using the load function</i> in <i>loading.htm</i> for details of the <i>system-library</i> argument.)</p> <p>The value of this keyword argument can also be a lambda expression (you cannot use the function special form: the value must be a list), accepting one argument, that is a predicate which determines if the loaded shared objects should be copied. The one argument is the pathname of the shared object file (the original pathname given to <b>load</b>). The predicate should return <code>t</code> if the shared object is to be copied, and <code>nil</code> if it is to be ignored by shared-library copying process. The predicate can also return a relative pathname, which will be used as the name of the copied shared library relative to the <i>destination-directory</i>. <b>Note:</b> shared libraries may be specified without a directory path and found using Operating System tools (such as <code>LD_LIBRARY_PATH</code> or <code>PATH</code>), as described in section <i>Load foreign code with cl:load</i> in the <i>foreign-functions.htm</i> document. However, the shared library copying functionality will not use Operating System tools to find shared libraries. If a shared library is specified without a directory path and is not in the current directory, it will not be found. The name of the shared libraries loaded on startup will be <i>sys:[name]</i>, for each <i>[name]</i> copied by <b>generate-application</b>.</p>
<b><i>:copy-file-function</i></b>	<p>If specified, the value should be a function object or a symbol naming a function. This function will be used to copy files to the destination directory. The default value is <b>copy-file</b> and that function is likely sufficient for most purposes. However, another function can be used if that is insufficient. This function will be called by the image that calls <b>generate-application</b> (not the image that builds the image). If the default is used, all keyword arguments to <b>copy-file</b> are called with their default values. It is simple to write your own function which calls <b>copy-file</b> with other argument values, if desired.</p> <p>Among the files copied are locale files (see the <i>include-locales</i> keyword argument). If <i>include-locales</i> is true and the default <i>copy-file-function</i> is used, the entire locale directory (about 2 Mbytes) is copied. If you do not want all locales copied, you must specify your own <i>copy-file-function</i> which will be selective about what locales to copy. See the description of <i>include-locales</i>.</p>
<b><i>:include-locales</i></b>	<p>A boolean, default is <code>nil</code>. If true, the <i>copy-file-function</i> is used to copy the <i>locales/</i> subdirectory and its contents of the Allegro directory to the application directory being created. Note that the size of the <i>locales/</i> subdirectory is about 2 Mbytes. You may wish to save space in your application by specifying a <i>copy-file-function</i> that copies only the locales that will actually be needed.</p> <p>You can also avoid copying locales if you know which specific locales an application needs. Instead of constructing a copy function for <b>generate-application</b>, it is simpler to include those locales in the application. Each locale is small, about 10K, so there is hardly any space savings in leaving the locales unloaded. Locales are included by having a file loaded by the application which contains top-level forms of the type <code>(find-locale &lt;locale&gt;)</code> (see <b>find-locale</b>). For example, you could have the file <i>my-locales.cl</i> as one of the files specified as the value of the <i>input-files</i> required argument, and this file would have something like the following contents:</p> <pre>;; file to cause specified locales to be loaded into an application (in-package :user) (find-locale "en_US")      ; English in the USA (find-locale "ja_JP")      ; Japanese in Japan (find-locale "fr_FR@euro") ; French in France using Euro currency (find-locale "fr_BE@euro") ; French in Belgium using Euro currency</pre>

	<pre>;; end of file</pre> <p>When you are unable to anticipate locale usage at runtime, the safest thing is simply to copy the entire directory and let the application autoloading.</p>
<b>:debug</b>	A boolean. If true, more information will be printed about progress as an aid to debugging.
<b>:icon-file:</b>	<p>[Windows only, ignored on UNIX.] If specified, the value must be a valid Windows .ico icon file. On the NT branch (which includes win2000 and XP), any .ico file should work. See <a href="#">Section 4.9.2 Icon files suitable as a value for icon-file</a> for more information on suitable icon files.</p> <p>If a non-nil value is specified, <b>win:set-application-icon</b> is called to embed the icon in the exe file.</p>
<b>:demo</b>	You must be licensed to produce demos (demonstration applications) to specify a non-nil value for this argument. If you are licensed, there is a maximum number of days that a demo will work specified in your license. The value of this argument, if specified and non-nil, should be an integer less than or equal to the maximum number of days that a demo is allowed to work. The application license written to the application directory will then be valid for that number of days. Contact your Franz Inc. Account Manager (send email to <a href="mailto:info@franz.com">info@franz.com</a> if you do not know who your Account Manager is) for information on the demo license.
<b>:image-only</b>	A boolean. If true, just build the image, and possibly the .pll file.
<b>:pure-files</b>	The value should be a list of .cvs and .str files to be put into the application's .pll file. See <i>Creating and using pll files in miscellaneous.htm</i> .
<b>:purify</b>	A boolean. If true, do automatic purification of Lisp and the application. This means all the strings and code vectors will be put into a .pll file. If you choose this option, do not also specify a value for :pure-files.
<b>:runtime-bundle</b>	A boolean. If specified t a bundle file named <i>files.bu</i> will be placed in the application directory. This file contains the modules allowable in a runtime image. This means that such modules need not be loading into the application image during the application build.
<b>build-lisp-image keyword args</b>	<b>generate-application</b> accepts and passes through to <b>build-lisp-image</b> all of build-lisp-image's keyword arguments except :lisp-files. The required <i>input-files</i> argument is used in place of :lisp-files. Even if a value is specified for :lisp-files, it is ignored. See also the description of the <i>build-executable</i> argument above, as it is used (differently) by both <b>generate-application</b> and <b>build-lisp-image</b> .
<i>dumplisp ignore-command-line-arguments keyword arg</i>	<b>generate-application</b> accepts and passes through to <b>dumplisp</b> the <i>ignore-command-line-arguments</i> keyword argument. When true, the resulting image will ignore command-line arguments prefixed by a dash (-). Command-line arguments prefixed by a + (used on Windows only) are never ignored. See <i>Command line arguments in startup.htm</i> for details of command-line arguments.

## 4.1 Resources

Resources are a way to specify default information for an application. The most common of which is command line arguments. Resources are handled differently on Windows and UNIX:

### UNIX

Resources are stored in a plain text file on UNIX. This file, *sys:lisprc*, if it exists can contain resource information for application startup. Currently, this is just command line arguments. The format of *lisprc* is:

```
.command-line: command line args...
```

or

```
appname.command-line: command line args...
```

where *appname* is the name of the Lisp executable used to start Lisp and *command line args...* are a list of valid command line arguments. *appname* should be used when there are multiple applications sharing the same directory and different command line argument resources are needed for each application.

For example, a *sys:lisprc* of

```
.command-line: -Q
```

would cause all applications in the directory this appears to start up quietly.

If there are both command line arguments in the resource file and given on the command line that starts the application, then command line arguments seen by the application are the concatenation of the resource command line and the given command line. This allows the given command line to override the resource command line.

## Windows

The newly exported function **windows:set-default-command-line-arguments** allows specifying the command-line arguments in an executable. Allegro CL executables come in two flavors on each platform: an 8-bit character version (for example *alisp8.exe* on Windows) and a 16-bit character version (for example *alisp.exe* on Windows). All Allegro CL executables (which, following standard Windows practice, have type *exe*) are copies of one or the other of *alisp8.exe* and *alisp.exe*. You can use **windows:set-default-command-line-arguments** to specify the command-line arguments for any such executable.

## 4.2 Defsystem

If the application is made up of many source files, then using the defsystem utility (described in *defsystem.htm*) will help the management (for compilation and loading) of the application. If defsystem is used, then it is easy, for example, to create a single *.fasl* file representing the compiled application. See *defsystem.htm* and the function **concatenate-system** for more information.

## 4.3 Tuning the application

The application should be optimized. Allegro CL contains space and time runtime analyzers that should be used to find places in the application which can be optimized. See *runtime-analyzer.htm*. The optimization of Common Lisp source code has two components, aside from optimizing algorithms used in the application:

- adding declarations, and
- setting the compiler optimization qualities speed, safety, space, and debug properly.

Both of the above can be done globally or locally to a particular function. Functions which are known to be used frequently should be optimized by declaring the types of the values bound to symbols, when the types are known and checked. Then, increasing the speed compilation quality and decreasing the safety and debug qualities will allow the compiler to produce smaller and faster code. These issues are discussed in *compiling.htm*. Note particularly the `:explain` declaration discussed in that document, in *Help with declarations: the `:explain` declaration*.

## 4.4 More on the development environment

If the application will not use the development environment of Allegro CL, then certain features of it can be turned off or compiled out of the application. For example, you might use the following global proclamation:

```
(proclaim '(optimize (debug 0)))
```

It will cause the compiler to compile with no consideration for easy debugging (presumably your users will not debug your application). Currently, this means local names of variables and the argument list for functions and macros will not be saved. Although the saving is not great, if these features are not to be used, then there is no reason to have the compiler annotate the *fasl* files with them.

Another saving can be achieved by evaluating:

```
(setf (argument-saving) nil)
```

This will cause the runtime calling sequence to be more efficient on some architectures (RS/6000 currently).

## 4.5 GC parameters and switches

Setting of GC parameters and switches appropriate to application-specific behavior is important for performance. *gc.htm* contains a complete discussion of the subject, and the only information included here is a check-list of items to consider.

### Switches

```
:clip-new (default: nil)
```

If keeping newspace small so that scavenges are short is important, then this feature should be enabled.

One negative aspect of this, however, is that garbage collections will be more frequent and this may cause more short-lived objects to be tenured, resulting in faster growth of the Allegro CL memory image. You should schedule global garbage collections more frequently to keep the image smaller if you set `:clip-new` to `t`.

```
:print (default: nil)
```

The users of many applications will not want to see the `gc` messages. Keeping this switch `nil` will prevent them from being printed. On the other hand, the `gc` message does explain why your application seems to have paused (during a `gc`). See also the discussion of `gc` cursors below.

## Parameters

```
:generation-spread (default: 4)
```

Depending on the behavior of the application, changing the generation spread may cause less garbage to be tenured. The default value has been chosen for development but not the runtime environment of applications.

```
:free-bytes-new-other (default: 131072)
:free-percent-new (default: 25)
:free-bytes-new-pages (default: 131072)
:expansion-free-percent-new (default: 35)
:quantum (default: 32)
```

These parameters determine the size of newspace after a scavenge. There must be at least `:free-bytes-new-pages` + `:free-bytes-new-other` bytes free, in addition to there being at least `:free-percent-new` percent of newspace free. `:quantum` specifies the number of pages (8k each) for newly created newspaces. The initial value is 32 for 256kb newspaces. `:expansion-free-percent-new` specifies the percent free in newly created newspaces.

```
:expansion-free-percent-old (default: 35)
```

This specifies how much must be free in an oldspace after it is created. A new oldspace is created because there is some amount of data that needs to be tenured and there is no current oldspace that can hold it.

See *gc.htm* for more information on memory layout.

## 4.6 GC cursors

A `gc` cursor facility provides some visual clue to the user that a garbage collection is taking place. Application writers find `gc` cursors useful since their users may think the application has hung while in fact it is just `gc`'ing. Unfortunately, implementing a `gc` cursor is difficult. See *Gc cursors* in *gc.htm* for more information.

In addition to the above parameters and switches, the variable `*global-gc-behavior*` determines whether or not a global gc is automatically performed when a certain number of bytes have been tenured (moved into oldspace). `*tenured-bytes-limit*` specifies this limit. It is very important to note that an interactive application would have execution suspended for an indeterminate amount of time if a global gc is performed--scavenges are normally quite short, in comparison.

The initial values for the above parameters are reasonable defaults, but there may be better defaults for individual applications. See *gc.htm* for more information.

---

## 4.7 Allegro Presto

In 7.0, the Allegro Presto facility has been removed. See *The Allegro Presto facility has been removed* in *loading.htm* for further information.

---

## 4.8 Allegro Runtime

Allegro Runtime is a Franz Inc. product which licenses distribution of applications written in Allegro CL. Please contact your Franz Inc. account manager if you want more information on Allegro Runtime and its terms. See the document *runtime.htm* for technical details of Allegro Runtime, including a list of restrictions on runtime images. **generate-application** only produces runtime images. The *runtime* defaults to `:standard` and must be either `:standard`, `:dynamic`, or `:partners`. See *runtime.htm* for allowable values.

---

## 4.9 Windows specific information

The file *msvcrt.dll* is needed by all Allegro CL applications generated on Windows. If the *copy-shared-libraries* argument is true, **generate-application** copies all needed system DLL's to a subdirectory of the *destination-directory* called *system-dlls*. When your application is installed, these DLL's should be copied to the Windows system directory *if necessary* (i.e. if they are not already there with the same or a later version). In the *system-dlls* subdirectory, they will not be seen by your application or any other program.

These system DLL's have presented a problem for Allegro CL applications. They are needed if the application is to run successfully but having them in more than one location where Windows sees them can create difficulties. Therefore, they cannot be put in the *destination-directory*, for that might result in two copies being visible (one there and one in the Windows system directory). And they cannot be copied blindly to the Windows system directory because that might overwrite an existing copy and result in a version mismatch (since other programs that depend on the copied-over later version may then fail).

We have tried to mitigate this problem by providing an install wizard (described in section [Section 4.10 Installation of your application on Windows using the Install Wizard](#) below) that does the right thing: it finds out if the DLL files in the *system-dlls* subdirectory are in the Windows system directory already. Any that are not are copied to the Windows system directory. The versions of the ones that are present are compared to the versions of the files in *system-dlls*.

Earlier versions are then updated (more precisely, are either then updated or things are arranged so they will be updated when Windows is restarted, so other running programs will not be affected).

Note that DLL's loaded with **load** with `:system-library` specified as true (see *Using the load function* in *loading.htm*) are not copied to *destination-directory* or the *system-dlls* subdirectory. The only files that are copied to the *system-dlls* subdirectory are *mfc42.dll* and *msvcrt.dll*. (This may seem counter-intuitive, but we feel free to copy *mfc42.dll* and *msvcrt.dll* because Microsoft explicitly allows it.)

**Autorun options.** On a different subject, it is possible to have a program autorun (that is, started automatically when, for example, a CD containing the software is inserted into the CD drive). See the MicroSoft document [http://msdn.microsoft.com/library/psdk/shellcc/shell/Shell\\_basics/Autoplay\\_cmds.htm](http://msdn.microsoft.com/library/psdk/shellcc/shell/Shell_basics/Autoplay_cmds.htm) for useful information.

## 4.9.1 The console window in applications

If your application is started so the console window is created (i.e. the `+c` command-line argument `-- no console --` was not specified on the command line), the function `console-control` provides control over whether the tray icon and/or the console window is visible, and whether you can exit the application using the tray icon and the close button on the console. The method `close-console` also provides control over the actions of the close button on the console (and the Alt-F4 key combination and the Exit choice on the tray icon menu).

## 4.9.2 Icon files suitable as a value for icon-file

The value of the *icon-file* to **generate-application**, if non-`nil`, must be a valid Windows icon file containing the icon that will be used for the application (in place of the bust of Franz Liszt icon used by Allegro CL). Such files normally have the *.ico* file type. A *.bmp* pixmap file or other non-icon image file will not work as an application icon.

## 4.9.3 Creating a Console APP on Windows

If you are unfamiliar with console apps, you probably do not want to create one.

Console apps should not use multiprocessing now **excl:run-shell-command** (in any form, which means do not use **excl:shell** or any of the subprocess API in `excl.osi -- os-interface.htm`). Doing so can cause the console app to hang. The problem has to do with I/O and the Windows console API and is a current Allegro CL restriction.

With that in mind, create a console app as follows:

1. Build your application (with a call to **generate-application** in the usual way. Assume the name (first argument) is "app" and the directory (second argument) is "dist").
2. Execute the following code (replacing "dist/app.exe" with the actual directory and application name):

```
(progn
  (delete-file "dist/app.exe")
  (sys:copy-file #+ics "sys;buildi.exe" #-ics "sys;build.exe"
    "dist/app.exe"))
```

In the application, using **format** to `*terminal-io*` will result in the output going to the console command window.

## 4.10 Installation of your application on Windows using the Install Wizard

The Allegro CL *Install Wizard* is a tool that application programmers writing Allegro CL-based applications can use to help deliver applications on the Windows platform. It is designed to deal with the problem of ensuring that system DLL's supplied with the application distribution are installed if necessary, but are not installed if a later version is already present. Note that the Install Wizard is a bare bones installer and is not meant to compete in the Windows installer market. We recommend the Install Wizard for only the simplest of installations. If your needs are anything beyond the bare bones, please consider using a commercially available installation program. We list some with which we are familiar next:

- <http://www.installshield.com/>
- <http://www.wisesolutions.com/default.htm>
- Installer/GD: <http://www.proggie.com/>
- And a site that provides general information on Windows installers: <http://www.installsite.org/>

In accordance with your license agreement, you can distribute Allegro CL-based applications. The files in the directory described in **1. Create the initial application directory** below are typically suitable for distribution, but again, this is controlled by your license. Ask your Franz Inc. account manager for information on what your license allows if you are unsure. We assume in the remainder of this section those files are licensed for distribution.

**Reasons for providing the Install Wizard.** You can create a delivery directory, either with **generate-application** or the IDE's **File | Build Project Distribution** menu command. On Windows, this directory includes a *system-dlls/* subdirectory. System DLL's needed for the application, such as *mfc42.dll* and *msvcrt.dll*, are placed in that directory.

As described in section [Section 4.9 Windows specific information](#), there is a problem when you install your application on a customer's machine: you must ensure the right thing is done with the system DLL's. They should be installed on the customer's machine if necessary but should not be installed if not necessary. So, it must be determined whether such installation is necessary. Further, when installing on Windows, various bookkeeping tasks like updating the registry must be performed. A program, usually named *setup.exe*, is typically provided to perform such tasks. The Install Wizard generates an appropriate *setup.exe*. **Note: the user installing the application must have administrator privileges on Windows NT and Windows 2000.**

Here are the steps for using the Install Wizard. Note that the Install Wizard will ensure that the user doing the installation has the administrative privileges necessary to install the application.

### 1. Create the initial application directory

Use either **generate-application** or the IDE's **File | Build Project Distribution** to create a directory, which for

example purposes we will call *c:/foo/foo/*. This will be a directory of files and subdirectories. This directory is named by the *destination-directory* argument to **generate-application** and by the *Distribution directory* dialog when using the **Build Project Distribution** menu command.

When **generate-application** or **Build Project Distribution** complete, you now have the initial application directory.

## 2. Run the Install Wizard

Run the Install Wizard via a shortcut on the **Start | Programs | Allegro CL submenu**. This will display the Install Build Wizard dialog, which has the following fields:

- **Company name:** The name of your company (e.g., *Foo, Inc.*).
- **Application name:** The name of your application (e.g., *WinFoo*).
- **Version number:** The version number of your application (e.g., *1.0.20*).
- **Source directory:** The directory containing the output of **generate-application** or **Build Project Distribution** (*c:/foo/foo\* in our example)
- **Output directory:** The name of a *non-existent* directory.

The Install Wizard will create a new directory (called the *Output directory*) and copy the files from the source directory to it. Further, it will generate a program named *setup.exe* and put it in the output directory.

The following buttons are on the bottom of the dialog: **Build**, **Quit** and **Help**. After filling in the fields, click on the **Build** button to create the directory specified in *Output directory*. **Quit** will exit without building anything. Clicking on **Help** displays this section of this document in a browser.

When the Install Wizard completes, the output directory is suitable for distributing to application users. We assume it is placed on a CD. How the CD is organized is up to you, the application writer. We assume the CD contains a single directory *distdir/* which contains the contents of the output directory from above. (At the toplevel, it may have a file causing it to autorun when the CD is inserted into the drive. In any case, *setup.exe* must be in the same location with respect to all other files as it was in the output directory.) Note, this will work equally well if you are distributing on floppies or some other media.

Once you have the CD, then you as application developer have finished creating the distribution.

## 3. On the application user's machine

The application user (presumably your customer) receives the application CD and puts it into the CD drive. On the application user's machine, there is (presumably) no Allegro CL, and the source and output directories mentioned in 2 above do not exist either. Recall we assume the CD contains a single directory *distdir/* which contains the contents of the output directory from 2 above. That directory includes the installation program *setup.exe*. **To run *setup.exe* on Windows NT and 2000, the user must have administrator privileges.**

When the application user runs *setup.exe* (in our example, by inserting the CD into the drive and running *distdir/setup.exe*), *setup.exe* will ask the user for an installation directory for the application, which we will call *appdir*. This directory must not exist. *setup.exe* will create the directory and install the application. Specifically, *setup.exe* will do the following:

1. Create *appdir* and copy the application specific files to the *appdir* on the user's machine.

2. Determine which of the DLL's in the *system-dlls/* subdirectory need to be installed. It will either immediately copy any that do need installation to the Windows `system' directory, or will arrange for them to be installed upon the next reboot of Windows.
3. Update the registry. This includes adding or updating to `\\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs`.
4. Copy the uninstallation program to `[appdir]\uninstall\`. (See [Section 4.10.1 Uninstalling an application on Windows](#) for important information on uninstalling an application.)
5. Adds uninstall entries to the registry key `\\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\uninstall\`. (See [Section 4.10.1 Uninstalling an application on Windows](#) for important information on uninstalling an application.)
6. On some Windows versions, if shared DLL's were to be installed which already existed, then tell the end-user to reboot to complete the installation. On other Windows versions, this doesn't happen.

If at any time the installation program gets an error, it will undo any changes made to the disk containing *appdir* and to the Windows registry. That is, if the application does not install, then *appdir* will not exist and no application files will be left on the user's machine, and the registry will be left in the pre-installation state.

---

## 4.10.1 Uninstalling an application on Windows

Part of the installation process done by the Install Wizard on an application user machine is the creation of an uninstallation program and the addition of appropriate entries in the registry having to do with uninstalling the application. An *uninstall/* subdirectory is placed in the application directory. It contains the programs necessary for uninstallation.

**Uninstallation of the application must be done using the Add/Remove Programs control, which can be displayed using the Windows Control Panel.** User should not directly run the programs in the *uninstall/* directory.

---

## 4.11 Testing your application

It is best to test an application on a different machine from the one it was developed on, as this ensures that machine-specific dependencies (presence of system libraries, for example) are better controlled for. However, even testing on the same machine can provide useful information. Because most necessary files are in the application directory, even after installation, the machine dependency problem is reduced. (Note that on Windows, Allegro CL DLL's are installed in system directories so you must be sure that the application is not working just because the ones from the Allegro CL installation are present.)

---

## 4.12 Expiration warnings

Evaluation copies of Allegro CL have a built-in expiration date (paid-up licensed copies do not have an expiration date). Applications built from evaluation copies will display the following when started:

```
WARNING: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
WARNING: Application will expire on [date] (XXX days).
WARNING: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

where [date] is a specific date in the future and `XXX' is the number of days until expiration. If you believe you have a paid-up, licensed copy of Allegro CL and you see that warning when your application starts, please contact Franz Inc. for advice, since such a warning should not appear. (Submit a bug report as described in *Reporting bugs in introduction.htm*.)

---



---

## 5.0 generate-executable: a wrapper for generate-application

The function **generate-executable** is a wrapper for **generate-application**, producing an application whose input is the command-line arguments. It can be used either as a quick way to create an application, or as an example of using **generate-application**. The source for **generate-executable** can be found in the file *[allegro-cl-dir]/src/genexe.cl*.

---



---

## 6.0 Patching your application after delivery

There may be bugs in the version of Allegro CL used to deliver your application and there may be bugs in your application code. Both need to be fixed for your users. Allegro CL bugs are typically fixed by patches supplied by Franz Inc. You, of course, have to decide how to provide fixes for bugs in your application, but you may wish to mimic the patch system used by Allegro CL. That system and the tools associated with it are the subject of this section.

The following two features are useful in a patch system:

- The patch files should be named following a predetermined convention, so users and the system know what is and what is not a patch file.
- The fact that a patch is loaded into an image should be recorded in the image in some fashion.

The tools provided support both features. The tools are the **defpatch** macro, the **load-patches** function, and the **featurep** predicate function.

---

### 6.1 The Allegro CL patch naming scheme

We first describe the Allegro CL patch scheme and then discuss how you can adapt it to your application's needs. Allegro CL patch files are named as follows:

```
p[m][p][n].[v]
```

So the first letter is *p*, followed by

- [m], one character denoting the version of Allegro CL, followed by
- [p], one character denoting the product, followed by
- [n], three characters denoting the patch sequence number, followed by a period (.), followed by
- [v], three characters denoting the patch version.

The version ([m]) changes with each new release. Its value is the value of `excl::*cl-patch-version-char*`. For example, `pca001.001` is the first version of the first patch file on version 9 (Allegro CL 8.0) for product `a` (Allegro CL base Lisp). `pca001.002` is the second version of the first patch file. `pca011.002` is the second version of the eleventh patch file.

## 6.2 Loading Allegro CL patches

All Allegro CL patches are placed in one directory, `sys::update`, that is the `update` subdirectory of the Allegro directory, where Allegro CL was installed.

Patches are loaded by **load-patches**. It takes only keyword arguments and the arguments are:

<code>update-directory</code>	The directory in which to look for patch files. Defaults to the Allegro CL patch directory, <code>sys::update</code> .
<code>product</code>	Value should be <code>nil</code> , meaning load all patch files regardless of the product code (the third letter of the filename, [p] <a href="#">above</a> ), or a character or list of characters, meaning load only those files whose product code (third letter) match the single character or is in the list of characters.
<code>patch-file-filter</code>	A function of three arguments, a product code, a pathname, and a version ([m] <a href="#">above</a> ). Returns true if the pathname names a valid patch file (based on parsing the name and location only).
<code>patch-file-sorter</code>	A function of three arguments, a product, a list of patch files (validated by the <code>patch-file-filter</code> ), and a version ([m] <a href="#">above</a> ). Sorts the list into the order in which the files should be loaded (from highest version to lowest).
<code>version</code>	Specifies the version ([m] <a href="#">above</a> ). Should be a character object naming a decimal digit ( <code>#\0 - #\9</code> ) or a lowercase alphabetic letter ( <code>#\a-#\z</code> ). This is for use with application patches only. Defaults to the value of <code>excl::*cl-patch-version-char*</code> .

You should have an `update/` subdirectory to your application directory (or wherever `sys:` translates to in your application). Then you can distribute post-loadable Allegro CL patches file to customers. *post-loadable* means that the patch can be loaded into an existing image. However, not all Allegro CL patches are post-loadable. You must distribute a new image with patches loaded if you need to deliver a non-post-loadable Allegro CL patch to your customers.

Patches are loaded latest version first and earlier versions are only loaded if the later version fails for some reason.

## 6.3 Patches for your application

The easiest way to provide loadable patches to your own application is to have a separate directory (say `sys::myapp-update`) where your patches will go. Then mimic the Allegro CL patch naming scheme and call **load-patches**, specifying `update-directory` to be the directory you chose. **load-patches** should be called when your application starts up. As long as your patch files are created with **sys:defpatch**, the scheme should work with your application. Make sure that the version (`[m]` parameter) is the value of `excl::*cl-patch-version-char*` for the version of Allegro CL you are using.

If you want to use a different naming scheme, you will have to supply your own `patch-file-filter` and `patch-file-sorter` functions. See the description of **sys:load-patches** for advice on how to do that.

Again, please do not mix your application patch files with Allegro CL patch files in the same directory (unless you use your own naming scheme that cannot be confused with the Allegro CL naming scheme, and even then it is a bad idea). Franz Inc. reserves the right to use any product code at any time and so you cannot guarantee the uniqueness of filenames simply by using an apparently unused product code.

The value of the variable `sys:*patches*` is a list of loaded patches.

## 6.4 Creating patch files in general

The following table describes the three attributes of patch files.

Attribute	Meaning
<i>post-loadable</i>	Can be loaded into a running image (so named because loadable after <code>-- post --</code> the original image build).
<i>superseded</i>	This attribute is no longer used. Since the latest version of a patch is loaded (and if it cannot be loaded, the next latest, and so on), a superseded version is not looked at. Therefore this attribute is not necessary.

## 6.5 Creating a patch file

A patch file is a compiled Lisp file. At the start of the patch file, there should be a **sys:defpatch** form, followed by the code that implements the patch. Therefore, a skeleton patch file will look like the following:

```
;; Our application
;; patch for report XXXX

(sys:defpatch "mpnnn" 1 ;; replace mpnnn with the product version (m),
                    ;; product code (p), the patch id number (nnn) and
                    ;; 1 with the patch version
```

```
"MESSAGE"           ;; Brief patch information (should fit on one line)
:type :myapp         ;; Type should be a keyword of your choosing.
;; Other arguments may be specified.
)

;; Put patch code after here ...

(in-package :blah)
```

The required arguments to **sys:defpatch** are:

*id*

A string identifying the patch number or name. This is usually the [m][p][nnn] of the patch file name and typically includes zero-filled numeric characters -- e.g. "0a001", "1j195", "0z234" -- but can include alphabetic characters and need not be exactly five characters long. It is not the patch file prefix. This id is unique to the patch.

*version*

A fixnum in the range 1 to 999 inclusive. This is the [v] of the patch file name.

*desc*

A string containing a brief description of the patch. Short strings are better because this string is printed by **dribble-bug** when it reports information of patches and long strings may mess up the printing (by forcing line wraps). Example "Fixes filename bug" or "Speeds up processing employee info".

The keyword arguments to **sys:defpatch** are:

*type*

A keyword specifying the type of the patch. Default is :unknown. Application programmers should decide on a single type or a group of types for their application and classify their patches according to that scheme. When information on patches in an image is printed by **dribble-bug**, they are organized by type. The following types are reserved by Allegro CL and should not be used by application programmers: :lisp, :aclwin, :clim, :system, and :allegro\* (any keyword starting with :allegro).

*defpatch-version*

Default is 1. If a new version of sys:defpatch is supplied by Franz Inc., the default will be changed and patches with the old version will be rejected. In general, do not worry about this argument unless a new version of **sys:defpatch** is distributed (that distribution will include additional instructions).

*post-loadable*

Default `t`. When `t`, the patch file can be loaded into a running image. When `nil`, the patch file can only be included in an image during image creation with **build-lisp-image**. The patch load will abort if **load-patches** tries to load it into a running image.

### *feature*

Default `nil`. When true, value can be any form acceptable as an argument to **featurep**. If **featurep** returns `nil` when applied to the form, the patch loading is aborted. The reason for aborting printed by the system is the form that is the value of this argument (made into a string).

### *compile-feature*

Default `nil`. When true, value can be any form acceptable as an argument to **featurep**.

The *compile-feature* keyword argument is designed to facilitate producing patches for different platforms. For example, suppose a patch is only applicable to versions of Allegro CL that use `os-threads` for multiprocessing. Specifying `:os-threads` as the value of *compile-feature* will cause compilation to proceed when compiled by a platform that uses OS threads but to abort when compiled by a non-`os-threads` Allegro CL. Aborting is what you want in that case, since the patch is not needed for such platforms. The aborting of compilation will signal a condition which looks for a `sys::abort-patch-compiling` restart. If that restart is not present, an error is signaled (and the programmer must intervene to do something). More typically, compilation of patch files are done in a form like the following:

```
(dolist (x patch-files)
  (restart-case (compile-file x)
    (sys::abort-patch-compiling (patch)
      ;; Actions of your choice, e.g printing a message like:
      (format t "Aborted patch file ~s, featurep returned
nil"
              x))))
```

Compilation of the remaining patch files will continue and all relevant patch fasl files will be present when the **dolist** form completes.

## 6.6 What to do with patch files

How you and your application team will manage patch files depends on how you deliver your application and whether or not your customers can build new images with **build-lisp-image**. Only customers of properly licensed VARs and customers who hold an appropriate license from Franz Inc. for Allegro CL will be able to build new images. Customers (of yours) who receive runtime images (and are not independently licensed by Franz Inc.) cannot make new original images because **dumplisp** (called by **build-lisp-image**) does not work in runtime images.

This is an issue because (1) patches which are not post-loadable (i.e. cannot be loaded into a running Lisp) can only be included in a new original image; and (2) post-loadable patches can be loaded into a running image but should not be loaded into an image which already contains them. Therefore, if you have runtime customers (who cannot build original images), you can send them post-loadable patches and arrange for those to be loaded automatically, but you may also send them new images from time to time (which include non-post-loadable patches but will usually include all available post-loadable patches as well). You must ensure that such users do not load the post-loadable patches in their possession which are already included in the current image.

Here is a possible scheme which will work for applications which are distributed as runtime images. (This is not the only possible scheme or even the best for your situation. It illustrates how the tools and their features can be used to produce a scheme that works.)

1. Your application, *myapp*, is distributed in a directory. That directory contains a subdirectory *update-myapp/* and your users are informed to put patches from you in that directory. Patch files are named *p[m]a[nnn].[vvv]*.
2. As part of the *\*restart-init-function\**, the following **load-patches** forms are evaluated:  

```
(sys:load-patches :update-directory "sys:iupdate-myapp;" :product #\a)
(sys:load-patches);;
```

to load any Allegro CL patches in *update/*  
Each time the application is started, all patch files named in the *update-myapp* directory will be loaded automatically along with any in *update*.
3. You create a new image for distribution to customers. This image includes all *myapp* patches, as well as non-post-loadable patches (which you have been testing internally) and perhaps other features and enhancements unrelated to patches. In this image, the first **sys:load-patches** form in *\*restart-init-function\** is changed to  

```
(sys:load-patches :update-directory "sys:iupdate-myapp;" :product #\b)
```
4. You distribute the new image to customers, telling them to delete all *p[m]a[nnn].[vvv]* files from *update-myapp* and remarking that patches associated with this image will be named *p[m]b[nnn].[vvv]*.

Note that even if your customer ignores your instruction to delete the *p[m]a[nnn].[vvv]* files from *myapp-update*, those (no longer valid) patches will not be loaded because **load-patches** is looking for product *b* and those files have product *a*.

## 6.7 Including application patches in an image at build time

The discussion under the previous heading concerns distributing patches to users who cannot themselves build an original image with **build-lisp-image**. You, however, will build original images (and perhaps your customers are licensed to do so as well). How do you include patch files in the image when it is built? You put appropriate **sys:load-patches** forms in *custom.cl* and make sure all your patches are in the directories specified in the **sys:load-patches** forms. Allegro CL patches can be put in the *update* subdirectory (they will be included automatically by the image build process). See *Use of custom.cl* in *building-images.htm*.

## 6.8 Superseding a patch

Say you have sent a patch, *p0a001.001* to your users (the first 0 would actually be the value of `excl::*cl-patch-`

version-char\*). The **sys:defpatch** form at the top of the patch file is:

```
(sys:defpatch "0a001" 1 "Fixes whatever" :type :myapp)
```

A user complains that including the patch fixes the problem reported but seems to cause another problem. You check it out and find that the patch does introduce new problems. So you create a new version. You put it into a file and compile it to *p0a001.002*. The **sys:defpatch** form at the head of the file is:

```
(sys:defpatch "0a001" 2 "Fixes whatever" :type :myapp)
```

When the user gets the new patch, the patch loader will load it. But you should also provide a new, withdrawn version of the original patch file. Then the actual bad patch file no longer exists and cannot be accidentally loaded.

## 6.9 Withdrawing a patch

Bad (for whatever reason -- introduces a new bug or does not do what it was supposed to or whatever) patch files should be simply withdrawn. A speed-enhancement patch which actually slows things down is one example (your idea for a speedup failed and you do not have other ideas). We recommend replacing the defective patch file with a new file (same filename and version, so the bad file is overwritten) marked withdrawn. This shows the fact that the patch is withdrawn in the **dribble-bug** output. So, the **sys:defpatch** form in the replaced *p0a001.001* would be (like above):

```
(sys:defpatch "0a001" 1 "Fixes nothing" :type :myapp :withdrawn t)
```

When the compiler processes this **sys:defpatch** form, it stops compiling the file. Therefore, you can leave the original patch source (for later reference) without worrying that the patch fasl file will be larger than necessary or contain bogus compiled code.

## 6.10 Distributing patches

In the era of the World Wide Web, ftp, and users around the world, a typical way to distribute patches to users is having them download the patches from an ftp (or www) site directly into the appropriate directory without actual human contact between you and your users. If you use this model, you should tell your users to download every patch, you should use the version mechanisms we describe above, and you should tell your users to expect files to be overwritten.

You can also, of course, distribute patches on request, one at a time, with instructions (which usually include `delete all earlier versions of this patch!') The more patch distribution has a human-contact element, the less you have to worry about old version and bad patches not being deleted. The more the system is automated, with less handholding, the more being very careful about version numbers becomes necessary.

## 6.11 Loading patches

Patches are compiled lisp files, and such files can be loaded in a number of ways. There is no reason a post-loadable patch file cannot be loaded with **load**. Often that is useful for quick tests. However, Allegro CL provides a patch-loading function carefully integrated with the patch system described in this section. As far as possible we recommend that you load with **sys:load-patches**.

The function **featurep** returns true or nil as the features called for in its argument are or are not on *\*features\**. It is thus a functional analog of the #+/#- reader macros. It is used by **sys:load-patches** to process the *feature* argument in **sys:defpatch** forms.

Things to note

- No attempt has been made at this time to allow for co-requisite patches. Pre-requisite patches can be created using the *feature* keyword, but the best philosophy, if possible, is to always create patches in self-contained files.
- Loading patches with **load**. Patch files are *fasl* files that can be loaded with **load** and its friends but note the following: if something causes the load of a patch to be aborted (the required features are not present, the patch is not post-loadable, a patch with the same id is already loaded) the system looks for a restart named `sys::abort-patch-loading`. If that restart is found, it is invoked. If no such restart is found, an error is signaled. **sys:load-patches** sets up the restart but no such restart will (likely) be present when loading a patch file with **load**. Therefore, using **load** is best done with single or just a few patches but **sys:load-patches** should be used for any automated procedure or when many patches are involved. Note that loading patches marked withdrawn does not signal an error.
- Never lambda-bind *\*features\** while loading a patch since the patch file may add a feature to that list, but the addition will be lost when the binding is undone.