

Building Images

This document contains the following sections:

- [1.0 Comparison with `excl:dumplisp`](#)
- [2.0 Comparison with `excl:generate-application`](#)
- [3.0 The template of a call to `build-lisp-image`](#)
 - [3.1 The character size in the resulting image](#)
- [4.0 Arguments to `build-lisp-image` 1: defaults inherited from the running image](#)
- [5.0 Arguments to `build-lisp-image` 2: defaults not inherited from the running image](#)
- [6.0 Debugging an image build problem or failure](#)
- [7.0 Use of `custom.cl`](#)
- [8.0 Building an image to include patches](#)
- [9.0 Minimal top levels](#)
 - [9.1 Using the default minimal top-level](#)
 - [9.2 Requiring the normal top-level in a minimal top-level lisp](#)
 - [9.3 Top-level variables](#)

There are many links to **build-lisp-image** in this document. In fact, essentially all relevant information about **build-lisp-image** is contained in this document.

The function **build-lisp-image** can be used to create a new image (*dxi*) file. This will be a fresh image, fresh in the sense that it inherits little from the running image (only values of certain global variables used as argument defaults). Typical reasons for building images with **build-lisp-image** include:

- Building a new image that includes patches.
- Building a custom image which includes local functionality or with locally-desired initializations.

build-lisp-image is a lisp function, so it is called from a running Lisp. We distinguish between the image which calls **build-lisp-image**, the *running image*, from the image being created, the *new image*.

build-lisp-image spawns a separate process. On Windows, a "Create image" window displays information about the build process (you may have to expose this window). On Unix, information is printed to the listener. You will see many prompts: do not attempt to type to them.

1.0 Comparison with `excl:dumplisp`

Images can also be created with **dumplisp**, as described in the documentation for that function and in the document **dumplisp.htm**. An image created by **excl:dumplisp** is essentially a copy of the currently running image, with all loaded functionality included and all values of variables preserved. In contrast, **build-lisp-image** spawns a separate process which builds a new image out of constituent parts. The currently running image supplies only default values for certain arguments. The new image does not capture loaded or newly-defined functionality of the running image (thus, if the function **user::foo** is defined in the currently running image, it will not be defined in the new image created by **build-lisp-image**).

This is why **build-lisp-image** can be used to create a new image with new patches. Patches in the currently running image are not captured by the newly created image. Instead (as described below), patch files are built into the newly created image but the patches in the current image are not involved.

Note that **build-lisp-image** calls **dumplisp** in the spawned process. All arguments accepted by **dumplisp** are also accepted by **build-lisp-image** except the *name* argument (which is accepted but ignored since the required argument to **build-lisp-image** specifies the new image name).

2.0 Comparison with `excl:generate-application`

`generate-application` itself calls `build-lisp-image`. `generate-application` is designed to produce a directory of files suitable for shipping to another machine or site (note that you must be licensed to distribute software built in Allegro CL – contact your Franz Inc. account manager if you are unsure of licensing terms). In contrast, `build-lisp-image` builds a single image file only. See the description of `generate-application` and the document `delivery.htm`.

3.0 The template of a call to `build-lisp-image`

The "BUILD" module must be loaded for `build-lisp-image` to work (that is, "BUILD" should be on the `*modules*` list). It will be loaded automatically when `build-lisp-image` is called. You can force loading of the module by evaluating

```
(require :build)
```

`build-lisp-image` takes one required and many keyword arguments. The required argument must be a string naming a file, with or without path information. If no path information is specified, the image file will be placed in the current directory (as returned by `current-directory`). If relative path information is supplied, it will be resolved relative to the current directory. The image file name must have an extension (type). If it does not, it will not be found when Lisp is started. The standard extension (type) of an image file is `.dxl` but any extension will do. Note that on Windows machines, `.dxl` is registered and associated with Allegro CL, so double-clicking on a `dxl` file (in the Windows Explorer, say) executes `mlisp.exe` with `-I [that file].dxl` as arguments. When `mlisp.exe` is called without an image specified with `-I`, it looks for an image (`dxl`) file with the same name (i.e. `mlisp`) and in the same directory. You can copy `mlisp.exe` to `[anything else].exe` if you want it to find a different image file named `[anything else].dxl` automatically.

Here is the template for a call to `build-lisp-image`:

```
(excl:build-lisp-image image-file
 &key ...keywords...
      ...dumplisp-keywords...)
```

3.1 The character size in the resulting image

Allegro CL supports two character sizes: 16-bit and 8-bit. Thus images and executables either use 16-bit characters or 8-bit characters. No image or executable supports characters of both sizes. See `iacl.htm` for information on character sizes in Allegro CL.

`build-lisp-image` creates a 16-bit character or an 8-bit character image as the new image is created using a 16-bit executable (`mlisp` and `alisp` on UNIX, `mlisp.exe` and `alisp.exe` on Windows) or an 8-bit executable (`mlisp8` and `alisp8` on UNIX, `mlisp8.exe` and `alisp8.exe` on Windows). The `build-executable` keyword argument to `build-lisp-image` specifies the executable that will be used to create the new image. The value of that argument defaults to the executable used to start the running image.

Therefore, if you want the character size of the new image to be the same as the character size in the running image, you need not specify a value for the `build-executable` keyword argument. If you want a different character size, specify an appropriate executable as the value of that argument.

For example, if you are running a 16-bit character size image and you want to build an 8-bit character size image, call `build-lisp-image` like this:

```
(build-lisp-image "my-image.dxl"
 :build-executable "mlisp8" ;; On Windows "mlisp8.exe")
```

```
;; < other keyword arguments and values >
)
```

(That call also works if you are running an 8-bit executable already, of course.)

For example, if you are running an 8-bit character size image and you want to build a 16-bit character size image, call **build-lisp-image** like this:

```
(build-lisp-image "my-image.dxl"
 :build-executable "mlisp" ;; On Windows "mlisp.exe"
 ;; < other keyword arguments and values >
)
```

(That call also works if you are running a 16-bit executable already, of course.)

Note: Unless you have created your own executable (which is uncommon, but see **main.htm**, all 16-bit character size executables are identical and all 8-bit character size executables are identical. Therefore, any executable of the correct character size may be specified.

4.0 Arguments to build-lisp-image 1: defaults inherited from the running image

Certain defaults for keyword arguments to **build-lisp-image** are inherited from the currently running image. It is important to understand that these inheritances are the only effect that values in the currently running image have on the new image being built.

| Argument | Default | Description |
|---------------|---|--|
| :case-mode | *current-case-mode* | Sets the case mode of the new Lisp image. The default is the case mode of the running image. The value can be one of. <ul style="list-style-type: none"> • :case-sensitive-lower (the Modern Common Lisp value) • :case-insensitive-lower • :case-insensitive-upper (the standard Common Lisp value) |
| :dst | *daylight-saving-time-observed* | Controls daylight savings time inclusion in time computations. If <code>t</code> , United States Daylight Savings Time schedules are used. If <code>nil</code> , daylight savings time is assumed to never be in force. There is no direct support for non-United States schedules. Users outside the United States can contact Franz Inc. technical support (support@franz.com) for assistance in implementing a different schedule. |
| :include-clim | <code>t</code> if <code>clim</code> is in the currently running image, <code>nil</code> if it is not. | When true, include CLIM in the resulting image. |

| | | |
|---------------------------------------|--|---|
| <code>:include-common-graphics</code> | This argument is no longer supported. | Windows only. If you are building a development image, specify <code>:include-ide true</code> and Common Graphics and the IDE will be in the image. If building an application image, use the project system in the IDE. |
| <code>:include-compiler</code> | <code>t</code> if the compiler is in the currently running image, <code>nil</code> if it is not. When building a standard runtime image (i.e. when the <code>:runtime</code> argument (in the next table) is specified <code>:standard</code> and you are calling generate-application , not build-lisp-image), the compiler cannot be in the image created. Thus, either this argument must be specified <code>nil</code> or both this argument and <code>:discard-compiler</code> must be specified <code>t</code> . | When true, include the compiler as the resulting image is being built. Whether the compiler is in the built image depends on the value of the <code>:discard-compiler</code> argument. When <code>nil</code> , the compiler will not be available at any point during the image building process or in the built image. See the <code>:discard-compiler</code> option in the next table. |
| <code>:include-composer</code> | <code>t</code> if Allegro Composer is in the currently running image, <code>nil</code> if it is not. | UNIX only. When true, include Composer in the resulting image. |
| <code>:include-debugger</code> | <code>t</code> | (The default for this argument is not inherited from the running image, so this argument belongs in the next table. It is repeated here to be with the other <code>:include-*</code> arguments.) When true, include the debugger in the resulting image. |
| <code>:include-devel-env</code> | <code>t</code> if the development environment is in the currently running image (i.e. <code>:develenv</code> is on the <code>*modules*</code> list), <code>nil</code> if it is not. | When true, include the non-graphical development environment in the resulting image. The file <code>sys::develenv.cl</code> (i.e. <code>develenv.cl</code> in the Allegro directory) lists the modules (as arguments to require) that make up the development environment. This argument must be specified <code>nil</code> when <code>:runtime</code> is true since some modules are not allowed in a runtime image. |
| <code>:include-ide</code> | <code>t</code> if the IDE is in the currently running image, <code>nil</code> if it is not. | Windows only. When true, include the graphical development environment in the resulting image. |
| <code>:include-tpl</code> | <code>t</code> if the top level is in the currently running image, <code>nil</code> if it is not. | When true, include the normal top-level in the resulting image. When <code>nil</code> a minimal top level, as described in Section 9.0 Minimal top levels below, is available. Also specify <code>nil</code> when supplying your own top level. |
| <code>:include-xcw</code> | <code>t</code> if Common Windows is in the currently running image, <code>nil</code> if it is not. | UNIX only. When true, include Common Windows in the resulting image. |

| | | |
|------------------------|-------------------------|---|
| :init-file-names | *init-file-names* | <p>The value should be a list of strings naming files without directory information, like</p> <pre>("clinit.cl" "clinit.cl")</pre> <p>The directories to be searched for these files is determined at runtime (the current directory and the home directory – see startup.htm).</p> <p>The default comes from the value of the indicated variable in the running image. See also :read-init-files below.</p> |
| :load-local-names-info | *load-local-names-info* | <p>The value of this argument serves as the default value for</p> <pre>*load-local-names-info*</pre> <p>in the image to be built. By default, the value is inherited from the value in the running image. The value can be <code>t</code> or <code>nil</code>.</p> |
| :load-source-file-info | *load-source-file-info* | <p>The value of this argument serves as the default value for</p> <pre>*load-source-file-info*</pre> <p>in the image to be built. By default, the value is inherited from the value in the running image. The value can be <code>t</code> or <code>nil</code>.</p> |
| :load-xref-info | *load-xref-info* | <p>The value of this argument serves as the default value for</p> <pre>*load-xref-info*</pre> <p>in the image to be built. By default, the value is inherited from the value in the running image. The value can be <code>t</code> or <code>nil</code>.</p> <p>When building a runtime (i.e. <code>:runtime</code> is true), it is best to explicitly specify this argument <code>nil</code>.</p> |
| :pll-file | (pll-file) | <p>The value can be a string naming an existing pll file or <code>nil</code>. The default is the pll file used by the running image (as returned by pll-file). If <code>nil</code>, no pll file will be used by the new image. Instead, data that would be placed in the pll file (code vectors and string constants) are placed in the Lisp heap.</p> <p>If the value has no directory information (i.e. is just a filename or a filename and a file type), the pll file will be looked for on startup as described in the pll-file entry. If directory information is included, that location (relative to the current directory if relative), will be looked in but nowhere else. The file</p> |

| | | |
|---------------------------------------|--|---|
| | | <p>type defaults to <i>.pll</i> if unspecified. Note that some pll files have type <i>.epll</i>.</p> <p>Note that <code>build-lisp-image</code> does not create a pll file so it is an error to specify a non-existent file as the value of this argument.</p> <p>PlI files are created by the program cvdclt. See the definition of that program and also the discussion of pll files in miscellaneous.htm. See also generate-application and delivery.htm.</p> |
| <code>:print-startup-message</code> | <code>*print-startup-message*</code> | The value of <code>*print-startup-message*</code> is set to the value of this keyword argument in the new image being created. |
| <code>:read-init-files</code> | <code>*read-init-files*</code> | <p>Specifies the value of <code>*read-init-files*</code> in the new image. The value can be:</p> <p><code>t</code>, meaning look in both the current directory and the home directory for init files and load <code>sys:siteinit.cl</code>;</p> <p><code>nil</code>, meaning read no initialization files;</p> <p><code>:nohome</code>, meaning look only in the current directory and load <code>sys:siteinit.cl</code>.</p> <p>The default is the value of <code>*read-init-files*</code> in the running image.</p> <p>See the document startup.htm for information on initialization files and the meaning of the current and the home directories. See also <code>:init-file-names</code> above.</p> |
| <code>:record-source-file-info</code> | <code>*record-source-file-info*</code> | <p>The value of this argument serves as the default value for <code>*record-source-file-info*</code> in the image to be built. By default, the value is inherited from the value in the running image.</p> <p>When building a runtime (i.e. <code>:runtime</code> is true), it is best to explicitly specify this argument <code>nil</code>.</p> |
| <code>:record-xref-info</code> | <code>*record-xref-info*</code> | The value of this argument serves as the default value for <code>*record-xref-info*</code> in the image to be built. By default, the value is inherited from the value in the running image. |

| | | |
|------------------------|-------------------------|---|
| :restart-app-function | *restart-app-function* | <p>Causes <code>*restart-app-function*</code> to be set to this value. The value must be a symbol (it cannot be a function object). See startup.htm or delivery.htm.</p> <p>Warning 1: if you call <code>build-lisp-image</code> or <code>generate-application</code> while running the Integrated Development Environment (on Windows), be sure to specify a value for this argument (<code>nil</code> explicitly if you want no restart app function) unless you want the IDE to start when the image does, since the IDE has its own value for <code>*restart-app-function*</code> (which starts the IDE) which will otherwise be inherited.</p> <p>Warning 2: A function object is a legal value for <code>*restart-app-function*</code> but not a legal value for this argument. If the value of the variable is a function object and no value is specified for this argument, the build will fail.</p> |
| :restart-init-function | *restart-init-function* | <p>Causes <code>*restart-init-function*</code> to be set to this value. The value must be a symbol (it cannot be a function object). See startup.htm or delivery.htm.</p> <p>Note that the value of this variable on Windows in an image that starts the IDE, is <code>cg:start-ide</code>.</p> <p>Warning: A function object is a legal value for <code>*restart-init-function*</code> but not a legal value for this argument. If the value of the variable is a function object and no value is specified for this argument, the build will fail.</p> |

5.0 Arguments to build-lisp-image 2: defaults not inherited from the running image

Most of the remaining keyword arguments are listed in the next table. The keyword arguments to `dumplisp` are also acceptable to `build-lisp-image`. None of these arguments inherit values from the currently running image.

--+

| Argument | Default value | Description/compatibility notes |
|----------|---------------|---------------------------------|
|----------|---------------|---------------------------------|

| | | |
|-------------------|-----|--|
| :autoload-warning | nil | <p>If true, then a report of autoloadable functions will be made to <i>autoloads.out</i>. If a string, it is interpreted as a filename which will be used in place of <i>autoload.out</i> for the report.</p> <p>An autoloadable function is one for which the function definition is in fact not present in the image. Instead, instructions to load a particular file, usually a fasl file out of the bundle in the Allegro directory are present. Once the file is loaded, the true function definition is used to evaluate the call to the function that triggered the autoload.</p> <p>The autoloading feature is designed to keep unneeded functionality from unnecessarily increasing the size of the running image, but to make use of functionality transparent to users. If you are building an image for use locally (say, making an image with patches) this report is likely of no interest since autoloading is typically transparent, as we said. If however, you are creating an application, you may want to know what functionality might be autoloaded since you are not permitted to distribute the <i>files.bundle</i> file from where most autoloading is done. For this reason, generate-application changes the default for this argument to <code>t</code>.</p> <p>The format of <i>autoload.out</i> is two columns, a fasl file name on the left and the function that triggers its loading on the right. Some files are loaded by many functions and so appear many times on the left.</p> |
| :build-debug | t | <p>Value can be <code>t</code> (meaning provide extra debug information, but not interactively), <code>:interactive</code> (meaning allow interactive debugging of the build), and <code>nil</code> (meaning do not provide extra debug information -- this option is intended for batch builds). See Section 6.0 Debugging an image build problem or failure.</p> |
| :build-input | nil | <p>Allows specifying a file which collects the input to the spawned process. The value, if supplied, must be a string naming a file or <code>t</code> (meaning use the filename <i>buildin.out</i> or <code>nil</code> (meaning do not write a file). This argument was previously called <code>:internal-debug</code>.</p> |

| | | |
|-------------------|--|---|
| :build-output | nil | Allows specifying a dribble file which contains a transcript of the entire process of building the image. The value, if supplied, must be a string naming a file. This argument was previously called :dribble-file. |
| :build-executable | Defaults to running image executable. Value, if specified, should be a string. | <p>this argument specifies the name of the Lisp executable that will be invoked when the process that builds the new image file is spawned. Typically, you specify a value for this argument only when</p> <p>(1) You want to use a custom executable built as described in main.htm</p> <p>(2) You want a character size in the new image that is different from the character size in the running image. See Section 3.1 The character size in the resulting image for more information and examples.</p> <p>Note: if build-lisp-image is being called by generate-application, the value of this argument is used by generate-application as well. The specified executable will be the one copied to the application directory. See delivery.htm.</p> |
| :c-heap-size | nil | Allows specification of the total size of the C heap. See Table note 1: Finding available memory addresses immediately following this table. |
| :c-heap-start | nil | <p>Allows specification of the start of the C heap. This value must be above the value of :lisp-heap-start. See Table note 1: Finding available memory addresses immediately following this table. The value should be a decimal integer or a string of an integer followed by M or K (for megabytes or kilobytes) -- e.g. "2883584K". We recommend using M or K format because it is a smaller value and may avoid unexpected sign wrapping -- 0xb0000000 being interpreted as decimal -1342177280, which has bit pattern 0xb0000000 on some machines. Thus, suppose you want the value #xb0000000. Specify :c-heap-start "2883584K" (2883584 is the result of (/ #xb0000000 1024)).</p> |

| | | |
|--|--|---|
| <code>:close-oldspace</code> | <code>nil</code> | If true, reorganizes oldspace and newspace sizes just before the new image is created. See Table note 2: :close-oldspace argument details immediately following this table for more information. |
| <code>:copy-shared-libraries</code> | | See generate-application and delivery.htm . This argument is only relevant when used with generate-application . |
| <code>:debug-on-error</code> | Unsupported, causes an error if specified | This argument is replaced by :build-debug . See Section 6.0 Debugging an image build problem or failure . |
| <code>:discard-arglists</code> | <code>nil</code> | Value can be <code>nil</code> , <code>:medium</code> , and <code>t</code> . <code>nil</code> means keep all arglist information; <code>:medium</code> means discard actual symbols used and use dummy ones (thus reducing the number of symbols in the new image at the cost of the information contained in argument names); <code>t</code> means discard all arglist information. |
| <code>:discard-compiler</code> | <code>nil</code> | Allows the compiler to be discarded after the input files are loaded. This might be necessary for some applications. <code>:include-compiler</code> must be <code>t</code> if this argument is <code>t</code> . |
| <code>:discard-local-name-info</code> | <code>t</code> | Controls throwing away local name information loaded from <i>.fasl</i> files as a result of the compiler switch <code>comp:save-local-names-switch</code> . If <code>t</code> , local name information is discarded. If <code>nil</code> , it is maintained. |
| <code>:discard-source-file-info</code> | (null [value of <code>:load-source-file-info</code> argument]) | Controls throwing away source file information. A true value both discards source file info in the image being created and causes the initial value of <code>*record-source-file-info*</code> and <code>*load-source-file-info*</code> to be <code>nil</code> . |

| | | |
|--|---|--|
| <code>:discard-xref-info</code> | (null [value of <code>:load-xref-info</code> argument]) | Controls throwing away of cross reference information. A true value both discards cross reference info in the image being created and causes the initial value of <code>*record-xref-info*</code> and <code>*load-xref-info*</code> to be nil. When building a runtime (i.e. <code>:runtime</code> is true), the value of this argument must be nil. (This is counter-intuitive but since both <code>:load-xref-info</code> and <code>:record-xref-info</code> must be nil, there will be no xref info to discard.) |
| <code>:dribble-file</code> | nil | This argument has been renamed <code>:build-output</code> although <code>:dribble-file</code> is accepted for backwards compatibility. It is an error to specify both <code>:dribble-file</code> and <code>:build-output</code> . See the description of <code>:build-output</code> for details. |
| <code>:exit-after-image-build</code> | Unsupported, causes an error if specified | This argument is replaced by <i>build-debug</i> . See Section 6.0 Debugging an image build problem or failure . |
| <code>:generate-fonts</code> | nil | UNIX only. Generate (X) fonts from server specified by <code>:server-name</code> . |
| <code>ignore-command-line-arguments</code> | nil | This is actually an argument to dumplisp which build-lisp-image accepts and passes to dumplisp when dumplisp is called at the end of the build process. 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 Command line arguments in startup.htm for details of command-line arguments. |
| <code>:include-*</code> | [See text at right] | All <code>:include-*</code> arguments except <code>:include-debugger</code> , documented just below, are documented in Section 4.0 Arguments to build-lisp-image 1: defaults inherited from the running image above since their default values are inherited from the running image. |
| <code>:include-debugger</code> | t | When true, include the debugger in the resulting image. |
| <code>:internal-debug</code> | nil | This argument has been renamed <code>:build-input</code> although <code>:internal-debug</code> is accepted for backwards compatibility. It is an error to specify both <code>:internal-debug</code> and <code>:build-input</code> . See the description of <code>:build-input</code> for details. |

| | | |
|-------------------------------|--------------------|---|
| <code>:lisp-heap-size</code> | <code>nil</code> | Allows specification of the total size to which the Lisp heap is expected to grow. The value for an image is printed by <code>(room t)</code> output (as <code>Lisp Heap Limit</code>). If you see gaps in room output, you may wish to build an image with a larger heap. See gc.htm . See Table note 1: Finding available memory addresses immediately following this table. |
| <code>:lisp-heap-start</code> | <code>nil</code> | Allows specification of the start of the Lisp heap. The value must be a decimal integer below the value of <code>:c-heap-start</code> . See Table note 1: Finding available memory addresses immediately following this table. |
| <code>:lisp-files</code> | <code>nil</code> | Allows Lisp files to be loaded before the image is created. The value of this argument is <code>nil</code> or a list of files to load. The files can be a pathname, string or keyword. Keywords (examples would be <code>:trace</code> and <code>:defsystem</code>), are passed to require . These files are loaded after <code>custom.cl</code> is loaded, as described below . This argument is ignored by generate-application . Instead, files to load are specified by the required <i>input-files</i> argument. |
| <code>:newspace</code> | See text at right. | Specifies the size of the newspace in the new image. See the document gc.htm for information on newspace sizes. Default value is 2mb if the <code>:include-devel-env</code> value is true, 256k (or larger) otherwise. The value should be an integer, like 2000000 for 2 megabytes. |
| <code>:oldspace</code> | See text at right. | Specifies the amount of free oldspace in the new image. See the document gc.htm for information on oldspace sizes. Default value is 2mb if the <code>:include-devel-env</code> value is true, 256k otherwise. The value should be an integer, like 2000000 for 2 megabytes. |
| <code>:opt-debug</code> | 2 | The initial value of the debug optimization quality in the new image. Value must be one of 0, 1, 2, 3. See Declarations and optimizations in compiling.htm for information on how these values are used. |

| | | |
|--|---|---|
| <code>:opt-safety</code> | 1 | The initial value of the safety optimization quality in the new image. Value must be one of 0, 1, 2, 3 but 0 is strongly discouraged. See Declarations and optimizations in compiling.htm for information on how these values are used. |
| <code>:opt-space</code> | 1 | The initial value of the space optimization quality in the new image. Value must be one of 0, 1, 2, 3. See Declarations and optimizations in compiling.htm for information on how these values are used. |
| <code>:opt-speed</code> | 1 | The initial value of the speed optimization quality in the new image. Value must be one of 0, 1, 2, 3. See Declarations and optimizations in compiling.htm for information on how these values are used. |
| <code>:post-load-form</code> | <code>nil</code> | A form to be evaluated just after the files given by <code>:lisp-files</code> are loaded. |
| <code>:pre-load-form</code> | <code>nil</code> | A form to be evaluated just before the files given by <code>:lisp-files</code> are loaded. |
| <code>:preserve-documentation-strings</code> | <code>t</code> when build-lisp-image is called, <code>nil</code> when generate-application is called. | <p>This argument sets the value of the <code>*load-documentation*</code> variable in the resulting image. If the value of that variable is true, then documentation strings are loaded into and can be created in the new image being created. If the value of that variable is <code>nil</code>, no documentation strings are loaded and documentation strings appearing in definitions are ignored.</p> <p>Note that the default value of this argument is different depending on whether build-lisp-image is called directly (default <code>t</code>) or is called from generate-application (default <code>nil</code>).</p> |
| <code>:restart-app-function</code> | | This argument is documented in Section 4.0 Arguments to build-lisp-image 1: defaults inherited from the running image above since its default value is inherited from the running image. |
| <code>:restart-init-function</code> | | This argument is documented in Section 4.0 Arguments to build-lisp-image 1: defaults inherited from the running image above since its default value is inherited from the running image. |

| | | |
|----------------------|--------------------|---|
| :runtime | nil | <p>When calling build-lisp-image, the value must be <code>nil</code>, as runtime images cannot be created with calls from build-lisp-image. Runtime images must be created with a call to generate-application. In a call to generate-application, the value can be <code>nil</code>, <code>:standard</code>, or <code>:dynamic</code>. Must be <code>nil</code> unless you have a runtime license and have installed necessary functionality. (Standard runtime is included with the Enterprise edition of Allegro CL and is not available with the Professional edition. Contact your account manager for information on upgrading Allegro CL Professional to Enterprise so you get Allegro Runtime, or for information on adding Allegro CL Dynamic Runtime to an enterprise edition.)</p> <p>See runtime.htm for more information. See particularly the discussion in that document of what values other arguments (such as <code>:include-compiler</code> and <code>:include-devel-env</code>) must be.</p> |
| :server-name | nil | Specifies X server from which to get fonts. Ignored if :generate-fonts is <code>nil</code> . |
| :show-window | :showna | Windows only. The value of the <code>:show-window</code> keyword to run-shell-command , used to start the process to build the image being created. See the page for run-shell-command to see the other allowable values for this argument. |
| :splash-from-file | nil | Windows only. Allows specification of a splash bitmap file. If true, the value must be a string naming a bitmap file. |
| :temporary-directory | See text at right. | Default value is architecture dependent. Temporary files created during the build will be stored in this directory, either the default or the specified location. |
| :us-government | nil | Must be specified <code>t</code> by any user employed by or acting on behalf of the United States Government. The only effect is to change the copyright banner to include language required by US copyright laws relating to US Government use. All other users can specify <code>nil</code> or leave unspecified. Only for the United States Federal Government. State and local governments in the United States and all governmental entities outside the United States can specify <code>nil</code> or leave unspecified. |

| | | |
|-------------------------------------|--|---|
| <code>:user-shared-libraries</code> | <code>nil</code> | Allows loading DLLs (Windows) or shared objects (.so or .sl, UNIX) files into the new application image. The value, if not <code>nil</code> , should be a list of pathnames or strings naming pathnames. These files are copied to the destination directory and they are loaded upon application image startup. An error will be signaled by generate-application if files listed cannot be found in order to be copied to the destination directory. |
| <code>:verbose</code> | <code>nil</code> | Causes informative messages to be printed while the image is created. |
| <code>:wait</code> | Unsupported, causes an error if specified | This argument is replaced by <i>build-debug</i> . See Section 6.0 Debugging an image build problem or failure . |

Table note 1: Finding available memory addresses

To successfully allocate the heap, you will need to move the starting address of the Lisp heap to a location large enough to support a contiguous address range specified by the heap size you chose. If you evaluate the following forms in Allegro CL, a memory map of the current state of virtual memory on your machine will be printed to the file *filename.ext* (except on the IBM RS/6000 where the output file will say only that the information is not available). Note that you may use any filename and extension.

```
(ff:def-foreign-call memory_status_dump ((filename (* :char)))
  :arg-checking nil :strings-convert t)

(memory_status_dump "filename.ext")
```

The output differs for different platforms, but in all (except the RS/6000) cases, a set of address ranges is provided showing what ranges are allocated.

The value of the *filename* may be 0, causing the information to be printed to the terminal (on Windows, if the *filename* argument is 0, then the **Console** is used as output). You may get a warning if you specify 0 as the *filename* argument. It can be ignored.

Permissions on Windows are shown in the last four characters in the output:

- r - read access
- w - write access
- x - execute access
- c - mapped copy on write (and has not been copied yet).

You will need to locate a large chunk of free memory and specify to **build-lisp-image** a starting address that will support your heap size.

Table note 2: `:close-oldspace` argument details

The default value of the *close-oldspace* keyword arguments in `nil`. If its value is specified true, the following is evaluated just before the image being built is created (*newspace* is the value of the `:newspace` keyword argument described [above](#), *oldspace* is the value of the `:oldspace` keyword argument also described [above](#)):

```
(sys:resize-areas
  :old 0
```

```

      :new (- (/ newspace 2) (* 1025 50))
      :global-gc t
      :tenure t)
(sys:resize-areas :old oldspace :sift-old-areas nil)
(setf (sys:gsgc-parameter :open-old-area-fence) -1)

```

See:

- **resize-areas**
- **gsgc-parameter**
- **Parameters that control generations and tenuring in gc.htm**, where the `:open-old-area-fence` gsgc parameter is described and closed old areas are described. The value `-1` means close all old areas except the newest one.

As described in **gc.htm**, closed old areas are never gc'ed, so objects in them and objects they point to are never considered garbage. The idea is that the image being created is an application, and the application machinery is being loaded. That machinery will never become garbage, so it saves time if the garbage collector never looks at it. Data to be used by the application is loaded after the image is created. That data will be tenured to the open old areas, and thus will be garbage collected. This allows faster global gc's to clear out data sets when they are no longer needed.

table Note 3: arguments removed in Allegro CL 7.0

The three keyword arguments relating to Allegro Presto, *presto*, *presto-flush-to-code-file*, and *presto-lib*, are no longer supported since Allegro Presto is no longer supported. The arguments are accepted and ignored. A warning is signaled if *presto* is specified (the other two are silently ignored). See **The Allegro Presto facility has been removed in loading.htm** for further information.

6.0 Debugging an image build problem or failure

Starting in Allegro CL 6.2, the method for debugging a failed or problematic image build has been simplified, and unified for all platforms. (In earlier releases, behavior was different for Windows and UNIX.) As part of the changes, the *debug-on-error*, *exit-after-image-build*, and *wait* keyword arguments to **build-lisp-image** (and therefore to **generate-application**) have been removed. An error is signaled if they are specified. In their place is a new keyword argument *build-debug*. In this section, we discuss that argument and the general issue of debugging builds.

build-lisp-image spawns another Lisp to build the desired image (that is one reason why the new image does not inherit from the calling image). If there is a problem, it is this spawned image that one wants to debug. Debugging the image in which you called **build-lisp-image** or **generate-application** is not useful because the problem is not manifested in that image.

The problem is usually caused by a Lisp error being signaled. If the spawned image exits at that point, it will exit with a non-zero status. As we describe just below, specifying the *build-debug* keyword argument as `:interactive` causes the spawned image not to exit and allows interactive debugging.

It is possible that a warning will be signaled during the build. Warnings will not typically cause the build to fail, but are legitimate causes for concern. If you want to debug a warning, call **build-lisp-image** or **generate-application** specifying `:pre-load-form '(setf *break-on-signals* t)`. This will cause an error when the warning is signaled.

The choices for debugging are:

- **Interactively debugging the spawned process.** For this option, specify `:build-debug :interactive` (see [:build-debug](#)). In the event of an error, the spawned process started to build the image does not exit, allowing the problem to be debugged. When you find the cause of the problem, correct the argument values or the input files and run **build-lisp-image** or **generate-application** again. Do not try to continue the process you are debugging through to a new image or distribution.
- **Get debugging information but do not debug interactively.** For this option, specify `:build-debug t` or leave the value unspecified, as `t` is the default (see [:build-debug](#)). The spawned process will exit with a non-zero status at the end of the build, error or not, but will print a backtrace at the point of error. On Windows, the default behavior, with no +

command-line arguments, is that when Lisp exits with a non-zero status, the console requires manual closing.

- **Get minimal debugging information.** For this option, which is intended for batch (no operator) builds, specify `:build-debug nil` (see [:build-debug](#)). The spawned process is invoked with **-batch** command-line arguments. No backtrace is provided. The console will close automatically regardless of error status.

Here are examples using the various values of *build-debug*. The file *foo.cl* includes the form at the top-level (`setq x y`), but *y* does not have a value.

```
;;*****
;; This call in the calling process spawns the process which
;; will attempt the build. Note that :build-debug is t,
;; so the spawned process will exit (its window will remain, however,
;; until you close it by clicking on the Close button).
cl-user(3): (build-lisp-image "foo.dxl"
                          :lisp-files '("foo.cl") :build-debug t)
Initial generation spread = 1
Allocated 10492920 bytes for old space
Allocated 5242880 bytes for new space
iiiiiiii...
...
#|
(let ((*libfasl* nil) (*global-gc-behavior* nil))
  (tenuring (load "foo.cl")))
|#
; Loading /stuff1/acl/acl70/src/foo.cl
Error (from error): Script /tmp/genappal49d19b aborted due to error:
    Attempt to take the value of the unbound variable
    `y'.
Evaluation stack:
  (excl::internal-invoke-debugger "Error" #<simple-error @ #x71f252aa>
    nil)
  (error "Script ~a aborted due to error: ~a" #1="/tmp/genappal49d19b"
    #<unbound-variable @ #x71f2507a>)
  (excl::process-script #1#)
->(tpl:start-interactive-top-level
  #<terminal-simple-stream [initial terminal io] fd 0/1 @
  #x711751f2>
  #<Function top-level-read-eval-print-loop> nil)
  (excl::start-lisp-execution)
; Auto-exit
; Exiting Lisp

[Perhaps a message about the exit status]
;; You may examine the window of the spawned Lisp.
;; When you have finished examining it, click the Close
;; button to close it, and then you will see back in the
;; calling process:

Error: image creation failed
  [condition type: simple-error]

Restart actions (select using :continue):
  0: Return to Top Level (an "abort" restart).
  1: Abort entirely from this process.
```

```
[1] cl-user(4):
```

```
;;*****
;; This call in the calling process spawns the process which
;; will attempt the build. Note that :build-debug is nil,
;; so the spawned process will exit.
cl-user(5): (build-lisp-image "foo.dxl" :lisp-files '("foo.cl")
                        :build-debug nil)
Initial generation spread = 1
Allocated 10492920 bytes for old space
Allocated 5242880 bytes for new space
;;;;;...
...
#|
(let ((*libfasl* nil) (*global-gc-behavior* nil))
  (tenuring (load "foo.cl")))
|#
; Loading /stuff1/acl/acl70/src/foo.cl
Error (from error): Script /tmp/genappal49d19c aborted due to error:
                    Attempt to take the value of the unbound variable
                    `y'.

; Auto-exit
; Exiting Lisp
Error: image creation failed
      [condition type: simple-error]
```

```
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this process.
[1] cl-user(6):
```

```
;;*****
;; This call in the calling process spawns the process which
;; will attempt the build. Note that :build-debug is :interactive,
;; so the spawned process will not exit.
cl-user(1): (build-lisp-image "foo.dxl" :lisp-files '("foo.cl")
                        :build-debug :interactive)
; Fast loading /stuff1/acl/acl70/src/code/build.fasl
Initial generation spread = 1
Allocated 10492920 bytes for old space
Allocated 5242880 bytes for new space
;;;;;...
...
#|
(let ((*libfasl* nil) (*global-gc-behavior* nil))
  (tenuring (load "foo.cl")))
|#
; Loading /stuff1/acl/acl70/src/foo.cl
Error: Attempt to take the value of the unbound variable `y'.
      [condition type: unbound-variable]
```

```
Restart actions (select using :continue):
 0: Try evaluating y again.
 1: Use :y instead.
 2: Set the symbol-value of y and use its value.
```

Building Images

```
3: Use a value without setting y.
4: retry the load of foo.cl
5: skip loading foo.cl
6: recompile /stuff1/acl/acl70/src/foo.cl
7: Abort entirely from this process.
```

```
[Current process: Initial Lisp Listener]
```

```
[1] cl-user(1): :zo :all t
```

```
Evaluation stack:
```

```
... 4 more newer frames ...
```

```
(excl::cer-general-error-handler-one 5 y)
(sys:...context-saving-runtime-operation)
(sys:...runtime-operation)
(excl::%eval y)
->(setq x y)
[... excl::%eval ]
(eval (setq x y))
(excl::sloload
  #<file-simple-stream
    #p"/stuff1/acl/acl70/src/foo.cl" for input pos 32 @
    #x71f237ba>)
(excl::load-from-stream-or-bundle
  #<file-simple-stream
    #p"/stuff1/acl/acl70/src/foo.cl" for input pos 32 @
    #x71f237ba>
  #p"/stuff1/acl/acl70/src/foo.cl" ...)
(excl::load-2 "foo.cl" t ...)
```

```
... more older frames ...
```

```
[Current process: Initial Lisp Listener]
```

```
[1] cl-user(2):
```

```
;; You are still at a prompt in the spawned process.
```

It is possible that the spawned image will fail without signaling a Lisp error. In that case, it will also exit with a non-zero status, but interactive debugging is not, of course, possible. However, this is rare. If it happens, please save any messages that are printed and try the build again (to ensure that it was not a transient problem that caused the failure). If the retry fails, send a bug report to support@franz.com.

The call that invokes the spawned process and input to it

To get maximal information about the invocation of the spawned process caused by your call to **build-lisp-image** (or **generate-application** add the arguments:

```
:build-input "input.txt" :verbose t :build-output "output.txt"
```

"input.txt" and "output.txt" are filenames -- any filename with any valid path will do, of course. Here is what the build-lisp-image form looks like:

```
(excl:build-lisp-image <image-file>
  :build-input "input.txt" :verbose t
  :build-output "output.txt"
  <your arguments>)
```

This causes the actual command starting the Lisp that builds the new dxi (along with other stuff) to be printed to the listener

where the `build-lisp-image` command was issued.

Generated (by the `:build-input` argument) is the file "input.txt" which contains the Lisp forms passed to the spawned Lisp. It looks roughly like:

```
(EXCL:SET-CASE-MODE :CASE-SENSITIVE-LOWER) (COMMON-LISP:FORCE-OUTPUT)
(COMMON-LISP:PROGN (COMMON-LISP:SETQ EXCL::*STORE-DOCUMENTATION* COMMON-LISP:NIL)
(COMMON-LISP:SETQ EXCL:*RECORD-SOURCE-FILE-INFO* COMMON-LISP:NIL)
(COMMON-LISP:SETQ EXCL:*LOAD-LOCAL-NAMES-INFO* COMMON-LISP:NIL)
[...]
```

Generated (by the `:build-output` argument) is the file "output.txt" which contains essentially a dribble output from the spawned process.

7.0 Use of custom.cl

The file `sys:custom.cl` in the Allegro directory is loaded into the new image at the end of the building process but just before the files specified by `:lisp-files` are loaded. If you are running **generate-application**, `sys:custom.cl` is loaded before the files specified by the (required) `input-files` argument are loaded.

`sys:custom.cl`, as delivered, contains various things, either commented out or marked with `#+ignore`. For example, certain forms setting values to the defaults used in previous versions of Allegro CL are provided, preceded by `#+ignore` so they will not be read unless the `#+ignore` is removed. (See **Features present or missing from *features* in Allegro CL in implementation.htm** for information on `#+ignore`.)

8.0 Building an image to include patches

During the build, all patch files in `[Allegro Directory]/update` relevant to the products included in the image (as coded by the filenames) are loaded into the image during the build.

9.0 Minimal top levels

When building a lisp image, specifying `nil` for `include-tpl` will cause a greatly reduced top-level functionality to be built into the lisp. The sole purpose of this minimal top-level is to reduce the space used by the full top-level. Applications which do not require a top-level or which provide their own will often specify `include-tpl nil`. The minimal top level described here will be available in such images.

The entire text of the minimal top level functionality is given below. Note that this code is loaded only if `include-tpl` is `nil`. The top-level code when `include-tpl` is true is quite different.

```
(defpackage :top-level
  (:nicknames :tpl)
  (:use :common-lisp :excl)
  (:size 20)
  (:import-from :excl excl::read-eval-print-loop)

  ;; These are the function handlers for the top-level commands.
  ;; They are user visible.
  (:export #:*read-eval-print-loop* ; user-defined read-eval-print-loop
           #:*read* ; the top-level reader)
```

```

    #:*eval*           ; the top-level evaler
    #:*print*         ; the top-level printer
  ))

```

```
(provide :tpl-user)
```

```
(in-package :top-level)
```

```
;; simple default tpl handlers:
```

```
(defvar *read-eval-print-loop* 'default-read-eval-print-loop)
(setq *read* 'read)
(setq *eval* 'eval)
(setq *print* 'print)
```

```
(declare (special *break-level*))
(setq *break-level* 0)
```

```
(defun start-interactive-top-level (*terminal-io*
                                   function args
                                   &key initial-bindings
                                   &aux vars vals)
```

```

  (declare (:discard-source-file-info))
  (setf (getf (excl::stream-property-list *terminal-io*) 'initial-listener)
        sys::*current-process*)

```

```
;; Compute the list of special variables and bindings for prog.
```

```
(dolist (b initial-bindings)
  (unless (member (car b) vars :test #'eq)
    (push (car b) vars)
    (push (eval (cdr b)) vals)))
(prog (prog vars vals
        (setq vars nil vals nil)           ;free up space
        (apply function args)))
```

```
(defun top-level-read-eval-print-loop ()
  (declare (:discard-source-file-info))
  (loop
    (setq *evalhook* nil *applyhook* nil)
    (catch ':top-level-reset (read-eval-print-loop :level 0))
  ))
```

```
(defun read-eval-print-loop (&key &allow-other-keys)
  (declare (:discard-source-file-info))
  (let (pop-type cval1 cval2)
    (loop
      (multiple-value-setq (pop-type cval1 cval2)
        (catch 'top-level-break-loop
          (funcall *read-eval-print-loop*)))
      ;; If we get here and pop-type is not null, then a throw
      ;; to 'top-level-break-loop was done (by a different toplevel)
      (case pop-type
        (:pop :debug-pop)
        (when (plusp cval1)
          (excl::funcall-in-package :debug-pop :debugger
```

```

                                nil (1- cval1) (1- cval2))))
    (error "user toplevel can't handle this pop type: ~s" pop-type))))))

```

```

(defun default-read-eval-print-loop ()
  (loop
   ;; print the prompt
   (fresh-line *terminal-io*)
   (princ "// " *terminal-io*)
   (force-output *terminal-io*)
   (let* ((exp (funcall *read*))
          (res (funcall *eval* exp)))
     (funcall *print* res *terminal-io*))))

```

9.1 Using the default minimal top-level

The minimal top-level is set up by default to issue a `//` prompt. It only accepts lisp evaluable expressions as "commands", and does not interpret any other top-level commands.

Example on a sparc:

```

% mlisp -I umsloadxcomp.dxl -qq
Loading /release/duane/acl70/src/libacl70pf23.so.
Mapping umsloadxcomp.dxl...done.
Mapping umclxcomp.pll.
Allegro CL 7.0
Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights Reserved.
// (room)
area  address(bytes)          cons          symbols          other bytes
                                8 bytes each  24 bytes each
                                (free:used)  (free:used)    (free:used)
Top #x81dc000
New #x8180000(376832)          918:3158      254:0            246536:37944
New #x8124000(376832)          -----      -----          -----
Old #x8000c40(1192896)        781:15523     135:5628         518896:397344
Root pages: 34
Lisp heap limit: 67108864

NIL
// (exit)
; Exiting Lisp
%

```

9.2 Requiring the normal top-level in a minimal top-level lisp

If the power of the normal top-level is needed after a (non-runtime) minimal top-level lisp is built, `:toplevel` can be required. However, simply requiring `:toplevel` is not enough to start the regular top-level listener; instead, the listener must be invoked recursively, either by an error or by any command (such as **inspect**) that starts a new listener level. At that time, top-level commands (such as **zoom**, etc) can be invoked.

Note however that when `:toplevel` is required, the read-eval-print-loop is reset, and so a **reset** command will make it

appear as if the Lisp had always had a normal top-level.

Example 1:

```
;; In this example, an error can be debugged after the fact by requiring
;; the normal top-level.
```

```
% mlisp -I umsloadxcomp.dxl -qq
Loading /release/duane/acl70/src/libacl70pf23.so.
Mapping umsloadxcomp.dxl...done.
Mapping umclxcomp.pll.
Allegro CL 7.0
Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights Reserved.
// (require :toplevel)
; Fast loading /acl70/src/code/toplevel.fasl
; Fast loading /acl70/src/code/frame.fasl
; Fast loading /acl70/src/code/r/rframe.fasl
```

```
T
// a
Error: Attempt to take the value of the unbound variable `A'.
[condition type: UNBOUND-VARIABLE]
```

```
Restart actions (select using :continue):
0: Try evaluating A again.
1: Use :A instead.
2: Set the symbol-value of A and use its value.
3: Use a value without setting A.
```

```
[1] USER(1): :zo
; Autoloading for TOP-LEVEL::ZOOM-COMMAND:
; Fast loading /acl70/src/code/tpl-debug.fasl
; Autoloading for package "DEBUGGER":
; Fast loading /acl70/src/code/debug.fasl
```

```
Evaluation stack:
```

```
->(EXCL::INTERNAL-INVOKE-DEBUGGER "Error" #<UNBOUND-VARIABLE @ #x81b9f8a> ...)
(ERROR #<UNBOUND-VARIABLE @ #x81b9f8a>)
(SYS:...CONTEXT-SAVING-RUNTIME-OPERATION)
(EVAL A)
(TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
(TPL:START-INTERACTIVE-TOP-LEVEL #<BIDIRECTIONAL-TERMINAL-STREAM [initial terminal
io] fd 0/1 @ #x80439ba>
                                     #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP @
#x804d22a> ...)
```

(to see any ghost frames, the disassembler must be loaded)

```
[1] USER(2): :res
USER(1):
```

Example 2:

```
;; Note in this example a variable is inspected after setting it,
;; to indicate the state of the lisp before the new top-level is
;; pulled into the lisp. Note also that we wrap a progn which will
```

```
;; return a final nil value, so as not to see a huge printout due to
;; the lack of *print-level*/*print-length* controls.
```

```
% mlisp -I umsloadxcomp.dxl -qq
Loading /acl70/src/libacl70pf23.so.
Mapping umsloadxcomp.dxl...done.
Mapping umclxcomp.pll.
Allegro CL 7.0
Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights Reserved.
// (progn (setq x (excl::get-objects 7)) nil)
```

```
NIL
// (require :toplevel)
; Fast loading /acl70/src/code/toplevel.fasl
; Fast loading /acl70/src/code/frame.fasl
; Fast loading /acl70/src/code/r/rframe.fasl
```

```
T
// (inspect x)
; Autoloading for INSPECT:
; Fast loading /acl70/src/code/inspect.fasl
A simple T vector (5649) @ #x81a506a
 0-> fixnum 5629 [#x000057f4]
 1-> The symbol CL:NIL
 2-> The symbol T
 3-> The symbol EXCL::ER-WNAERR
 4-> The symbol EXCL::ER-GENERAL-ERROR-HANDLER-ZERO
 5-> The symbol EXCL::ER-GENERAL-ERROR-HANDLER-ONE
 6-> The symbol EVAL
 7-> The symbol EXCL::INTERPRETED-FUNCALL
 8-> The symbol EXCL::+_2OP
 9-> The symbol EXCL::GC-AFTER
10-> The symbol EXCL::*WITHOUT-INTERRUPTS*
11-> The symbol EQUAL
12-> The symbol *PACKAGE*
13-> The symbol *LISP-PACKAGE*
14-> The symbol *KEYWORD-PACKAGE*
15-> The symbol EXCL::INTERN*
16-> The symbol EXCL::FASL-FIND-PACKAGE
17-> The symbol EXCL::CONVERT-TO-INTERNAL-FSPEC
18-> The symbol *COMPILER-PACKAGE*
19-> The symbol *SYSTEM-PACKAGE*
20-> The symbol EXCL::CONVERT-TO-EXTERNAL-FSPEC
21-> The symbol SYS::LISP-BREAKPOINT
22-> The symbol EXCL::HANDLE-PENDING-SIGNAL
23-> The symbol EXCL::SET-FUNCTION
24-> The symbol EXCL::.INV-MACRO-FUNCTION
...
5648-> The symbol NIL
[1i] USER(1):
```

9.3 Top-level variables

The following variables are maintained or used by the minimal top-level:

| Variable | Notes |
|---|--|
| <code>*read*</code> | Must be true - the function to be used to read top-level input. Initially set to read . |
| <code>*eval*</code> | Must be true - the function to be used to evaluate top-level input. Initially set to eval . |
| <code>*print*</code> | Must be true - the function to be used to print top-level input. Initially set to print . |
| <code>tpl:*read-eval-print-loop*</code> | Must be true - the function to be used as the read-eval-print-loop. Initially set to the <code>tpl::default-read-eval-print-loop</code> as shown in the code above , where <code>tpl::default-read-eval-print-loop</code> is also defined. This variable only exists in minimal top-level lisps. This variable does not have a separate descriptions page. |

If `*read-eval-print-loop*` is set to a value other than `tpl::default-read-eval-print-loop`, then the three read/eval/print variables are not used. Normally, this variable should not be set unless it is desired to remove all possible user interaction with lisp. If a replacement top-level is supplied, it is recommended that all possible errors be handled explicitly with handlers.

The top-level variables in the table with links are also discussed in **Top-level variables** in [top-level.htm](#).