A
Concurrent Event I/O Library

This Appendix presents the types and function definitions of the concurrent event I/O library, version 1.0. These definitions have been used throughout this thesis. Appendix A.1 contains the general definition modules. Appendix A.2 presents the process combinator definitions that are needed when creating interactive processes. The appendices A.3 upto A.7 contain the definitions and operations of each of the possible abstract devices. These are controls, menus, receivers, timers, and windows respectively. For reasons of completeness, we have also included the definition of the standard file operations in Appendix A.8.

A.1 General definitions

This Appendix consists of the definition modules `StdIOCommon`, `StdIOState`, `StdSystem`, `StdFont`, and `StdPicture`. `StdIOCommon` contains types that occur frequently within the I/O library. `StdIOState` contains operations that are application directed, such as setting the global cursor shape and process attributes. `StdSystem` contains all platform dependent constants (special key codes) and operations (screen resolution functions). `StdFont` contains all font operations, and `StdPicture` all drawing operations.

A.1.1 StdIOCommon

```
definition module StdIOCommon
```

Common types for the event I/O system and their access rules:

```
from StdPicture import Rectangle, Point

::  Id            :== Int
::  Index         :== Int
::  Title         :== String
::  Size          :== (!Int,!Int)          // (width,height) in pixels

::  SelectState   =   Able | Unable
::  MarkState     =   Mark | NoMark

::  KeyboardState :== (!KeyCode, !KeyState, !Modifiers)
::  KeyCode       :== Char
::  KeyState      =   KeyUp | KeyDown | KeyStillDown
::  MouseState    :== (!MousePosition,!ButtonState,!Modifiers)
::  MousePosition :== Point
::  ButtonState   =   ButtonUp
                  |   ButtonDown
                  |   ButtonDoubleDown
                  |   ButtonTripleDown
                  |   ButtonStillDown
::  Modifiers     :== (!Bool,!Bool,!Bool,!Bool)
```

Modifiers indicates the meta keys that have been pressed (True) or not (False): (Shift, Option, Command, Control).

```
::  PictureDomain   :== Rectangle
::  UpdateArea      :== [Rectangle]

::  ItemPos         :== (   ItemLoc,
                            ItemOffset
                        )
::  ItemLoc         =   LeftTop     | RightTop    | LeftBottom | RightBottom
                    |   Left        | Center      | Right
                    |   LeftOf   Id | RightTo  Id
                    |   Above    Id | Below    Id
::  ItemOffset      :== (!Int,!Int)
```

The layout language used for windows and controls.

```
::  IOAttribute ps  =   IOActivate   (IOFunction ps)
                    |   IODeactivate (IOFunction ps)
                    |   IOHelp       (IOFunction ps)
```

Attributes for interactive processes.

```
::  IOFunction     ps :==                  ps -> ps
::  ModsIOFunction ps :== Modifiers     -> ps -> ps
::  UpdateFunction ps :== UpdateArea    -> ps -> ps
::  MouseFunction  ps :== MouseState    -> ps -> ps
::  KeysFunction   ps :== KeyboardState -> ps -> ps
```
Frequently used function types.

```
EqualSelectState :: !SelectState !SelectState -> Bool
EqualMarkState   :: !MarkState   !MarkState   -> Bool
EqualButtonState :: !ButtonState !ButtonState -> Bool
Enabled          :: !SelectState              -> Bool
Checked          :: !MarkState                -> Bool
MarkSwitch       :: !MarkState                -> MarkState
```

A.1.2 StdIOState

```
definition module StdIOState
```

Operations on the IOState that have a global effect.

```
import StdEventIO, StdWindowDef
```

```
Beep :: !(IOState .l .p) -> IOState .l .p
```
If the interaction is active, Beep emits a sound alert. If the interaction is inactive, Beep does nothing.

Operations on the global cursor:

```
SetGlobalCursor :: !CursorShape !(IOState .l .p) -> IOState .l .p
```
Set the shape of the cursor globally. This shape overrules the local cursor shapes of windows.

```
ResetCursor :: !(IOState .l .p) -> IOState .l .p
```
Undoes the effect of SetGlobalCursor.

```
ObscureCursor :: !(IOState .l .p) -> IOState .l .p
```
ObscureCursor hides the cursor until the mouse is moved.

Operations on the DoubleDownDistance:

```
SetDoubleDownDistance :: !DoubleDownDist !(IOState .l .p) -> IOState .l .p
```
Set the maximum distance the mouse is allowed to move to generate a ButtonDouble (Triple) Down button state. Negative values are set to zero.

Operations on the attributes of an interaction:

```
SetIOActivate  :: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
SetIODeactivate:: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
SetIOHelp      :: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
```
Change the IOActivate, IODeactivate, IOHelp attribute of the interaction.

```
from iostate import  IOState, PState, Files,
                     IOStateSetDoubleDownDist, DoubleDownDist
```

A.1.3 StdSystem

```
definition module StdSystem
```

Operating System dependent constants and functions.

```
import StdIOCommon
```

```
UpKey          :== '\036'    // Arrow up
DownKey        :== '\037'    // Arrow down
LeftKey        :== '\034'    // Arrow left
RightKey       :== '\035'    // Arrow right
PgUpKey        :== '\013'    // Page up
PgDownKey      :== '\014'    // Page down
BeginKey       :== '\001'    // Begin of text
EndKey         :== '\004'    // End of text
BackSpKey      :== '\010'    // Backspace
DelKey         :== '\177'    // Delete
TabKey         :== '\011'    // Tab
ReturnKey      :== '\015'    // Return
EnterKey       :== '\003'    // Enter
EscapeKey      :== '\033'    // Escape
HelpKey        :== '\005'    // Help
```
Keyboard constants.

```
DirSeparator   :== ':'       // Separator between folder- and filenames in a pathname
```
File constants.

```
ShiftOnly      :== (True, False,False,False)
OptionOnly     :== (False,True, False,False)
CommandOnly    :== (False,False,True, False)
ControlOnly    :== (False,False,False,True )
```
Constants to check which of the Modifiers is down.

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

```
mmperinch      :== 25.4

hmm            :: !Real -> Int
vmm            :: !Real -> Int
hinch          :: !Real -> Int
vinch          :: !Real -> Int
```
Screen resolution functions. h(mm/inch) convert millimeters/inches into pixels, horizontally. v(mm/inch) convert millimeters/inches into pixels, vertically.

```
MaxScrollWindowSize :: Size
MaxFixedWindowSize  :: Size
```

Maximum ranges of window PictureDomains:
MaxScrollWindowSize yields the range at which scrollbars are inactive.
MaxFixedWindowSize yields the range at which the window does not change into a ScrollWindow.

A.1.4 StdFont

```
definition module StdFont
```

Operations on Fonts.

```
::  Font

::  FontDef
    =   {   fName    :: !FontName,
            fStyles  :: ![FontStyle],
            fSize    :: !FontSize
        }
::  FontMetrics
    =   {   fAscent  :: !Int,
            fDescent :: !Int,
            fLeading :: !Int,
            fMaxWidth:: !Int
        }
::  FontName   :== String
::  FontStyle  :== String
::  FontSize   :== Int

MinFontSize    :== 6
MaxFontSize    :== 128
```

```
FontSelect  :: !FontDef -> (!Bool,!Font)
```

FontSelect creates the font as specified by the name, the stylistic variations and size. In case there are no FontStyles ([]), the font is selected without stylistic variations (i.e. in plain style). The size is always adjusted between MinFontSize and MaxFontSize. The boolean result is True in case this font is available and needn't be scaled. In case the font is not available, the default font is chosen in the indicated style and size.

```
DefaultFont :: FontDef
```

DefaultFont returns name, style and size of the default font.

```
GetFontDef  :: !Font -> FontDef
```

GetFontDef returns the name, stylistic variations and size of the argument Font.

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

FontNames	returns the FontNames of all available fonts.
FontStyles	returns the FontStyles of all available styles.
FontSizes	returns all FontSizes of a font that are available without scaling.

In case the font is unavailable, the styles or sizes of the default font are returned.

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

FontCharWidth(s) (FontStringWidth(s)) return the width(s) in terms of pixels of given character(s) (string(s)) for a particular Font.

```
GetFontMetrics   :: !Font -> FontMetrics
```

GetFontMetrics yields the metrics of a given Font in terms of pixels. FontMetrics is a record which defines the metrics of a font:

- fAscent — is the height of the top most character measured from the base
- fDescent — is the height of the bottom most character measured from the base
- fLeading — is the vertical distance between two lines of the same font.
- fMaxWidth — is the width of the widest character including spacing

The full height of a line is fAscent+fDescent+fLeading.

A.1.5 StdPicture

```
definition module StdPicture
```

Drawing functions and other operations on Pictures.

```
import StdFont

::  *Picture

::  Point        :== (!Int,  !Int)
::  Line         :== (!Point,!Point)
::  Curve        :== (!Oval, !Int,!Int)
::  Rectangle    :== (!Point,!Point)
::  Oval         :== Rectangle
::  Polygon      :== (!Point,!PolygonShape)

::  PolygonShape :== [Vector]
::  Vector       :== (!Int, !Int)
```

The predefined figures that can be drawn.

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

The pen attributes which influence the way figures are drawn.

```
::  Colour     =   RGB Real Real Real
               |   BlackColour | RedColour
               |   WhiteColour | GreenColour
               |   BlueColour  | YellowColour
               |   CyanColour  | MagentaColour
MinRGB         :== 0.0
MaxRGB         :== 1.0
```

Rules setting the attributes of a Picture:

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

SetPenSize (w,h) sets the PenSize to w pixels wide and h pixels high.
SetPenMode sets the interference how new figures 'react' to drawn ones.
SetPenPattern sets the way new figures are drawn.
SetPenNormal sets the PenSize to (1,1), PenMode to CopyMode and PenPattern to BlackPattern.

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

Using colours: There are basically two types of Colours: RGB 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 a RGB colour, all amounts are adjusted between MinRGB and MaxRGB. Applications that use RGB colours may not run on all computers (e.g. Macintosh Plus). A small set of basic colours is defined that can be used on all systems.
SetPenColour sets the colour of the pen.
SetBackColour sets the background colour.

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

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

Using fonts: The initial font of a Picture is 12 point Chicago in PlainStyle.
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 (StdFont.dcl).
PictureCharWidth (PictureStringWidth) yields the width of the given Char (String) given the current font of the Picture.
PictureFontMetrics yields the FontInfo of the current font.

```
DrawClip          :: !Polygon [DrawFunction] !Picture -> Picture
```

Drawing within in a polygonal clipping area: the Polygon argument defines the shape of the clipping area in which the drawing functions will be applied.

```
GetPenPos         :: !Picture -> (!Point, !Picture)
```

Determines the position of the pen.

Rules changing the position of the pen:

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

Absolute and relative pen move operations (without drawing).

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

Absolute and relative pen move operations (with drawing).

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

DrawChar (DrawString) draws the Char (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 Char (String) including spacing.

Rules not changing the position of the pen after drawing:

```
DrawPoint          :: !Point  !Picture -> Picture
DrawLine           :: !Line   !Picture -> Picture
DrawCurve          :: !Curve  !Picture -> Picture
```

Non plane figures:

DrawPoint	draws the pixel in the Picture.
DrawLine	draws the line in the Picture.
DrawCurve	draws the curve in the Picture. A Curve is part of an Oval o starting from angle a upto angle b (both of type Int in degrees modulo 360): (o, a, b).

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

MoveRectangle      :: !Rectangle !Vector !Picture -> Picture
CopyRectangle      :: !Rectangle !Vector !Picture -> Picture
```

A Rectangle is defined by two diagonal corner Points (A,B) with A=(Ax,Ay), B=(Bx,By) such that Ax <> Bx and Ay <> By. In case either Ax==Bx or Ay==By, the Rectangle is empty.

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.
MoveRectangle	moves	the contents of the rectangle over the given vector.
CopyRectangle	copies	the contents of the rectangle over the given vector.

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

Ovals: an Oval is defined by its enclosing Rectangle. Note: the Oval of a square Rectangle is a Circle.

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

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

Polygons: a Polygon is a figure drawn by a number of lines without taking the pen of the Picture, starting from some Point p. The PolygonShape s==[$v_1 \ldots v_N$] defines how the Polygon is drawn:

MoveTo p, DrawLine from v_1 upto v_N, DrawLineTo p to close it.
So a Polygon with s==[] is actually the Point p.

ScalePolygon	by scale k sets shape [$v_1 \ldots v_N$] into [k $v_1 \ldots$k v_N]. Negative and 0 are valid scales.
MovePolygonTo	changes the starting point into the given Point
MovePolygon	moves the starting point by the given Vector.

A.2 Process combinators

This Appendix consists only of the definition module `StdEventIO`. `StdEventIO` contains all interactive process combinators and basic operations on interactive processes.

A.2.1 StdEventIO

```
definition module StdEventIO
```

Definition of IOState: the environment on which all GUI I/O functions operate.

```
::  *IOState l p
::  *PState  l p
    =   {   pLocal   :: l,               // the local (and private) data of an interaction
            pPublic  :: p,               // the shared data (in a group) of an interaction
            pFiles   :: !Files,          // the current state of the file system
            pIOState :: !IOState l p     // the IOState environment of this process
        }
::  InitIO l p
    :== [IOFunction (PState l p)]

::  IODef l p
    =   {   ioDefInit  :: InitIO l p,    // The initial actions of the process
            ioDefAbout :: String         // The name of the process
        }
```

Coercing PState component operations to PState operations:

```
seqPIO :: ![IOFunction (IOState .l .p)] !(PState .l .p) -> PState .l .p
seqPFs :: ![IOFunction Files]           !(PState .l .p) -> PState .l .p
seqPLoc:: ![IOFunction .l]              !(PState .l .p) -> PState .l .p
seqPPub:: ![IOFunction .p]              !(PState .l .p) -> PState .l .p
```

Starting an interaction:

```
OpenIO :: !(IODef .l .p) (.l,.p) !*World -> *World
```

OpenIO starts an interaction specified by the IODef argument. The program state argument consisting of a local and public part serves as initial program state. If the interaction has been successfully created, the functions in ioDefInit are evaluated from left-to-right. This is followed by the actual evaluation of the interaction. In the cause of the evaluation many new sub interactions can be created and terminated. The interaction created by OpenIO is the root interaction. OpenIO terminates as soon as all sub interactions (including the root interaction) have terminated. OpenIO returns the final world environment.

```
NewIO :: !(IODef .l .p) (.l,.p) !(IOState .l` .p`) -> IOState .l` .p`
```

If the interaction is active, NewIO starts a new sub interaction that will run interleaved with the current sub interactions. The new sub interaction is specified by the IODef argument. Creation of the new sub interaction is done as in OpenIO. The functions in ioDefInit are evaluated from left-to-right before any abstract event handler of the new sub interaction is evaluated. The new sub interaction becomes the active sub interaction (so the current sub interaction is deactivated). If the interaction is inactive, NewIO does nothing.

```
ShareIO :: !(IODef .l .p) .l !(IOState .l` .p) -> IOState .l` .p
```

If the interaction is active, ShareIO starts a new sub interaction that will run interleaved with the current sub interactions. The new sub interaction is specified by the IODef argument. Creation of the new sub interaction is done as in OpenIO. The functions in ioDefInit are evaluated from left-to-right before any abstract event handler of the new sub interaction is evaluated. The new sub interaction becomes the active sub interaction (so the current sub interaction is deactivated). The new sub interaction can communicate with all sub interactions by means of the file system or by message passing. The new sub interaction can communicate with all sub interactions of the interaction group of the sub interaction that spawned it by means of the shared program state component. If the interaction is inactive, ShareIO does nothing.

```
QuitIO   :: !(IOState .l .p) -> IOState .l .p
```

QuitIO removes all devices that are held in the sub interaction. As a result evaluation of this sub interaction will terminate. QuitIO is the only function that causes OpenIO to terminate.

```
HideIO   :: !(IOState .l .p) -> IOState .l .p
ShowIO   :: !(IOState .l .p) -> IOState .l .p
```

If the interaction is active, HideIO hides the sub interaction, and ShowIO makes it visible. Note that hiding a sub interaction does *not* disable the sub interaction, but simply makes it invisible. If the interaction is inactive, HideIO and ShowIO do nothing.

```
ActivateIO :: !(IOState .l .p) -> IOState .l .p
```

If the interaction is active, ActivateIO activates the sub interaction. As a result, all open windows and dialogs of the sub interaction will be moved top-most on the desktop. If the interaction is inactive, ActivateIO does nothing.

```
RequestIO :: !String !(IOState .l .p) -> IOState .l .p
```

If the interaction is inactive, RequestIO alerts the user that the interaction needs to become active. If the string argument is not empty, then this alert will consist of a Notice displaying the string. An interaction can issue an arbitrary amount of requests. If the interaction is active, RequestIO does nothing.

```
from   iostate     import IOState, Files, PState
from   StdIOCommon import IOFunction
```

A.3 Control device

This Appendix presents the definition modules that concern the definitions and operations on controls. Controls are the elements of windows and dialogues, such as buttons, (input) text fields, and so on. The predefined controls result in a platform dependent look-and-feel occurence of the controls. The definition modules are `StdControlDef` and `StdControl`. `StdControlDef` consists of the control definitions as they can be incorporated in the definitions of windows and dialogues. `StdControl` contains all operations on controls, once created in windows and dialogues.

A.3.1 StdControlDef

```
definition module StdControlDef
```

Definition of controls

```
import StdIOCommon
from   StdPicture  import DrawFunction, Picture


::  ControlDef ps
=   RadioControl        TextLine MarkState                             [ControlAttribute ps]
|   CheckControl        TextLine MarkState                             [ControlAttribute ps]
|   PopUpControl        [PopUpItem ps] Index                           [ControlAttribute ps]
|   SliderControl       Direction Length SliderState (SliderAction ps) [ControlAttribute ps]

|   TextControl         TextLine                                       [ControlAttribute ps]
|   EditControl         TextLine Width NrLines                         [ControlAttribute ps]
|   ButtonControl       TextLine                                       [ControlAttribute ps]
|   CustomButtonControl Size ControlLook                               [ControlAttribute ps]
|   CustomControl       Size ControlLook CustomState                   [ControlAttribute ps]
|   CompoundControl     [ControlDef ps] ControlLook                    [ControlAttribute ps]

::  TextLine         :== String
::  NrLines          :== Int
::  Width            :== Int
::  Length           :== Int
::  PopUpItem    ps  :== (TextLine, IOFunction ps)
::  ControlLook      :== SelectState -> Size -> [DrawFunction]
::  SliderAction ps  :== SliderMove  -> ps -> ps
::  SliderState      =   { sliderMin      :: !Int,
                           sliderMax      :: !Int,
                           sliderThumb    :: !Int
                         }
::  SliderMove       =   SliderIncSmall
                     |   SliderDecSmall
                     |   SliderIncLarge
                     |   SliderDecLarge
                     |   SliderThumb Int
::  Direction        =   Horizontal
                     |   Vertical

::  CustomState      =   BoolCS Bool | IntCS Int | RealCS Real | StringCS String
                     |   PairCS CustomState CustomState
                     |   ListCS [CustomState]
```

```
::  ControlAttribute ps                                    // Default:
=   ControlId           Id                                 // no id
|   ControlPos          ItemPos                            // (RightTo previous, (0,0))
|   ControlSize         Size                               // system derived
|   ControlMinimumSize  Size                               // (0,0)
|   ControlResize       ControlResizeFunction              // no resize
|   ControlSelectState  SelectState                        // control Able
|   ControlFunction     (IOFunction     ps)                // I
|   ControlModsFunction (ModsIOFunction ps)                // ControlFunction
|   ControlMouse SelectState (MouseFunction ps)            // no mouse input
|   ControlKeys  SelectState (KeysFunction  ps)            // no keyboard input

::  ControlResizeFunction
:== Size ->                                                // current control size
    Size ->                                                // old     window size
    Size ->                                                // new     window size
    Size                                                   // new     control size
```

A.3.2 StdControl

```
definition module StdControl
```

StdControl specifies all functions on controls. Changing controls in a window requires a *(Window .l .p). Reading the status of controls requires a (Window .l .p).

```
::  u:Window l p
```

```
GetWindow           :: !Id !(IOState .l .p) -> (!Bool, !Window .l .p, !IOState .l .p)
```

GetWindow returns a read-only Window for the indicated window. The Boolean result indicates whether the indicated window exists. In case it is False a dummy Window is returned.

```
SetWindow           :: !Id ![*(Window .l .p)->*Window .l .p] !(IOState .l .p) -> IOState .l .p
```

Apply the control changing functions to the current state of the indicated window. Invalid Ids are ignored.

```
GetControlSizes     :: ![ControlDef .ps] !(IOState .l .p) -> (![Size], !IOState .l .p)
```

GetControlSizes calculates the sizes of the given control definitions in the size as they would be opened as elements of a window.

Functions applied to unknown ids are ignored.

```
EnableControls      :: ![Id]            !*(Window .l .p) -> *Window .l .p
DisableControls     :: ![Id]            !*(Window .l .p) -> *Window .l .p
MarkCheckControls   :: ![Id]            !*(Window .l .p) -> *Window .l .p
UnmarkCheckControls:: ![Id]             !*(Window .l .p) -> *Window .l .p
SelectRadioControl :: !Id               !*(Window .l .p) -> *Window .l .p
SetEditTextControl :: !Id !String       !*(Window .l .p) -> *Window .l .p
SetTextControl      :: !Id !String      !*(Window .l .p) -> *Window .l .p
SetControlLook      :: !Id !ControlLook !*(Window .l .p) -> *Window .l .p
```

These functions change the state of controls.

```
SetSliderState     :: !Id !SliderState !*(Window .l .p) -> *Window .l .p
SetSliderThumb     :: !Id !Int         !*(Window .l .p) -> *Window .l .p
```

SetSlider(State/Thumb) sets the SliderState/Thumb and redraws the settings of the slider.

```
SetControlHandler :: !Id !(ControlAttribute (PState .l .p)) !*(Window .l .p) -> *Window .l .p
```

Set the abstract event handler of a (Radio/Check/Text/Edit/(Custom)Button/Custom/Compound)Control. The ControlAttribute argument must be a Control (Resize/Function/ModsFunction/Mouse/Keys) attribute.

```
DrawInControl            :: !Id ![DrawFunction] !*(Window .l .p) -> *Window .l .p
```

Draw in a (Custom(Button)/Compound)Control.

```
GetEditTextControl       :: !Id !(Window .l .p) -> (Bool,String)       // ""
GetTextControl           :: !Id !(Window .l .p) -> (Bool,String)       // ""
GetSelectedPopUpItem     :: !Id !(Window .l .p) -> (Bool,Index)        // 0
GetSelectedRadioControls::      !(Window .l .p) -> [Id]
GetSelectedCheckControls::      !(Window .l .p) -> [Id]
RadioControlMarked       :: !Id !(Window .l .p) -> (Bool,Bool)         // False
CheckControlMarked       :: !Id !(Window .l .p) -> (Bool,Bool)         // False
GetCustomState           :: !Id !(Window .l .p) -> (Bool,CustomState)  // BoolCS False
GetSliderState           :: !Id !(Window .l .p) -> (Bool,SliderState)  // {sliderMin = 0,
                                                                       //   sliderMax = 0,
                                                                       //   sliderThumb = 0}
```

Functions that return the current contents of controls that can be changed by the user. The first Boolean result is False in case of invalid ids (if so dummy values are returned - see comment). The id passed to GetSelectedPopUpItem must be the id of a PopUpControl.
Important: controls with no ControlId attribute, or illegal ids, can not be found in the WindowInfo!

A.4 Menu device

This Appendix presents the definition modules that concern the definitions and operations on menus. The definition modules are `StdMenuDef` and `StdMenu`. `StdMenuDef` consists of the menu definitions. `StdMenu` contains all operations on menus.

A.4.1 StdMenuDef

```
definition module StdMenuDef
```

MenuDefinitions:

```
import StdIOCommon
```

```
:: MenuDef     ps =   Menu        Title [MenuElement ps] [MenuAttribute ps]
:: MenuElement ps =   SubMenuItem Title [MenuElement ps] [MenuAttribute ps]
                  |   MenuItem    Title                  [MenuAttribute ps]
                  |   MenuSeparator

:: MenuAttribute ps                            // Default:
=  MenuId           Id                         // no Id
|  MenuSelectState  SelectState                // menu(item) Able
|  MenuShortKey     KeyCode                    // no KeyCode
|  MenuAltKey       Index                      // no AltKey
|  MenuMarkState    MarkState                  // NoMark
```

Attributes ignored by (sub)menus:

```
|  MenuFunction     (IOFunction     ps)        // I
|  MenuModsFunction (ModsIOFunction ps)        // MenuFunction
```

A.4.2 StdMenu

```
definition module StdMenu
```

Operations on menus.

Operations on unknown Ids are ignored.

```
OpenMenu          :: !Int ![MenuDef (PState .l .p)] !(IOState .l .p) -> IOState .l .p
```

Open the given menu definition for this interactive process behind the menu indicated by the integer index. The index of a menu starts from one for the first present menu. If the index is negative or zero, then the new menu is added before the first menu. If the index exceeds the number of menus, then the new menu is added behind the last menu.

```
CloseMenu         :: !Id                            !(IOState .l .p) -> IOState .l .p
```

Closes the given menu (and all of its elements including submenus).

```
EnableMenuSystem  :: !(IOState .l .p) -> IOState .l .p
DisableMenuSystem :: !(IOState .l .p) -> IOState .l .p
```

Enable/disable the MenuSystem. When the menu system is enabled the previously selectable menus and menu items will become selectable again. Operations on a disabled menu system take effect when the menu system is re-enabled.

```
EnableMenus  :: ![Id] !(IOState .l .p) -> IOState .l .p
DisableMenus :: ![Id] !(IOState .l .p) -> IOState .l .p
```

Enable/disable PullDownMenus. Disabling a menu overrides the SelectStates of its menu elements, which become unselectable. Enabling a disabled menu re-establishes the SelectStates of the menu elements.

```
OpenMenuItems:: !Id !Int ![MenuElement (PState .l .p)] !(IOState .l .p) ->IOState .l .p
```

Adding menu elements in a (sub)menu. OpenMenuItems adds menu elements after the item with the specified index. The index of a menu element starts from one for the first menu element in the (sub)menu. If the index is negative or zero, then the new menu elements are added before the first menu element of the (sub)menu. If the index exceeds the number of menu elements in the (sub)menu, then the new menu elements are added behind the last menu element of the (sub)menu. Only MenuItems and MenuSeparators can be added to (sub)menus.

```
CloseMenuItems      :: ![Id]      !(IOState .l .p) -> IOState .l .p
CloseMenuIndexItems :: !Id ![Int] !(IOState .l .p) -> IOState .l .p
```

Removing menu elements from the indicated (sub)menu(s).

CloseMenuItems	removes elements by their Id.
CloseMenuIndexItems	removes elements of the indicated (sub)menu by their indices.

Analogous to OpenMenuItems, indices range from one to the number of menu elements in a (sub)menu. Invalid indices (less than one or larger than the number of menu elements of the (sub)menu) are ignored.

```
EnableMenuItems     :: ![Id]                                !(IOState .l .p) -> IOState .l .p
DisableMenuItems    :: ![Id]                                !(IOState .l .p) -> IOState .l .p
MarkMenuItems       :: ![Id]                                !(IOState .l .p) -> IOState .l .p
UnmarkMenuItems     :: ![Id]                                !(IOState .l .p) -> IOState .l .p
SetMenuItemTitles   :: ![(Id,Title)]                        !(IOState .l .p) -> IOState .l .p
SetMenuItemFunctions:: ![(Id,MenuAttribute (PState .l .p))] !(IOState .l .p) -> IOState .l .p
```

Enable/disable, mark/unmark, and change titles/functions of MenuElements (including SubMenuItems).

A.5 Receiver device

This Appendix presents the definition modules that concern the definitions and operations on receivers. Receivers are used to do message passing between interactive processes. The definition modules are `StdReceiverDef` and `StdReceiver`. `StdReceiverDef` consists of the definition of receivers. `StdReceiver` contains all operations on receivers, and in particular the message passing primitives.

A.5.1 StdReceiverDef

```
definition module StdReceiverDef
```

ReceiverDefinitions:

```
import StdIOCommon
```

```
::  ReceiverDef        mess ps = Receiver [ReceiverAttribute mess ps]

::  ReceiverAttribute mess ps                        // Default:
=   ReceiverSelect     SelectState                   // receiver Able
|   ReceiverFunction   (ReceiverFunction mess ps)    // f _ x = x
::  ReceiverFunction   mess ps
:== mess -> ps -> ps
```

A.5.2 StdReceiver

```
definition module StdReceiver
```

The identification of a Receiver:

```
::  RId m

eqRId :: !(RId m) !(RId m) -> Bool
```

Operations on the ReceiverDevice.

```
OpenReceiver    :: !(ReceiverDef mess (PState .l .p)) !(IOState .l .p)
                -> (!RId mess,       ! IOState .l .p)
CloseReceiver   :: !(RId mess)       !(IOState .l .p) -> IOState .l .p

EnableReceiver :: !(RId mess)        !(IOState .l .p) -> IOState .l .p
DisableReceiver:: !(RId mess)        !(IOState .l .p) -> IOState .l .p
```

Enable/Disable receiving events that have been raised to this interaction. If there is no receiver device, nothing happens.

```
ASyncSend       :: !(RId mess) mess !(IOState .l .p) -> IOState .l .p
```

ASyncSend posts an event in the event stream environment that is addressed to the interaction with the given RId.

A.6 Timer device

This Appendix presents the definition modules that concern the definitions and operations of timers. The definition modules are `StdTimerDef` and `StdTimer`. `StdTimerDef` consists of the timer definitions. `StdTimer` contains all operations on timers, and other time related operations such as pausing and getting the current system time and date.

A.6.1 StdTimerDef

```
definition module StdTimerDef
```

TimerDefinitions:

```
import StdIOCommon
```

```
::  TimerDef        ps =   Timer TimerInterval [TimerAttribute ps]
::  TimerInterval      :== Int
::  NrOfIntervals      :== Int

::  TimerAttribute ps                  // Default:
=   TimerId       Id                   // no Id
|   TimerSelect   SelectState          // timer Able
|   TimerFunction (TimerFunction ps)   // f _ x = x
::  TimerFunction ps
:== NrOfIntervals -> ps -> ps
```

A.6.2 StdTimer

```
definition module StdTimer
```

Operations on the TimerDevice.

```
TicksPerSecond  :== 60

::  CurrentTime =   { hours   :: !Int,  // hours       (0-23)
                      minutes :: !Int,  // minutes     (0-59)
                      seconds :: !Int   // seconds     (0-59)
                    }
::  CurrentDate =   { year    :: !Int,  // year
                      month   :: !Int,  // month       (1-12)
                      day     :: !Int,  // day         (1-31)
                      dayNr   :: !Int   // day of week (1-7,Sunday==1,Saturday==7)
                    }
```

```
OpenTimer            :: !(TimerDef (PState .l .p))          !(IOState .l .p) -> IOState .l .p
```
Open a new timer. This function has no effect in case the interactive process already contains a timer with the same Id. Negative TimerIntervals are set to zero.

```
CloseTimer           :: !Id                                 !(IOState .l .p) -> IOState .l .p
```
Close the timer with the indicated Id.

```
EnableTimer          :: !Id                                !(IOState .l .p) -> IOState .l .p
DisableTimer         :: !Id                                !(IOState .l .p) -> IOState .l .p
SetTimerFunction     :: !Id !(TimerFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
SetTimerInterval     :: !Id !TimerInterval                 !(IOState .l .p) -> IOState .l .p
```

Enable/disable, change the TimerFunction and TimerInterval of the indicated timer. Negative TimerIntervals are set to zero.

```
Wait                 :: !TimerInterval .x -> .x
```

Wait suspends the interaction for TimerInterval ticks.

```
GetTimerBlinkInterval:: !(IOState .l .p) -> (!TimerInterval, !IOState .l .p)
```

Returns the TimerInterval that should elaps between blinks of a cursor. This interval can change during the interaction!

```
GetCurrentTime       :: !(IOState .l .p) -> (!CurrentTime,   !IOState .l .p)
GetCurrentDate       :: !(IOState .l .p) -> (!CurrentDate,   !IOState .l .p)
```

GetCurrentTime(Date) returns the current time(date).

A.7 Window device

This Appendix presents the definition modules that concern the definitions and operations on windows. Windows can either be dialogue windows or more general windows. In a window one can place an arbitrary number of controls (see Appendix A.3). The definition modules are `StdWindowDef`, `StdWindow`, and `StdFileSelect`. `StdWindowDef` consists of the definitions of windows. `StdWindow` contains all operations on windows. `StdFileSelect` contains two special operations that open file selector windows, if possible, in a platform dependent look-and-feel.

A.7.1 StdWindowDef

```
definition module StdWindowDef
```

Window definitions.

```
import StdControlDef
from   StdFont import Font
```

```
::  WindowDef ps
=   DialogWindow  Title                [ControlDef ps] [WindowAttribute ps]
|   Window        Title PictureDomain [ControlDef ps] [WindowAttribute ps]
::  WindowFrame  :== Rectangle

::  WindowAttribute   ps                        // Default:
```

Attributes for all windows:

```
=   WindowId           Id                       // no id
|   WindowPos          ItemPos                  // system dependent
|   WindowSize         Size                     // screen size
|   WindowItemSpace    Size                     // system dependent
|   WindowOk           Id                       // system dependent
|   WindowStandBy                               // system dependent
|   WindowHide                                  // initially visible
|   WindowClose        (IOFunction     ps)      // window can't close window
|   WindowUpdate       (UpdateFunction ps)      // update by system
```

Attributes for DialogWindows only:

```
|   WindowMargin       Size                     // system dependent
```

Attributes for Windows only:

```
|   WindowMinimumSize Size                      // system dependent
|   WindowResize       (WindowResizeFunction ps)   // fixed size
|   WindowActivate     (IOFunction ps)          // I
|   WindowDeactivate   (IOFunction ps)          // I
|   WindowMouse  SelectState (MouseFunction ps)    // no mouse    input
|   WindowKeys   SelectState (KeysFunction  ps)    // no keyboard input
|   WindowCursor CursorShape                    // no cursor change

::  WindowResizeFunction ps :== Size ->         // old window size
                                Size ->         // new window size
                                ps   -> ps
::  CursorShape      = StandardCursor | BusyCursor     | IBeamCursor
                     | CrossCursor    | FatCrossCursor | ArrowCursor | HiddenCursor

DialogFont         :: Font
```

A.7.2 StdWindow

```
definition module StdWindow
```

StdWindow specifies all functions on windows.

Functions applied to unknown Ids are ignored.

```
OpenWindow     :: !(WindowDef (PState .l .p)) !(IOState .l .p) -> IOState .l .p
OpenModalWindow:: !(WindowDef (PState .l .p)) !( PState .l .p) ->  PState .l .p
```

If the interactive process is active, Open(Modal)Window opens the given window. In case a window with the same Id is already open then that window will be activated. OpenModalWindow terminates when the window has been closed (by means of CloseWindow). If the interactive process is inactive, Open(Modal)Window does nothing.

```
CloseWindow    :: !Id !(IOState .l .p) -> IOState .l .p
```

If the interactive process is active, CloseWindow closes the indicated window. If the interaction is inactive, CloseWindow does nothing.

```
HideWindows    :: ![Id] !(IOState .l .p) -> IOState .l .p
ShowWindows    :: ![Id] !(IOState .l .p) -> IOState .l .p
```

If the interactive process is active, (Hide/Show)Windows hides/shows the indicated windows. If the interactive process is inactive, (Hide/Show)Windows does nothing.

```
ActivateWindow :: !Id !(IOState .l .p) -> IOState .l .p
```

If the interaction is active, ActivateWindow makes the window with the given Id the active window. In case the Id is unknown ActivateWindow has no effect. If the interaction is inactive, ActivateWindow does nothing.

```
GetActiveWindow:: !(IOState .l .p) -> (Bool, !Id, !IOState .l .p)
```

GetActiveWindow returns the Id of the currently frontmost and visible window of the interactive process. The Boolean result reports whether the window exists. In case it is False, Id==0.

```
StackWindow    :: !Id !Id !(IOState .l .p) -> IOState .l .p
```

If the process is active, StackWindow id1 id2 places the window with id1 behind the window with id2. If id2 is unknown, the window becomes the active window. If id1 is unknown, or the process is inactive, StackWindow does nothing.

```
SetWindowPos   :: !Id !ItemPos !(IOState .l .p) -> IOState .l .p
```

If the interactive process is active, SetWindowPos places the window to the indicated position. If the ItemPos argument refers to the Id of an unknown window (in case of LeftOf/ RightTo/ Above/ Below), SetWindowPos has no effect. SetWindowPos also has no effect if the window is moved of the screen, if the Id is unknown, or if the interactive process is inactive.

```
GetWindowPos   :: !Id !(IOState .l .p) -> (!Bool, !ItemOffset, !IOState .l .p)
```

GetWindowPos returns the current item offset position of the indicated window. The corresponding ItemPos is (LeftTop,offset). In case the window does not exist the Boolean result is False and ItemOffset==(0,0).

```
GetWindowFrame :: !Id !(IOState .l .p) -> (!WindowFrame, !IOState .l .p)
```

GetWindowFrame returns the currently visible frame of the window in terms of the PictureDomain. In case of dialogues this is ((0,0), size). In case the id is unknown, the WindowFrame==((0,0),(0,0)).

```
MoveWindowFrame:: !Id Vector !(PState .l .p) -> PState .l .p
```

MoveWindowFrame moves the orientation of the window over the given vector, and updates the window if necessary. The window frame is not moved outside the PictureDomain of the window. In case of unknown Id, or of DialogWindows, MoveWindowFrame has no effect.

```
SetPictureDomain:: Id !PictureDomain !(PState .l .p) -> PState .l .p
```

If the interactive process is active, SetPictureDomain resets the current PictureDomain of the indicated window, and updates the window if necessary. In case the new PictureDomain is smaller than the current WindowFrame, the window is resized to fit the new domain exactly. The window frame is moved only if it gets outside the new PictureDomain. In case of unknown Ids, of DialogWindows, or of inactive processes, SetPictureDomain has no effect.

```
SetWindowMinimumSize :: Id Size !(PState .l .p) -> PState .l .p
```

If the interactive process is active, SetWindowMinimumSize sets the minimum size of the indicated window as given. The new minimum size is set to be smaller than the current PictureDomain of the window. The window is resized and updated if the current size of either edge of the window is smaller than the new minimum size. In case of unknown Ids, of DialogWindows, or of inactive processes, SetWindowMinimumSize has no effect.

```
SetWindowSize  :: Id Size !(PState .l .p) -> PState .l .p
```

If the interactive process is active, SetWindowSize sets the size of the indicated window as given, and updates the window if necessary. The size is fit between the minimum size and the PictureDomain of the window. In case of unknown Ids, of DialogWindows, or of inactive processes, SetWindowSize has no effect.

```
SetWindowTitle      :: !Id !Title                           !(IOState .l .p)-> IOState .l .p
SetWindowCursor     :: !Id !CursorShape                     !(IOState .l .p)-> IOState .l .p
SetWindowUpdate     :: !Id !(UpdateFunction (PState .l .p)) !(IOState .l .p)-> IOState .l .p
SetWindowClose      :: !Id !(IOFunction     (PState .l .p)) !(IOState .l .p)-> IOState .l .p
SetWindowResize     :: !Id !(WindowResizeFunction (PState .l .p))
                                                            !(IOState .l .p)-> IOState .l .p
SetWindowActivate   :: !Id !(IOFunction     (PState .l .p)) !(IOState .l .p)-> IOState .l .p
SetWindowDeactivate :: !Id !(IOFunction     (PState .l .p)) !(IOState .l .p)-> IOState .l .p
```

These functions set the indicated attributes. If the indicated window does not have the corresponding attribute then in case of SetWindow(Close/Resize) the new attribute is not set. In the other cases the window obtains the new attribute.

```
EnableWindowMouse     :: !Id                                  !(IOState .l .p)-> IOState .l .p
DisableWindowMouse    :: !Id                                  !(IOState .l .p)-> IOState .l .p
EnableWindowKeys      :: !Id                                  !(IOState .l .p)-> IOState .l .p
DisableWindowKeys     :: !Id                                  !(IOState .l .p)-> IOState .l .p
SetWindowMouseFunction:: !Id !(MouseFunction (PState .l .p)) !(IOState .l .p)-> IOState .l .p
SetWindowKeysFunction :: !Id !(KeysFunction  (PState .l .p)) !(IOState .l .p)-> IOState .l .p
```

These functions change the state of mouse and keyboard input.

```
DrawInWindow          :: !Id ![DrawFunction]                  !(IOState .l .p)-> IOState .l .p
```

Draw in the window (behind all Controls).

A.7.3 StdFileSelect

```
definition module StdFileSelect


from StdFile    import Files
from StdEventIO import IOState
```

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

```
SelectInputFile :: !Files !(IOState .l .p) -> (!Bool, !String, !Files, !IOState .l .p)
```

SelectInputFile opens a window in which the user can traverse the file system to select an existing file. The boolean result indicates whether the user pressed the Open button (True) or the Cancel button (False). The String result contains the complete pathname of the selected file. When Cancel was pressed an empty string is returned.

```
SelectOutputFile:: !String !String !Files !(IOState .l .p)
                -> (!Bool, !String,!Files,! IOState .l .p)
```

SelectOutputFile opens a dialog in which the user can specify the name of a file to write to in a certain directory. The first argument is the prompt of the dialog (default: "Save As:"), the second argument is the default filename. The boolean result indicates whether the user pressed the Save button (True) or the Cancel button (False). The String result contains the complete pathname of the selected file. When Cancel was pressed an empty string is returned. When a file with the indicated name already exists in the indicated directory a confirm dialog will be opened.

A.8 StdFile

```
system module StdFile
```

File modes synonyms:

```
FReadText   :== 0    // Read from a text file
FWriteText  :== 1    // Write to a text file
FAppendText :== 2    // Append to an existing text file
FReadData   :== 3    // Read from a data file
FWriteData  :== 4    // Write to a data file
FAppendData :== 5    // Append to an existing data file
```

Seek modes synonyms:

```
FSeekSet    :== 0    // New position is the seek offset
FSeekCur    :== 1    // New position is the current position plus the seek offset
FSeekEnd    :== 2    // New position is the size of the file plus the seek offset

::  *Files
```

Opening and closing a File from the file system:

```
openfiles :: !*World -> (!*Files,!*World)
closefiles:: !*Files !*World -> *World

fopen     :: !String !Int !*Files -> (!Bool,!*File,!*Files)
```

Opens a file for the first time in a certain mode (read, write or append, text or data). The boolean output parameter reports success or failure.

```
fclose    :: !*File !*Files -> (!Bool,!*Files)

freopen   :: !*File !Int    -> (!Bool,!*File)
```

Re-opens an open file in a possibly different mode. The boolean indicates whether the file was successfully closed before reopening.

Reading from a File:

```
freadc    :: !*File -> (!Bool,!Char,!*File)
```

Reads a character from a text file or a byte from a datafile. The boolean indicates succes or failure.

```
freadi    :: !*File -> (!Bool,!Int,!*File)
```

Reads an Integer from a textfile by skipping spaces, tabs and newlines and then reading digits, which may be preceded by a plus or minus sign. From a datafile freadi will just read four bytes (a Clean Int).

```
freadr    :: !*File -> (!Bool,!Real,!*File)
```

Reads a Real from a textfile by skipping spaces, tabs and newlines and then reading a character representation of a Real number. From a datafile freadr will just read eight bytes (a Clean Real).

```
freads    :: !*File !Int -> (!String,!*File)
```

Reads n characters from a text or data file, which are returned as a String. If the file doesn't contain n characters the file will be read to the end of the file. An empty String is returned if no characters can be read.

```
freadline :: !*File -> (!String,!*File)
```

Reads a line from a textfile (including a newline character, except for the last line). freadline cannot be used on data files.

Writing to a File:

```
fwritec   :: !Char !*File -> *File
```
Writes a character to a textfile. To a datafile fwritec writes one byte (a Clean Char).

```
fwritei   :: !Int !*File -> *File
```
Writes an Integer (its textual representation) to a text file. To a datafile fwritei writes four bytes (a Clean Int).

```
fwriter   :: !Real !*File -> *File
```
Writes a Real (its textual representation) to a text file. To a datafile fwriter writes eight bytes (a Clean Real).

```
fwrites   :: !String !*File -> *File
```
Writes a String to a text or data file.

Testing:

```
fend      :: !*File -> (!Bool,!*File)
```
Tests for end-of-file.

```
ferror    :: !*File -> (!Bool,!*File)
```
Has an error occurred during previous file I/O operations?

```
fposition :: !*File -> (!Int,!*File)
```
Returns the current position of the file pointer as an Integer. This position can be used later on for the fseek function.

```
fseek     :: !*File !Int !Int -> (!Bool,!*File)
```
Move to a different position in the file, the first Integer argument is the offset, the second argument is a seek mode (see above). True is returned if successful.

Predefined files:

```
stdio     :: !*Files -> (!*File,!*Files)
```
Open the 'Console' for reading and writing.

```
stderr    :: *File
```
Open the 'Errors' file for writing only. May be opened more than once.

Opening and reading Shared Files:

```
sfopen    :: !String !Int !*Files -> (!Bool,!File,!*Files)
```
With sfopen a file can be opened for reading more than once. On a file opened by sfopen only the operations beginning with sf can be used.

```
sfreadc   :: !File           -> (!Bool,!Char,!File)
sfreadi   :: !File           -> (!Bool,!Int, !File)
sfreadr   :: !File           -> (!Bool,!Real,!File)
sfreads   :: !File !Int      -> (!String,    !File)
sfreadline:: !File           -> (!String,    !File)
sfseek    :: !File !Int !Int -> (!Bool,      !File)
sfend     :: !File           -> Bool
sfposition:: !File           -> Int
```
The sf… operations work just like the corresponding f… operations. They can't be used for files opened with fopen or freopen. The functions sfend and sfposition work like fend and fposition, but don't return a new file on which other operations can continue. They can be used for files opened with sfopen or after fshare, and in guards for files opened with fopen or freopen.

*Convert a *File into:*

```
fshare    :: !*File -> File
```

Change a file so that from now it can only be used with sf... operations.