

Common Graphics Integrated Development Environment

This document contains the following sections:

[1.0 About Common Graphics and IDE documentation](#)

[2.0 About Menus and Dialogs in the IDE](#)

[3.0 Common Graphics and Simple Streams](#)

[4.0 About IDE startup](#)

[4.1 How to create an 8-bit image which starts the IDE](#)

[5.0 About submitting a bug report from a break in the IDE](#)

[6.0 About child, parent, and owner windows](#)

[7.0 About how to get sample code for creating controls](#)

[8.0 About using multiple windowing threads in a Common Graphics application](#)

[8.1 Modal CG utility dialogs are not shared between threads](#)

[8.2 CG re-entrancy](#)

[8.3 Enhanced Break Key functionality](#)

[8.4 Debugging Multiple Threads in the IDE](#)

[8.5 Using the IDE while user code is busy](#)

[9.0 About design considerations for event-driven applications](#)

[9.1 Message-handling routines that run for a long time](#)

[9.2 Message-handling routines that block](#)

[10.0 Widget and window classes](#)

[11.0 About creating a Standalone Common Graphics Application without using the Project System](#)

[12.0 About the position class](#)

[13.0 The I\[cl-math-function\] functions](#)

[Appendix A. A Windows API Program with windows but without CG](#)

This document provides an introduction to Common Graphics (a windowing system used with Allegro CL on Windows and on Linux using the GTK window interface) and the Integrated Development Environment (the set of application-building tools available with Allegro CL on Windows and Linux/GTK). Note that Common Graphics and the IDE are only available on Windows and on Linux.

1.0 About Common Graphics and IDE documentation

Each object (operator, variable or constant, class) has an HTML page in *operators/cg/*, *variables/cg/*, or *classes/cg/*, as appropriate. The index (*index.htm*) includes Common Graphics symbols.

There are a number of essays, including the IDE User Guide, which are also provided. They are listed here.

- *Ide User Guide*. The link goes to the first chapter.
- *cg-application-help.htm*. A discussion of adding help facilities to Common Graphics applications.
- *cg-clipboard.htm*. A discussion of the use of the clipboard in Common Graphics.
- *cg-color-palettes.htm*. A discussion of the use of color palettes in Common Graphics.
- *cg-coordinates.htm*. A discussion of coordinates in Common Graphics.
- *dde.htm*. A discussion of DDE support. This is now separated from Common Graphics but only works on Windows.
- *cg-drawing.htm*. A discussion of drawing graphics in Common Graphics.
- *cg-events.htm*. A discussion of event handling in Common Graphics.
- *cg-icons.htm*. A discussion of icons handling in Common Graphics.
- *cg-mci.htm*. A discussion of MCI support in Common Graphics.
- *cg-pixmap.htm*. A discussion of pixmaps.
- *cg-rich-text.htm*. A discussion of rich text editing in Common Graphics.
- *cg-system.htm*. A discussion of the **system** variable and its value.
- *cg-timers.htm*. A discussion of the Common Graphics timer facility.

Information on projects, forms, and menus is found in the *Ide User Guide*, as follows:

- **Projects** are described in *Chapter 4: Projects*.
- **Forms** are described in *Chapter 6: Designing a user interface using forms*.
- **Menus** are described in *Chapter 7: Menus*.

2.0 About Menus and Dialogs in the IDE

This section describes the menus on the Allegro CL IDE menu bar and the various dialogs that present information about the running Lisp and about your project. Note that the IDE is available on Windows and Linux only.

Dialogs:

- **Add Component**

- **Allegro Tree of Knowledge**
- **Apropos**
- **Check for New Patches**
- **Class Browser**
- **Clipboard**
- **Debug Window**
- **Definitions**
- **Download Patches**
- **Editor Workbook**
- **Find**
- **Find in Files**
- **Form**
- **Inspect**
- **Menu Editor**
- **Navigator**
- **New Form**
- **New Project Directory**
- **Options**
- **Package List**
- **Process**
- **Project Manager**
- **Project Parent Directory**
- **Replace**
- **Runtime Analyzer Control**
- **Runtime Analyzer Results**
- **Select a Completion**
- **Shortcut Keys**
- **Startup Action**
- **Tab Order**
- **Trace**
- **Window List**

Menus:

- **File menu**
- **Edit menu**
- **Search menu**
- **View menu**
- **Windows menu**
- **Tools menu**
- **Run menu**
- **Form menu**
- **Recent menu**

- **Help menu**
-
-

3.0 Common Graphics and Simple Streams

Common Graphics streams are simple-streams. Simple-streams are described in *streams.htm*.

Because Common Graphics streams are simple-streams, Common Graphics uses the buffered I/O used by simple-streams. This can require **force-output** to be called to make still-buffered text output visible. Force-output is called internally by Common Graphics after any **redisplay-window** method runs and before any graphical output is done, to ensure that any text that is drawn onto a Common Graphics window by calling a Common Lisp text output function is always visible when appropriate under normal circumstances. But if an application performs Common Lisp text output to a Common Graphics stream in some other way, then it may need to add explicit calls to **force-output**.

Another consequence of this change is that a closed Common Graphics stream no longer changes class, but **window-p** still returns `nil` for a closed stream for backward compatibility (this name may change later). The print name of a closed Common Graphics stream now includes the string "(closed)".

open-stream is still exported for creating printer streams and bitmap-streams, but the location and direction arguments no longer need be specified for these classes. The arguments are now optional rather than required. And bitmap-streams now have a default page size, which is equal to the size of the screen. So a printer stream could be created simply with the form `(open-stream 'printer)` and a bitmap-stream could be created with `(open-stream 'bitmap-stream)`. But if additional initargs are to be passed, the old location and direction arguments still need to be passed as placeholders for backward compatibility, though the value will be ignored. Examples:

```
(open-stream 'printer nil nil :orientation :vertical)
(open-stream 'bitmap-stream nil nil :page-width 200 :page-height 300)
```

4.0 About IDE startup

There are pre-built images that, when invoked, start the IDE automatically. These images are *allegro.dxl* and *allegro-ansi.dxl* (the first is a modern -- case-sensitive -- image and the second an ANSI -- case-insensitive -- image). Both use 16-bit characters. There are no pre-built 8-bit character images that contain the IDE. (See [Section 4.1 How to create an 8-bit image which starts the IDE](#) for information on

starting the IDE in an 8-bit image and information on building an 8-bit image that includes Common Graphics and the IDE and starts the IDE when invoked.)

There are menu items on the Allegro CL submenu of the Windows Start menu for Modern Images and ANSI Images. Each has an 'Allegro CL (w IDE)' item. Choosing one of those starts Allegro CL and automatically starts the IDE. See *Starting on Windows machines* in *startup.htm* for more information on starting Allegro CL on Windows.

The IDE is started by calling **start-ide** with no arguments. This function is called automatically when running one of the IDE images, or can be called explicitly in a base lisp after requiring the `:ide` module by evaluating `(require :ide)`. When **start-ide** is called, it performs the following steps:

1. If the command line that started the lisp contains the `-batch` flag (see *Command line arguments* in *startup.htm*), then **start-ide** exits immediately, returning nil, and nothing else is done.
2. Multiprocessing is started (if necessary), and the "IDE GUI" process is created. **start-ide** returns the new IDE GUI thread. The rest of the startup procedure is performed in the IDE GUI thread.
3. The `*system*` object is created (or recreated if this is a `dumplisp`'ed image).
4. Common Graphics is initialized, and the main IDE owner window (see **development-main-window**) and the IDE menu-bar window are created.
5. If a user prefs file exists, or if no user prefs file exists, if the file `prefs.cl` exists in the main Allegro directory, it is loaded to restore configuration options that were saved most recently in an earlier IDE session. (See the description of **save-options-to-user-specific-file** for information on user prefs files.) If no prefs file exists, default settings are used.
6. The IDE toolbars and status-bar are added according to the **display-toolbars** and **display-status-bar** configuration options.
7. If an initial project was specified either by using the `-project` command line argument or by double-clicking a `.lpr` project definition file to start the IDE, then **open-project** is called to open that existing project. If no initial project was specified, then a new empty project is created instead.
8. Additional IDE windows are created if the following configuration options are true:
 - **new-project-show-form** exposes the initial form window of the project and creates an inspector window inspecting that form.
 - **new-project-show-project-manager** shows the **Project Manager** dialog.
 - **new-project-show-editor** creates and displays an **Editor Workbook**.
9. The "Listener 1" thread is created to evaluate expressions in the initial lisp listener pane of the Debug Window, and to evaluate user code when commands such as **Tools | Incremental Evaluation** and **File | Load** are invoked when this listener is the selected one. `*default-cg-bindings*` is passed as the `:initial-bindings` argument to **process-run-function** when creating this thread (this should always be done when creating a thread that may create Common Graphics windows or to allow debugging that thread in the IDE). The user may create additional similar listeners later by using the **View | New Listener** command.
10. The IDE GUI thread enters an event-handling loop to handle events for IDE windows, which are created in this thread. An abort restart around the loop ensures that the IDE GUI thread re-enters

this event-handling loop when it is reset or otherwise aborted.

11. The Listener 1 thread tells the IDE GUI thread to create its listener pane. This follows the convention where all IDE windows are created in the IDE GUI thread so that their events are all received and handled in that thread. The value of (`ide-evaluator-listener` `*system*`) (see **ide-evaluator-listener**) is set to this initial listener pane.
12. If the file `startup.cl` exists in the main allegro directory, then the Listener 1 thread loads this file. Users may create this file to evaluate arbitrary forms whenever the IDE is started. (If the file `startup.fasl` exists and is not older than `startup.cl`, then `startup.fasl` is loaded instead.)

If an error is signaled while loading this file (or the `prefs.cl` file above), a modal dialog informs you of this and then the IDE continues to start up without loading the rest of the file. Debugging is not possible at this point because the thread is not yet in an event-loop to which it can return when aborting from the debugger. You can, however, load the file explicitly after the IDE is running in order to debug it.

13. If the value of `*ide-startup-hook*` is non-`nil`, then its value is expected to be a list of function names and/or function objects. Each function in the list is funcalled with no arguments in the Listener 1 thread.
14. The Listener 1 thread sets the variable `*ide-is-running*` to `t`. The thread that called **start-ide** could check this variable to know when the IDE has completely finished starting up.
15. The Listener 1 thread passes **top-level-read-eval-print-loop** to **start-interactive-top-level** to enter a read-eval-print loop. As with other lisp listeners, an abort restart ensures that the read-eval-print loop continues whenever this thread is reset or otherwise aborted.

Note on the initial package

By default, the initial package when the IDE starts up is the `common-graphics-user` package, nicknamed `cg-user`. The IDE starts with this as the initial package rather than the more traditional `common-lisp-user` package (nicknamed `cl-user`) so that IDE users do not need to add package qualifiers to external Common Graphics symbols, since the `cg-user` package uses the `cg` package in addition to the packages used by the `cl-user` package. On startup the Debug window prints **[changing package from "common-lisp-user" to "common-graphics-user"]**, which notes the change of package. This message is normal and no user action is necessary.

One of the entries in the list of `*default-cg-bindings*` (which establishes bindings for listeners started by the IDE) binds `*package*` to value returned by **initial-package**. As we said, that value is initially the `common-graphics-user` package. If you would rather have IDE listeners start up in a different package, you can set the **initial-package** configuration option. You do this with the following steps (we assume you want the initial package to be the package named `:mypackage` -- replace `:mypackage` with a keyword naming the package you actually want):

1. Start Allegro CL and the IDE. (The initial package will be the `common-graphics-user`

package).

2. Evaluate the following form:

```
(setf (initial-package (configuration *system*)) :mypackage)
```

3. Choose the menu command **Tools | Save Options Now**.

The next time you start Allegro CL with the IDE, all IDE listeners will have (find-package :mypackage) as the initial value of *package*.

Note that:

- The information on the desired value for **initial-package** is stored in the *prefs.cl* file. If that file is deleted, the value returned by **initial-package** reverts to `common-graphics-user` and this process must be repeated.
- The value of **initial-package** must be a keyword whose symbol-name is the name of the desired package. It must not be a package object (as returned by **find-package**).

4.1 How to create an 8-bit image which starts the IDE

As delivered, Allegro CL only provides 16-bit character size images that contain the IDE and start the IDE when invoked. These images are *allegro.dxl* (and its associated executable *allegro.exe*), which is modern (case-sensitive), and *allegro-ansi.dxl* (and its associated executable *allegro-ansi.exe*), which is ANSI (case-insensitive). See *Allegro CL Executables* in *startup.htm* for a general discussion of executable and image names.

If you want to run an 8-bit character size image with the IDE, you can start an 8-bit image (*alisp8.exe/alisp8.dxl* or *mlisp8.exe/mlisp8.dxl*, that is **Start | Programs | Allegro CL 8.0 | ANSI Images | Allegro CL (non Int'l, ANSI)** or **Start | Programs | Allegro CL 8.0 | Modern Images | Allegro CL (non Int'l, Modern)**), and then load the IDE and call **start-ide**, by evaluating these forms:

```
(require :ide)
(start-ide)
```

You can also build an 8-bit image that has Common Graphics and the IDE already loaded into it, and that starts up the IDE automatically. Evaluating the following expression, for example, would build an 8-bit Modern IDE:

```
(progn
```

```
(build-lisp-image
 "sys:allegro8.dxl"
 :build-executable (namestring (translate-logical-pathname
                               "sys:mlisp8.exe")))

:include-ide t
:restart-init-function 'ide:start-ide
:case-mode :case-sensitive-lower)
(sys:copy-file "sys:mlisp8.exe" "sys:allegro8.exe"
 :overwrite t)
```

That example builds a Modern (case-sensitive) IDE. To build an ANSI (case-insensitive) IDE instead, then you would need to specify the *:case-mode* argument as **:case-insensitive-upper**. And to match the names of the 16-bit IDE images, you might want to use the name "allegro8-ansi" in place of "allegro8". On Linux/Unix you should exclude the ".exe" suffixes in either case.

After the build completes, the files **allegro8.exe** and **allegro8.dxl** will be in the Allegro directory. You can use the new 8-bit IDE by running **allegro8.exe**. (As usual, you could run this file by double-clicking on **allegro8.exe** in the file explorer, or by entering "allegro8" in the proper directory in a command window, or by setting up a shortcut that invokes **allegro8.exe**).

5.0 About submitting a bug report from a break in the IDE

This page describes how to generate an automatic textual bug report when an error occurs in the IDE environment. If the error appears to be due to an Allegro bug, emailing this report to Franz will often help us to debug the error.

When an error occurs and the Restarts dialog (shown on the **Debug Windows after an error** page) appears with options for proceeding from the error, click the Debug button. This will show the current function stack as a graphical outline control. (The control will appear as a pane in the Debug Window if that window is tall enough, and otherwise in its own top-level window. Again see **Debug Windows after an error**.)

If the keyboard focus is not already in the stack outline control, move the focus there by clicking anywhere in the stack outline or perhaps by using the **View | Manage** menu commands for selecting windows that are near the top.

Then write the bug information to a file. This can be done in several ways:

- **Click on the Write a Bug Report button above the Debug Pane.**

- **Invoke the File | Save command.**
- **Invoke the File | Save As command.**

Whichever you choose, a modal dialog appears asking for a pathname (even if you have chosen **Save** or **Save As** before). Select a pathname to write the bug report to. The entire function stack will be saved textually to that file, with brief platform information at the top of the file, and complete dribble-bug (as generated by **dribble-bug**) and *prefs.cl* information at the bottom. The stack information will include all of the arguments and local variables for each stack frame, regardless of whether you have opened the outline items to show the arguments and variables in the IDE. The bug report that was written will then be shown in the IDE editor for your review.

The stack information will include the normally "hidden" frames only if the "Include Hidden Frames" button on the stack outline's toolbar is currently pressed. Though the additional information is not always necessary, it is best to click on this button before generating the bug report.

6.0 About child, parent, and owner windows

Some definitions

- **Owner:** every window has an owner, which is either another window or the screen. If the owner is another window, then the window will always remain in front of the owner window and will shrink, expand, minimize, maximize, and close whenever the owner window does.
- **Parent:** every window has a parent, which is either another window or the screen. If the parent is a window, then the window will be visible only within the interior of the parent window (in other words, it is "clipped" at the interior border of the parent window), it will move along with the parent window whenever the parent moves, and it will scroll within the interior of the parent window whenever the parent scrolls. When the parent of a window is another window, the parent is also the owner. An existing window may be given a new parent by calling (**setf parent**).
- **Child window:** a child window is a window whose parent is a window (not the screen).
- **Top-level window:** a top-level window is a window whose parent is the screen (and not another window).
- **Owned window:** an owned window is a window whose owner is a window.
- **Non-owned or free window:** a non-owned window, also called a free window, is a window whose owner is the screen.
- **Child windows:** the child windows of a window are the windows whose parent is that window.
- **Child windows of the screen:** the child windows of the screen are the windows whose parent is the screen. Note the slight ambiguity here: child windows of the screen are not themselves *child windows* but *top-level windows*.
- **Owned windows:** the owned windows of a window or the screen are the windows whose owner

is that window or screen.

Since a window can be either a child window or a top-level window, and can also be either an owned window or not, there are four possible combinations of these attributes. But since a child window is always also an owned window (where the parent is also the owner), that leaves three actual types of windows, in terms of their relationships to a parent or owner:

1. Child windows, where the parent is a window. Such windows are created by passing an existing window as the value of the `:owner` keyword argument to **make-window**, and `true` as the value of the `:child-p` argument. (Note that the `:child-p` argument defaults to `t`, so it's not necessary to specify a value when creating a child window.)
2. Owned top-level windows, where the owner is a window but the parent is the screen. These windows appear to be independent, moving freely about on the desktop (screen), but still have an owner window with which they shrink and so on. Such windows are created by passing an existing window as the value of the `:owner` argument to **make-window**, and passing `nil` as the value of the `:child-p` argument. Note that the owner of an owned top-level window must always be a top-level window; if a child window is passed as the owner, its top-level parent (or ancestor) will become the owner, not the child window specified.
3. Non-owned top-level windows, where both the owner and parent are the screen. These windows are truly independent, and have their own icons in the Windows taskbar and alt-tab window (unless their **border** property is `:palette`). Create a non-owned top-level window by passing `(screen *system*)` as the value of the `:owner` argument to **make-window** (in which case the `:child-p` argument will be essentially ignored).

In the IDE, the default value of the `:owner` argument to **make-window** is `(development-main-window *system*)`, which is the invisible owner window of the various IDE dialogs (see **development-main-window** and `*system*`). This default allows a user-created window to access the IDE menubar commands by using the menubar's keyboard shortcuts when the user-created window has the keyboard focus, and also allows the window to intermingle with the various IDE windows, rather than being either behind all of the IDE windows or in front of all of them. In a generated standalone application, on the other hand, the default value of the `:owner` argument is the screen. So to test a top-level window just as it would behave in a standalone application, specify `(screen *system*)` as the owner of the window rather than letting the owner default to the IDE owner window. These two alternatives are used by the **Run Form** and **Run Project** commands (both on the **Run** menu): The **Run Form** command places the running window on the IDE owner window for easy access within the IDE, whereas the **Run Project** command creates the main window of the project on the screen to more closely emulate the standalone application that would be created from the project.

The IDE is multi-threaded, so you may find that a user window created on the IDE owner window leads to "message timeout" errors if it interacts with IDE windows in certain ways, due to the windows having been created in different threads and therefore handling their messages in those different threads. If this should happen, the workaround is to not create those user windows on the IDE owner window.

Some related functions: **parent** returns the parent of a window, while **owner** returns the owner. **windows** returns a list of all of the child or owned windows of a window. **child-p** returns whether a window is a child window. **top-level-window** returns the top-level ancestor window of a window.

Some Notes

- Because the `:owner` argument was inappropriately called `:parent` in earlier releases (inappropriate because the argument was only an owner and not a parent when creating an owned top-level window), the `:parent` argument still works for compatibility, but `:owner` is preferred.
- In earlier releases, passing the `:pop-up` argument to **make-window** as true simply created a top-level window. Now it more specifically creates a top-level window appropriate for use as a modal dialog, by coercing `:child-p` to `nil`, `:state` to `:shrunk`, `:minimize-button` and `:maximize-button` to `nil`, and `:scrollbars` to `nil`.

7.0 About how to get sample code for creating controls

All controls (buttons, single-item-lists, combo-boxes, multiline-editable-text controls, etc.) are represented in the Allegro CL Integrated Development Environment (IDE) as classes. Instances are created with **make-instance**, which takes a class name and initialization arguments.

Using the IDE, you can add a control to a form. The code for creating an instance of the control is generated automatically. That automatically generated code provides examples of code that creates instances of controls (and also windows).

You can add a form to a project with the **File | New Form** command. The first form added will typically be labeled `form1`. If you click **Run | Run Project**, files named `form1.cl` and `form1.bil` are saved (along with `project1.lpr`, which does not concern us here). Here is the contents of `form1.bil` (slightly edited, note: do not try to run this code as it has pathnames likely invalid on your machine):

```
;;;
;;; Define :form1

(in-package :common-graphics-user)

;; Return the window, creating it the first time or when it's closed.
;; Use only this function if you need only one instance.
(defun form1 () (find-or-make-application-window :form1 'make-form1))
```

```

;; The maker-function, which always creates a new window.
;; Call this function if you need more than one copy,
;; or the single copy should have a parent or owner window.
;; (Pass :owner to this function; :parent is for compatibility.)
(defun make-form1
  (&key parent (owner (or parent (screen *system*)))
   (exterior (make-box 256 149 960 519)) (name :form1)
   (title "Form1") (border :frame) (child-p nil) form-p)
  (let ((owner
        (make-window name
                      :owner owner
                      :class 'dialog
                      :exterior exterior
                      :border border
                      :child-p child-p
                      :close-button t
                      :cursor-name :arrow-cursor
                      :font (make-font-ex :swiss "MS Sans Serif / ANSI" 11 nil)
                      :form-state :normal
                      :maximize-button t
                      :minimize-button t
                      :name :form1
                      :form-package-name nil
                      :path #p"C:\\Program Files\\acl70\\form1.bil"
                      :help-string nil
                      :pop-up nil
                      :resizable t
                      :scrollbars nil
                      :state :normal
                      :status-bar nil
                      :system-menu t
                      :title title
                      :title-bar t
                      :dialog-items (make-form1-widgets)
                      :toolbar nil
                      :form-p form-p
                      :help-string nil))))
    owner))

```

Note in the definition of the function **make-form1** is a call to **make-window** suitable for creating a dialog window (that is, an instance of class `dialog`). Note that only some of the possible arguments are included. Now, stop running the form and place a button on it by clicking on the button icon on the

component toolbar and clicking on the blank form1. Run the form again, saving *form1.cl*. Again, look at the *form1.bil* file:

```

;;;
;;; Define :form1

(in-package :common-graphics-user)

;; Return the window, creating it the first time or when it's closed.
;; Use only this function if you need only one instance.
(defun form1 () (find-or-make-application-window :form1 'make-form1))

;; The maker-function, which always creates a new window.
;; Call this function if you need more than one copy,
;; or the single copy should have a parent or owner window.
;; (Pass :owner to this function; :parent is for compatibility.)
(defun make-form1
  (&key (owner (or parent (screen *system*)))
        (exterior (make-box 256 149 960 519)) (name :form1)
        (title "Form1") (border :frame) (child-p nil) form-p)
  (let ((owner
        (make-window name
          :owner owner
          :class 'dialog
          :exterior exterior
          :border border
          :child-p child-p
          :close-button t
          :cursor-name :arrow-cursor
          :font (make-font-ex :swiss "MS Sans Serif / ANSI" 11 nil)
          :form-state :normal
          :maximize-button t
          :minimize-button t
          :name :form1
          :form-package-name nil
          :path #p"C:\\Program Files\\acl70\\form1.bil"
          :help-string nil
          :pop-up nil
          :resizable t
          :scrollbars nil
          :state :normal
          :status-bar nil
          :system-menu t

```

```

      :title title
      :title-bar t
      :dialog-items (make-form1-widgets)
      :toolbar nil
      :form-p form-p
      :help-string nil)))
owner))

```

```

(defun make-form1-widgets ()
  (list (make-instance 'button :font
                      (make-font-ex nil "Tahoma / ANSI" 11 nil) :left
                      114 :name :button4 :top 80)))

```

Note that the call to **make-window** now has another argument provided, `:dialog-items`. Its value is a call to **make-form1-widgets**, which is defined (in the file, appearing above as well) as:

```

(defun make-form1-widgets ()
  (list (make-instance 'button :font
                      (make-font-ex nil "Tahoma / ANSI" 11 nil) :left
                      114 :name :button4 :top 80)))

```

If you further customize the button, additional arguments will be provided. Here is the call after we have changed the **title** to "Here", added an **on-click** event handler, and modified the **width** from their default values:

```

(defun make-form1-widgets ()
  (list (make-instance 'button :font
                      (make-font-ex nil "Tahoma / ANSI" 11 nil) :left
                      114 :name :button4 :on-click
                      'form1-button4-on-click :title "Here" :top 80
                      :width 33)))

```

The *title*, *width*, and *on-click* arguments have all been added.

You can reasonably easily generate similar examples creating forms of various classes and by adding controls to a form, running the form, and looking at the resulting *.bil* file.

8.0 About using multiple windowing threads in a Common

Graphics application

Multiple application threads can now create windows and handle events and can be debugged from the IDE. In earlier releases, only the single CG/IDE thread could create windows and only a single one could be debugged. Now multiple threads can create independent hierarchies of windows, each in its own thread, without needing to coordinate the activities of each thread in order for one to be responsive when the other is busy.

We advise against creating windows in multiple threads within a single window hierarchy, though, because deadlocks may occur when messages are sent from a window in one thread to a window in another thread within the hierarchy.

The generic function **creation-process** applied to a window returns the process that called **make-window** to create the window.

The function **set-foreground-window** makes the thread that created the specified window be the foreground thread, and selects the specified window.

A thread that is to create windows must be set up as follows:

1. When creating the thread by calling **process-run-function**, pass `*default-cg-bindings*` as the value of the `:initial-bindings` keyword argument. If other bindings are needed, a union of those bindings with `*default-cg-bindings*` may be passed, but of course do not modify the `*default-cg-bindings*` list.
2. At the end of the preset-function passed to **process-run-function**, enter an event-handling loop by calling **event-loop**. This allows any messages that are sent to windows that are created in this thread to be handled. Typically a "main window" is passed to **event-loop** so that the event-loop and its process will exit when the user has closed the specified window.

A convenient way to do the above two steps is to use the macro **in-cg-process**.

These steps are not necessary when using the project system to create an application with a single windowing thread (which is typical), because these steps are done automatically for the thread created by the **Run | Run Project** command in the IDE and by the corresponding initial thread of the generated standalone application.

Trivial example

```
;; This example simply starts up a thread to create a window,
;; and exits its event-loop (and therefore the thread) when
;; the user closes the window.
```

```
(mp:process-run-function
 (list :name "My dummy thread"
       :initial-bindings cg:*default-cg-bindings*)
 #'(lambda ()
     (let* ((win (cg:make-window :my-window
                                :owner (cg:screen cg:*system*)
                                :title "A window in its own thread.")))
       (event-loop :window win))))
```

Simple example

```
;; This example lets the user click the window to specify a position.
;; A list is kept of the positions, and the window draws a circle
;; at each one. As soon as the user adds the third circle, the
;; event-loop exit-test causes the event-loop to exit, and so
;; the thread dies and its window is therefore closed (this
;; will happen before you actually see the third circle).
```

```
(defclass my-frame (frame-window)
 ((circle-centers :initform nil :accessor circle-centers)))

(defmethod redisplay-window ((window my-frame) &optional box)
 (declare (ignore box))
 (call-next-method) ;; Clear the window
 (dolist (center (circle-centers window))
  (draw-circle window center 50))

(defmethod mouse-left-down ((window my-frame) buttons cursor-pos)
 (declare (ignore buttons))
 (push cursor-pos (circle-centers window)) ;; Add a new circle
 (invalidate window)) ;; Redraw the window to include the new circle

(mp:process-run-function
 `(:name "Three Circles" :initial-bindings ,*default-cg-bindings*)
 #'(lambda ()
     (let* ((window (make-window :three-circles
                                :class 'my-frame
                                :owner (screen *system*)
                                :title "Click to give me three circles")))
       (event-loop :window window
                   :exit-test
                   #'(lambda (win)
```

```
(>= (length (circle-centers win)) 3))))))
```

When the Run | Run Project command in the IDE is invoked, a new thread is created automatically to run the project, and is set up as described above for debugging in the IDE and for handling events.

DDE can now work in multiple threads. See *dde.htm*.

8.1 Modal CG utility dialogs are not shared between threads

Multiple threads may simultaneously invoke modal dialogs without interference, even if the two dialogs are the "same" CG utility dialog, such as the **ask-user-for-choice-from-list** dialog.

8.2 CG re-entrancy

Various global objects have been modified to avoid re-entrancy problems when multiple threads enter the same CG functions simultaneously. Among the things modified are many box and position constants. The functions **with-boxes**, **with-positions**, and **with-positions-and-boxes** are provided for applications that similarly need to remove box and position constants.

8.3 Enhanced Break Key functionality

When the break key is pressed, the Restarts dialog will be created and presented in a new thread that exists solely for handling the break; this may avoid problems with interrupting another thread that is in a problematic state.

Before the Restarts dialog appears, the **process-quantum** of every thread is set to 0.1 seconds, to make any threads that are used for debugging more responsive if another thread is in a busy loop. The process quanta are set back to their earlier values when the break thread goes away, which happens when you either abort from the break key's Restarts dialog or from the backtrace pane that is created for the break in the Debug Window if you select Debug from the Restarts dialog.

Also before the Restarts dialog appears, any modal dialogs that are currently invoked will be brought to the front. This may help to recover from a possible problem where a modal dialog gets buried and then prevents further work due to its modality.

This break key functionality exists in generated CG applications as well as in the IDE. If it is not appropriate for a delivered application, it could be disabled by the application at startup time with this form which uses **remove-global-keyboard-accelerator** and the constant `vk-pause`:

```
(cg:remove-global-keyboard-accelerator cg:vk-pause)
```

8.4 Debugging Multiple Threads in the IDE

Multiple threads may be debugged in the IDE. Any thread can be debugged in the IDE if it is set up using `*default-cg-bindings*` initial bindings as described [above](#). See also the section [Section 8.5 Using the IDE while user code is busy](#).

Multiple Listeners may be used. A new **View | New Listener** menu command allows for the creation of additional all-purpose lisp listeners. Each listener uses an independent thread for evaluations, and has its own command history and backtrace window (when needed). All of the listeners are grouped into a single frame window, with a tab for each listener. The name of the thread and its listener will be "EvaluatorX", where X is a number to make the name unique.

The "Listener 1" listener window always exists while the IDE is running, along with the "Listener 1" thread. Forms evaluated in this listener or elsewhere in the IDE are evaluated in the Listener 1 thread, which is distinct from the "IDE GUI" thread, which handles the actual user gestures in the IDE such as mouse clicks and keypresses (since the IDE windows are created in the IDE GUI thread). The main implication of this is that if the evaluation of a user form is taking a while, the IDE GUI itself will still respond to interactive gestures since these are handled in a different thread (though it may be very sluggish if the evaluation is in a tight loop). To print output to the Listener 1 listener from any thread, the expression

```
(frame-child (ide-evaluator-listener *system*))
```

will return the debug-pane that can be printed to as a stream [see below].

The additional listeners may be closed with the **File | Close Pane** command (control-F4). The initial IDE Evaluator listener, named Listener 1, cannot be closed.

Each thread being debugged will have its own listener and backtrace pane. If a break occurs in one of the Evaluator threads, the listener that already exists for that thread will be used for the backtrace if the debugger is selected from the Restarts dialog. For other threads, a new listener will be created, assuming that IDE debugging has been enabled for the thread.

Listeners that are created for debugging a break will go away automatically when the break is aborted or entirely popped out of. When a thread is debugged by clicking the Debug button of the Processes dialog, the listener that is created does evaluations in a new separate "proxy" process, similar to focusing on a thread in non-IDE listener; this is unlike listeners created when a break occurs and is debugged, which do evaluations in the broken thread itself.

Shortcut keystrokes can be used for moving amongst the different listeners and break levels with the keyboard. Shortcut keys are shown on the right-button shortcut menus of the listener tabs.

Dialog modality in user threads will not disable IDE interaction. Only modal dialogs invoked by the IDE itself in the IDE GUI thread will prevent further interaction with the IDE while the modal dialog is present. Modal dialogs invoked by user code will run independently.

The trace dialog reports which thread each call was made in. When a function call is selected in the trace dialog's outline control, the thread in which that call occurred is displayed in the titlebar of the trace dialog.

The **View | Debug Window** command will generate its new prompt in the main IDE Evaluator listener unless the focus is in a listener already, in which case the prompt is generated in that listener.

The Debug button on the Processes dialog will arrest the selected thread for debugging, and create a new thread with a listener that is focused on the selected thread. The new thread and its listener tab will be named "Proxy for FOO", where FOO is the thread that is focused on. Aborting out of the new listener will unarrest the focused thread.

8.5 Using the IDE while user code is busy

Multiple threads are used by the IDE to allow IDE windows to be responsive while arbitrary user code is busy executing. This is accomplished by creating all of the IDE windows in the "IDE GUI" thread, but evaluating user code in an IDE Listener thread such as the initial "Listener 1" thread, which is associated with the Debug window. The listener threads are used not only for evaluations in the listener pane itself, but also by IDE commands that involve user code, such as the **Tools | Incremental Evaluation** and **File | Load** commands.

The IDE windows will not respond at all while a listener thread is busy, however, if there are any open user windows (windows that were created in a listener thread) on the IDE owner window (see **development-main-window**). There are two cases (described below) where a user window may commonly end up on the IDE owner window, and so these cases should be avoided (and any existing user windows on the IDE owner should be closed) at times when it is desirable to use IDE windows while user code is busy running.

The first case is the **Run | Run Form** command. **Run Form** creates the running window in an IDE listener thread, with the IDE owner window as the owner. This is done so that particular IDE windows may be used alongside the running window without bringing the entire IDE in front of the running window, and so that keyboard shortcuts for IDE commands may be used while the running window is selected. **Run Project** (also on the **Run** menu), on the other hand, does not use the IDE owner window, since its purpose is to simulate the final standalone application more closely. So if a widget on a dialog of the current project initiates a long procedure, and it is desirable to use the IDE while this procedure is running, you should use **Run Project** rather than **Run Form**. (If you need to interrupt something that is running and the IDE windows are not responsive, you can still do so by right-clicking the Franz icon in the Windows Tray and selecting "Interrupt Lisp".)

The second case involves arbitrary user calls to **make-window** where no `:owner` argument is specified. When this is done in the IDE, the owner of the new window defaults to the IDE owner window. This default is used for the same reasons as **Run Form** above. If it is desirable to use the IDE while such a window is open and while user code is busy running, then the window should be created with the screen as its owner by specifying the value of the `:owner` argument to **make-window** as `(screen *system*)`. See **screen** and `*system*`.

As an alternative, see **process-pending-events** on how to use cooperative multitasking, which allows other threads to run in any situation.

9.0 About design considerations for event-driven applications

As documented in [Section 8.0 About using multiple windowing threads in a Common Graphics application](#) above, any process that is set up to create windows and handle the messages that are sent to them needs to call **event-loop** at the end of its **process-run-function** preset-function. The process will then spend its remaining time inside **event-loop**, running code that is triggered by the messages that are sent to any windows that are created in that process. The messages include user mouse and keyboard events as well as messages sent by code that the application is running and messages from the operating system.

This interactive event-driven model requires an application to be designed somewhat differently than one that simply runs from start to end to complete a pre-defined task. We describe two kinds of potential problems that are good to keep in mind when designing an interactive Common Graphics application: [problems with message-handling routines that run for a long time](#) and [problems with message-handling routines that block](#).

9.1 Message-handling routines that run for a long time

When mouse and keyboard events (and other messages) are sent to various windows of a Common Graphics application, the messages are held in a queue, and generally the application code that handles each message is run completely before the next message in the queue is handled. This allows the code to run in a predictable order, even though the messages themselves are queued asynchronously.

This can be a problem when the code that handles a message runs for a long time, because no other messages for windows in the same process will be handled until that code returns, and so end users will see no response to their gestures in the meantime. If future user actions might depend on the current routine completing, then not much can be done about this except showing an hourglass cursor (see **with-hourglass**) or using other cues to tell the user to wait. In this default case, further messages will be queued and handled later in order, and this is how most windowing applications work.

But sometimes it is desirable for the user to be able to go ahead and perform other independent actions (when there are any). One way to do this is to call **process-run-function** to create a new process that performs the time-consuming operation. Note that if the new process needs to create windows that will handle messages, then it needs to follow the guidelines for creating a Common Graphics process described in [Section 8.0 About using multiple windowing threads in a Common Graphics application](#) above; otherwise any ordinary process may be created and used. Another general option is to hand off a command to an existing process, perhaps with `process-interrupt` or using a custom queue of some sort.

A different approach is to call **process-pending-events** at frequent intervals in the long-running code, which handles subsequent queued messages at that time. This function can cause unknown messages to be handled in a different order than usual, though, and so it should be used with care.

process-pending-events should not be called when an exclusive resource (such as a process lock) is currently being held, if it is possible that the processing of subsequent events may lead to a request for that resource that will block until the resource is no longer held. If it is the same process that requests the resource again, then this would always cause a deadlock, since the process will not unwind to the earlier use of the resource in order to free it (unless the second request knows how to see that that process already has the resource, and then either proceed or return, as appropriate). Even if the second request is from another process, complex interactions could still lead to deadlock unless this is carefully avoided in the application design.

Similarly, a Common Graphics process that an application creates normally should not grab an exclusive resource in its **process-run-function** preset-function and not release it before it calls **event-loop**, as this would hoard the resource for the entire life of the process. Likewise, a project's **on-initialization** function should avoid returning while holding an exclusive resource.

With either of the above approaches (handing a long-running command off to another thread, or calling **process-pending-events** frequently), it is up to the application to prevent the user from doing an action that depends on the result of an earlier action that has not yet completed.

9.2 Message-handling routines that block

A less obvious kind of problem may arise due to the fact that there are certain other exceptions to the general rule where one message is handled completely before the next message is handled. In particular, there are certain functions such as **process-wait** that wait an arbitrary amount of time until some condition is met (this is usually called blocking). When such functions are called in a Common Graphics process, any messages that were already queued or that occur during the waiting period are handled while the call is blocking.

This handling of further messages during blocking is done to avoid failing to respond to arbitrary messages for long periods, including messages sent by other processes, or messages that the operating system may send to all top-level windows, expecting a timely reply. Also, the process may be waiting on a condition that will not be reached until further messages are handled by that same process, and so the process would be hung if further messages were not handled while blocking.

This design means that when a blocking function is called in code that is itself handling a message, later messages are handled while the code handling the current message is still running, and so the messages are not handled totally in the usual order. In particular, if the same type of message occurs again during the waiting period, the code that handles the message may be re-entered a number of times, which the application may not be written to handle.

And normally almost all Common Graphics code in an application is message-handling code (namely everything except the **on-initialization** function of a standalone application, or setup code that a newly-created Common Graphics process calls before it calls **event-loop**). So this warning applies to nearly all Common Graphics code in a typical application, if it calls blocking functions that process intervening messages.

The functions and macros that cause intervening messages to be handled immediately include the following:

- **process-wait**
- **with-process-lock**
- **sleep**
- **process-pending-events**
- **astore:with-transaction** (in AllegroStore applications)

Other functions (either in Allegro itself or in the application) that call the above functions would exhibit this behavior as well, and there may be other primitive functions in Allegro that behave this way but are not noted here.

A particular kind of deadlock can result if one of these functions is called while holding an exclusive resource of some kind, such as a process lock. For example, say an application has a **mouse-in** method that grabs a process lock and then calls either **process-wait** or **process-pending-events**. If the user moves the mouse into a window a second time and that event is handled while the call to the method for the first **mouse-in** is still inside the call to **process-wait** or **process-pending-events**, then the **mouse-in** method will be re-entered and wait for the lock. This will deadlock (with the default arguments to **with-process-lock**) because that event-handling process will wait for the lock forever and therefore never unwind to the first call to the **mouse-in** method, as it would need to do to release the lock.

The function **post-funcall-in-cg-process** has been supplied as a general single-process solution to this kind of problem. If code that calls any of the blocking functions listed above is passed to this function instead of being called directly, then the code will be run later after all window messages in this process have been handled and all code that was queued by earlier calls to this function have run and returned. The important point is that all function calls that are passed to **post-funcall-in-cg-process** for a given process are guaranteed to run sequentially (in the order of posting), eliminating problems that might arise if the calls overlapped. See **post-funcall-in-cg-process** for more information and an example. Further window messages are handled as usual by the process whenever a queued function is not running, and users will still see response to their interactive gestures while posted function calls are queued or blocking.

10.0 Widget and window classes

The **Tree of Knowledge** displays the classes associated with windows and widgets: follow **Common Graphics | Interface Objects | Windows | Window Classes** or **Common Graphics | Interface Objects | Controls | Control Classes** (and look at the subentries). Here we list the classes for windows and widgets:

Windows classes

```
cg-stream
  graphical-stream
  screen-stream
    basic-pane
    bitmap-pane
    frame-window
```

- dialog
 - rich-edit-dialog
 - status-bar
 - toolbar
- form
- frame-with-single-child
 - bitmap-window
 - non-refreshing-window
 - text-edit-window
 - lisp-edit-window
- non-refreshing-pane
 - transparent-pane
 - form-pane
- widget-window
 - lisp-widget-window
 - grid-drawing-pane
 - lisp-widget-top-window
 - drawable-pane
 - grid-top-pane
 - group-box-pane
 - multi-picture-button-pane
 - outline-widget-pane
 - outline-display-pane
 - outline-pane
 - outline-top-pane
 - outline-dropping-pane
 - rich-edit-ruler-pane
- os-widget-window
 - combo-box-pane
 - common-status-bar
 - header-control-pane
 - item-list-pane
 - multi-item-list-pane
 - single-item-list-pane
 - list-view-pane
 - progress-indicator-pane
 - scroll-bar-pane
 - horizontal-scroll-bar-pane
 - vertical-scroll-bar-pane
 - tab-control-pane
 - text-edit-pane
 - lisp-edit-pane
 - rich-edit-pane

- text-widget-pane
 - editable-text-pane
 - lisp-text-pane
 - static-text-pane
- toggling-widget-pane
 - button-pane
 - check-box-pane
 - picture-widget-pane
 - picture-button-pane
 - static-picture-pane
 - radio-button-pane
- trackbar-pane
- up-down-control-pane

Widget classes

- dialog-item
 - lisp-widget
 - drawable
 - grid-widget
 - group-box
 - multi-picture-button
 - rich-edit-multipic
 - outline
 - dropping-outline
 - rich-edit-ruler
- os-widget
 - combo-box
 - rich-edit-combo-box
 - font-face-combo-box
 - font-size-combo-box
 - header-control
 - item-list
 - multi-item-list
 - single-item-list
 - list-view
 - progress-indicator
 - scroll-bar
 - horizontal-scroll-bar
 - vertical-scroll-bar
 - tab-control
 - text-widget
 - editable-text

```
lisp-text
multi-line-editable-text
  multi-line-lisp-text
  rich-edit
static-text
togglng-widget
  button
    cancel-button
    default-button
  check-box
  picture-widget
    picture-button
    static-picture
  lamp
  radio-button
trackbar
up-down-control
```

11.0 About creating a Standalone Common Graphics Application without using the Project System

A Common Graphics application is typically built as a project in the IDE, and then turned into a standalone application with the IDE's **File | Build Project Distribution**. See *Chapter 4: Projects* of the *Ide User Guide* for information on using projects to create applications.

You can also create a standalone Common Graphics application with a direct call to **generate-application**, but you must make sure you do everything that needs to be done correctly. Most importantly, note that it is **not** sufficient to specify as a value for the `:restart-app-function` or `:restart-init-function` argument (to **generate-application**) a function that would work as the **on-initialization** function of a project. Instead, the restart function must do additional setup that would have been handled automatically by the project system. Generally, the restart function should do the following:

1. Call **initialize-cg**.
2. Create any initial windows and other setup that would be done by a project's **on-initialization** function.
3. Call **event-loop** to handle events until the application exits.
4. Call **excl:exit**, passing the return code that was returned from the call to event-loop.

In addition, the `:pre-load-form` argument to **generate-application** should require any needed Common Graphics modules, since these will not be included automatically as the project system would do.

12.0 About the position class

Because of a design flaw which is hard to back out of, there is a class named `position` which is the class of position objects in Common Graphics. This is a design flaw because `position` is a Common Lisp symbol (naming a sequence function). It is actually outside the ANSI spec to overload Common Lisp symbols with additional functionality. However, because the violation is not very serious and because changing it would involve substantial costs, we have decided to leave the class `position` rather than renaming it.

The `position` class is documented here because we arrange our documentation by package and documenting the `position` class with other Common Lisp symbols is inappropriate.

The `position` class is the class of position objects. A position is created with **make-position**, and indicates a location in some coordinate system by specifying its x and y coordinates. Positions are useful for determining such things as a window's location or where to draw something on a graphical stream.

13.0 The I[cl-math-function] functions

An artifact left over from a very early release of Common Graphics is various mathematical functions named *i<cl-math-function>*. In the very earliest release, these were quite fast on Windows machines because they used 16-bit arithmetic. With the release of 5.0, the 16-bit operators were replaced by corresponding regular Common Lisp math functions with wrapped declarations identifying the arguments as fixnums. These were named by symbols in the `aclwin` package and were not documented. These symbols are now in the `cg` package. Because we document all exported symbols in that package, we document the *i<cl-math-function>* operators here. They are defined for backward compatibility only. Please do not use them in new code. Instead, use the Common Lisp function determined by removing the initial **i** from the symbol name.

i*, **i+**, **i-**, **i/**, **i/=**, **i1+**, **i1-**, **i>**, **i<**, **i<=**, **i>=**, **i=**, **iabs**, **iceiling**, **idecf**, **ievenp**, **ifloor**, **iincf**, **ilogand**, **ilogandc1**, **ilogandc2**, **ilogbitp**, **ilogior**, **ilognand**, **ilognor**, **ilognot**, **ilogorc1**, **ilogorc2**, **ilogtest**, **ilogxor**, **imin**, **iminusp**, **imod**, **ioddp**, **iplusp**, **irem**, **iround**, **isquare**, **itruncate**, **izerop**.

Appendix A: A Windows API Program with windows but without CG

It is, of course, possible to write a Windows program that makes direct calls to the Windows API and does not use Common Graphics. Below is an example of a trivial but complete program written using Allegro CL's foreign function interface to the Windows API. Writing directly in the Windows API may be useful if you want to port an existing Windows program from C, or if you want the application to be smaller than it would be if it included Common Graphics.

This program simply creates a window that draws a box inside itself, but it illustrates how to set up the usual basic structure of a Windows program in Allegro CL. This example runs in a base lisp that does not include Common Graphics. (Note that the package is the **cl-user** package, not the **cg-user** package -- which may not exist in the base Lisp. If you run this program in the IDE, be sure to use the **cl-user::** package qualifier when necessary.)

```
(in-package :cl-user)

(eval-when (compile load eval)
  (require :winapi)
  (require :winapi-dev))

;;; An arbitrary Windows class name to use for our windows.
(defconstant *my-window-class-name* "My Window Class")

;;; Call this function to run the application in a development lisp.
(defun run-my-windows-app ()

  ;; Create a new process for this app since it will
  ;; remain tied up in its message-handling loop.
  (mp:process-run-function "My Windows App" 'my-windows-app))

;;; Call this function to run the application as a standalone app.
(defun my-windows-app ()

  ;; Register a window class with Microsoft for all of the
  ;; windows of this application. Their messages will be
  ;; received by my-window-procedure-callback.
  (register-window-class *my-window-class-name*
```

```

                                'my-window-procedure-callback)

;; Make an initial top-level window.
(let* ((window-handle
      (with-native-string (native-class-name *my-window-class-
name*)
        (with-native-string (native-window-name "My Window")
          (with-native-string (dummy-string "x")
            (win:CreateWindowEx
              0          ;; extended style
              native-class-name
              native-window-name

              ;; Specify the style of this window.
              #.(logior
                win:WS_CAPTION      ;; title bar
                win:WS_SYSMENU     ;; system menu and close button
                win:WS_MAXIMIZEBOX ;; maximize button
                win:WS_MINIMIZEBOX)

              100      ;; left
              200     ;; top
              400     ;; width
              300     ;; height
              0       ;; parent window (0 is the screen)
              0       ;; system menu
              (lisp-hinstance)
              dummy-string)))))) ;; value to pass to WM_CREATE

;; Expose the window.
(win:ShowWindow window-handle win:SW_SHOWNOACTIVATE)

;; Give the window the keyboard focus.
(win:SetForegroundWindow window-handle)

;; Process incoming messages in a loop.
(ff:with-stack-fobject (message 'win:msg)
  (loop (unless (win:GetMessage message 0 0 0)

        ;; Exit the application when a WM_QUIT message is
        ;; received, causing GetMessage to return nil.
        (return))

```

```

;; Dispatch each message so that it is passed
;; to our window procedure callback function.
(win:DispatchMessage message))

```

```

;; Return an exit code to the operating system
;; (if this is a standalone app). This is the
;; exit code that we passed to PostQuitMessage.
(ff:fslot-value-typed 'win:msg :foreign message :wparam)))

```

```

(ff:defun-foreign-callable my-window-procedure-callback
  (window-handle message-number wparam lparam)

```

```

;; This is the window procedure that is called for all
;; messages that are sent to "My Window Class" windows.

```

```

;; It is very important to use the :stdcall convention for
;; winapi callback functions. Otherwise lisp will crash.
(declare (:convention :stdcall))

```

```

;; Make this foreign callback function call a generic function
;; that we can specialize for various messages.
(my-window-procedure window-handle message-number wparam lparam))

```

```

(defmethod my-window-procedure (handle message wparam lparam)

```

```

;; This default message-handling method passes the message back
;; to the operating system to let it do its default behavior.
;; It needs to return whatever the DefWindowProc returns.
(win:DefWindowProc handle message wparam lparam))

```

```

(defmethod my-window-procedure (handle (message (eql win:WM_DESTROY))
  wparam lparam)

```

```

(declare (ignore handle wparam lparam))

```

```

;; When our only window is being closed, post a WM_QUIT message.
;; Our call to GetMessage will return nil when it reads this
;; WM_QUIT message, and then we exit our event-handling loop.
(win:PostQuitMessage 0)

```

```

;; Call the default message-handling method to pass
;; the message back to the OS for default behavior.
;; (If we didn't call call-next-method, then we would need
;; to return the proper type of value for this message,

```

```
;; which is usually win:FALSE.)
(call-next-method))
```

```
(defmethod my-window-procedure (handle (message (eql win:WM_PAINT))
                                   wparam lparam)

  (declare (ignore wparam lparam))
  (declare (optimize (speed 3)(safety 1))) ;; for with-stack-fobject

  ;; The WM_PAINT message is sent by the operating system whenever
  ;; we need to redraw all or part of our window.

  ;; Do nothing if the window currently has no update region.
  (unless (eq (win:GetUpdateRect handle 0 0) 0)

    ;; Do the standard preparation for painting.
    (ff:with-stack-fobject (ps 'win:paintstruct)
      (win:BeginPaint handle ps)
      (unwind-protect

        ;; Find the area of the window that needs to be redrawn.
        (let* ((left (ff:fslot-value-typed
                      'win:paintstruct :foreign ps :rcPaint :left))
              (top (ff:fslot-value-typed
                    'win:paintstruct :foreign ps :rcPaint :top))
              (right (ff:fslot-value-typed
                     'win:paintstruct :foreign ps :rcPaint :
right))
              (bottom (ff:fslot-value-typed
                      'win:paintstruct :foreign ps :rcPaint :
bottom))))

          ;; We're not actually using the refresh area in this
          ;; simple example, so just ignore it. A real app can
          ;; be made more efficient by not drawing anything that
          ;; doesn't intersect this refresh rectangle.
          (declare (ignore left top right bottom))

          ;; Draw and fill a simple rectangle in our window.
          ;; This illustrates the convolutions required in Windows
          ;; to simply draw colored lines and areas.
          (let* ((hdc (win:GetDC handle))
                 (brush (win:CreateSolidBrush (win-color 0 255 0)))
                 (pen (win:CreatePen win:PS_SOLID 1
```

```

                                (win-color 0 0 128)))
      old-brush old-pen)
(unwind-protect
  (progn
    (setq old-brush (win:SelectObject hdc brush))
    (setq old-pen (win:SelectObject hdc pen))
    (win:Rectangle hdc 20 20 100 100))
  (win:SelectObject hdc old-brush)
  (win:SelectObject hdc old-pen)
  (win>DeleteObject brush)
  (win>DeleteObject pen)
  (win:ReleaseDC handle hdc))))

;; Do the standard painting cleanup.
(win:EndPaint handle ps)

;; Always return zero to the OS for WM_PAINT messages.
0)))

;;; -----
;;; Utility Functions --- These could be used as is.

(defun register-window-class
  (class-name window-procedure
   &key
   (style-flags #.(logior win:CS_OWNDC win:CS_DBLCLKS))
   (large-icon-handle (franz-icon-handle))
   (small-icon-handle 0))
  (declare (optimize (speed 3)(safety 1))) ;; for with-stack-fobject

  ;; Registers a window class in the Windows OS.
  ;; Messages sent to windows of this class will call
  ;; the specified window-procedure callback function.
  (let* ((hinstance (lisp-hinstance)))
    (ff:with-stack-fobject (class 'win:wndclassex)

      ;; If we have already registered this Windows class name
      ;; in this lisp session, then don't don't do so again.
      (when (with-native-string (native class-name)
        (win:GetClassInfo hinstance native class))
        (return-from register-window-class))

      ;; Fill the WNDCLASSEX foreign structure with the

```

```

;; attributes for our window class.
(setf (ff:fslot-value-typed 'win:wndclassex nil class :cbSize)
      (ff:sizeof-fobject 'win:wndclassex))
(setf (ff:fslot-value-typed 'win:wndclassex nil class :style)
      style-flags)
(setf (ff:fslot-value-typed 'win:wndclassex nil class :
lpfnWndProc)
      (ff:register-foreign-callable window-procedure
                                     :reuse :return-value))
(setf (ff:fslot-value-typed 'win:wndclassex nil class :
cbClsExtra)
      0)
(setf (ff:fslot-value-typed 'win:wndclassex nil class :
cbWndExtra)
      win:DLGWINDOWEXTRA)
(setf (ff:fslot-value-typed 'win:wndclassex nil class :
hInstance)
      hinstance)
(setf (ff:fslot-value-typed 'win:wndclassex nil class :hIcon)
      large-icon-handle)
(setf (ff:fslot-value-typed 'win:wndclassex nil class :hCursor)
      0)
(setf (ff:fslot-value-typed 'win:wndclassex nil class
:hbrBackground)
      (1+ win:COLOR_WINDOW))
(setf (ff:fslot-value-typed 'win:wndclassex nil class
:lpzMenuName) 0)
(setf (ff:fslot-value-typed 'win:wndclassex nil class
:lpzClassName)
      (string-to-native class-name))
(setf (ff:fslot-value-typed 'win:wndclassex nil class :hIconSm)
      small-icon-handle)

;; Register the class, signaling an error on failure.
(when (zerop (win:RegisterClassEx class))
      (error "Failed to register the window class ~s."
             class-name))))

(defun franz-icon-handle ()
  (with-native-string (native "aclicon")
    (win:LoadIcon (lisp-hinstance) native)))

(defun lisp-hinstance ()

```

```
(declare (optimize (speed 3)(safety 1))) ;; for with-stack-fobject

;; Returns the handle of the lisp executable that's running.
(ff:with-stack-fobject (vector '(:array :long 4))
  (win:GetWinMainArgs vector)
  (ff:fslot-value-typed '(:array :long 4) nil vector 0)))

(defun win-color (red green blue)
  (+ (ash blue 16)
     (ash green 8)
     red))
```