oodles-of-utils version 1.2

Michael S. Engber

Neither I, Northwestern University, or the Institute for the Learning Sciences make any warranties about this code. It is provided free of charge. If you redistribute oodles-of-utils, please redistribute the entire package.

oodles-of-utils is a collection of Macintosh Common LISP (MCL) code to help access the Mac ToolBox and create user interfaces. The full source is provided so you can use the code or just look at the source for ideas on how to do things.

This document will make using oou a lot easier for both ILS and non-ILS users. Its target audience is both novice and neophyte MCL users. Familiarity with Common LISP and CLOS is assumed. Parts of the document may make not make sense to non-ILS users (or anyone else).

Send questions, suggestions, bug reports, and cash (in small, unmarked, bills) to:

Mike Engber
The Institute for the Learning Sciences
1890 Maple Ave
Evanston, IL 60201

(708)467-1006

from InterNet: `engber@ils.nwu.edu`
from AppleLink: `engber@ils.nwu.edu@INTERNET#`
from CompuServe: `>INTERNET:engber@ils.nwu.edu`

Table of Contents

Acknowledgments & History

oodles-of-utils is not a commercial product. It has evolved over the past 2+ years to satisfy the in house development needs of The Institute for the Learning Sciences. oodles-of-utils has its roots in my "custom" folder of utilities, written for MACL 1.3.2. When MCL 2.0b1 came out in June '91, it made "custom" almost completely obsolete. Thus began a frantic race to write a replacement so we could start using MCL 2.0 productively. After all, without on screen video meaningful software development is impossible.

I would like to thank

My in house users - or bug hallucinators as I usually refer to them:

Mark Chung, Ken Greenlee, Mike Korcuska, Rich Lynch, Tamar Offer, John Welch, Pete Welter, and the rest of the ILS programming staff.

Brian Slator and Ray Bareiss, for letting me have the time to work on oou and the foresight to see that a shared code library is a worthwhile investment.

Martha, for agreeing that it was worthwhile to spend $100[1] on a copy of Coral Common LISP[2] so I could do my AI homework at home on my MacPlus.

And lastly, the MCL team for producing a truly outstanding product and listening to all my whining.

[1] educational price

[2] Yes, that's what `ccl` stands for, Coral Common LISP, the predecessor to MCL which actually ran quite well on a MacPlus with 1 MegaByte of RAM.

Using oodles-of-utils

First you must learn to correctly pronounce oodles-of-utils. "utils" is short for utilities and should be pronounced with the accent on the first syllable - so it rhymes with "oodles."

Then you should put a copy of the oodles-of-utils folder into your MCL folder and load `oou-init.lisp`.

> Do **not** load in `Traps.lisp` or `records.lisp`. This can accidentally happen if you use some old MACL 1.3.2 code that contains `(require :Traps)` or `(require :records)`. These two files support the old MACL mechanism for accessing the ToolBox and loading them interferes with MCL 2.0's mechanism.

You then load in the specific files you need using the `oou-dependencies` macro.

`oou-dependencies &rest module-names`

> Each module name should be specified as a keyword or a string corresponding to an oou file name (file type omitted) For example:
>
> ```
> (oou-dependencies :QuickDraw-u
>                   :Resources-u
>                   :draggable-svm)
> ```

You **must** use `oou-dependencies` instead of `require`. Reasons:

> The oou folder hierarchy is not in the normal search paths `require` uses.
>
> oou's `.fasl` files are all kept in their own folder, which would confuse `require` even if its search path were set up correctly.
>
> Many of the oou files contain macros which must be loaded at compile time or your code won't compile correctly. `oou-dependencies` expands into an appropriate `eval-when` form to ensure they're loaded.

After you load `oou-init.lisp`, the `oou` package is created and `cl-user` package is set up to use the `oou` package, giving you access to all the exported oou symbols (functions, macros, methods, …).

All of the symbols (functions, methods, classes, etc.) documented here should be exported. If you find one that's not, notify me and in the meantime, use a fully qualified symbol name, `oou::some-symbol`. If you dig around in the source you may run into things that are not exported. If you find one you think would be generally useful, notify me and I'll consider cleaning it up and exporting it.

System Requirements

oou was developed using a variety of beta versions of MCL 2.0. It has not yet been tested under MCL 2.0 final, but that shouldn't introduce any problems (yeah, right).

- it will **not** work with any of the MCL 2.0b1pX versions
- parts of it require color QuickDraw (the GDevice stuff comes to mind)
- parts of it is require 32 bit QuickDraw (the GWorld stuff comes to mind)
- I don't think any of it is currently System 7 dependent, but parts probably will be in the future

Of course, the more memory you have allocated to MCL, the better things will work. I don't have any specific guidelines here, but the `.fasl` files take up +300K on disk. If you're really tight on memory you may want to try recompiling oou with some of the memory hogging options turned off.

Compiling oodles-of-utils

All of the oou `.fasl` files are kept in a single folder, `oou-fasl`, which should be located in the `oodles-of-utils` folder. Since these folders are so large, I'm distributing the `oodles-of-utils` folder and the `oou-fasl` folder separately.

For a variety of reasons, you may want to compile oou yourself. For example, you may want to turn off some of the memory expensive compiler options, like `*fasl-save-local-symbols*`. To compile or re-compile oou, use the `compile-oou` function.

`compile-oou`

This function compiles all `.lisp` files that are more recent than their corresponding `.fasl` file (located in the `oou-fasl` folder).

Modifying oodles-of-utils

If, god forbid, you find a bug or something else tempts you to make changes to the oou source, resist. You shouldn't modify its files directly or it will be a nightmare trying to reincorporate your changes into new versions of oodles of utils. Instead, you should place a copy of the file in question into the oou-mods folder and make your changes in the copy. The `oou-dependencies` macro first looks in the `oou-mods` folder, so your changed file will be loaded in preference to the original file.

If you decide to compile your modified file, you will need to do it yourself, `compile-oou` won't. You should place the `.fasl` in the `oou-mods` folder. The `oou-mods` folder should be kept flat, don't put other folders in it.

Reading the Documentation

Documenting object libraries is difficult because understanding a class requires understanding all the classes it inherits from. This is one reason I've moved all my documentation to a single file rather than keeping it spread out in the headers of the source files.

Many source files have commented out example code at the bottom. If after reading a file's documentation you still have some questions, you should try out the example code. A line of working example code is worth a thousand lines of documentation. Remember to first load the file before trying out the example code.

The document is broken up into sections that correspond the hierarchy of folders in oou. For each file, I give a brief discussion of its purpose in life and then document the functions, macros, and classes it provides.

Functions, macros, and methods have their names shown in bold followed by a list of their arguments.

Initargs are shown as keywords followed by braces containing their default initarg values or `[`*no default*`]` indicating no default initarg.

If a description of a method states you should specialize or shadow it, that means you should define your own primary method and not use `call-next-method`. If a method description states you should augment the method, that means you should define your own before, after, around, or primary methods, being sure to use `call-next-method` if you write around or primary methods.

Reading the Source

I would like to encourage people to read the source. Significant effort went into making it legible. I even put in a few comments. It may help if you first read Appendix C which explains some coding conventions used.

I use a few macros and functions from the `ccl` package which are not currently exported or documented. The most common of these is the macro which creates local `macptr` variables of dynamic extent. The other unexported `ccl` stuff can usually be figured out by its name, how it's used, or by expanding the macros.

Many source files have commented out example code at the bottom. If after reading a file's documentation you still have some questions, you should try out the example code. A line of working example code is worth a thousand lines of documentation. Remember to load the file before trying out the example code.

mixin-madness

Mixin classes are not designed to be instantiated themselves, but instead, used in defining other classes (i.e. mixed into their definitions). Well designed mixins are the ultimate in code reusability, allowing you to define complex objects using only a defclass.

In general, I try to keep the mixin classes orthogonal so it doesn't matter which order you use them in an object precedence list. However, some conflicts are inevitable. I've tried to document the conflicts I know about. So far, they are all resolvable by proper ordering of the precedence list.

The mixins in oou use a naming convention which indicate what classes they are to be mixed with. Mixin class names end in the suffix; `-svm`, `-dim`, or `-wm`, indicating they work with the classes; `simple-view`, `dialog-items`, or `window`, respectively. For more information on designing your own mixins, see Appendix D.

Simple View Mixins

Simple view mixins are all named with a `-svm` suffix. They can be mixed with classes that inherit from `simple-view`, including `dialog-item` and `view`.

Notes:

Simple views and views get drawn with different coordinate systems conventions. Simple views are drawn using their container's coordinate system, while views are drawn in their own coordinate system. This distinction is being made in the documentation when I use the term, focusing view coordinates, which means the view container's coordinates for simple views and the view's own coordinate system for views. All the oou simple view mixins are designed to handle these cases properly, so you shouldn't normally have to worry about this detail. On those occasions when you do need to worry about this distinction, the methods `focused-corners` and `focusing-view` are provided to simplify things.

Some simple view mixins require modification to work with the `window` class. This is because `install-view-in-window` and `remove-view-from-window` are not called for windows as they are for other views. So mixins which specialize these methods should probably specialize `initialize-instance` and `window-close` instead. For an example of this, see `back-PICT`, which specializes `PICT-svm` to install background pictures in a window.

This mixin allows a view to be dragged around much like you drag icons around in the Finder.

See Also

`droppable-svm` - dropping the object onto a target
`selectable-svm` - dragging groups of items

Initargs

`[:none]`

Determines the legal drag area (a rectangle). Allowed values are

```
:window    item's                                                window
:container item's                          containing                 view
:none         entire desktop
```

[20]
[20]

These control how far the user can move the cursor outside the drag bounds before the drag becomes void (and the outline disappears). Try dragging the thumb on a standard scroll bar for an example of this behavior.

[:both]

Constrains the drag direction. Allowed values are :vertical, :horizontal, or :both. Try dragging the thumb on a standard scroll bar for an example of this behavior.

[t]

It non-nil, only a dotted outline of the view is dragged around.

```
[t]
   [nil]
[t]
```

These can be used in conjunction with a non-nil :drag-outline-p to achieve different effects.

```
:drag-pre-hilite-p  is    the    original    view    hilited    during    the    drag
:drag-pre-erase-p   is    the    original    view    erased    during    the    drag
:drag-post-erase-p  is the dragged image erased after the drag
```

[#@(2 2)]

This point specifies the delta-h and delta-v the mouse must travel before a click turns into a drag. This prevents accidentally dragging something you only intended to click on. Try dragging an icon in the Finder for an example of this behavior. If you observe carefully, you'll notice you have to move the mouse a few pixels before the drag actually starts.

[*no default*]

The default drag-action method calls the function stored in this slot. The function should accept one argument, the view being dragged. Use this function to perform some action continuously during the drag.

[*no default*]

The default drag-end-action method calls the function stored in this slot. The function should accept 3 arguments, the item dragged, the change in mouse position as a point (delta-h, delta-v), and the final mouse position as a point (in local coordinates for a view, or in its container's coordinates for a simple-view) Use this function to perform an action after a legal drag.

Methods of Interest

(sv draggable-svm) hilite-flag

Specialize this to customize the highlight effect on the original position of the item being dragged. hilite-flag indicates whether to highlight or unhighlight the item (t/nil). The default method uses inversion.

(sv draggable-svm)

Specialize this to control when an item can be dragged. The default method always return t.

(sv draggable-svm) drag-rgn

Specialize this method to customize the shape of the region that gets dragged around. The default method uses RectRgn to set drag-rgn to the rectangular region defined by the view's corners. The drag-rgn argument is a region handle which you should modify to define the desired drag region in global coordinates.

This mixin is a specialization of draggable-svm which allows a view to be dragged and dropped onto targets. It handles highlighting during the drag when the item is over a legal drop target. This functionality is much like dragging files over a folder in the Finder. If the item is released over a legal target, drop-action is executed rather than drag-end-action.

See Also

`draggable-svm` - inherited behavior
`selectable-svm` - dropping groups of items

Initargs

`['dialog-item]`

All legal drop targets must inherit from this class.

`[*no default*]`

A list of view-nick-names to be considered legal drop targets. If not supplied, any object in drop-target-class is legal. Items can't be dropped onto themselves, so don't worry about excluding yourself from your own drop-targets list.

`[*no default*]`

The default drop-action method calls the function stored in this slot. The function should accept 4 arguments, the item dragged, the drop target, the change in mouse position as a point (delta-h,delta-v), and the final mouse position as a point (in local coordinates for a view, or in its container's coordinates for a simple-view). Use this function to perform an action after dropping the item over a legal drop target. (`drag-end-action` is called to handle non-dropping drags)

Methods of Interest

`(sv simple-view) hilite-flag`

Specialize this method to customize the highlighting effect when the item is dragged over a legal drop target. `hilite-flag` indicates whether to highlight or unhighlight the item (`t/nil`). The default method uses inversion.

`(sv droppable-svm) (target simple-view)`

Specialize this method for more sophisticated legal drop target discrimination. The default method finds the view the mouse is in and makes sure it isn't yourself, checks its class against drop-target-class, and finally checks for membership in drop-targets (if drop-targets is bound).

These mixins allow for selecting groups of objects. They group views into clusters and handle selection of items within a cluster. Three classes are defined, differing in how they handle multiple selections:

`selectable-svm`	- shift click to extend selections (à la the Finder)
`selectable-rb-svm`	- no multiple selections (à la radio buttons)
`selectable-cb-svm`	- click to toggle selections (à la check boxes)

See Also

`draggable-svm` - dragging current selection
`droppable-svm` - dragging + dropping current selection onto targets

Note: When combining selectable and draggable behavior, `selectable-svm` must appear before `draggable-svm` in the class precedence list (it specializes some of `draggable-svm`'s methods). Keep this in mind when using `selectable-svm` with any items that inherit from `draggable-svm` like `droppable-svm`.

Initargs

`[nil]`

The selection cluster to which the item belongs. (test with eq) Members of a cluster are required to have the same containing view. Multiple selection constraints are enforced within each cluster.

`[nil]`

Determines if the item is initially selected.

`[t]`

If the item is draggable, this determines if the drag actions of all items in the selections will be called during the drag.

`[t]`

If the item is draggable, this determines if the drag end actions of all items in the selections will be called after the drag.

`[t]`

If the item is droppable, this determines if the drop actions of all items in the selections will be called after the drop.

Methods of Interest

`(sv selectable-svm)`

Returns a list of the items currently selected in dialog item's cluster. Use with setf to change the current selection.

`(sv selectable-svm) hilite-flag`

Specialize this to customize the highlighting effect for selected items. hilite-flag indicates whether to highlight or unhighlight the item (`t/nil`). The default method uses inversion.

This mixin provides a frame for simple views.

Initargs

`[1]`

Pixel width of the frame

Methods of Interest

`(sv frame-svm) rect`

Specialize this method for custom frame types. The default method frames the specified rect.

This mixin is a specialization of `frame-svm` which provides a 3D frame for simple views. This effect only looks right over patterned or colored backgrounds.

See Also

`frame-svm` - inherited behavior

Initargs

`[:botRight]`

Determines the shadow position and consequently if the 3D effect makes the view looked pushed in or popped out. Allowed values are `:botRight` (popped out) and `:topLeft` (pushed in).

[*no default*]

The additional part key, :frame-lite, is accepted and used for the lighter unshadowed half of the frame. The darker shadowed part of the frame is drawn using the :frame part color.

[2]

Pixel width of the frame.

This mixin provides static text imaging for views.

Initargs

["hi,ho"]

Specifies the initial string.

[:center]

Determines the text justification. Allowed values are :left, :right, and :center.

Methods of Interest

(sv static-text-svm)

Returns the current justification, :left, :right, or :center. Use with setf to change the justification.

(sv static-text-svm)

Returns margins for indenting the text as two points (topLeft, botRight). Specialize this method to control text placement.

(sv static-text-svm)

This accessor method returns the string to be displayed. Use with setf to change the text.

This mixin automates resource access for simple views. When the view is installed in a window the resource is read in. When the view is removed from a window, the resource is disposed of (optionally). For graphic resources, there is a specialized class, graphic-rsrc-svm.

Some familiarity with the ResourceManager may be helpful. A common problem with rsrc-svm happens when views share the same handle. One of the views disposes of the handle leaving the other view holding an invalid handle. Two common ways this situation occurs are:

1) Obviously, if you reuse the same handle as the :rsrc-handle initarg, objects are going to share resource handles. In this case, specifying a nil :dispose-rsrc-on-remove-p initarg avoids the problem, but leaves you responsible for disposing of the resource handle when you're through with it.

2) With most types of resources, if you reuse the same :rsrc-id or :rsrc-name initarg, the objects will share the same resource handles. To avoid this problem you can still use the above solution, specifying a nil :dispose-rsrc-on-remove-p initarg, but unless you go to a fair amount of trouble, you'll end up never disposing of the resource handle. An alternate solution is to pass t for the :detach-p initarg, but then you end up with multiple copies of the resource in memory.

More really needs to be said here than I have time for. Resources are a relatively complex subject, but are fundamental to the Macintosh. It is worth your time to do some reading about them. Inside Macintosh Volume I is a good place to start (be sure to ignore

all the errata), then you need to read up on the specific type of resource you're using. The utilities provided in `Resources-u` can simplify things a lot, especially the various `with-xxx` macros.

Initargs

`[no default]`

The resource type (4 character string, e.g. "PICT")

`[no default]`
`[no default]`

The resource can be specified by id or name. The resource will be read in when the view is installed in a window. It is assumed that the appropriate resource file will be open at that time.

`[no default]`

If you already have a handle, it can be specified directly.

`[nil]`

Determines if the resource should be detached after it's read in.

`[t]`

Determines if the resource handle should be disposed of when the view is removed from its window. It also controls whether the old resource handle is disposed of when a new one is installed via `set-view-resource`.

Methods of Interest

```
(sv rsrc-svm) &key
     :rsrc-type :rsrc-id :rsrc-name :rsrc-handle
```

Changes the resource of view. The keywords have the same meaning as in `make-instance`.

`(sv rsrc-svm) rsrc-type rsrc-id-or-name`

The default version uses GetResource. This should be specialized for resource types that have their own get functions. (e.g. `"cicn"` resources use `GetCIcon`)

`(sv rsrc-svm) rsrc-handle rsrc-handlep`

The default version uses `DisposeResource` for resource handles and `DisposeHandle` on vanilla handles. This should be specialized for resource types that have their own dispose functions. (e.g. `"cicn"` resources use `DisposCIcon`)

This mixin is a specialization of `rsrc-svm` which provides a framework for creating graphical objects based on resources.

See Also

`rsrc-svm` - inherited behavior
`PICT-svm` - example of a mixin class based on `graphic-rsrc-svm`
`ICON-di` - example of a dialog item based on `graphic-rsrc-svm`
`cicn-di` - slightly more complex example of a dialog item based on `graphic-rsrc-svm`

Initargs

All the `rsrc-svm` initargs are accepted.

`[:adjust-view-size]`

Determines if the graphic is scaled to the view size or vice-versa. Allowed values are the keywords:

```
:adjust-view-size  - the view size is adjusted to fit the graphic
:scale-to-view     - the graphic is scaled to the view size
:clip-to-view   - the graphic is drawn clipped to the view
```

`[#@(32 32)]`

Used as the default value for a graphic's size.

`[t]`

Determines if the view is erased when the resource is changed.

Methods of Interest

`(sv graphic-rsrc-svm) rsrc-handle`

Returns the size of the specified resource as a point. The default version simply returns the value of the `graphic-default-size` slot. Unless the type of resource you're supporting is always displayed at the same size, you'll need to specialize this function. e.g. PICT-svm's return:

```
(subtract-points (href           rsrc-handle           :Picture.picFrame.botRight)
                 (href rsrc-handle :Picture.picFrame.topLeft))
```

`(sv graphic-rsrc-svm) rsrc-handle rect`

Specialize this method to draw the type of resource you're supporting. It should draw the specified resource scaled to `rect`.

`(sv graphic-rsrc-svm)`

Returns margins for indenting the graphic as two points (topLeft, botRight). The default method returns zero margins. Specialize this method to control placement.

This mixin is a specialization of `graphic-rsrc-svm` which provides a convenient way to display PICTs in views. It supports PICTs stored as resources or in PICT files. In addition, PICTs stored in PICT files can optionally be spooled in from disk as needed rather than read in and kept in RAM. When a PICT is created, it has a bounding rectangle, but can be displayed scaled(yuck) to any size.

See Also

`graphic-rsrc-svm` - inherited behavior
`PICT-di` - this dialog item will probably suffice for most needs

Initargs

All the `graphic-rsrc-svm` initargs are accepted.

All the `rsrc-svm` initargs are accepted.

```
   [no default]
[no default]
```

The PICT can be specified by id or name. The PICT resource will be read in when the view is installed in a window. It is assumed that the appropriate resource file will be open at that time.

`[no default]`

If you already have a PICT handle, it can be specified directly.

`[no default]`

The name of a PICT file to use instead of looking for a PICT resource in the open resource files.

[:memory]

Applicable only when using a PICT file. It determines if the PICT will be kept memory or will be spooled in from disk. Allowed keywords are :memory and :disk.

[:adjust-view-size]

Determines if the PICT is scaled to the view size or vice-versa. Allowed values are the keywords:

```
:adjust-view-size   -   the   view   size   is   adjusted   to   fit   the   PICT
:scale-to-view      -    the    PICT    is    scaled    to    the    view    size
:clip-to-view    - the PICT is drawn clipped to the view
```

Methods of Interest

```
(sv PICT-svm) &key
     :PICT-id :PICT-name :PICT-handle :PICT-file :PICT-storage
```

Changes the PICT. The keywords have the same meaning as in make-instance.

This mixin uses a GWorld to cache an image (or series of images) offscreen for fast updates using spiffy special effects. You'll need to read the Graphic Device Manager chapter of IM VI to appreciate some of the more esoteric options.

There is a series of slides underlying the view. Only one slide is current at a time. When the view needs to draw itself it copies from the offscreen slide to the screen. You can use spiffy effects when you change the current slide, like dissolving from one slide to another.

You'll need to draw into each slide to initialize its contents. You can later change a slide's contents by drawing into it again. Drawing into a slide uses a coordinate system with the slide's upper left corner at #@(0 0). The slide's size is returned by GWorld-slide-size. Don't use the view's coordinate system when drawing into a slide or assume a slide is the same size as the view.

You can get quite a lot of neat effects by just specifying the copy mode and the fore and back colors. For instance, if you want a non-rectangular image, this be achieved using the transparent copy mode. You don't have to resort to using a non-rectangular copy region, which will be slower. The fore and back colors can colorize a slide as it's copied on screen. You can use colorizing to avoid wasting memory with two similar slides.

There is more to work to do here, like supporting color table animation.

Initargs

[*no default*]

With this initarg you can specify a GWorld-view to use rather than have a new one created (the default). This can save memory if you have several views that share the same set of slide. It's potentially dangerous if one view deallocates a GWorld that other views are still using. Use a non-nil :GW-free-on-remove-p initarg to prevent this. There will also be problems if any of the views draw to it. You can create the initial GWorld-view on your own, but it's probably safest to let one GWorld-svm create it and pass it out to subsequent views that want to share it.

[*no default*]

When the GWorld is created, this function is called to initialize it. It is called once for each slide. It should accept 3 arguments, a GWorld-svm, the slide-number to draw, and the GWorld itself. You might very well choose to ignore all three arguments, they're there in case you need them.

[1]

Specifies how many offscreen slides are used.

`[0]`

Specifies which offscreen slide is initially the current slide.

`[8]`

The depth of the offscreen bitmap.

`[#$srcCopy]`

The copy mode used for copying the slide onto the screen. You may want to look into some of the new color QuickDraw modes, like `transparent` or `blend`, to get some ideas on interesting effects that can be achieved using the copy mode.

`[(%null-ptr)]`

Specifies a region to clip the slide to. The region should be in focusing view coordinates.

`[*black-color*]`
`[*white-color*]`

These fore and back colors will be using when copying of the slide on screen. You can change them from the defaults to get interesting colorizing effects. See IM VI or Develop #6 for details on colorizing.

`[:none]`
`[:transporter]`

`:GW-update-fx` specifies what effect to using when doing normal updates. `:GW-slide-fx` specifies the effect to use when switching from one slide to another. The currently supported effects are:

```
:none               no                          effect                          used
:transporter        random                                                      bits
:screen-door        more            organized           than            :transporter
:waynes-world    wavy  lines  àla  Wayne's  World  dream  sequences  -  excellent!
:h-blind            horizontal                  blinds                    turning
:v-blind            vertical                   blinds                     turning
:left-to-right      left                to                right                wipe
:right-to-left      right                to                left                wipe
:top-to-bottom      top                to                bottom                wipe
:bottom-to-top      bottom                to                top                wipe
:round-iris-in      expanding                       round                       iris
:round-iris-out     contracting                      round                      iris
:square-iris-in     expanding                      square                       iris
:square-iris-out contracting square iris
```

`[0]`

The delay in ticks used to slow down the special effects, in case you find them too intense. The wipes are very fast - you'll probably need to use a non-zero delay for them.

The number of stages used in the various wipe effects. Make sure it's less than the width or height of the GWorld.

`[t]`

If nil, the offscreen `GWorld-view` won't be deallocated when the view is removed from its window. This is useful if the `GWorld-view` is being shared.

Methods of Interest

`(sv GWorld-svm)`

Returns the offscreen `GWorld-view` being used.

`(sv GWorld-svm) slide-num &key inval-p draw-now-p`

Changes the current slide. If `:draw-now-p` is `t` (the default), the change will take place immediately using the `GW-slide-fx` special effect. If `:inval-p` is `t` (defaults to `nil`) then the view will be invalidated and will get re drawn through the normal update process using the `GW-update-fx` special effect. And yes, you can use `t` for both `:inval-p` and `:draw-now-p` , but what's the point?

`(sv GWorld-svm) slide-num draw-fn`

Use this method to change the contents of the specified slide after it has already been initialized. `draw-fn` should be of the same form at the `GW-init-fn`. See the description of the `:GW-init-fn` initarg for more details.

```
(sv GWorld-svm) from-slide-num to-slide-num
     &key from-rect to-rect copy-mode copy-rgn fore-color back-color
```

Use this method to copy from one slide to another. This is useful if your slides are based on each other. It's much more efficient to copy from one slide to another than to grab images off disk. The meanings of the keys are as follows:

`:from-rect [`*`full slide`*`]`	specifies the portion of source slide to copy
`:to-rect [`*`full slide`*`]`	specifies the destination rectangle - if it doesn't match the source rectangle, scaling is performed
`:copy-mode [#$srcCopy]`	the copy mode to use (if you're scaling you may want to use ditherCopy)
`:copy-rgn [(%null-ptr)]`	the copy is clipped to this region (specified in slide drawing coordinates)
`:fore-color [*black-color*]`	foreground color to use during copy
`:back-color [*white-color*]`	background color to use during copy

```
(sv GWorld-svm) slide-num
     &key from-rect to-rect copy-mode copy-rgn fore-color back-color
```

Use this method to copy from the screen to a slide (sort of an inverse to displaying a slide). This is useful if you need a slide to contain the current screen image so you can later restore it. The meanings of the keys are the same as for `GWorld-slide-to-slide-copy`.

`(sv GWorld-svm)`

Returns margins for indenting the slide as two points (topLeft, botRight). The default method returns zero margins. Specialize this method to control placement.

`(sv GWorld-svm)`

Returns the size of the slides. Call this method if you need to determine the `botRight` of the slide's rectangle for drawing into the slide (the `topLeft` is always `#@(0 0)`)

`(GWorld-svm)`

This method updates the GWorld to use the current settings. You need to call this method after changing some setting (see below). If you're using a pixel depth of 0 to get optimized copying speed, you need to call this method whenever the view's global position changes (i.e. if the window containing the view moves).

```
(GWorld-svm)
   (GWorld-svm)
(GWorld-svm)
(GWorld-svm)
(GWorld-svm)
   (GWorld-svm)
   (GWorld-svm)
(GWorld-svm)
```

These accessor methods can be used with `setf` to change various settings.

video-digitizer-svm (described in Video section)

video-svm (described in Video section)

Dialog Item Mixins

Dialog item mixins are all named with a `-dim` suffix. They can be mixed with classes that inherit from `dialog-item`.

This mixin allows a dialog item to be used as a button. It handles tracking the mouse while appropriately highlighting/unhighlighting the item.

Methods of Interest

`(di button-dim) hilite-flag`

Specialize this to customize the highlighting effect when the mouse is pressed on a button. hilite-flag indicates whether to highlight or unhighlight the item (t/nil). The default method uses inversion.

This mixin gives a disabled dialog item that familiar grayed out look by bit clearing alternating bits in the item's rectangle. This dimming is implemented by defining an :after method on view-draw-contents. So you'll need to place disable-dim early in the class precedence list to ensure the graying out is done after the dialog-item is already drawn. This is actually a pretty wimpy way to gray out disabled dialog items. It's more efficient to draw them gray pen in the first place. (esp. now that system 7 provides the grayishTextOr txMode) But hey, life is short.

Initargs

`[#$patBic]`

ToolBox pen mode used to PaintRect the item's rectangle.

`[*gray-pattern*]`

ToolBox pattern used to PaintRect the item's rectangle.

This mixin provides a double click action for dialog items. The first click on the item is handled normally (i.e. the dialog-item-action may be run) The second click of the double click will trigger the double click action to be called.

Initargs

`[`*`no default`*`]`

A function to be called when the item is double clicked. The function should accept one argument, the item.

Methods of Interest

`(di double-click-dim)`

This is called when the dialog item is double clicked. The default method calls the function in the slot, `dialog-item-double-click-action-function`.

This mixin provides styled text edit for dialog items. It will become a simple view mixin as soon as I can figure out how to work around the problems with MCL's key-handle-mixin class. Most needs will probably be satisfied by the `te-view` which is much simpler to use. Take a look at `te-view` before attempting to roll your own.

Currently, this class give every object its own TextEdit record (~32K). For most purposes this should be fine. But if you want to have a zillion icons on the screen, each with it's own editable label underneath, this might be a problem. In the future I may add support (a subclass) for a group of objects sharing a common TextEdit record.

See Also

`te-view` - simpler way to get at most of the desired functionality

Initargs

`[:default]`

Controls the text justification. Allowed values are `:default`, `:left`, `:right`, and `:center`.

`[t]`

Determines if the text will be word wrapped.

`[nil]`

This flag allows the text to be read only. Selection and copying are still allowed, but editing is not.

`["hi,ho"]`

A LISP string that will be used to initialize the text.

`[`*`no default`*`]`

The resource id or name of a `'TEXT'` resource to be used to initialize the text. If there is a `'styl'` resource with the same id or name, it will be used to initialize the style of the text. These `'TEXT'/'styl'` resource pairs can be created and edited with ResEdit. Make sure the resource file containing the resource is open.

`[`*`no default`*`]`
`[`*`no default`*`]`

These allow scroll bars to be associated with the text. Operating the scroll bars will scroll the text as you would expect. Plus, the scroll bars will be updated if the user auto scrolls (i.e. drags the selection past the bounds of the text box to get the text to automatically scroll). You need to pass in a `scroll-bar-dialog-item` or its nickname. If you pass a nickname, the scroll bar has to already be installed. Appearing in the `:view-subviews` list before the `te-dim` will ensure this.

`[5]`
`[5]`

This controls how fast text will scroll. Five seems to be a pretty good choice for average sized text (9-12 point). If you use exceptional small or large point sizes you may want to vary it.

Methods of Interest

`(di te-dim)`

Returns two values, the character positions of the first and last character selected.

`(di te-dim) sel-start sel-end`

Sets the current selection.

`(di te-dim) font-spec &key font-color mode`

Sets the font of the current selection. You can use the `:font-color` key to change the color by passing in an MCL style encoded color like `*red-color*` (not an `RGBColor` record). The `:mode` key is use to specify which attributes you want to affect (the default is all). The value you pass in is a sum comprised of the following ToolBox constants: `#$doFont`, `#$doFace`, `#$doSize`, `#$doColor`, `#$doAll`, and `#$addSize`. The `#$addSize` constant allows you to increment/decrement the font size. See IM V p.269-270 for details.

`(di te-dim)`

This accessor returns the text as a LISP string. Expensive for large amounts of text. It can be used with setf to change the text.

`(di te-dim) rsrc-id-or-name`

This method lets you change the text using `'TEXT'/'styl'` resource pairs a per the `:te-init-rsrc` initarg.

`(di te-dim) &key rsrc-id rsrc-name`

This save the current text as a `'TEXT'/'styl'` resource pair in the current resource file. Make sure the file you want it save in is current (you can use `with-res-file`).

`(di te-dim)`

Returns the margins for indenting the text box from the view boundaries. Specialize this method to control placement.

Window Mixins

Window mixins are all named with a `-wm` suffix. They can be mixed with classes that inherit from `window` including `dialog`.

video-wm (described in Video section)

dialog-items

This dialog item is just a `PICT-button-di` with a 3D frame. This effect only looks right over patterned or colored backgrounds.

See Also
`PICT-button-di` - inherited behavior
`frame-3D-svm` - inherited behavior

Initargs

`[2]`
Determines the thickness of the 3D frame.

This dialog item is just a `static-text-di` with a 3D frame. This effect only looks right over patterned or colored backgrounds.

See Also
`static-text-di` - inherited behavior
`3D-frame-svm` - inherited behavior

Initargs

`[2]`
Determines the thickness of the 3D frame

These dialog items are specialization's of `graphic-rsrc-svm` which provide a convenient way to display color icon resources (`cicn`'s). `cicn-button-di` additionally provides button behavior.

Some notes to those unfamiliar with `cicn` resources:

- a `cicn` occupies a fairly large amount of memory, and unlike `GetIcon`, `GetCIcon` allocates a new handle on every call.
- Unlike `GetIcon`, `GetCIcon` makes a copy of the information it needs from the `cicn` resource. This means you can make `cicn` resources purgeable as advised in IM V p. 76.
- Unlike black and white `ICON`s, `cicn`'s are not limited to 32x32 bits in size.
- ResEdit provides a nice `cicn` editor

See Also
`graphic-rsrc-svm` - inherited behavior

Initargs

All the `graphic-rsrc-svm` initargs are accepted.

All the rsrc-svm initargs are accepted.

[*no default*]
[*no default*]

The cicn can be specified by id or name. The cicn resource will be used to create a cicn when the item is installed in a window. It is assumed that the appropriate resource file will be open at that time.

[*no default*]

If you already have a cicn handle, it can be specified directly.

[:adjust-view-size]

Determines if the cicn is scaled to the view size or vice-versa. Allowed values are the keywords:

```
:adjust-view-size   -   the   view   size   is   adjusted   to   fit   the   cicn
:scale-to-view      -    the    cicn    is    scaled    to    the    view    size
:clip-to-view   - the cicn is drawn clipped to the view
```

Methods of Interest

(di cicn-di) &key :cicn-id :cicn-name :cicn-handle

Changes the cicn. The keywords have the same meaning as in make-instance.

These dialog items are specialization's of graphic-rsrc-svm which provide a convenient way to display ICON resources. ICON-button-di additionally provides button behavior. ICONs are defined 32x32 pixels in size, but can be displayed scaled(yuck) to any size.

See Also

graphic-rsrc-svm - inherited behavior

Initargs

All the graphic-rsrc-svm initargs are accepted.

All the rsrc-svm initargs are accepted.

[*no default*]
[*no default*]

The ICON can be specified by id or name. The ICON resource will be read in when the view is installed in a window. It is assumed that the appropriate resource file will be open at that time.

[*no default*]

If you already have a ICON handle, it can be specified directly.

[:adjust-view-size]

Determines if the ICON is scaled to the view size or vice-versa. Allowed values are the keywords:

```
:adjust-view-size   -   the   view   size   is   adjusted   to   fit   the   ICON
:scale-to-view      -    the    ICON    is    scaled    to    the    view    size
:clip-to-view   - the ICON is drawn clipped to the view
```

Methods of Interest

`(di ICON-di) &key :ICON-id :ICON-name :ICON-handle`

Changes the `ICON`. The keywords have the same meaning as in make-instance.

These dialog items are a specialization of `PICT-svm` which provide a convenient way to display PICT resources. `PICT-button-di` additionally provides button behavior. Most of these item's functionality is inherited from the mixin they're based on, `PICT-svm`.

See Also

PICT-svm - inherited behavior

Initargs

All the `PICT-svm` initargs are accepted.

All the `graphic-rsrc-svm` initargs are accepted.

All the `rsrc-svm` initargs are accepted.

This dialog item is similar to MCL's `static-text-dialog-item`, but supports left, center, & right justification. In addition, it word wraps the text. Most of this item's functionality is inherited from the mixin it's based on, `static-text-svm`.

See Also

`static-text-svm` - inherited behavior

Initargs

`["hi,ho"]`

Use either to specify the initial string.

`[:center]`

Determines the text justification. Allowed values are :left, :right, and :center.

`[1]`
`[1]`

Determines the horizontal and vertical indent (in pixels) used in drawing the text.

Methods of Interest

```
   (di static-text-dim)
(di static-text-dim) string
```

These can be used as alternatives to the standard `static-text-svm` accessor method, `text-string`. They use the `text-string` slot rather than the `dialog-item-text-slot`.

`(di static-text-dim)`

Returns the current justification, :left, :right, or :center. Use with setf to change the justification.

```
(di static-text-dim)
(di static-text-dim)
```

These accessor methods return the horizontal & vertical text indents. Use with `setf` to change these values.

NotInROM-u provides a way to define trap calls which MCL does not provide. These are generally traps that are listed as "Not in ROM" in Inside Macintosh. To call a "Not in ROM" trap you should use the following syntax:

```
(#~SomeTrap                                                    args…)
or
(require-trap-NotInROM #~SomeTrap args…)
```

The reader macro will first check if there is a Not in ROM definition in effect for the trap name. If so, that's what will be used, if not, it will expand as if you'd used the regular `#_` MCL syntax. The `require-trap-NotInROM` macro works analogously to MCL's `require-trap` macro.

For example:

`#~Control` expands into the high level `Control` call documented in IM II
`#_Control` expands into the low level trap call documented under `PBControl` in IM II

In the `NotInROM` folder you will find a variety of files which define "Not in ROM traps." The files use a naming convention of starting with a + followed by the name of the interfaces file, in MCL's `interfaces` folder, from which they were omitted.

Unlike regular traps, the Not in ROM traps are not automatically loaded on an as needed basis (maybe some day I'll get around to it). So you will have to ensure the *+interface* files you need are loaded.

If you want to just load all the Not in ROM calls at once, load the file `NotInROM.lisp`. You may want to do this because selectively loading the files requires you to know which trap calls are Not in ROM and which interface file they belong to. In addition, both #~ and `require-trap-NotInROM` behave equivalently to `#_` and `require-trap` for "in ROM" traps, so you can choose to always use them and not worry about which traps are "Not in ROM."

The set of "Not in ROM" calls defined is not complete. It includes the most commonly needed ones; all the high level File Manager calls, all the high level Device Manager calls, all the Serial Manager calls, plus others. I've been adding to it on an as needed basis.

Macros of Interest

To define your own Not In ROM traps use these macros. **Note:** all trap symbols should start with an underscore character, `'_'`.

`symbol result-type (&rest typed-arglist) &body body`

This macro defines a Not In ROM function. See the various `+XXX.lisp` files in the NotInROM folder for examples of it's usage.

`trap-symbol &rest arglist`

This macro works like MCL's `require-trap` except works properly for traps of the form `#~SomeTrap` as well as traps of the form `#_SomeTrap`.

`alt-trap-symbol asm-trap-symbol`

This macro is for defining traps whose high level names differ from their assembly language trap names. You can consider this a special case of the "Not in ROM" traps which only requires name mapping to handle. For example, the function `DisposeHandle` uses assembly language trap `_DisposHandle`. Currently, MCL seems to define all these traps in both their high level and assembly language names. This macro is provided in case this changes or omissions are discovered.

MCL provides a file of it's own "Not In ROM" function definitions, . However, these are defined as normal lisp functions, they do not use the `#_` syntax, and the arguments they accept and the values they return are not always the same as those documented in Inside

Macintosh. NotInROM-u is provided as an alternative which makes the "Not in ROM" calls more consistent with the other traps calls and behave as per Inside Macintosh.

It may be the case, that a version of NotInROM will be provided by Apple in the MCL folder. If this is the case, you still want to use the version in oou because it may be more up to date. If you use `oou-dependencies` to load in "Not in ROM" routines, you'll get the proper files.

brutal-utils

Utilities for working with GDevices.

Functions of Interest

`&optional globalRect`

Returns the GDHandle of the deepest screen device intersecting the specified rectangle (global coordinates). If `globalRect` is not specified, then it returns the GDHandle of the deepest available screen.

`fn &optional active-screens-only-p`

Maps `fn` over the GDevice list. If `active-screens-only-p` is `t` (the default), then `fn` is only applied to active screen GDevices, else it's applied to the entire GDevice list. `fn` should accept one argument, a GDHandle.

`&optional where`

Returns the GDHandle of the GDevice containing the specified point (in global coordinates). If where is omitted, it defaults to the current mouse position.

Right now, this file only contains pop-up menu related utilities.

Functions of Interest

`rect &key :width :right-indent`

This function draws a standard pop-up menu down arrow in the right end of the specified rectangle. You can customize it's size and location using `:width` and `:right-indent`.

```
                                  item-list                                    &key
   :where   :default-item   :checked-items   :other-p   :test   :item-string-function
   :hier-p-fn :hier-parent-fn :hier-items-fn :hier-select-p
```

This function is a pop-up menu equivalent to select-item-from-list. It returns two values. The first value is either the item selected, `nil` (no choice), or `:other` (`Other…` menu item). The second value is the item's position (0 based) in the menu or sub-menu. The keys are described below:

```
:where [current mouse position]       point      where      the      menu      appears
:default-item [0]                     elt#  of  item  in  list  initially  highlighted
:checked-items [nil]                  list     items     to     have     check     marks
:other-p [nil]                        if  an  Other…  item  is  added  to  the  menu
:test [eql]                           used  for  membership  test  in  checked-items
:item-to-string-fn [princ-to-string]  returns a menu item title for an item
```

These keys allow pup-menu-select to traverse item-list as a tree and create an isomorphic hierarchical pop-up menu (ugh!)

```
:hier-p-fn [no default]       returns    whether    an    item    has    a    sub-menu
:hier-items-fn [no default]   returns    a    list    of    an    item's    sub-items
:hier-parent-fn no default]   returns    the    parent    item    of    a    sub-menu
:hier-select-p [t]            can parents of sub-menus be selected?
```

Items that have sub-menus can be selected (unless `:hier-select-p` is `nil`), but they cannot be checked - due to limitations in the Menu Manager.

Also, see the example code at the bottom of Menus-u.lisp.

Utilities for working with resources

Macros of Interest

Resources-u provides a variety of with- macros which execute a body of code with the resource environment temporarily changed.

```
(pathname &key :if-does-not-exist                                    :if-no-rsrc-fork
    :if-not-open :if-open :write-changes-p) &body body
```

Executes body with the current resource file set to pathname. The meanings of the keys are:

:if-does-not-exist [:error]	What to do if the file doesn't exist. Allowed values are :error and :create.
:if-no-rsrc-fork [:error]	What to do if the file exists, but doesn't have a resource fork. Allowed values are :error and :create.
:if-not-open [:close-when-done]	Specifies what to do if the resource file was not previously open. Allowed values are :close-when-done and :leave-open.
:if-open [:leave-open]	What to do if the resource file is already open. Allowed values are :close-when-done and :leave-open.
:write-changes-p [nil]	Whether or not to call _UpdateResFile.

```
&body body
```

Executes body with resource loading turned off.

```
(rsrc-handle &key :changed-p nil) &body body
```

Executes body with rsrc-handle loaded and non-purgeable (it does not lock rsrc-handle). The changed-p key indicates whether to mark rsrc-handle as changed (defaults to nil).

Functions of Interest

```
rsrc-type rsrc-id-or-name &key :errorp
```

Returns a handle to a resource of the specified type. The resource can be specified by id (fixnum) or by name (string). The errorp key determines if an error is signaled if the resource is not able to be loaded. It defaults to t. If errorp is nil and the resource is not loaded, a null macptr is returned.

```
rsrc-handle
```

Releases the specified resource. An error is signaled if rsrc-handle is not a handle to a resource.

```
rsrc-type rsrc-name &key :errorp
```

Returns the id of the resource with the specified type and name. The errorp key determines if an error is signaled if the resource is not found.

```
handle
```

Returns t/nil indicating if handle is a resource handle.

```
handle
```

Returns t/nil indicating if handle is a purgeable resource handle.

`pathname`

Returns `t/nil` indicating if `pathname` is a currently opened resource file.

`pathname &key :if-does-not-exist :if-no-rsrc-fork`

Opens the specified resource file. The keys specify what to do if the file does not exist or it exists but has no resource fork. Their allowed values are `:error` (the default) and `:create`.

`refNum-or-pathname`

Closes the resource file specified by a reference number or a pathname.

Utilities for working with PICT files.

Functions of Interest

`pathname`

This function allocates and returns a PICT handle created from the data fork of the specified PICT file.

`PICT-handle pathname &key :creator  :if-exists`

The functions creates a `PICT` file from the specified PICT handle. The file creator can be specified with the `:creator` key which defaults to `"????"`. The `:if-exists` key specifies what to do if the file already exists. It can be either `:error` (the default) indicating an error is signaled or `:overwrite` indicating the file should be overwritten.

`pathname rect &optional (scale-to-rect-p t)`

This function draws a PICT from the specified PICT file by spooling it from disk. (i.e. without first reading the entire PICT into memory) If `scale-to-rect-p` is non-nil, the PICT is drawn scaled to `rect`, else it is drawn at `rect.topLeft`, but in it's original size (from its `picFrame`). The PICT size (in bytes) is returned.

`pathname picture-record-ptr`

This function returns the size of the PICT (in bytes) in the specified PICT file. `picture-record-ptr` is a pointer to a `:Picture` record that will have it's `picSize` and `picFrame` filled in from the PICT file. Note: QuickDraw now ignores the `picSize` field because it wasn't large enough to support PICTs > 32K. The return value of this function is the correct size.

Utilities for working with QuickDraw.

Macros of Interest

`QuickDraw-u` provides a variety of `with-` macros which execute a body of code with the drawing environment temporarily changed. These macros all assume that the body code doesn't leave the current port changed.

`(&key :pnLoc :pnSize :pnMode :pnPat :pnPixPat) &body body`

Executes body with the specified pen characteristics.

`(&key :txFont :txFace :txMode :txSize) &body body`

Executes body with the specified text characteristics.

`font-spec &body body`

Executes body with the specified pen characteristics.

`portBits &body body`

Executes `body` with `portBits` (which must be a symbol) bound to a the `portBits` field of the current port. If the current port is a GrafPort this will actually be a `BitMap`. If the current port is a `CGrafPort` it will be a pointer to its `pmVersion` field. In either case it can be passed to `CopyBits` and related traps which accept either a `BitMap`., a pointer to a `CGrafPort`'s `pmVersion` field, or a `PixMapPtr`.

`pattern &body body`

Executes body with the specified back pattern. This works with either a `GrafPort` or a `CGrafPort`.

`pix-pat &body body`

Executes body with the specified back pixel pattern. This works with either a `GrafPort` or a `CGrafPort`.

`rgb &body body`

Executes body with the specified hilite color. Normally, the hilite color is set by the user via the Color Control Panal This works with either a `GrafPort` or a `CGrafPort`.

`clip-rgn &body body`

Executes body with the specified clip region.

```
                                                                    (&key
                   :textProc :lineProc :rectProc :rRectProc :ovalProc :arcProc :p
olyProc :gnProc :bitsProc :commentProc :txMeasProc :getPicProc :p
utPicProc :opCodeProc :newProc1 :newProc2 :newProc3 :newProc4 :ne
wProc5                                                        :newProc6
)&body body
```

Executes body with the specified QuickDraw bottlenecks installed.

Functions of Interest

`rgn-handle h &optional v`

Moves a region to the specified point by offsetting it.

`rect frame-width shadow-position`

Draws a 3D frame inside the border of the specified rectangle. The `shadow-position` can be `:topLeft` (for a pushed in effect) or `:botRight` (for a popped out effect). The shadowed half of the frame is drawn in the current foreground color, the unshadowed half of the frame is drawn in the current background color. Note: This effect only looks right over patterned or colored backgrounds.

QD-fx-u

Utilities for creating special bit copying effects that will amaze your friends. `GWorld-svm` already supports these effects, so you probably won't need to use these functions directly unless you're rolling your own. You should be familiar with `CopyBits` as described in IM I and V.

Functions of Interest

The `src-portBits` argument to each of these function needs to be either a `BitMap`, a `pixMapPtr`, or the `portBits` field of either a `GrafPort` or a `CGrafPort` (or `GWorld`). The `view-portBits` method will return an appropriate value for a view or simple view.

```
                src-portBits                    src-rect                    dest-rect
  &key copy-mode copy-rgn delay-ticks dissolve-type
```

Does a dissolve from the specified rectangle in `src-portBits` to the specified rectangle in the current port. The meanings of the keys are:

```
:copy-mode [#$srcCopy]       QuickDraw                transfer                mode
:copy-rgn [(%null-ptr)]         clipping   region   for   transfer   (wptr   coords)
:delay-ticks [0]             time   delay   between   stages   of   the   transfer
:dissolve-type [:transporter]  the type of dissolve used
```

The dissolve effects currently supported are:

```
:transporter     random                                                    bits
:screen-door     more            organized            than            :transporter
:waynes-world wavy   lines   àla   Wayne's   World   dream   sequences,   excellent!
:h-blind         horizontal                    blinds                     turning
:v-blind         vertical blinds turning
```

```
                src-portBits                    src-rect                    dest-rect
  &key copy-mode copy-rgn delay-ticks dissolve-type
```

Does a wipe from the specified rectangle in `src-portBits` to the specified rectangle in the current port. The meanings of the keys are:

```
:copy-mode [#$srcCopy]          QuickDraw               transfer               mode
:copy-rgn [(%null-ptr)]            clipping  region  for  transfer  (wptr  coords)
:delay-ticks [0]                time   delay   between   stages   of   the   transfer
:wipe-direction [:left-to-right]   :left-to-right,:right-to-left,
                                   :top-to-bottom,:bottom-to-top
:wipe-count [8]                    number of stages to the copy
```

```
                src-portBits                    src-rect                    dest-rect
  &key copy-mode copy-rgn delay-ticks dissolve-type
```

Does an iris expansion/contraction from the specified rectangle in `src-portBits` to the specified rectangle in the current port. The meanings of the keys are:

```
:copy-mode [#$srcCopy]    QuickDraw                  transfer                  mode
:copy-rgn [(%null-ptr)]      clipping    region    for    transfer    (wptr    coords)
:delay-ticks [0]          time    delay    between    stages    of    the    transfer
:iris-direction [:outward]   :outward                   or                   :inward
:iris-count [8]              number      of      stages      to      the      copy
:iris-shape [:round]         :round or :square
```

Advanced Functions

```
         fn mask-rect pat-hex-string
fn mask-rect iris-direction iris-count
fn mask-rect iris-direction iris-count
```

These four functions repeatedly call `fn` with successive masks used in creating the bit copying effects described above. `fn` should accept one parameter, the `view-portBits` of a 1 bit `GWorld` containing the mask to use (this value is suitable for passing to `CopyBits` - for more info see `view-portBits`). The `mask-rect` parameter is used to define the size of the mask. For best results, its `topLeft` should be `#@(0 0)` and its `botRight` the size of the destination.

If you're putting bits into a window, you should be using one of the `x-o-rama` functions. If the destination of your spiffy effect is not a window, you can use these functions to generate the sequence of masks and use them as you please. The only plausible use I can think of is for showing/hiding on screen video with an effect.

Note: The successive masks do not overlap. The union of all the masks is pure black. This is precisely what's needed for copies to the screen. You don't want the masks to overlap or you'll end up copying a bit twice, which is a problem for some copy modes. For doing a fade into live video you'll want to accumulate the series of masks (oring them) into the mask the video uses. For doing a fade out of live video you'll want to subtract (xoring them) them from a video mask that started at all black.

Utilities for moving images around.

```
                                    drag-rgn                                    start-pt
  &key  :bounds-rect    :slop-rect    :drag-axis    :action-fn    :erase-at-start-p
        :erase-at-end-p :border-size :saved-bits-init-fn
```

This function is similar to the `DragGrayRegion` trap call - see IM I. Unlike `DragGrayRegion`, it drags the whole image, not just a dotted outline. Two values are returned; change in mouse position as a point (delta-h,delta-v) and the final mouse position as a point. The meanings of the keys are listed below. The first four are analogs to the arguments to `DragGrayRegion`. The rest have no analog because dragging the whole image raises issues that do not arise when dragging just an outline.

`:bounds-rect [nil]`	A ptr to a rectangle record that limits the allowable drag area. If nil, the drag is unconstrained.
`:slop-rect [nil]`	A ptr to a rectangle record that controls the amount of slop tolerated before the mouse is considered outside the drag area. If nil, the bounds-rect is used.
`:drag-axis [:both]`	Determines the directions which the region can be moved. Allowed values are: :both, :vertical, or :horizontal.
`:action-fn [nil]`	If specified, this function is funcalled continuously during the drag. It takes no arguments.
`:erase-at-start-p [nil]`	If this key is non-nil, the bits underneath the drag-rgn will be erased in the current ports background color an pattern - to give the effect of moving the object (rather than moving a copy)
`:erase-at-end-p [t]`	If this key in non-nil, the dragged bits will be erased when the mouse is released. This might be desirable if the drag conveyed an object being deposited into a folder. It would probably be undesirable if the drag conveyed an object being moved to a new location.
`:border-size [#@(50 50)]`	This controls how much screen is buffered for use in saving and restoring the bits underneath the region. It has a strong effect on the smoothness of the animation. You can adjust it to suit your tastes. Some factors to consider are: the speed of the machine, the amount of available memory, the depth of the monitor, the drag-axis.
`:saved-bits-init-fn [nil]`	This function can be used to draw the bits which are initially obscured by the object, but which are revealed when the object is moved. This is a more complex alternative (or addition) to the :erase-at-start-p key. It would be useful if there were an object obscured by the region. The function is passed two arguments, the region being dragged and the bounding rectangle of the offscreen GWorld used to save the bits (both in global coordinates).

`:drag-over-p-fn [see desc]` This function returns t when the drag is over. The default simply calls the WaitMouseUp trap. This function takes no arguments.

`:drag-cur-pos-fn [see desc]` This function returns the current position (in global coordinates) of the drag. The default uses the GetMouse trap. This function takes no arguments.

```
                                drag-rgn                                    point-list
  &key  :erase-at-start-p :erase-at-end-p :border-size :saved-bits-init-fn
```

This function is similar to `drag-region`, but instead of tracking the mouse it uses the points in `point-list`. The keys have the same meanings as in `drag-region`. Note that movement of the region is determined by the deltas between the points rather than the actual positions of the points. You don't have to worry about making the first point line up exactly with the topLeft of the region. This also means two objects that follow a parallel path can share the same point list.

Utilities for working with MCL records

Macros of Interest

These macros attempt **not** to generate runtime code. For example, if a record's length can be determined at macro expansion time, the macro will expand into that value rather than generating code for a run time lookup of the length. In general, all you need to do is make sure your defrecords have evaluated - MCL's defrecords get auto-loaded.

`record-type`

Returns the record-type's length (in bytes). Does not round to an even number of bytes like MCL's `record-length` macro (ouch!).

`record-type`

Returns the record-type's default storage, `:pointer` or `:handle`.

`record-type`

Returns the record-type's fields.

`record-type field`

Returns the field's type.

`record-type field`

Returns the field's offset (in bytes).

`record-type field`

Returns the field's length (in bytes).

Utilities for working with macptrs.

Functions of interest

These functions are additions to the set of `%get-xxx` and `%put-xxx` functions MCL provides.

```
                    ptr                         &optional                       offset
ptr bool &optional offset
```

Gets/puts boolean values. Especially useful when passing/getting boolean values as var parameters to ToolBox calls.

```
                    ptr                         &optional                       offset
ptr char &optional offset
```

Gets/puts character values. Especially useful when passing/getting character values as var parameters to ToolBox calls.

```
              ptr               length               &optional               offset
ptr string &optional offset
```

Gets/puts blocks of text. `%get-text` needs to be passed the number of characters to get (`%put-text` can figure it out from `(length string)`).

```
     ptr       elt-%get-fn       elt-size       elt-count       &optional       offset
ptr list elt-%put-fn elt-size &optional offset
```

Gets/puts arbitrary lists. You pass in a `%get-xxx` or `%put-xxx` function to get/put individual elements of the list. You also need to specify the size (in bytes) of each element. `%get-list` also needs to be passed the length of the list it's getting (`%put-list` can figure it out from `(length list)`).

```
             ptr               byte-count               &optional               offset
ptr hex-str &optional offset
```

Gets/puts hex values (which you specify as LISP strings) into memory. `%put-hex-str` accepts a LISP string which can be any length, so it is a useful alternative to the ToolBox function, `StuffHex`, which is limited to 255 characters. Unlike `StuffHex`, `%put-hex-str` ignores (skips over) characters which are not hex digits ([`1…9,A…F,a…f`]). This allows you to group your hex character into blank-separated groups without affecting their value. `%get-hex-str` is useful for taking a quick look at the contents of a chunk of memory, it returns a string of hex digits with every four hex digits (word) separated by a blank. Note: a byte corresponds to two hex digits, so if you specify `n` as the `byte-count` to `%get-hex-str`, you'll get back a string containing `2n` hex digits. Similarly, if you pass in a `2n` hex digits to `%put-hex-str`, `n` bytes get written to memory.

Macros of Interest

`trap-call`

This macro is a simple way to error check trap calls for non-zero results. If `trap-call` returns a non-zero result an error message is generated which shows the trap called and the arguments passed.

`trap-form &rest body`

This macro checks for errors like `trap-nz-echeck`. In addition, if an error is detected, `body` is executed. Note: `body` is only executed if an error occurs. If you have code that needs to execute either way, you should use `unwind-protect` in conjunction with `trap-nz-echeck`.

`(trap-number new-trap-addr &optional old-trap-addr) &body body`

This with- macro temporarily patches a trap. The `old-trap-addr` argument should be a symbol which will be bound to the original address of the trap (in case you need to call the original trap). See the example code at the bottom of `Traps-u.lisp` for a plausible usage.

MCLs-funniest-home-video

How do I get video into my program? The quickest way to learn enough to get video up and running is to try the example code in `video-example.lisp` and read the Video Mixins section. The rest of the sections can be read on an as needed basis.

This is a first cut at video classes. I've tried to anticipate the advent of QuickTime, hopefully it will smoothly integrate into the architecture. When I get the time I will do this.

Before diving into the details of each class, it helps to have a big picture of how all the video classes are used together to get video into a program.

 Extremely Confusing Representation of oou Video Classes

Control of video sources and control of on screen video digitizing are handled by separate classes. It's quite possible to need one without the other. For instance, your video source may be a home VCR with no serial port interface. In this case, digitizing is controlled by the computer and the VCR is controlled manually.

Video sources are controlled through `video-player` objects. The `video-player` class provides a basic model of video control, defining methods like `vp-play` and `vp-stop` which each hardware specific class must implement.

Video digitizers are controlled through `video-digitizer` objects. The `video-digitizer` class provides a basic model of digitizer control, defining methods like `vd-start-digitizing` and `vd-stop-digitizing` which each hardware specific class must implement.

Built on top of the video players and digitizers are two simple view mixins, `video-digitizer-svm` and `video-svm`. Using these mixins, you can define view classes that will display and control video. It's far easier to use these mixins than to deal directly with the video objects. For example, `video-digitizer-svm` automatically takes care of reconfiguring the video hardware if the view is resized or moved.

Neither `video-digitizer-svm` or `video-svm` inherit from the `video-digitizer` or `video-player` classes. Instead they use slots to hold `video-digitizer` or `video-player` objects. When you issue a video command to a video view, the view passes the command on to the appropriate object. The `video-digitizer-svm` defines a slot for holding a `video-digitizer` object. The `video-svm` class inherits from `video-digitizer-svm`. It defines a slot for holding a `video-player` object and it inherits the digitizer slot from `video-digitizer-svm`.

Due to the way digitizers blast their video on screen, windows containing digitized video must be handled specially. The `video-wm` mixin is designed to handle these problems. You can either use `video-wm` to create your own classes or use the pre-defined `video-window` and `video-dialog` classes which are just the standard MCL `window` and `dialog` classes with the `video-wm` mixed in.

The hardware this code has been tested on to date is:

- Pioneer 8000 laserdisc player
- RasterOps 364 digitizing board
- RasterOps 24STV digitizing board (not very extensively)
- MoonRaker digitizing board

I anticipate most people's needs will be satisfied by installing a `video-view` in a `video-window` or a `video-dialog`.

If you're controlling a video player, but not doing on screen digitizing, see the `video-player` class.

If you're using on screen video, but you're manually controlling the video source, see the `video-digitizer` and `video-digitizer-svm` classes.

If you dig into the rest of the documentation and the source you can see how to get more control over the hardware and how to create classes for other digitizer boards or video players.

Digitizers

This object forms a framework for creating digitizer objects for specific video boards. Currently, this class is a dummy class (it provides no functionality) The default routines must be shadowed by board specific specialization's.

Using QuickTime this can change and it will be possible to write most of the default routines independent of any particular digitizing board. (provided, of course, the board supports QuickTime)

Initargs

`[1]`

If multiple video cards of the same type are in use, this specifies which card to use.

`[no default]`

A pointer to the color window to contain the digitized image.

```
   [no default]
[no default]
```

The source rectangle from which video is digitized. (MaxSrcRect coordinates)

```
   [no default]
[no default]
```

Portion of the source rectangle from which video is grabbed. (MaxSrcRect coordinates)

```
[no default]
[no default]
```

The destination rectangle where the video is displayed. (display port coordinates)

`[:composite]`

This controls the video format. Allowed values are :composite, :s-video,:RGB :NTSC, :PAL, :SECAM. Not all digitizer boards support all these formats.

`[:NTSC]`

This controls the signal standard. Allowed values are :RGB, :NTSC,:PAL,:SECAM. Not all digitizer boards support all these standards.

```
[:unsupported]
[:unsupported]
   [:unsupported]
         [:unsupported]
   [:unsupported]
   [:unsupported]
```

These initargs control various aspects of the picture. The allowed range of values is 0 -65535. Not all digitizer boards support all these settings. It is expected that each board specific classes will provide appropriate default initarg values.

Methods of Interest

`(vd video-digitizer) error-code`

Used to check the result codes of commands to the digitizer board.

`(vd video-digitizer)`

Returns an alist (error-code . message-string) used by `vd-nz-error-check`. Should be shadowed to return a board specific error code alist.

`(vd video-digitizer)`

Should be called before the vd object is used. Should be augmented with board specific methods to do whatever initialization is required.

`(vd video-digitizer)`

Should be called when you're through with a vd object. Should be augmented with board specific methods to do whatever deallocation is needed.

`(vd video-digitizer)`

Configures the digitizer board to the settings of a particular vd object. Necessary when a single board is shared by more than one vd object. This should be augmented with an after method that installs board specific settings.

`(vd video-digitizer)`

Grabs a single frame of video. Should be augmented with board specific after methods. The default routine calls `vd-install-settings` and nothing else. So your after method needs to issue a board specific command to digitize a single frame. The default method make no attempt to do the grab because, most boards support some way to do this other than issuing a start digitizing and stop digitizing command in rapid succession.

```
(vd video-digitizer)
(vd video-digitizer)
```

Starts/stops the video digitizing. These should be augmented with board specific after methods. The default method maintains a flag that keeps track if the video object is currently digitizing. The default start routine calls vd-install-settings.

`(vd video-digitizer)`

Returns `t/nil` indicating whether the board is currently digitizing video. The default method returns the value of the `digitizing-flag` slot. This should probably be specialized by a primary method that also makes some board specific test which actually interrogates the hardware. e.g.

```
   (defmethod              vd-digitizing-p              ((vd              mr-vd-mx))
      (and (call-next-method) (board-specific-test vd)))

   (vd video-digitizer)
   (vd video-digitizer)
   (vd video-digitizer)
   (vd video-digitizer)
   (vd video-digitizer)
(vd video-digitizer)
         (vd video-digitizer)
      (vd video-digitizer)
         (vd video-digitizer)
         (vd video-digitizer)
            (vd video-digitizer)
                  (vd video-digitizer)
         (vd video-digitizer)
            (vd video-digitizer)
```

These accessors are all usable with `setf` to change settings. The settings are stored away in slots and get used to configure the board when the video object starts digitizing. The `setf` methods have after methods which will immediately update the digitizer board if the video object is digitizing when you change a setting.

```
      (vd video-digitizer) topLeft botRight
      (vd video-digitizer) topLeft botRight
      (vd video-digitizer) topLeft botRight
   (vd video-digitizer) format
(vd video-digitizer) standard
   (vd video-digitizer) level
   (vd video-digitizer) level
      (vd video-digitizer) contrast
            (vd video-digitizer) hue
      (vd video-digitizer) saturation
      (vd video-digitizer) sharpness
```

These methods should be shadowed by board specific ones that really do something. These methods only change the settings on the digitizing hardware. They should NOT be used to change settings of a video object. Instead, `setf` on the corresponding slot accessors should be used.

This class is a specialization of `video-digitizer` for MoonRaker digitizing boards. The MoonRaker board requires that the WTI-VideoMgr init be installed in your system.

There is a bug in versions of the WTI-VideoMgr init involving clipping. The problem shows up when the origin of the windows coordinate system is not at the topLeft corner of the window. This will happen when you focus on a view whose position is not `#@(0 0)`. The work around for now is to only use MR video objects in simple views whose container is the window. A new version of WTI-VideoMgr will be released to fix the problem.

The `MR-vd` file also contains a number of MoonRaker specific functions which I haven't taken the time to include in this document. See their source + the MoonRaker Developer's manual for more information.

See Also

`video-digitizer` - inherited behavior

Initargs

`[nil]`

Flag controlling whether or not to map 8 bit colors into grays.

`[nil]`

Flag controlling whether or not to extract the sync information from the green channel when using :RGB1 or :RGB2 input formats.

`[255]`

Size of the color table.

`[:composite]`

In addition to the normal formats, the MoonRaker boards supports two RGB inputs, specified with :RGB1 and :RGB2. (:RGB defaults to :RGB1)

```
[:unsupported]
[:unsupported]
   [:unsupported]
```

MoonRaker boards don't support these settings.

```
[(ash 12 12)]
      [(ash 8 12)]
[(ash 8 12)]
```

MoonRaker boards support 16 values [0-15] for each of these settings. Map these 16 values into the required range (0-65535) using: (ash MR-setting 12) {or equivalently (* MR-setting 4096)}

Methods of Interest

```
       (vd MR-vd)
(vd MR-vd)
          (vd MR-vd)
```

These accessors are all usable with `setf` to change settings.

`(vd MR-vd)`

This method installs an optimal color table into the window containing the video object. It takes several seconds to execute and if digitizing is in progress it will be temporarily halted.

This class is a specialization of `video-digitizer` which forms the basis for model specific RasterOps classes. It cannot be used by itself.

The `RO-vd` file also contains a number of RasterOps specific functions which I haven't taken the time to include in this document. See their source + the RasterOps Developer's ToolKit for more information.

See Also

`video-digitizer` - inherited behavior

Initargs

`[t]`

This flag determines if the RasterOps board will use an alternate Phase-lock-loop. Enabling this option may improve the video signal stability on some video media.

```
[nil]
[nil]
```

These flags allow you to flip the image horizontally and/or vertically.

`[nil]`

This flag determines if the RasterOps board will swap the order in which it display the odd/even fields. Enabling this option may improve the video signal quality on some video media.

`[:full]`

Controls rate of digitizing. Allowed values are :full(30 fps) and :half(15 fps).

`[:both]`

Controls what fields are used in half size images. Allowed values are: :both, :odd, :even. Additionally, the 24STV supports: :both-dls, :odd-sls, :even-sls.

This class is a specialization of `RO-vd` for model 24STV digitizing boards.

A common problem in using the RO24STV-vd is access to it's driver. The disks that comes with the RasterOps board have the driver on them. You will find the DRVR resource (ID=56) in the 24STV XCMDs and XObjects. Use ResEdit to copy the driver into your own resource file. You must ensure:

- The resource file containing the driver is open. This can be done with code similar to:

```
(oou-dependencies                                                    :Resources-u)
(open-res-file "oou:MCLs-funniest-home-videos;RO24STV-driver.rsrc")
```

- The driver name is correct in the definition of RO24STV-drvr-name below. Different versions of the driver have different names. The version used in the development of this code used the name, `".RasterOps24STVPIP1.1d3"`. If you upgrade the driver, be sure to check the name.

High level functions have been provided to access all of the 24STV features, but they have not been incorporated into the normal object behavior. Probably, the most useful of these features is masking. It would be nice to use masking to get the RO24STV-object to respect the current clipping region.

The `RO24STV-vd` file also contains a number of 24STV specific functions which I haven't taken the time to include in this document. See their source + the RasterOps Developer's ToolKit for more information.

See Also

`RO-vd`	- inherited behavior
`video-digitizer`	- inherited behavior

Initargs

```
[nil]
[nil]
[nil]
```

These flags allow suppression of individual RGB video components.

```
[:unsupported]
[:unsupported]
```

RO24STV boards don't support these settings.

```
      [(ash   0 8)]
[(ash 59 8)]
```

RO24STV boards support 256 values [0-255] for each of these settings. Map these 256 values into the required range (0-65535) using:

```
(ash MR-setting 8) {or equivalently (* MR-setting 256)}
```

Note: Some of the early 24STV board ROMs don't support saturation. An error code gets returned if you try to get the saturation. I believe that RasterOps is making ROM upgrades available to fix the problem. Consequently, the definitions of `vd-set-saturation` & `vd-get-saturation` are commented out. If your board supports saturation (or you want to try it to find out), uncomment their definitions.

This class is a specialization of `RO-vd` for model 364 digitizing boards. (364 is a discontinued model)

The `RO364-vd` file also contains a number of 364 specific functions which I haven't taken the time to include in this document. See their source + the RasterOps Developer's ToolKit for more information.

See Also

`RO-vd` - inherited behavior
`video-digitizer` -inherited behavior

Initargs

`[32]`

RO364 boards support brightness control. Allowed range is 0-63.

```
[nil]
[nil]
[nil]
```

These flags allow suppression of individual RGB video components.

`[:unsupported]`

RO364 boards don't support sharpness.

```
[(ash 29 10)]
[(ash 56 10)]
   [(ash 32 10)]
         [(ash 32 10)]
   [(ash 32 10)]
```

RO364 boards support 64 values [0-63] for each of these settings. Map these 64 values into the required range (0-65535) using:

```
(ash RO364-setting 10) {or equivalently (* RO364-setting 512)}
```

Methods of interest

`(vd RO364vd-install-332-table)`

This method installs a 3-3-3 table in the RO364 GDevice. This allows for reasonable looking video in 8 bit mode.

Players

This class is a first attempt to create a basis for controlling video input devices (laserdisc player, PC-VCRs, DVI hardware, ...) Currently it's only been used for Pioneer laserdisc players. As more hardware becomes available (to me) it will be expanded. Hopefully, the selection of video control methods will be sufficient for most purposes. Not all hardware will support everything. The `vp-features` method should be used to see what's available.

Initargs

`[`*`no default`*`]`

This initarg should be a function which takes one argument, the current frame. It will be called as the current frame number changes. It allows displaying the current frame number. Some considerations:

- a slow framehook-fn will degraded MCL's performance
- it won't get called often enough to see every frame change, count on gaps
- the frame number will be decreasing during reverse scans

Methods of Interest

Unless otherwise indicated, these methods return `t` if successful.

`(vp video-player)`

Should be called before the vp object is used.

`(vp video-player)`

Should be called when you're through with a vp object.

```
   (vp video-player) &key &allow-other-keys
(vp video-player)
```

Make sure that the video player is loaded before using it. Use `vp-loaded-p` to check. Call `vp-load` when new media is put into the player. `vp-load` accepts player specific keyword args (e.g. 330's use `:disk` and `:side`)

`(vp video-player)`

Returns a list of keywords indicating what features are available. Features can vary with the loaded media (e.g. CAV vs. CLV laserdiscs) Currently, the possible features are:

```
:freeze       -    stop    at    and    continuously    display    a    single    frame
:step-forward -         move         forward         a         single         frame
:step           -     move     forward     and     backward     a     single     frame
:scan           - play forward and reverse at high speed
```

```
(vp video-player)
(vp video-player)
```

Returns the range of allowable frame number. Can vary with loaded media. (e.g. CAV laserdisc's max = 65535, CLV laserdisc's max = 863970)

Core Feature Control Methods

`(vp video-player)`

Returns the current frame number.

`(vp video-player) frame &key &allow-other-keys`

The hardware searches to the specified frame (in preparation for play). Player specific keyword args are accepted as in `vp-load`.

`(vp video-player)`

Video is played starting from the current position.

`(vp video-player)`

Stops video playing.

`(vp video-player) min-frame max-frame`

Sets up limits for video play (`nil` for no limit). Use `vp-play-clip` if you want to set up temporary limits for the duration of a clip. Note: `vp-limit` has no effect on video in progress. The new limits will not take effect until the current video stops or freezes.

Optional Feature Control Methods

`(vp video-player)`

Freezes the video on the current frame.

`(vp video-player) direction`

Advances the video by one frame. The direction key can be `:forward` (the default) or `:reverse`.

`(vp video-player) direction speed-x`

The video plays at an accelerated or decelerated speed. The direction key can be `:forward` or `:reverse`. The `:speed-x` key is a number indicating at how many times the normal rate the video should play (fractions can be used - e.g. `1/2` for half speed).

High Level Feature Control Methods

`(vp video-player) start-frame end-frame &key &allow-other-keys`

Plays the specified clip. Player specific keyword args are accepted as in `vp-load`.

`(vp video-player) direction frame-count`

Jumps `:forward` or `:reverse` the specified number of frames.

Designing Video Players

Should be augmented with before/after methods to do player specific initialization. (e.g. Pioneer-vp, vp-init initializes the serial port)

These primary methods need to be written. `vp-load` should do whatever player specific initialization is necessary when new media is put in. `vp-loaded-p` checks if the player is loaded.

Should be specialized to return a list of available features.

These primary methods need to be written. The video-player class provides some around and after methods.

These methods should start the video and then return immediately (not waiting for completion of the play). The framehook mechanism cuts play off at the limits, but it doesn't get called every frame, so it can miss by a few. I try to compensate, but if your device supports stop markers or some other mechanism for controlling the limits of play, you should use them in your `vp-play` and `vp-scan` methods. Check `(frame-limit-p vp)` and use `(max-frame-limit vp)` and `(min-frame-limit vp)` as the limits. Also, be sure not to accidentally leave stop markers in place. You might fix this by first clearing them in `vp-play` & `vp-scan` if `(frame-limit-p vp)` is `nil` or by clearing them in after methods to `vp-stop` and `vp-freeze`.

These are considered higher level methods because they can be implemented in terms of the other methods. If your device can perform them in a more efficient way, then their primary methods should be shadowed.

This file contains various utilities for working with Pioneer laserdisc players.

Functions of Interest

```
   frame
   time
frame
time
```

These functions convert between frame numbers and the two time formats used by Pioneer, hmmss (hours minutes seconds) and hmmssff (hours minutes seconds frames)

`port &key :timeout`

Returns a plist describing the current player configuration. The port arg specifies whether to check the :printer or :modem serial port. The timeout parameter controls how long to wait for the player to respond. This function is slow to execute, so call it just once when your code starts up and save the plist. The plist format is:

```
indicator     values
:port         :modem,                                                    :printer
:baud         1200,                          4800,                          9600
:stop-bits 1.0,                                                           2.0
:parity    :none,                          :even,                         :odd
:data-bits 7,                                                               8
:model        P330-vp, P4200-vp, P8000-vp
```

These info plists are convenient for creating `Pioneer-vp` objects. e.g.

```
(defvar *pld-info* (or   (Pioneer-player-info                                :modem)
                      (error     "Player     on     modem     port     not     responding.")))

(apply 'make-instance (getf *pld-info* :model) :framehook-fn #'my-hook *pld-info*)
```

Pioneer-player-info works by trying combinations (up to 36) of serial port settings until getting a response. This can take a while. I've attempted to optimize the process is by trying the most likely combinations first. The timeout parameter determines how long to wait for a response with each combination. The default is 2 seconds. Too short a value may miss the response.

`port`

This function returns the format of the currently loaded disk. (`:CAV, :CLV, :CLV-E`) CLV-E is a CLV disk with Extended Philips Code. Pioneer models which don't handle CLV-E (e.g. 4200 and 330) will report such disks as vanilla CLV. (model 8000 supports CLV-E)

This class is a specialization of `video-player` for Pioneer laserdisc players with serial port control.

See Also

`video-player`	- inherited behavior
`serial-port`	- inherited behavior
`Pioneer-u`	- utility functions

This class is a specialization of `video-player` which forms the basis for the model specific `Pioneer-vp` classes. The features supported vary with model and the type of disk in use. So be sure to use the `vp-features` method to see what's available.

Pioneer video players inherit from `serial-port`. When you create one you specify the baud rate, serial port, etc.…. The `Pioneer-player-info` function can be used to determine what the player's current configuration is. This function is potentially slow, so you probably want to call it once, when you start-up. e.g.

```
(defvar *pld-info* (or   (Pioneer-player-info                                    :modem)
                         (error     "Player     on     modem     port     not     responding.")))

(apply 'make-instance (getf *pld-info* :model) :framehook-fn #'my-hook *pld-info*)
```

Initargs

`:framehook-fn` - see docs on video-player

`:port, :baud, …` - see docs on serial-port

Methods of Interest

`(vp Pioneer-vp) code-string &key`
`:arg :frame :disk-format :response-p :error-p :flush-p`

This function issues a command string to a Pioneer laserdisc player. It can prefix the commands with an arbitrary arg or an address (specified as a frame number) It handles formatting the frame number into an appropriate address.

`:arg` [*no default*] - pre-pended to the code-string
`:frame` [*no default*] - address is pre-pended to the code-string
`:response-p [t]` - if non-nil, pld-cmd waits for & returns the player's response code
`:error-p [t]` - if non-nil, an error code response will signal an error
`:flush-p [t]` - if non-nil, any pending result codes are flushed (if response-p is t, a flush is performed regardless of flush-p)

`(vp Pioneer-vp) &key :wait-p`

Reads the next pending response from the serial port buffer. If no response is available and wait-p is nil (the default) it will return immediately, else it waits for a response. Note: The default serial port buffer holds only 64 bytes.

`(vp Pioneer-vp)`

Flushes any pending responses from the serial port buffer.

`(vp Pioneer-vp) disk-format`

This primary method needs to be implemented for each model specific class. It returns what addressing format (:frame,:hmmss,:hmmssff) should be used with a given disk format.

This class is a specialization of `Pioneer-vp`. The model 8000 supports Extended Philips Coding on CLV disks. This allows CLV-E disks to be accessed by frame number. Plain CLV disks are only accessible to the nearest second and do not support step or scan.

Features supported:

```
CLV   :freeze
CLV-E :freeze, :step, :scan
CAV   :freeze, :step, :scan
```

The model 4200 does not support Extended Philips Coding on CLV disks. They are treated as ordinary CLV disks. This means that CLV disks are only accessible to the nearest second and do not support freeze, step, or scan.

Features supported:
`CLV` *none*
`CLV-E` *none*
`CAV` `:freeze, :step, :scan`

P330-vp

The model 330 is basically a model 4200 with additional capabilities for selecting a disk (JukeBox style) and playing either side of the disk.

See Also
`P4200-vp` - inherited behavior

Features supported:
`CLV` *none*
`CLV-E` *none*
`CAV` `:freeze, :step, :scan`

Methods of Interest

```
(vp P330-vp) &key :disk :side
(vp P330-vp) frame &key :disk :side
```

These two methods accept additional keywords, `:disk` and `:side`, for specifying disk number and side (`:A` or `:B`). The `vp-seek` command handles loading the specified disk, so it's not necessary to call `vp-load` when you seek to a new disk. When `:disk` or `:side` is not specified, the currently loaded disk and side are used.

Video Mixins

This mixin class attempts to solve some of the problems video digitizers cause. You should display video in windows which inherit from `video-wm`. Since the most common use of `video-wm` is to mix it with the standard classes, `window` or `dialog`, two classes are provided which do just that:

Some of the problems (and solutions) addressed by video-wm are:

If you drag a window containing live video to a new position, the video gets left behind.
This is solved by stopping digitization during the drag and restarting it afterwards.

Digitizers only support digitizing on a single screen.
This is solved by a variety of means:
- unless an initial position is specified, windows are created positioned on the proper screen
- user drags are limited to the proper screen

- set-view-position generates an error if the new position isn't on the proper screen
- all video views installed in the same window must use the same screen.

If you obscure live video with another window, the live video blasts right through.

This problem is currently unsolved. If you need to address this, here are some things to think about. You could just stop digitizing when the window gets deactivated. This is pretty foolproof, but too restrictive for many applications. You could try clipping the video to the exposed area, but not all digitizers support clipping. Additionally, you'll need to figure out how to get notified when your visible region changes.

Methods of Interest (changes to existing methods)

`:after (w video-wm) &rest initargs`

If no `:view-position` is specified and the window has some digitizer subviews, then the window will be moved onto the appropriate screen. Note: the default value for the `:color-p` initarg is `t`.

`:around (w video-wm) h &optional v`

Digitizing is stopped and restarted.

`:after (w video-wm) h &optional v`

An error is generated if the new position is not entirely on the digitizer's GDevice.

`:after (w video-wm) &rest subviews`

An error is generated unless all the video subviews in the window use the same GDevice.

`(w video-wm)`

The digitizer's GDevice rectangle is used as the dragging rectangle.

`:before (w video-wm)`

Digitization is stopped before hiding the window.

This mixin provides video display for views. It allows controlling when and where the incoming video is placed on the screen, but not control over the video player. This mixin will be useful in situations when the video source is not controlled by the computer. A specialized mixin, `video-svm`, addresses computer control of the player.

`video-digitizer-svm` uses a slot to hold a video digitizer object and provides methods which issue the basic digitizing commands to this digitizer object. More control can be achieved by using the digitizer object directly.

Note: there is currently a problem with the MoonRaker drivers. See `mr-vd` for more details. The current work around is to only use it with simple-views and to install them in a `window` (as opposed to a sub-view of the window)

Note: you should use video in windows of class `video-window`, `video-dialog`, or other classes which inherit from `video-wm`.

See Also

`video-digitizer` - for more info on video digitizer objects.
`??-vd` - board-specific `-vd` files for individual digitizers

Initargs

All the `video-digitizer` initargs are also accepted and used in creating the digitizer object. (the entire initarg list is passed along)

[video-digitizer]

The class of digitizer object to use. (e.g. 'mr-vd for a MoonRaker board)

[*no default*]

This initarg can be used to provide an already existing digitizer object. Allows multiple views to share a common digitizer object. For shared video objects you may want to use a nil :dispose-vd-on-remove-p. See below.

[t]

This flag determines if vd-dispose is called on the digitizer object when the view is removed from it's window. All the video-digitizer initargs are also accepted and used in creating the digitizer object. (the entire initarg list is passed along)

Methods of Interest

(sv video-digitizer-svm)

Returns t/nil indicating whether the board is currently digitizing.

```
(sv video-digitizer-svm)
(sv video-digitizer-svm)
```

Starts/stops continuous digitizing.

(sv video-digitizer-svm)

Grabs a single frame of video.

(sv video-digitizer-svm)

Returns margins for indenting the video as two points (topLeft, botRight). The default method returns zero margins. Specialize this method to control placement.

(sv video-digitizer-svm)

This accessor provides access to the digitizer object used to control the digitizing board. See video-digitizer for the more info.

This mixin is a specialization of video-digitizer-svm which provides video display in views and control over the video source. It incorporates control over the video player, along with control of the on screen digitizing.

video-svm uses a slot to hold a player object and provides methods which issue the basic video commands to this player object. More control can be achieved by using the player object directly.

Note: you should use video in windows of class video-window, video-dialog, or other classes which inherit from video-wm.

See Also

video-digitizer-svm	- inherited behavior
video-player	- for more info on video player objects
??-vp	- board-specific -vp files for individual player classes

Initargs

All the video-digitizer-svm initargs are accepted

All the video-player initargs are also accepted and used in creating the player object. (the entire initarg list is passed along)

All the video-digitizer initargs are also accepted and used in creating the digitizer object. (the entire initarg list is passed along)

`[no default]`

The class of video player object to use. (e.g. 'P330-vp for a Pioneer 330 laserdisc jukebox/boat-anchor)

`[no default]`

This initarg can be used to provide an already existing player object. Allows multiple views to share a common player object. For shared player objects you may want to use a `nil :dispose-vd-on-remove-p`. See below.

`[t]`

This flag determines if `vp-dispose` is called on the player object when the view is removed from it's window.

Methods of Interest

`(sv video-svm)`

This accessor provides access to the video player object used to control the video source. See `video-player` for the more info.

See `video-player` for descriptions of these methods. The `video-svm` versions behave the same plus make appropriate calls to the digitizer. (e.g. `vp-play` also starts digitizing, `vp-stop` also stops digitizing, ...)

objects-of-desire

This class provides access to the Mac serial ports. In its current implementation it is not designed for high speed transfer of lots of data. It's intended use is for controlling serial devices with an ASCII character command set, like laserdisc players and plotters.

Initargs

	allowed values
[:modem]	:modem,:printer
[9600]	300,600,1200,2400,3600,4800,7200,9600,19200,57600
[1.0]	1.0,1.5,2.0
[:none]	:none,:odd,:even
[8]	5,6,7,8

These initargs control the serial port configuration.

[t]
[t]
[t]

These determines if the serial port will be opened, configured, and flushed automatically when the object is created (actually, when the object is initialized). That latter two options are not applicable if `:open-on-init-p` is `nil`.

[#\return]

This character is used to delimit lines for the purposes of `read-line` and `write-line`.

Methods of Interest

(sp serial-port)

Returns which serial port the object is using, `:modem` or `:printer`.

(sp serial-port)
(sp serial-port)
(sp serial-port)
(sp serial-port)

These reader methods return the corresponding serial port params for the `serial-port` object. These are not necessarily the settings currently installed in the physical serial port.

(sp serial-port) baud &key :config-p
(sp serial-port) stop-bits &key :config-p
(sp serial-port) parity &key :config-p
(sp serial-port) data-bits &key :config-p

Use these to change the serial port param. If `:config-p` is `t` (the default) then the change will also be installed in the physical serial port.

(sp serial-port)

Return `t/nil` indicating if the serial port has been opened.

`(sp serial-port) &key :config-p :flush-p`

Opens and configures the physical serial port. `:config-p` and `:flush-p` control whether or not the physical port is configured and flushed when it is opened (both default to `t`). A port must be opened before it can be used. This is done automatically when the object is created unless suppressed by a `nil :open-on-init-p` initarg.

`(sp serial-port)`

Closes the port. Should be done when you're through with the object.

`(sp serial-port)`

Flushes any data current in the serial port buffer.

`(sp serial-port)`

Returns the number of characters currently in the serial port buffer.

`(sp serial-port) &key :wait-p`

Returns a single character from the serial port buffer. If `:wait-p` is `nil` (the default) the function will return immediately if no characters are available, else it waits for a character.

`(sp serial-port) &key :wait-p :wait-eoln-p :eoln-char`

Reads characters one at a time from the serial port buffer until the eoln-char is read. The non-eoln characters are accumulated into a string. It returns two values, the accumulated string, and `t/nil` indicating if the eoln-char was actually read. The `:eoln-char` keyword can be used to override the character in the object's eoln-char slot. If `:wait-p` is `nil` (the default) the function will return immediately if no characters are available, else it waits for a character. If at least 1 character has been read, `:wait-eoln-p` controls whether the function will return as soon as no more characters are available or wait until the eoln-char is read. The value of `:wait-eoln-p` defaults to the value of `:wait-p`.

`(sp serial-port) char`

Writes a single character to the serial port.

`(sp serial-port) string &key :eoln-char`

Writes a string of characters + an eoln-char to the serial port. The `:eoln-char` keyword can be used to override the character in the object's eoln-char slot.

room-with-a-view

Two classes are defined, `back-PICT-view` and `back-PICT-window`, which create views and windows with background pictures. Most of these item's functionality is inherited from the mixin they're based on, `PICT-svm`. The main reason these are defined at all is because it's a little tricky to get `PICT-svm` to work with windows.

See Also

`PICT-svm` - inherited behavior

Initargs

All the `PICT-svm` initargs are accepted.

This class creates views based on the window manager port. Rather than instantiating the class yourself, you should probably just use the pre-defined variable `*WMgr-view*`. To draw into the window manager port you'd just write something like:

```
(with-focused-view                                                    *WMgr-view*
   ;nasty                                drawing                            code
)
```

Note: Drawing into the window manager port is generally considered evil. You should only use it for temporary images like an something being dragged from one window to another. You should clean up after yourself by erasing any drawing you did (xor mode drawing or saved bits) and restoring the drawing environment.

This class creates views based on offscreen `GWorld`s. You should familiarize yourself `GWorld`s by reading the GDevice chapter of IM VI before attempting to use them.

Generally you'll use the GWorld-views in one of two ways:

Drawing to them:

```
(with-focused-view                                                    my-gw-view
   ;drawing                                                              code
)
```

Copying them on-screen:

```
(with-locked-GWorld-view                                              my-gw-view
   (#_CopyBits (view-portBits my-gw-view) (view-portBits dest-view) …))
```

It's important to use the `with-locked-GWorld-view` macro to lock the GWorld before using it (focusing on a `GWorld-view` locks it's bits, but for copying on-screen you want to be focused on the destination view).

Initargs

`[8]`

Specifies the depth of the GWorld.

```
GWorld-views use the view-position as their origin (coordinates of the top left
corner). The view-size determines the size of the GWorld.
```

`[(%null-ptr)]`
`[(%null-ptr)]`

These allow you to specify an alternate color table or graphics device.

`[0]`
`[0]`

These are the flags passed to `NewGWorld` and `UpdatedGWorld`. See IM VI for details.

Methods of Interest

`(view GWorld-view)`
`(view GWorld-view)`

Both of these methods return the offscreen GWorld.

```
          (view GWorld-view)
          (view GWorld-view)
       (view GWorld-view)
    (view GWorld-view)
 (view GWorld-view)
```

These accessors can be used with `setf` to change the various setting. However, the changes will not be propagated to the offscreen GWorld until `GWorld-alloc` or `GWorld-realloc` is called.

```
(view GWorld-view)
(view GWorld-view)
   (view GWorld-view)
```

These methods allocate, reallocate, and deallocate the GWorld underlying the view. It's important that you allocate the GWorld before you use it and that you deallocate it when you're done. Forgetting to deallocate GWorlds will quickly chew up your Mac heap.

Macros

`GWorld-view &body body`

It's essential to lock a GWorld before drawing to it or copying bits from it. Focusing on a GWorld automatically locks it - so this macro isn't generally needed if you're drawing to a GWorld-view.

This view provides styled text editing. It consists of up to one to three subviews: a text-editing box, a vertical scroll bar and a horizontal scroll bar.

See Also

`te-dim` - basis for the text box subview

Initargs

`[:default]`

Controls the text justification. Allowed values are `:default`, `:left`, `:right`, and `:center`.

`[t]`

Determines if the text will be word wrapped.

`[nil]`

This flag allows the text to be read only. Selection and copying are still allowed, but editing is not.

`["The horror…"]`

A LISP string that will be used to initialize the text.

`[`*`no default`*`]`

The resource id or name of a `'TEXT'` resource to be used to initialize the text. If there is a `'styl'` resource with the same id or name, it will be used to initialize the style of the text. These `'TEXT'/'styl'` resource pairs can be created and edited with ResEdit. Make sure the resource file containing the resource is open.

`[t]`
`[nil]`

Specifies if you want vertical and/or horizontal scroll bars.

Methods of Interest

`(di te-dim)`

Returns two values, the character positions of the first and last character selected.

`(di te-dim) sel-start sel-end`

Sets the current selection.

`(di te-dim) font-spec &key font-color mode`

Sets the font of the current selection. You can use the `:font-color` key to change the color by passing in an MCL style encoded color like `*red-color*` (not an `RGBColor` record). The `:mode` key is use to specify which attributes you want to affect (the default is all). The value you pass in is a sum comprised of the following ToolBox constants: `#$doFont`, `#$doFace`, `#$doSize`, `#$doColor`, `#$doAll`, and `#$addSize`. The `#$addSize` constant allows you to increment/decrement the font size. See IM V p.269-270 for details.

`(di te-dim)`

This accessor returns the text as a LISP string. Expensive for large amounts of text. It can be used with setf to change the text.

`(di te-dim) rsrc-id-or-name`

This method lets you change the text using `'TEXT'/'styl'` resource pairs a per the `:te-init-rsrc` initarg.

`(di te-dim) &key rsrc-id rsrc-name`

This save the current text as a `'TEXT'/'styl'` resource pair in the current resource file. Make sure the file you want it save in is current (you can use `with-res-file`).

low-class-extensions

Additional methods for the dialog items.

Methods of Interest

These methods hide/show dialog-items by adjusting their positions as per the Dialog Manager calls, HideDItem and ShowDItem (IM IV p. 59).

`(di dialog-item)`

Hides the specified dialog item by moving it to a position off screen.

`(di dialog-item)`

Shows a hidden dialog item by moving it back on screen.

`(di dialog-item)`

Returns `t/nil` indicating if the dialog item is currently hidden.

`(di dialog-item)`

If the dialog item is hidden, it returns the original position of the dialog item. If the item is not hidden it just returns its current position.

Additional methods for simple views.

Methods of Interest

`(sv simple-view) dh &optional dv`

Moves a view relative to its current position.

These methods hide/show views by adjusting their positions as per the Dialog Manager calls, HideDItem and ShowDItem (IM IV p. 59).

`(di dialog-item)`

Hides the specified view by moving it to a position off screen.

`(di dialog-item)`

Shows a hidden view by moving it back on screen.

`(di dialog-item)`

Returns `t/nil` indicating if the view is currently hidden.

`(di dialog-item)`

If the view is hidden, it returns the original position of the view. If the view is not hidden it just returns its current position.

`(sv simple-view) topLeft botRight`

Adds the specified rectangle (window coordinates) to the view's containing window's erase region. It doesn't actually erase the rectangle itself. The erase will be performed by the normal updating process.

`(sv simple-view)`

Adds the view's rectangle to the window's erase region. It doesn't actually erase the view itself. The erase will be performed by the normal updating process.

Note: You should not normally need to call `erase-corners` and `erase-view`. Instead you should probably be using `invalidate-corners` and `invalidate-view`. Neither of these methods change the update region, so unless the update region already contains the region you cause to be erased, it won't get re drawn. For a discussion of when you might need to use these methods, see Appendix D.

`(sv simple-view) hilite-flag`

This method highlights a view by inverting it. The hilite-flag determines whether to highlight or unhighlight the view. The default method uses InvertRect (in highlight mode).

Note: `hilite-view` is not a general purpose drawing method. It is intended for temporary highlighting effects, like those used in button tracking. You may want to specialize for views you create to get custom highlighting effects. Be sure to read the source for the default method before trying to specialize it.

```
(sv simple-view)
(v view)
```

This method returns the view that should be focused on for drawing and event handling. For simple views, this will be their container. For views, this will be the view itself.

```
(sv simple-view)
(v view)
```

This method is like `view-corners`, except that it returns the corners in the coordinate system the view uses for drawing and event handling. For a simple views, this will be its container. For a views, this will be itself.

`(sv simple-view)`

This method returns a pointer to be passed into `CopyBits`. Nominally, it returns the `portBits` field of the specified view's port. Even though `CGrafPort`s don't have a `portBits` field, the value returned is still valid to be passed to `CopyBits` (see IM V p. 70)

```
(sv simple-view) point
(sv simple-view) point
(sv simple-view) point
(sv simple-view) point
```

These methods return `point` converted between various coordinate systems.

```
(sv simple-view)
(sv simple-view)
```

These methods are similar to `view-corners`, except their return values are in the specified coordinate system Note: `view-corners` returns them in the coordinate system of the view's container.

Additional methods for the various classes of `window`.

Methods of Interest

```
(w window) which-screen &key
     :upper-3rd-p :move-now-p :GDevice :point
```

This method returns the position which will center a window on particular screen. The `which-screen` argument is a keyword that specifies which screen to use. Allowed values are:

```
:specified-GD    the    GDHandle    is    specified    by    the    :GDevice    key
:deepest            screen      with      the      greatest      pixel      depth
:main               screen           with           the           menu           bar
:containing-mouse   screen       currently       containing       the       mouse
:containing-point   screen containing the point specified by the :point key
```

If `:move-now-p` is `t` (the default) it will call `set-view-position` to move the window to this position. If `:upper-3rd-p` is `t` (the default) the window won't be vertically centered, but will be positioned with one third of the empty space above it and two thirds below. This is usually more visually pleasing than perfect vertical centering. The `:GDevice` and `:point` keys are only used in conjunction with certain types of screen specification .

Appendix A - Classic LISP Blunders

"nconc doesn't work."

```
(setf a '(1))
(setf b '(2 3 4))
(setf c nil)

(nconc a b) ;-> (1 2 3 4) ;OK
a ;-> (1 2 3 4) ;Big deal.
(nconc c b) ;-> (2 3 4) ;zzz…
c ;-> nil ;WHAT! It's still nil?
```

Explanation: You're using `nconc` for side effects only, ignoring the return value. If the first argument to `nconc` is `nil`, `nconc` can't modify it (i.e. destructively change the last cons in the list). If you still don't understand, try implementing your own version of `nconc`, it's a simple, but instructive, exercise.

Solution: Always use the result of `nconc`. In the above example, using `(setf c (nconc c b))` would have avoided the problem.

"Miscellaneous destructive sequence function doesn't work."

```
(setf a '(1 2 3 4))

(delete 3 a) ;-> (1 2 4) ;OK
a ;-> (1 2 4) ;So what?
(delete 1 a) ;-> (2 4) ;What's your point?
a ;-> (1 2 4) ;WHAT! How did 1 get back in there?
```

Explanation: When the need arises to change the first element of the sequence, the destructive sequence functions can only pass back a pointer to the new first element, they can't do anything about any pointers you still have to the old first element.

Solution: Always use the result of the destructive function. In the above example, using `(setf a (delete 3 a))` would have avoided the problem.

"I change one element of my list and all the elements change."

```
(defun foo () '(1))
(setf a nil)

(push (foo) a) ;-> ((1)) ;OK
(push (foo) a) ;-> ((1) (1)) ;Of course.
(push (foo) a) ;-> ((1) (1) (1)) ;It's obvious.
a ;-> ((1) (1) (1)) ;ho, hum.
(setf (first (first a)) 2) ;-> 2 ;
a ;-> ((2) (2) (2)) ;WHAT!
(foo) ;-> (2) ;Has LISP gone wild.
```

Explanation: Some or all of the elements of your list (or other data structure) are shared. An easy way for this to happen is by a function to return a quoted form. In most LISP implementations, such a function will always return the same lisp object. Therefore, any destructive changes to that object are reflected in subsequent function calls. In the above example, note that not only the variable `a` is affected, the function `foo` now returns `(2)`.

Solution: Never return a quoted form from a function. Also, if you use quoted forms in a function, be careful not to pass them off to any functions that might destructively change them. In the above example, using `(defun foo () (list 1))` would have avoided the problem.

"defmacro doesn't work."

Explanation: When you evaluate a `defmacro` it doesn't go back and update code that has already been compiled under the old definition.

Solution: When redefining a macro, be sure to go back and recompile functions (and `.fasl` files) that use the macro.

"When I compile my file, LISP warns me that my macros are undefined functions."

Explanation: `defmacro`s, `load`s, and `require`s are not normally evaluated at compile time. If your file uses macros, you must make sure they are in effect at compile time.

Solution: If the macros are used in the same file they are defined in, make sure they appear in an appropriate `eval-when` before they are used. If the macros are defined in a file you require, make sure your `require` or `load` statement is in an appropriate `eval-when`. Many people avoid all this nonsense by making sure to load all their files before compiling them.

"Conflict errors are driving me crazy! > Error: Name conflict detected by export"

Explanation: A package is trying to export a symbol that's already defined. You probably tried to use a function only to discover you'd forgotten to load its file. The failed attempt at using the function caused its symbol to be interned. So now when you try to load the file, you get a conflict. Unfortunately, understanding and correcting the code which caused the export problem doesn't make those nasty error messages go away. That symbol is still interned where it shouldn't be.

Conflicts can also occur if your code happens to use a symbol that later gets exported from some package. For example, if a file containing `(let ((foo 2)))` is loaded, and then later, some file is loaded that which exports `foo`, you'll get a conflict error. The fact that `foo` is a local variable has nothing to do with it, nor does whether or not the code was executed. The reader interns the symbol when the file is loaded.

One particularly insidious way export conflicts can happen is if you use `(require 'foo)`. Often the file "`foo.lisp`" will try to export the symbol `foo`, but you've just interned `foo` in your package when calling `require`. Kind of a chicken and egg problem.

Solution: No, you don't have to restart LISP to stop those error messages. You can use `unintern` to remove the symbol from a package, but make sure you remove it from the right package. Conflicts arising from accidental name overlaps can sometimes be avoided if you load in all the package you need up front (or at least import their external symbols), but this is often impractical. It's best to simply not use symbols that other packages export (consult your local seer for help). The `require` problem can be avoided by always passing keywords or strings to `require` - `(require :foo)` or `(require "foo")`

"defvar doesn't initialize the variable"

Explanation: `defvar` only initializes the variable if it doesn't already have a value. So, re-evaluating a `defvar` after changing its initializer doesn't have any effect.

Solution: You probably shouldn't be using global variables anyway, but if you insist, `defparameter` always evaluates its initializer and sets the variable's value.

Appendix B - Classic MCL Blunders

"All the drawing in my view is shifted down and to the right."

Explanation: Is the amount your drawing is offset the same as the amount your view's top left corner is offset from its container? If so, the problem is that views are drawn focused to themselves while dialog items (and other simple views) are drawn focused to their container. For example, if you try to frame a view by drawing the rectangle defined by the values returned from `view-corners`, it's using the wrong coordinates.

Solution: For the framing problem mentioned above, you should use the rectangle defined by `#@(0 0)` and the view's size. In general, you need to understand the subtle differences between views and simple views. oou provides some additional methods for views and simple views to help out. See `focused-corners` and `focusing-view`.

"I can't change my dialog-item-action without recreating the entire dialog."

Explanation: You're probably using a closure, `#'my-action`, as your dialog item action. Redefining the function after the barn door is closed doesn't affect the original closure. Your dialog item action is a closure which captures the original state of the function it closes off.

Solution: Pass in a symbol, `'my-action`, for the `:dialog-item-action` initarg. Funcall works with symbols as well as functions. When you pass `funcall` a symbol, its current function definition is used.

"This works from the Listener, but not from my dialog item action."

Explanation: There's so much to say about this one. Volumes could be written. Basically, you need to grok how MCL handles events. This comes with reading the Events chapter about 500 times and banging your head against the wall about a million times. Before reading the Events chapter in the MCL documentation, you need to have already read and fully understood the Events chapter in the MCL documentation.

What's probably happening is that your dialog item action is sitting around waiting for something to happen, that won't happen until it stops sitting around waiting for it to happen. *A watched dialog item action never boils*. Do you understand?

Enough Zen style explanation. There are two threads of execution going on in MCL. One handles events and one executes forms typed into the Listener. The Listener thread periodically gives time to the event thread, that's why you can pull down menus and click on dialog items while your program is running.

Dialog item actions and menu item actions are executed in the event thread. While one of these actions is executing, no other event handling can take place. So if your action doesn't exit, you block the event handling process.

Functions like `set-dialog-item-text`, do not directly make changes to the screen, they generate update events that will eventually be handled by the event thread. So if you call `set-dialog-item-text` from a dialog item action, it won't actually happen until after you exit.

There are a myriad of variations on this problem. Some quite baffling and difficult to explain. Fortunately, once you figure out the problem is a blocked event thread, it's usually not hard to fix.

Solution: Your dialog item action needs to use . As a rule of thumb, dialog item actions and menu actions should do something quick and exit. If they need to do something more complex, they should use `eval-enqueue` to have the listener thread execute it. (yes, I know you were told that `eval` was evil, but this is one of those cases your LISP instructor said rarely arises) Since `eval-enqueue` exits immediately, your dialog item action exits and frees up the event thread.

"This ToolBox call acts like its Boolean argument is always true."

Explanation: In days of old, you had to pass Boolean values to the traps as `-1` or `0`. Now you should pass `t` or `nil` and MCL's trap calling mechanism will take care of pushing `-1` or `0` onto the stack. Since both `-1` and `0`, are non-`nil`, they both are passed as true.

Solution: Don't do that! Pass `t` or `nil` for Boolean values.

"I can't set this Boolean record field to false."

Explanation: This is similar to the problem described above. However, it is not new to MCL 2.0. The `rset` macro takes a `nil` or non-`nil` value and fills in the field with `-1` or `0`. This makes sense considering that `rref` on a Boolean field returns `t` or `nil`, not `-1` or `0`.

Solution: Don't do that! Pass `t` or `nil` for Boolean values.

"Why is MCL crashing? And what's this `#<ccl::ivector subtype 0 length 7981958 #xF396D9>` thing?"

Explanation: You're probably using a variable with dynamic extent after it's extent has expired. If you're unfamiliar with the term extent, see Steele's discussion of scope and extent. Try issuing:

```
(rlet ((p :integer)) (print p) p)
-> #<A Mac Non-zone Pointer #xF396E8>                    ;what we expect
-> #<ccl::ivector subtype 0 length 7981958 #xF396D9> ;returned an expired object
```

But, make sure to save any work first because on occasion, just printing an expired object can crash the system. In this example, the `rlet` returns an expired object which the Listener then tries to print.

Solution: Don't do that! Be careful when using variables with dynamic extent.

"Double clicking MCL documents in the Finder launches a saved application instead of MCL."

Explanation: The saved MCL application has the same four character signature as MCL ("`MCL2`"). When you open a document from the Finder, the Finder examines the creator of the document and looks for an application with that same signature. If there are multiple applications with the same signature, the Finder chooses one (the one you don't want).

A similar problem occurs if you have multiple copies of MCL on your system.

Solution: When saving applications from MCL, give them a unique signature. You can change the signature of saved applications using ResEdit. If you have multiple copies of MCL on your system, changing their signatures won't work. You can use the freeware program, Save A BNDL, to reinstall the BNDL information of a particular copy of MCL.

Appendix C - LISP & MCL Coding Style

Below is a list of coding conventions and advice I tell to new MCL programmers. Of course, no one listens, but at least I get to say "I told you so" when they get burned. These conventions shouldn't adversely affect code efficiency (time or space), but they should make it more legible and more reliable. I try to adhere to them myself, so reading them may make the oou source easier to digest.

• Use mnemonic variable names. It's an excellent way to document your code. If you really hate typing, learn to use search and replace. Often I have a comment at the start of a function describing what its intended to do. Occasionally you'll see an inline comment or two, but the nature of LISP code (function calls whose arguments are function calls…) doesn't always lend itself to inline comments.

• The solution to all those compiler warnings is not to turn off the various compiler warning flags. Those warnings are your buddy, they're telling you, "Hey knucklehead! fix your code."

• Don't use global variables! If you find you really need them, redesign your code. If you give into temptation, use `defvar` or `defparameter` to define them.

• Use `when` and `unless` for conditionals with no else clause. They are more legible than `if` forms. You don't have to go hunting for the else clause. You don't need `progn` to group multiple statements. If you see an `(if foo …)` in my code, you can count on there being both an *if* and an *else* clause. I use `cond` for multiway decisions and sometimes to avoid using `progn` with `if`.

• Use `case` and `typecase` when applicable. These are very legible, but often overlooked, LISP constructs. For cheap error checking, use their `etypcase` and `ecase` versions.

• Learn the sequence functions. These functions, especially the power of their keyword arguments, are often overlooked by novice LISP coders (probably because their LISP instructor said they weren't allowed to use them). Almost anything can be accomplished using the right sequence function and appropriate keywords. If you find yourself writing a nasty dolist, check though the sequence function section of Steele.

• Use `declare`. It can really optimize your code. For instance, declaring a variable to be of type `fixnum` can make a big difference in a loop. It's hard to tell what will make a difference and what won't. Experiment, use `time` and `disassemble` to see if it really makes a difference. Let me know of any discoveries you make. I really ought to use some `declare`s in my code. One warning: Lying to declare is an offense punishable by System Error. Make sure your variables really conform to their declarations. One especially unforgiving declaration is `dynamic-extent`. Declaring a variable to be `dynamic-extent` can help you avoid accumulating garbage, but be sure that the variable is not used after its allocating block has exited.

• Don't use or , instead use , , , . You can get burned using `rref` and `rset` because they use the default record storage type which may or may not be how your particular record is allocated. And yes, this requires knowing whether your record is allocated as a pointer or a handle, but if you don't know, you are in serious trouble. Along similar lines, don't use the `with-pointers` macro which is for dereferencing things which may be pointers or handles. If they're pointers, they don't need dereferencing. If they're handles, use `with-dereferenced-handles`. If you don't know which they are, you're in the wrong business.

• Use to allocate stack space instead of . In MCL 2.0 final you can use `rlet` to allocate and initialize all mactypes, not just records. The `rlet` code is more legible, terser, and ends up expanding into the same thing anyway.

```
(rlet               ((refNum_p              :integer              0))              …)
   vs.
(%stack-block ((refNum_p 2)) (%put-word refNum_p 0) …)
```

My prediction is that `rlet` is destined to become the general purpose MCL stack allocation form of choice. Someday people won't

remember what the 'r' stands for, just like few people remember what the 'n' in the destructive functions (nreverse, nbutlast, …) means[3].

• Use to locally allocate macptrs.

• Use a naming convention to help avoid confusion between pointers and what they point to. For example, if you allocate space for a var integer parameter, don't call the pointer my-int or later on you'll accidentally use my-int as an integer, forgetting to use (%get-byte my-int) instead. Make sure the variable name indicates it's a pointer. Try my-int-ptr or my-int_p. Personally, I usually use an _p suffix to indicate pointers and an _h to indicate handles. This is a carryover from my C coding style. In MCL, I sometimes get sloppy in my record variable names because, in MCL, all record variables are pointers to records, something which isn't true in C.

3According to Chris Riesbeck, the explanation of the n prefix dates way back to the origins of LISP and the primordial destructive function, conc. conc accepted two lists and destructively appended them. conc begat nconc which took n lists and destructively appended them. The rest is history.

Appendix D - Designing Code for MCL

Designing Specialized Dialog Items and Views

If your dialog item is getting really complex it may be better to instead, create a view containing other views or dialog items. That way you can design and test the beast in simple pieces. To get the thing to really act as a unit, you can augment the initialize-instance method of the outer view with an after method that creates and adds the various subviews. For a barely plausible illustration of this, see `hairy-view.lisp` in the `examples` folder.

Another problem you may run into with your view classes is that sometimes they need to be erased before re-drawing. For example, view classes that don't erase as part of their normal drawing need to do an erase when being resized to larger size, otherwise they will re-draw superimposed over their old image. The standard `set-view-size` method does invalidate a view, but it doesn't add it to the window's erase region. Oou provides two methods to address this problem, `erase-view` and `erase-corners`. All these methods do is add a rectangle to a window's erase region. To solve the resizing problem, call `erase-view` in a `set-view-size` after method.

Designing View Mixins

When designing a mixin class for simple views (or dialog items), with a little more forethought, you can usually make it work for simple views and views. This can be a big win, especially if your dialog items get complex and you decide to turn them into views as was previously suggested.

The main consideration in designing mixins for both simple views and views is that simple views are normally drawn focused on their container while views are focused on themselves. Also, simple views get clicks in the coordinate system of their container while views get them in their own coordinate system. You can avoid writing separate code to handle simple views and views, by using `focused-corners` and `focusing-view` as alternatives to `view-corners` and `view-container`. These alternate methods are described in `simple-view-ce`. For examples of this type of design, see the source to the various simple view mixins in oou.

Initarg Conflicts

If you're designing mixins or classes to be used with oou mixins, you may want to check for potential initarg conflicts by looking in the initarg section of the index. Initarg conflicts are less likely to be detected by the compiler than method name conflicts. If you reuse a method name, unless it's lambda list is congruent to the original, the compiler will issue a warning. No such checking is done for initargs.

References & Suggested Reading

Apple Computer, "Macintosh Common LISP 2.0 Reference - Draft." 1991, Apple Computer.
If you've got a legal copy of MCL, you should have a copy of this one. I haven't seen the final version yet. The draft has plenty of errors and omissions. MCL's Apropos tool goes a long way toward making up for the deficiencies.

Apple Computer, "Inside Macintosh" - Volumes I-VI. 1985, Addison Wesley.
Essential references for Mac programming. Rumor is they're going to be replaced in late 1992 by a new and improved set.

Card, Orson Scott, "Ender's Game." 1991, Tom Doherty and Associates.
Very entertaining. Even people who don't normally like science fiction will enjoy this one.

Chernicoff, Stephen, "Macintosh Revealed" - Volumes I-IV. 1985, Hayden Books.
Digestible alternative to Inside Macintosh (or at least to many of the essential parts). Volumes I and II cover most of the basics. Volume IV has a good introduction to color QuickDraw. Volume III covers advanced topics.

Engber, Michael S., *The Sound Manager with LISP*. MacTutor, March 1991, pp. 84-89.
An informative and well written article, if I do say so myself. It covers the basics of ToolBox access and illustrates them by using the Sound Manager. It was written in the days of MACL 1.32, so parts are dated.

Irving, John, "The Cider House Rules." 1985, William Morrow and Company.
I haven't actually read this one, but my wife assures me it's quite excellent.

Keene, Sonya E., "Object-Oriented Programming in Common LISP." 1989, Addison Wesley.
A very complete and digestible introduction to CLOS. It can be read straight through (if you ignore that extended example on the lock class).

Steele, Guy L., "Common LISP - The Language" 2nd edition. 1990, Digital Press.
Comprehensive, precise, essential, very dense. This is one of those books whose prerequisite is a solid understanding of the subject it covers. Not a tutorial.

Steele, Guy L. and Harbison, Samuel P., "C - A Reference Manual" 2nd edition. 1987, Prentice Hall.
Steele does it again, a comprehensive and precise description of C, including ANSI C, thorough enough for C compiler writers to use. Mixed in with the language issues is a lot of practical computer science.

Sherwood T.K. and Wilcox F.C., Sabotage of Gasoline Engines. 1946, Office of Scientific Research .
A definite must.

Wilensky, Robert, "Common LISPcraft." 1984, WW Norton and Company.
If you already know something about programming and you're looking for a book you can sit down, read, and come away with the impression, albeit mistaken, that you know something about LISP, this is it. It has good, readable, explanations of the fundamentals. Appendix A is a good Common LISP reference, although not as encyclopedic as Steele. When you outgrow Wilensky, you'll be ready for Steele.

Index