# Appendix C
# Clean 0.8 I/O Library

This section describes the Clean 0.8 I/O library converted to 1.0 syntax. A new I/O library which allows more locality of state within interactive components is being developed.

## C I/O library

### C.1 General operations on the IOState (deltaEventIO)

```
definition module deltaEventIO
import deltaIOSystem

::   * EVENTS
```

The event stream.

```
::   * IOState * s
```

The environment on which all event I/O functions operate.

```
::   InitialIO *s  :==  [s ->  (IOState s) -> (s, IOState s)]
```

The I/O functions that will be evaluated before starting the interaction.

```
OpenEvents    ::   !* WORLD   ->   (!EVENTS, !* WORLD)
```

Retrieves the event stream from the world. Attempts to retrieve the event stream from the world more than once without putting it back result in a run-time error.

```
CloseEvents   ::   !EVENTS  !* WORLD   ->   * WORLD
```

Puts the event stream back into the world.

```
StartIO       ::   !(IOSystem s (IOState s)) !s !(InitialIO s)  !EVENTS  ->   (!s, !EVENTS)
```

Starts a new interaction. The initial state of this interaction is composed of the program state (s) and the initial I/O state, which consists of the devices participating in the interaction (defined in the `IOSystem` argument) and the event stream (`EVENTS`). Of each device in the initial `IOSystem` only the first occurrence is taken into account. The program state must be unique. Before starting the interaction the `InitialIO` functions are evaluated. At the end of the interaction `StartIO` returns the final program state and the resulting event stream, from which the events that have been handled during the interaction have been removed.

```
NestIO        ::   !(IOSystem s (IOState s)) !s !(InitialIO s) !(IOState t) -> (!s, !IOState t)
```

Starts a nested interaction. It replaces the current interaction (as specified by the `IOState` argument) with a completely new one (as specified by the `IOSystem` argument). It hides the devices of the current interaction (if any) and fills a new `IOState` with the devices that are specified in the `IOSystem` and with the event stream of the old `IOState`. The program state argument (s) serves as initial program state of the new interaction. Before starting the nested interaction the `InitialIO` functions are evaluated. `NestIO` returns the final program state of the new interaction and the original `IOState`, such that the original interaction reappears.

```
QuitIO        ::   !(IOState s)  ->   IOState s
```

Closes all devices that are held in the `IOState` argument. The resulting `IOState` will cause `StartIO` or `NestIO` to terminate.

```
ChangeIOState ::   ![(IOState s) -> (IOState s)] !(IOState s)   ->   IOState s
```

Applies all functions in its first argument in consecutive order to the second (`IOState`) argument.

## C.2 Definition of the I/O system (deltaIOSystem)

```
definition module deltaIOSystem
from deltaPicture import Picture, Rectangle, DrawFunction

::   * DialogState * s * io
::   DialogInfo

::   IOSystem * s * io      :==   [DeviceSystem s io]
::   DeviceSystem * s * io  =     TimerSystem   [TimerDef  s io]
                            |     MenuSystem    [MenuDef   s io]
                            |     WindowSystem  [WindowDef s io]
                            |     DialogSystem  [DialogDef s io]
```

The timer device responds only to timer events. A timer event occurs as soon as a certain `TimerInterval` has expired since the last time it was ‘sampled’. A `TimerInterval` is defined as a number of ticks. A macro `TicksPerSecond` is defined in `deltaTimer.dcl`. The timer event causes one of the programmer-defined `TimerFunctions` to be evaluated. The `TimerState` argument of a `TimerFunction` indicates how many times the `TimerInterval` has passed since the last timer event for that timer. When the `TimerInterval` of one of the timers is smaller than 1 the corresponding `TimerFunction` is called as often as possible.

```
::   TimerDef * s * io   =  Timer TimerId SelectState TimerInterval (TimerFunction s io)
::   TimerId                :==   Int
::   TimerInterval          :==   Int
::   TimerFunction * s * io :==   TimerState -> ( s -> * ( io -> (s, io) ) )
::   TimerState             :==   Int
```

The menu device consists of several `PullDownMenus`. `PullDownMenus` are logically grouped into a menu bar, in the same order as they are specified. They are selected by pressing the mouse on their `MenuTitle` in the menu bar. Menus contain `MenuElements`. The corresponding `MenuFunctions` are executed when these elements are selected.

```
::   MenuDef * s * io    =  PullDownMenu MenuId MenuTitle SelectState [MenuElement s io]
::   MenuElement * s * io
   =  MenuItem       MenuItemId ItemTitle KeyShortcut SelectState           (MenuFunction s io)
   |  CheckMenuItem  MenuItemId ItemTitle KeyShortcut SelectState MarkState (MenuFunction s io)
   |  SubMenuItem    MenuId     ItemTitle SelectState [MenuElement s io]
   |  MenuItemGroup  MenuItemGroupId                  [MenuElement s io]
   |  MenuRadioItems MenuItemId                       [RadioElement s io]
   |  MenuSeparator
::   RadioElement * s * io
   =  MenuRadioItem MenuItemId ItemTitle KeyShortcut SelectState (MenuFunction s io)
::   MenuFunction * s * io    :==   s -> *(io -> (s, io))
::   MenuId                   :==   Int
::   MenuTitle                :==   String
::   MenuItemId               :==   Int
::   MenuItemGroupId          :==   Int
::   KeyShortcut              =     Key KeyCode | NoKey
```

The window device consists of several `ScrollWindows` or `FixedWindows`. A `ScrollWindow` is defined by the following arguments:

- `WindowId`: the number by which the programmer refers to the window.
- `WindowPos`: the position of the upper-left corner of the window.
- `WindowTitle`: the title of the window.
- `ScrollBarDefs`: the horizontal and vertical scroll bars (in that order).
- `PictureDomain`: the range of the drawing operations in the window.
- `MinimumWindowSize`: the smallest dimensions of the window.
- `InitialWindowSize`: the initial dimensions of the window.
- `UpdateFunction`: the function to redraw parts (`UpdateArea`) of the window.
- An attribute list that may contain the following window attributes:
  - `Activate`: the way to respond to activation of the window.
  - `Deactivate`: the way to respond to de-activation of the window.
  - `GoAway`: the way to respond when the window is closed.
  - `Keyboard`: the way the window responds to keyboard input.
  - `Mouse`: the way the window responds to mouse events.
  - `Cursor`: the shape of the cursor (mouse pointer) inside the window.
  - `StandByWindow`: when this attribute is present the window will be a so-called stand-by window.

A `FixedWindow` has a fixed size, which is defined by its `PictureDomain`. Therefore it has no scroll bars and no size parameters. When the `PictureDomain` of a `FixedWindow` is or becomes greater than one of the screen's dimensions it becomes a `ScrollWindow`.

```
::   WindowDef * s * io
   = ScrollWindow WindowId WindowPos WindowTitle  ScrollBarDef ScrollBarDef PictureDomain
                MinimumWindowSize InitialWindowSize (UpdateFunction s) [WindowAttribute s io]
   | FixedWindow WindowId WindowPos WindowTitle PictureDomain
                                                     (UpdateFunction s) [WindowAttribute s io]

::   WindowId               :==   Int
::   WindowPos              :==   (!Int, !Int)
::   WindowTitle            :==   String
::   ScrollBarDef           =     ScrollBar ThumbValue ScrollValue
::   ThumbValue             =     Thumb Int
::   ScrollValue            =     Scroll Int
::   MinimumWindowSize      :==   (!Int, !Int)
::   InitialWindowSize      :==   (!Int, !Int)
::   UpdateFunction * s     :==   UpdateArea -> (s -> (s, [DrawFunction]))
::   UpdateArea             :==   [Rectangle]

::   WindowAttribute * s * io =  Activate      (WindowFunction s io)
                              |  Deactivate    (WindowFunction s io)
                              |  GoAway        (WindowFunction s io)
                              |  Mouse         SelectState (MouseFunction s io)
                              |  Keyboard      SelectState (KeyboardFunction s io)
                              |  Cursor        CursorShape
                              |  StandByWindow

::   WindowFunction      * s * io :==  s -> *(io -> (s, io))
::   MouseFunction       * s * io :==  MouseState -> (s -> *(io -> (s, io)))
::   KeyboardFunction    * s * io :==  KeyboardState -> (s -> *(io -> (s, io)))

::   CursorShape   =  StandardCursor   |  BusyCursor      |  IBeamCursor   |
                      CrossCursor      |  FatCrossCursor  |  ArrowCursor   |
                      HiddenCursor
```

The dialog device: dialogs given in the initial I/O system will be opened as modeless dialogs. Use the `Open(Modal)Dialog` function (`deltaDialog.icl`) to open dialogs during the interaction. `PropertyDialogs` are special dialogs that can only be modeless. They are used to change certain properties (default values, preferences etc.) of a program. They have two predefined buttons: the `Set` and the `Reset` buttons. A `CommandDialog` can be any kind of dialog. A `PropertyDialog` is defined by the following attributes:

- `DialogId`: a number by which the programmer can refer to the dialog.
- `DialogTitle`: the title of the dialog.
- A list of attributes that may contain the following dialog attributes:
  - `DialogPos`: the position of the dialog on the screen.
  - `DialogSize`: the size of the dialog.

  - `DialogMargin`: the horizontal and vertical margins between the borders of the dialog and the items.
  - `ItemSpace`: the horizontal and vertical space between the items of the dialog.
  - `StandByDialog`: when this attribute is present the dialog will be a so-called stand-by dialog.

  When none of these attributes is specified the dialog is centred on the screen, a size is chosen such that all items fit in the dialog and save default margins, and item spaces are chosen.
- `SetFunction`/`ResetFunction`: the button function for the set/reset button.
- A list of `DialogItems`: other items such as `CheckBoxes`, `Controls` etc.

A `CommandDialog` also has an id, a title, a position, a size and a list of `DialogItems`. Furthermore it has the following attribute:
- `DialogItemId`: the item id of the default button.

In the `AboutDialog`, information about the application (version, authors etc.) can be presented. It may also contain a button that should provide a help facility. The first `AboutDialog` that is encountered in the initial dialog device becomes the `AboutDialog` of the application. Attempts to open `AboutDialogs` with `Open(Modal)Dialog` are ignored. The `AboutDialog` will be accessible by the user during the interaction in a system-dependent way.

```
::   DialogDef * s * io
   = PropertyDialog DialogId DialogTitle [DialogAttribute]
                               (SetFunction s io) (ResetFunction s io) [DialogItem s io]
   | CommandDialog  DialogId DialogTitle [DialogAttribute] DialogItemId [DialogItem s io]
   | AboutDialog ApplicationName PictureDomain [DrawFunction] (AboutHelpDef s io)

::   DialogId           :==  Int
::   DialogTitle        :==  String
::   DialogAttribute     =  DialogPos  Measure Measure
                         |  DialogSize Measure Measure
                         |  DialogMargin  Measure Measure
                         |  ItemSpace  Measure Measure
                         |  StandByDialog
::   Measure             =  MM Real  |  Inch Real  |  Pixel Int
::   ApplicationName    :==  String
::   AboutHelpDef  * s * io   =    AboutHelp ItemTitle (AboutHelpFunction s io)
                              |    NoHelp
::   AboutHelpFunction * s * io  :==   s -> *(io -> (s, io))
```

A `DialogItem` can be a final button (`DialogButton`), a final button with a user-defined look (`DialogIconButton`), an unchangeable piece of text (`StaticText`), a changeable piece of text (`DynamicText`), an editable text field (`EditText`), a pop-up menu (`DialogPopUp`), a group of `RadioButtons`, a group of `CheckBoxes`, or a user-defined `Control`. The `ItemPos` specifies the position of the item relative to the other items. When the `ItemPos` is `Left`, `Center` or `Right` the item is placed left-aligned, centred or right-aligned, respectively, beneath all other items.

```
::   DialogItem * s * io
   = DialogButton     DialogItemId ItemPos ItemTitle SelectState (ButtonFunction s io)
   | DialogIconButton DialogItemId ItemPos PictureDomain
                                           IconLook SelectState (ButtonFunction s io)
   | StaticText   DialogItemId ItemPos String
   | DynamicText  DialogItemId ItemPos TextWidth String
   | EditText     DialogItemId ItemPos TextWidth NrEditLines String
   | DialogPopUp  DialogItemId ItemPos SelectState   DialogItemId [RadioItemDef s io]
   | RadioButtons DialogItemId ItemPos RowsOrColumns DialogItemId [RadioItemDef s io]
   | CheckBoxes   DialogItemId ItemPos RowsOrColumns [CheckBoxDef s io]
   | Control DialogItemId ItemPos PictureDomain SelectState ControlState
                                                ControlLook ControlFeel (DialogFunction s io)

::   DialogItemId  :==   Int
::   ItemPos        =    Left | Center | Right | RightTo DialogItemId |
                         Below DialogItemId | XOffset DialogItemId Measure |
                         YOffset DialogItemId Measure | XY Measure Measure |
                         ItemBox Int Int Int Int
::   IconLook      :==   SelectState -> [DrawFunction]
::   TextWidth     :==   Measure
::   NrEditLines   :==   Int
::   RowsOrColumns =     Rows Int  |  Columns Int

::   RadioItemDef * s * io = RadioItem DialogItemId ItemTitle SelectState (DialogFunction s io)
::   CheckBoxDef * s * io
                    = CheckBox DialogItemId ItemTitle SelectState MarkState (DialogFunction s io)
```

Attributes of a user-defined `Control`: the `ControlState` can be a boolean, an integer, a real or a string. The look of the control is defined by the DrawFunctions (see deltaPicture.dcl) returned by the ControlLook function. The ControlFeel defines the way to respond to mouse clicks inside the control's picture domain.

```
::   ControlState  =     BoolCS Bool         |  IntCS Int        |
                         RealCS Real         |  StringCS String  |
                         ListCS [ControlState] |  PairCS ControlState ControlState
::   ControlLook   :==   SelectState -> ControlState -> [DrawFunction]
::   ControlFeel   :==   MouseState  -> ControlState -> (ControlState,[DrawFunction])
```

Types of the several dialog item functions:

```
::   SetFunction      * s * io   :== ButtonFunction s io
::   ResetFunction    * s * io   :== ButtonFunction s io
::   DialogFunction   * s * io   :== DialogInfo -> (DialogState s io) -> DialogState s io
::   ButtonFunction   * s * io   :== DialogInfo -> (s -> *(io -> (s, io)))
```

A notice is a simple, modal dialog containing only text and final buttons. It can be used to inform the user about un-usual or dangerous situations. A notice is defined by the following attributes:

- A list of Strings: each string is a line of the message of the notice.
- A NoticeButtonDef: the default button of the notice.
- A list of NoticeButtonDefs: the other buttons of the notice.

```
::   NoticeDef          =     Notice [String] NoticeButtonDef [NoticeButtonDef]
::   NoticeButtonDef    =     NoticeButton NoticeButtonId ItemTitle
::   NoticeButtonId     :==   Int
```

Keyboard input: a window may respond to keyboard events. Each such event causes the KeyboardFunction to be evaluated. For certain special keys constants of type KeyCode are provided in deltaSystem.dcl.

```
::   KeyboardState    :==   (!KeyCode, !KeyState, !Modifiers)
::   KeyCode          :==   Char
::   KeyState         =     KeyUp    |  KeyDown  |  KeyStillDown
```

Mouse input: windows and controls may respond to mouse events. Each such event causes the MouseFunction to be evaluated.

```
:: MouseState          :==   (!MousePosition, !ButtonState, !Modifiers)
:: MousePosition       :==   (!Int, !Int)
:: ButtonState = ButtonUp | ButtonDown | ButtonDoubleDown | ButtonTripleDown | ButtonStillDown
```

For each modifier or meta-key (Shift, Option (Alternate), Command, Control) a boolean in Modifiers indicates whether it was pressed (TRUE) or not (FALSE). On keyboards that have no Command key both the third and the fourth boolean in Modifiers become TRUE when Control is pressed.

```
::   Modifiers        :==   (!Bool, !Bool, !Bool, !Bool)
```

Other common types:

```
::   ItemTitle        :==   String
::   SelectState      =     Able | Unable
::   MarkState        =     Mark | NoMark
::   PictureDomain    :==   Rectangle
```

## C.3 Operations on the timer device (deltaTimer)

```
definition module deltaTimer
import deltaIOSystem, deltaEventIO

TicksPerSecond   :==   …      // system dependent

::   CurrentTime :== (!Int, !Int, !Int)
```

(hours (0–23), minutes (0–59), seconds (0–59))

```
::   CurrentDate :== (!Int, !Int, !Int, !Int)
```

(year, month (1–12), day (1–31), day of week (1–7, Sunday=1, Saturday=7))

```
OpenTimer  ::   !(TimerDef s io)   !(IOState s)  ->   IOState s
CloseTimer ::   !TimerId           !(IOState s)  ->   IOState s
```

Open (install) a new timer, close (remove) an existing timer.

```
EnableTimer   ::   !TimerId   !(IOState s)  ->   IOState s
DisableTimer  ::   !TimerId   !(IOState s)  ->   IOState s
ChangeTimerFunction:: !TimerId   !(TimerFunction s (IOState s)) !(IOState s)  ->   IOState s
SetTimerInterval      ::   !TimerId   !TimerInterval   !(IOState s)  ->   IOState s
GetTimerBlinkInterval ::                           !(IOState s)  ->(!TimerInterval, !IOState s)
```

Enable, disable, change the `TimerFunction` and `TimerInterval` of a `Timer`. `GetTimerBlinkInterval` returns the number of ticks that should pass between a blink of the cursor (also called caret time).

```
Wait    ::   !TimerInterval   x     -> x
UWait   ::   !TimerInterval   * x   -> * x
```

Delay the evaluation of the second argument for a certain `TimerInterval`.

```
GetCurrentTime   ::   !(IOState s)  ->   (!CurrentTime,    !IOState s)
GetCurrentDate   ::   !(IOState s)  ->   (!CurrentDate,    !IOState s)
```

`GetCurrentTime` / `GetCurrentDate` return the current time and date.

## C.4 Operations on menus (deltaMenu)

```
definition module deltaMenu
import deltaIOSystem, deltaEventIO
```

Menu operations on unknown `MenuIds`/`MenuItemIds` are ignored.

```
EnableMenuSystem    ::   !(IOState s)  ->   IOState s
DisableMenuSystem   ::   !(IOState s)  ->   IOState s
```

Enable and disable the complete menu system. Enabling the menu system will make the previously enabled menus and menu items selectable again. Operations on a disabled menu system take effect when the menu system is re-enabled.

```
EnableMenus   ::   ![MenuId] !(IOState s)   ->   IOState s
DisableMenus  ::   ![MenuId] !(IOState s)   ->   IOState s
```

Enable, disable menus. Disabling a menu causes its contents to be unselectable. Enabling a disabled menu with partially selectable contents causes the previously selectable items to become selectable again.

```
InsertMenuItems:: !MenuItemGroupId !Int ![MenuElement s (IOState s)] !(IOState s) -> IOState s
AppendMenuItems:: !MenuItemGroupId !Int ![MenuElement s (IOState s)] !(IOState s) -> IOState s
RemoveMenuItems       ::   ![MenuItemId]           !(IOState s)  ->   IOState s
RemoveMenuGroupItems  ::   !MenuItemGroupId ![Int]  !(IOState s)  ->   IOState s
```

Addition, removal of `MenuItems` in `MenuItemGroups`. `InsertMenuItems` inserts menu items before the item with the specified index, `AppendMenuItems` inserts them after that item. Items are numbered starting from one. Indices smaller than one or greater than the number of elements cause the elements to be inserted, respectively, before the first and after the last item in the group. Only (`Check`)`MenuItems` and `MenuSeparators` are added to `MenuItemGroups`. `RemoveMenuItems` only works on items that are in a `MenuItemGroup`. `RemoveMenuGroupItems` removes the items with the specified indices (counting from one) from the specified `MenuItemGroup`.

```
SelectMenuRadioItem   ::   !MenuItemId !(IOState s)    ->   IOState s
```

Select a `MenuRadioItem`: the mark will move from the currently selected item in the group to the item with the specified id.

```
EnableMenuItems       ::   ![MenuItemId]            !(IOState s)  ->   IOState s
DisableMenuItems      ::   ![MenuItemId]            !(IOState s)  ->   IOState s
MarkMenuItems         ::   ![MenuItemId]            !(IOState s)  ->   IOState s
UnmarkMenuItems       ::   ![MenuItemId]            !(IOState s)  ->   IOState s
ChangeMenuItemTitles  ::   ![(MenuItemId, String)]  !(IOState s)  ->   IOState s
ChangeMenuItemFunctions:: ![(MenuItemId, MenuFunction s (IOState s))] !(IOState s)->IOState s
```

Enable, disable, mark, unmark, change titles and functions of `MenuElements`.

## C.5 Operations on windows (deltaWindow)

```
definition module deltaWindow
import deltaIOSystem, deltaEventIO, deltaPicture
```

Functions applied on non-existing windows or unknown `WindowIds` are ignored. Functions that operate on the active (or frontmost) window have the same effect as those operating on a (list of) `WindowId`(s) but are (slightly) faster.

```
OpenWindows        ::   ![WindowDef s (IOState s)] !(IOState s)   ->   IOState s
```

The windows are opened in the same order as specified in the list of `WindowDefinitions`. If one of these windows has the same `WindowId` as a window that is already open, then this window is not opened. Each new window is always the active window and is placed in front of all existing windows.

```
CloseWindows       ::   ![WindowId] !(IOState s) ->   IOState s
```

The windows are closed in the same order as specified in the list.

```
CloseActiveWindow  ::   !(IOState s)  ->   IOState s
```

The active window is closed.

```
GetActiveWindow    ::   !(IOState s)  ->   (!Bool, !WindowId, !IOState s)
```

Returns the id of the active window. If there is no active window the boolean will be `FALSE`.

```
ActivateWindow     ::   !WindowId !(IOState s)   ->   IOState s
```

Activates the indicated window. No effect, if the window is already active.

```
ChangeUpdateFunction        ::   !WindowId  !(UpdateFunction s) !(IOState s)-> IOState s
ChangeActiveUpdateFunction  ::              !(UpdateFunction s) !(IOState s) -> IOState s
```

Change the update function of the indicated window.

```
ChangeWindowTitle        ::   !WindowId  !WindowTitle !(IOState s)   ->   IOState s
ChangeActiveWindowTitle  ::              !WindowTitle !(IOState s)   ->   IOState s
```

Change the window title of the indicated window.

```
ChangeWindowCursor       ::   !WindowId  !CursorShape !(IOState s)   ->   IOState s
ChangeActiveWindowCursor ::              !CursorShape !(IOState s)   ->   IOState s
```

Change the local cursor shape of a window. When the mouse pointer moves over the content region of the window the cursor will take the indicated shape.

```
::   ScrollBarChange
   =    ChangeThumbs Int Int   // set new horizontal and vertical thumb values
   |    ChangeScrolls Int Int  // set new horizontal and vertical scroll values
   |    ChangeHThumb Int       // set new horizontal thumb value
   |    ChangeVThumb Int       // set new vertical thumb value
   |    ChangeHScroll Int      // set new horizontal scroll value
   |    ChangeVScroll Int      // set new vertical scroll value
   |    ChangeHBar Int Int     // set new horizontal thumb and scroll values
   |    ChangeVBar Int Int  `  // set new vertical thumb and scroll values
```

```
ChangeScrollBar       ::   !WindowId  !ScrollBarChange !s !(IOState s) -> (!s, !IOState s)
ChangeActiveScrollBar ::              !ScrollBarChange !s !(IOState s) -> (!s, !IOState s)
```

Change the scroll bar(s) of a window according to the `ScrollBarChange`.

```
ChangePictureDomain        ::   !WindowId  !PictureDomain !s !(IOState s) -> (!s, !IOState s)
ChangeActivePictureDomain  ::              !PictureDomain !s !(IOState s) -> (!s, !IOState s)
```

Change the `PictureDomain` of the indicated window. The settings of the scroll bars are automatically adjusted. Windows will be resized in cases where the size of the new `PictureDomain` is smaller than the current size of the window.

```
::   DrawInWindow !WindowId   ![DrawFunction] !(IOState s)   ->   IOState s
::   DrawInActiveWindow       ![DrawFunction] !(IOState s)   ->   IOState s
```

Apply the list of `DrawFunctions` (see `deltaPicture.dcl`) to the `Picture` of the window in the given order.

```
::   DrawInWindowFrame !WindowId !(UpdateFunction s) !s !(IOState s)
                                                          -> (!s, !IOState s)
::   DrawInActiveWindowFrame !(UpdateFunction s) !s !(IOState s)-> (!s,!IOState s)
```

The `UpdateFunction` has a list of visible rectangles as parameter. Using this list it is possible to return a list of drawing functions that only draw in the visible part of the window.

```
WindowGetFrame        ::   !WindowId  !(IOState s)  ->   (!PictureDomain, !IOState s)
ActiveWindowGetFrame  ::              !(IOState s)  ->   (!PictureDomain, !IOState s)
```

Return the visible part of the `Picture` of the indicated window in terms of the `PictureDomain`. Returns ((0, 0), (0, 0)), if the indicated window does not exist.

```
EnableKeyboard           ::   !WindowId  !(IOState s)  ->   IOState s
DisableKeyboard          ::   !WindowId  !(IOState s)  ->   IOState s
EnableActiveKeyboard     ::              !(IOState s)  ->   IOState s
DisableActiveKeyboard    ::              !(IOState s)  ->   IOState s
ChangeKeyboardFunction :: !WindowId  !(KeyboardFunction s (IOState s)) !(IOState s) -> IOState s
ChangeActiveKeyboardFunction ::      !(KeyboardFunction s (IOState s)) !(IOState s) -> IOState s
```

Enable, disable, change `KeyboardFunction` of a window.

```
EnableMouse           ::   !WindowId  !(IOState s)  ->   IOState s
DisableMouse          ::   !WindowId  !(IOState s)  ->   IOState s
EnableActiveMouse     ::              !(IOState s)  ->   IOState s
DisableActiveMouse    ::              !(IOState s)  ->   IOState s
ChangeMouseFunction   ::   !WindowId  !(MouseFunction s (IOState s)) !(IOState s) -> IOState s
ChangeActiveMouseFunction ::          !(MouseFunction s (IOState s)) !(IOState s) -> IOState s
```

Enable, disable, change `MouseFunction` of a window.

## C.6 Operations on dialogs (deltaDialog)

```
definition module deltaDialog
import deltaIOSystem, deltaEventIO
```

Functions applied on non-existent dialogs or unknown ids are ignored.

```
::   DialogChange s :== (DialogState s (IOState s)) -> (DialogState s (IOState s))
```

`OpenDialog` opens a `PropertyDialog` or `CommandDialog` as a modeless dialog.

```
OpenDialog :: !(DialogDef s (IOState s)) !(IOState s) -> IOState s
```

`OpenModalDialog` opens a `CommandDialog` as a modal dialog. The function terminates when the dialog is closed (by means of `Close(Active)Dialog`). Attempts to open property dialogs with this function are ignored.

```
OpenModalDialog ::  !(DialogDef s (IOState s)) !s !(IOState s) -> (!s, !IOState s)
```

`Close(Active)Dialog` closes the indicated dialog.

```
CloseDialog        ::   !DialogId  !(IOState s) -> IOState s
CloseActiveDialog  ::              !(IOState s) -> IOState s
```

`OpenNotice` opens a notice and returns the id of the selected notice button.

```
OpenNotice :: !NoticeDef !(IOState s) -> (!NoticeButtonId, !IOState s)
```

A `Beep` is the simplest kind of notice.

```
Beep ::  !(IOState s) -> IOState s
```

With the following functions the state of dialog items can be changed. They have no effect when an id is specified for an item for which the state change is invalid.

```
EnableDialogItems      :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
DisableDialogItems     :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
MarkCheckBoxes         :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
UnmarkCheckBoxes       :: ![DialogItemId]
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
SelectDialogRadioItem  :: !DialogItemId
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeEditText         :: !DialogItemId !String
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeDynamicText      :: !DialogItemId !String
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeIconLook         :: !DialogItemId !IconLook
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

Functions to change state, look and feel (behaviour) of `Controls`. When the id is not the id of a `Control` the functions have no effect.

```
ChangeControlState  ::   !DialogItemId !ControlState
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeControlLook   ::   !DialogItemId !ControlLook
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeControlFeel   ::   !DialogItemId !ControlFeel
                            !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

Functions to change the functions related to dialog items.

```
ChangeDialogFunction   ::   !DialogItemId !(DialogFunction s (IOState s))
                               !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeButtonFunction   ::   !DialogItemId !(ButtonFunction s (IOState s))
                               !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeSetFunction      ::   !(SetFunction s (IOState s))
                               !(DialogState s (IOState s)) -> DialogState s (IOState s)
ChangeResetFunction    ::   !(ResetFunction s (IOState s))
                               !(DialogState s (IOState s)) -> DialogState s (IOState s)
```

`Get(Active)DialogInfo` returns the current settings (`DialogInfo`) of the indicated dialog. The boolean indicates whether the indicated dialog is open. When it is `FALSE` a dummy `DialogInfo` is returned.

```
GetDialogInfo          ::   !DialogId  !(IOState s)  ->   (!Bool, !DialogInfo, !IOState s)
GetActiveDialogInfo    ::              !(IOState s)  ->   (!Bool, !DialogInfo, !IOState s)
```

`DialogStateGetDialogInfo` returns the `DialogInfo` corresponding to a `DialogState`.

```
DialogStateGetDialogInfo ::   !(DialogState s (IOState s))
                                        ->   (!DialogInfo, !DialogState s (IOState s))
```

The following functions return the current settings of certain dialog items (those that can be changed by the user). When the corresponding item does not exist a run-time error occurs. The id passed to `GetSelectedRadioItemId` must be the id of a `DialogPopUp` or a group of `RadioButtons`. The function `CheckBoxesMarked` returns the settings of a group of `CheckBoxes`. The id passed to it must be the id of such a group.

```
GetEditText              ::   !DialogItemId !DialogInfo  ->  String
GetSelectedRadioItemId   ::   !DialogItemId !DialogInfo  ->  DialogItemId
CheckBoxesMarked         ::   !DialogItemId !DialogInfo  ->  [(DialogItemId, Bool)]
CheckBoxMarked           ::   !DialogItemId !DialogInfo  ->  Bool
```

```
GetControlState          ::   !DialogItemId !DialogInfo  ->  ControlState
```

`ChangeDialog` can be used to modify open dialogs.

```
ChangeDialog  ::   !DialogId  ![DialogChange s]  !(IOState s)  ->   IOState s
```

## C.7 The file selector dialogs (deltaFileSelect)

```
definition module deltaFileSelect
import deltaFile, deltaEventIO
```

With the functions defined in this module, standard file selector dialogs can be opened, which provide a user-friendly way to select input or output files. The lay-out of these dialogs depends on the operating system.

```
SelectInputFile :: !Files !(IOState s)   -> (!Bool, !String, !Files, !IOState s)
```

Opens a dialog in which the user can traverse the file system to indicate an existing file. The Boolean result indicates whether the user has pressed the `Open` button (`TRUE`) or the `Cancel` button (`FALSE`). The `String` result is the complete path name of the selected file. If the `Cancel` button has been pressed, an empty string will be returned.

```
SelectOutputFile:: !String !String !Files !(IOState s) ->  (!Bool, !String, !Files, !IOState s)
```

Opens a dialog in which the user can specify the name of a file that should be created in a certain directory. The first parameter is the prompt of the dialog (default: "`Save As:`"). The second parameter is the default file name. The Boolean result indicates whether the user has pressed the `Save` button (`TRUE`) or the `Cancel` button (`FALSE`). The `String` result is the complete path name of the selected file. If the `Cancel` button has been pressed, an empty string will be returned. When a file with the indicated name already exists in the indicated directory a confirm dialog will be opened after pressing `Save`.

## C.8 Predefined Controls (deltaControls)

```
definition module deltaControls
import deltaIOSystem, deltaEventIO
```

General scrolling list and slider bar definition. These predefined dialog items are implemented entirely in Concurrent Clean as a user-defined `Control`.

```
::   NrVisible        :==  Int
::   SliderDirection  =    Horizontal | Vertical
::   SliderPos        :==  Int
::   SliderMax        :==  Int
```

A `ScrollingList` is defined by the following attributes:

- Id, `ItemPos` and `SelectState` (like other dialog items).
- The minimum width of the scrolling list (`Measure`).
  This attribute is important only when `ChangeScrollingList` is used. Use a minimum width of zero when the scrolling list is not changed.
- The number of items that is visible in the list (`NrVisible`).
- The item that is initially selected (`ItemTitle`).
- The list of items (`[ItemTitle]`).
- A `DialogFunction` that is called whenever a new item is selected.

The function `ScrollingList` returns a `DialogItem` (a `Control`) that can be used in any dialog definition.

```
ScrollingList ::   !DialogItemId !ItemPos !Measure !SelectState !NrVisible !ItemTitle
              ![ItemTitle] !(DialogFunction s (IOState s))   ->   DialogItem s (IOState s)
```

With `ChangeScrollingList` the items in the scrolling list can be changed. Its arguments are the id of the scrolling list, the newly selected item and the new list of items. When the id is not the id of a `ScrollingList` a run-time error is generated.

```
ChangeScrollingList ::  !DialogItemId !ItemTitle ![ItemTitle]
                         !(DialogState s (IOState s))   ->   DialogState s (IOState s)
```

`GetScrollingListItem` returns the currently selected item in the scrolling list with the indicated id from the `DialogInfo` parameter. When the id is not the id of a `ScrollingList` a run-time error is generated.

```
GetScrollingListItem  ::   !DialogItemId !DialogInfo   ->   ItemTitle
```

A `SliderBar` is defined by the following attributes:

- `Id`, `ItemPos` and `SelectState`, like other `DialogItems`.
- `SliderDirection`: `Horizontal` or `Vertical`.
- `SliderPos`: the initial position of the slider. This position is always adjusted between `0` and `SliderMax`.
- `SliderMax`: the slider can take on positions between `0` and `SliderMax`.
- A `DialogFunction` that is called whenever the slider moves.

```
SliderBar ::  !DialogItemId !ItemPos !SelectState !SliderDirection !SliderPos
        !SliderMax !(DialogFunction s (IOState s))   ->   DialogItem s (IOState s)
```

`ChangeSliderBar` moves the slider of the indicated slider bar to the new position. The position is always adjusted between `0` and `SliderMax`.

```
ChangeSliderBar     ::   !DialogItemId !SliderPos !(DialogState s (IOState s))
                                                 ->   DialogState s (IOState s)
```

`GetSliderPosition` returns the current slider position of the slider bar with the indicated id from the `DialogInfo` parameter. When the id is not the id of a `SliderBar` a run-time error is generated.

```
GetSliderPosition   ::   !DialogItemId !DialogInfo   ->   SliderPos
```

## C.9 Miscellaneous operations (deltaIOState)

```
definition module deltaIOState
import deltaIOSystem, deltaEventIO

SetGlobalCursor        ::   !CursorShape  !(IOState s)  ->   IOState s
```

Sets the shape of the cursor (mouse pointer) globally. This shape overrules the local cursor shapes of windows. Attempts to set the global cursor to `HiddenCursor` are ignored.

```
ResetCursor            ::   !(IOState s)  ->   IOState s
```

Undoes the effect of `SetGlobalCursor`. It resets the cursor to the standard shape outside windows with a local cursor shape and to the local shape inside such windows.

```
ObscureCursor          ::   !(IOState s)  ->   IOState s
```

Hides the cursor until the next time that the mouse is moved.

```
SetDoubleDownDistance  ::   !Int  !(IOState s)  ->   IOState s
```

Set the maximum distance (in pixels) between two mouse clicks such that they will be treated as a `ButtonDoubleDown` or `ButtonTripleDown`.

## C.10 Operations on pictures (deltaPicture)

```
definition module deltaPicture
import deltaFont

::   * Picture

::   DrawFunction  :== Picture -> Picture
```

The predefined figures that can be drawn:

```
::   Point            :==   (!Int, !Int)
::   Line             :==   (!Point, !Point)
::   Curve            :==   (!Oval, !Int, !Int)
::   Rectangle        :==   (!Point, !Point)
::   RoundRectangle   :==   (!Rectangle, !Int, !Int)
::   Oval             :==   Rectangle
::   Circle           :==   (!Point, !Int)
::   Wedge            :==   (!Oval, !Int, !Int)
::   Polygon          :==   (!Point, !PolygonShape)
::   PolygonShape     :==   [Vector]
::   Vector           :==   (!Int, !Int)
```

The pen attributes influence the way figures are drawn. The `PenMode` also influences the way text is drawn. The `Not...` modes do not work when text is drawn. When the `PenMode` is a `Not...` mode text is drawn in `OrMode`.

```
::   PenSize    :==  (!Int, !Int)
::   PenMode    =    CopyMode      |  OrMode        |  XorMode    |
                     ClearMode     |  NotCopyMode   |  NotOrMode  |
                     NotXorMode    |  NotClearMode  |  HiliteMode
::   PenPattern =    BlackPattern  |  DkGreyPattern |  GreyPattern   |
                     LtGreyPattern |  WhitePattern
```

The predefined colours:

```
::   Colour =   RGB Real Real Real              |
                BlackColour   |  WhiteColour    |  BlueColour    |  CyanColour    |
                RedColour     |  GreenColour    |  YellowColour  |  MagentaColour


   MinRGB   :==   0.0
   MaxRGB   :==   1.0
```

Rules setting the attributes of a Picture:

`SetPenSize (w, h)` sets the `PenSize` to `w` pixels wide and `h` pixels high.
`SetPenMode` sets the drawing mode of the pen.
`SetPenPattern` sets the pattern of the pen.
`SetPenNormal` sets the `PenSize` to `(1,1)`, the `PenMode` to `CopyMode` and the `PenPattern` to `BlackPattern`.

```
SetPenSize    ::   !PenSize      !Picture   ->   Picture
SetPenMode    ::   !PenMode      !Picture   ->   Picture
SetPenPattern ::   !PenPattern   !Picture   ->   Picture
SetPenNormal  ::                 !Picture   ->   Picture
```

Colour: there are two types of `Colour`: RGB colours and basic colours. An RGB colour defines the amount of red (`r`), green (`g`) and blue (`b`) in a certain colour by the tuple `(r,g,b)`. These are `Real` values and each of them must be between `MinRGB` and `MaxRGB` (`0.0` and `1.0`). The colour black is defined by `(MinRGB, MinRGB, MinRGB)` and white by `(MaxRGB, MaxRGB, MaxRGB)`. Given an RGB colour, all amounts are adjusted between `MinRGB` and `MaxRGB`.

`SetPenColour` sets the colour of the pen.
`SetBackColour` sets the background colour.

```
SetPenColour  ::   !Colour !Picture    ->   Picture
SetBackColour ::   !Colour !Picture    ->   Picture
```

Fonts: the initial font of a `Picture` is the default font (see `deltaFont.dcl`).

`SetFont` sets a new complete `Font` in the `Picture`.
`SetFontName` sets a new font without changing the style or size.
`SetFontStyle` sets a new style without changing font or size.
`SetFontSize` sets a new size without changing font or style. The size is always adjusted between `MinFontSize` and `MaxFontSize` (see `deltaFont.dcl`).

```
SetFont       ::   !Font         !Picture   ->   Picture
SetFontName   ::   !FontName     !Picture   ->   Picture
SetFontStyle  ::   ![FontStyle]  !Picture   ->   Picture
SetFontSize   ::   !FontSize     !Picture   ->   Picture
```

`PictureCharWidth` (`PictureStringWidth`) yield the width of the given `Char` (`String`) given the current font of the `Picture`. `PictureFontMetrics` yields the `FontInfo` of the current font.

```
PictureCharWidth    ::   !Char   !Picture   ->   (!Int,      !Picture)
PictureStringWidth  ::   !String !Picture   ->   (!Int,      !Picture)
PictureFontMetrics  ::           !Picture   ->   (!FontInfo,   !Picture)
```

Absolute and relative pen move operations without drawing.

```
MovePenTo  ::   !Point  !Picture   ->   Picture
MovePen    ::   !Vector !Picture   ->   Picture
```

Absolute and relative pen move operations with drawing.

```
LinePenTo  ::   !Point  !Picture   ->   Picture
LinePen    ::   !Vector !Picture   ->   Picture
```

`DrawChar` (`DrawString`) draws a character (string) in the current font. The baseline of the characters is the *y*-coordinate of the pen. The new position of the pen is directly after the (last) character (of the string).

```
DrawChar   ::   !Char      !Picture   ->   Picture
DrawString ::   !String !Picture   ->   Picture
```

All following rules do not change the position of the pen after drawing.

`Draw(C)Point` draws the pixel (in the given colour) in the `Picture`.
`Draw(C)Line` draws the line (in the given colour) in the `Picture`.
`Draw(C)Curve` draws the curve (in the given colour) in the `Picture`. A `Curve` is a part of an `Oval o` starting from angle `a` up to angle `b` (both in degrees modulo 360): `(o, a, b)`. Angles are always taken anticlockwise, starting from 3 o'-clock.

```
DrawPoint  ::   !Point  !Picture   ->   Picture
DrawLine   ::   !Line   !Picture   ->   Picture
DrawCurve  ::   !Curve  !Picture   ->   Picture
DrawCPoint ::   !Point  !Colour !Picture   ->   Picture
DrawCLine  ::   !Line   !Colour !Picture   ->   Picture
DrawCCurve ::   !Curve  !Colour !Picture   ->   Picture
```

A `Rectangle` is defined by two diagonal corner points.
`DrawRectangle` draws the edges of the rectangle.
`FillRectangle` draws the edges and interior of the rectangle.
`EraseRectangle` erases the edges and interior of the rectangle.
`InvertRectangle` inverts the edges and interior of the rectangle.
`MoveRectangleTo` moves the contents of the rectangle to a new top-left corner.
`MoveRectangle` moves the contents of the rectangle over the given vector.
`CopyRectangleTo` copies the contents of the rectangle to a new top-left corner.
`CopyRectangle` moves the contents of the rectangle over the given vector, but leaves the original untouched.

```
DrawRectangle    ::   !Rectangle !Picture   ->   Picture
FillRectangle    ::   !Rectangle !Picture   ->   Picture
EraseRectangle   ::   !Rectangle !Picture   ->   Picture
InvertRectangle  ::   !Rectangle !Picture   ->   Picture

MoveRectangleTo  ::   !Rectangle !Point   !Picture   ->   Picture
MoveRectangle    ::   !Rectangle !Vector  !Picture   ->   Picture
CopyRectangleTo  ::   !Rectangle !Point   !Picture   ->   Picture
CopyRectangle    ::   !Rectangle !Vector  !Picture   ->   Picture
```

Rounded corner rectangles: a `RoundRectangle` with enclosing `Rectangle r` and corner curvatures `x` and `y` is defined by the tuple `(r, x, y)`. `x` and `y` define the horizontal and vertical diameter of the corner curves. They are always adjusted between 0 and the width (height) of `r`.

```
DrawRoundRectangle     ::   !RoundRectangle  !Picture   ->   Picture
FillRoundRectangle     ::   !RoundRectangle  !Picture   ->   Picture
EraseRoundRectangle    ::   !RoundRectangle  !Picture   ->   Picture
InvertRoundRectangle   ::   !RoundRectangle  !Picture   ->   Picture
```

An `Oval` is defined by its enclosing `Rectangle`.

```
DrawOval   ::   !Oval   !Picture   ->   Picture
FillOval   ::   !Oval   !Picture   ->   Picture
EraseOval  ::   !Oval   !Picture   ->   Picture
InvertOval ::   !Oval   !Picture   ->   Picture
```

A `Circle` with centre `c` (a `Point`) and radius `r` (an `Int`) is defined by the tuple `(c, r)`.

```
DrawCircle    ::   !Circle !Picture   ->   Picture
FillCircle    ::   !Circle !Picture   ->   Picture
EraseCircle   ::   !Circle !Picture   ->   Picture
```

```
InvertCircle  ::   !Circle  !Picture   ->   Picture
```

A `Wedge` is a part of an `Oval o` starting from angle `a` up to angle `b` (both in degrees modulo 360): `(o, a, b)`. Angles are taken anticlockwise, starting from 3 o'-clock.

```
DrawWedge     ::   !Wedge   !Picture   ->   Picture
FillWedge     ::   !Wedge   !Picture   ->   Picture
EraseWedge    ::   !Wedge   !Picture   ->   Picture
InvertWedge   ::   !Wedge   !Picture   ->   Picture
```

A `Polygon` is a figure drawn by a number of lines without taking the pen off the `Picture`, starting from and ending at some `Point p`. The `PolygonShape` is a list of `Vectors` (`[`$v_1$`,...,`$v_n$`]`) that defines how the `Polygon` is drawn: `(p, [`$v_1$`,...,`$v_n$`])`.

`ScalePolygon` scales the polygon. Non-positive scale factors are allowed.
`MovePolygonTo` changes the starting point into the given `Point` and
`MovePolygon` moves the starting point by the given `Vector`.

```
ScalePolygon  ::   !Int     !Polygon   ->   Polygon
MovePolygonTo ::   !Point   !Polygon   ->   Polygon
MovePolygon   ::   !Vector  !Polygon   ->   Polygon

DrawPolygon   ::   !Polygon   !Picture   ->   Picture
FillPolygon   ::   !Polygon   !Picture   ->   Picture
ErasePolygon  ::   !Polygon   !Picture   ->   Picture
InvertPolygon ::   !Polygon   !Picture   ->   Picture
```

## C.11 Operations on fonts (deltaFont)

```
definition module deltaFont

::   Font

::   FontName   :==  String
::   FontStyle  :==  String
::   FontSize   :==  Int
::   FontInfo   :==  (!Int, !Int, !Int, !Int)


   MinFontSize   :==  …        // system dependent
   MaxFontSize   :==  …        // system dependent
```

`SelectFont` creates the font as specified by the name, the stylistic variations and size. The size is always adjusted between `MinFontSize` and `MaxFontSize`. The Boolean result is `TRUE` if the font is available and need not be scaled. In cases where the font is not available, the default font is chosen in the specified style and size.

```
SelectFont    ::   !FontName ![FontStyle] !FontSize  ->   (!Bool, !Font)
```

`DefaultFont` returns the default font, specified by name, style and size.

```
DefaultFont   ::   (!FontName, ![FontStyle], !FontSize)
```

`FontNames` returns the `FontNames` of all available fonts.
`FontStyles` returns the available `FontStyles` for a certain font.
`FontSizes` returns all `FontSizes` of a font that are available without scaling.
In cases where the font is not available, the styles or sizes of the default font are returned.

```
FontNames  ::   [FontName]
FontStyles ::   !FontName  ->   [FontStyle]
FontSizes  ::   !FontName  ->   [FontSize]
```

`FontCharWidth(s)` and `FontStringWidth(s)` return the width(s) in terms of pixels of given character(s) or string(s) in a certain `Font`.

```
FontCharWidth       ::   !Char      !Font    ->   Int
FontCharWidths      ::   ![Char]    !Font    ->   [Int]
FontStringWidth     ::   !String    !Font    ->   Int
FontStringWidths    ::   ![String]  !Font    ->   [Int]
```

`FontMetrics` yields the `FontInfo` in terms of pixels of a given `Font`. The `FontInfo` is a 4-tuple `(ascent, descent, maxwidth, leading)`, which defines the font metrics:

- `ascent` is the maximum height of a character measured from the base line.
- `descent` is the maximum depth of a character measured from the base line.
- `maxwidth` is the width of the widest character.
- `leading` is the vertical distance between two lines of the same font.

The full height of a line is the sum of the `ascent`, `descent` and `leading`.

```
FontMetrics   ::   !Font   ->   FontInfo
```

## C.12 System-dependent constants and functions (deltaSystem)

```
definition module deltaSystem
```

Keyboard constants (of type `KeyCode`):

```
   UpKey       :== ...              BackSpKey   :== ...
   DownKey     :== ...              DelKey      :== ...
   LeftKey     :== ...              TabKey      :== ...
   RightKey    :== ...              ReturnKey   :== ...
   PgUpKey     :== ...              EnterKey    :== ...
   PgDownKey   :== ...              EscapeKey   :== ...
   BeginKey    :== ...              HelpKey     :== ...
   EndKey      :== ...
```

Separator between directory and filenames in a pathname (of type `Char`):

```
   DirSeparator  :== ...
```

Constants to check which of the meta-keys (modifiers) is down (of type `Modifiers`):

```
   ShiftOnly     :== (TRUE,FALSE,FALSE,FALSE)
   OptionOnly    :== (FALSE,TRUE,FALSE,FALSE)
   CommandOnly   :== ... // depends on whether the keyboard of the system
   ControlOnly   :== ... // has a Command key.
```

The minimum and maximum sizes of a picture domain (of type `Int`):

```
   MinPictureDomain    :== ...
   MaxPictureDomain    :== ...
```

The functions `HomePath` and `ApplicationPath` prefix the filename given to them with the full pathnames of the 'home' and 'application' directory. These functions have been added for compatibility with the Sun version of the Clean system. In the 'home' directory settings-files (containing preferences, options etc.) should be stored. In the 'application' directory (i.e. the directory in which the application resides) files that are used read-only by the application (such as help files) should be stored. On the Macintosh these functions just return the filename given to them, which means that the file will be stored in the same folder as the application.

```
HomePath           ::   !String  -> String
ApplicationPath  ::   !String  -> String
```

Functions to retrieve the horizontal and vertical screen resolution.

```
MMToHorPixels    ::   !Real    ->   Int
MMToVerPixels    ::   !Real    ->   Int
InchToHorPixels  ::   !Real    ->   Int
InchToVerPixels  ::   !Real    ->   Int
```

The maximum width and height for the indicated type of window such that the window will fit on the screen. For `FixedWindows` this is the maximum size of the `PictureDomain` such that the window does not become a `ScrollWindow`.

```
MaxScrollWindowSize    ::   (!Int, !Int)
MaxFixedWindowSize  ::   (!Int, !Int)
```