A Tutorial to the Clean Object I/O Library - version 1.2

Peter Achten
Martin Wierich
Department of Functional Programming
University of Nijmegen
The Netherlands

February 2, 2000

Contents

Chapter 1

Preface

The functional programming language Clean has an extensive library to build graphical user interface applications, the *object I/O library*. In this tutorial the basic concepts of the object I/O library are explained. Many of the concepts are illustrated by means of Clean program examples. Clean code will be typeset in `type writer` style. The example programs are also available as Clean sources in the corresponding 'Tutorial Examples' folder.

This tutorial is not a technical reference manual: you will not find an extensive and detailed description of every data structure and function of the library. It is also assumed that the reader is familiar with functional programming and Clean.

In Chapter 2 a brief overview of the object I/O library is given, to give the reader a taste of what the object I/O system is all about. Chapter 3 presents the global structure of the object I/O system.

The remaining part of this document explains the individual components of the library. But before we can explain the graphical user interface elements, we first talk about object identification (Chapter 4), and drawing (Chapter 5).

To users of graphical user interfaces *the* interface elements are of course the *windows* and *dialogues*. These are discussed in Chapter 6. Windows and dialogues can contain *controls*. Because there are many aspects about control handling their treatment deserves a separate chapter (7). In all graphical user interface systems, the set of available commands is presented by means of *menus*, see Chapter 8. To support timing features, *timers* can be used, see Chapter 9. Flexible communication of arbitrary expressions between components can be achieved by using *receivers* and *message passing*, see Chapter 10.

All of the above objects are elements of one interactive process. The object I/O library enables the programmer to split up a large interactive program into several interactive processes that can be created and closed dynamically. This is presented in Chapter 11.

Two final chapters remain that deal with issues that are also related to graphical user interface applications. In Chapter 12 we discuss clipboard handling, a simple user driven mechanism to transfer data between interactive applications. In Chapter 13, we demonstrate the means that exist to send output to a printer.

In Chapter 10 communication of arbitrary Clean expressions between interactive objects is discussed. In Chapter 14 it is demonstrated how Clean programs can communicate with arbitrary programs using TCP.

Appendix A contains the definition modules of the Clean object I/O library, version

1.2. in alphabetic order.

The Clean object I/O library is maintained by Peter Achten. Martin Wierich wrote the chapters on printing (Chapter 13) and TCP (Chapter 14) and also developed the corresponding library modules.

Chapter 2

Introduction

In this chapter we give a brief overview of the main features of the object I/O library. We first discuss what the basic components are and how they can be used to construct more complex components (Section 2.1). When these elements have been constructed, they must be opened to create an actual working image on the underlying platform. Elements can be opened and closed dynamically, but it is of course also possible to change them dynamically (Section 2.2). Once we know how to construct graphical user interfaces we can start an interactive program. This is explained in Section 2.3. Finally, to wrap things up Section 2.4 presents the first complete interactive Clean object I/O program of this tutorial, the ubiquitous "Hello world!".

2.1 What are interactive objects

One way of looking at the object I/O library is to regard it as a collection of building blocks, the *interactive objects*, that the programmer can use to construct graphical user interfaces. For instance, Figure 2.1 summarises the standard set of *control* objects that can be placed in a *window* or *dialogue* object.

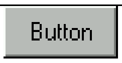
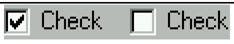
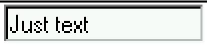
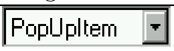
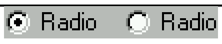
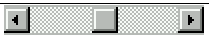
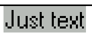
Control object:	**What does it look like:**
`ButtonControl`	
`CheckControl`	
`CompoundControl`	Program defined combination of controls
`CustomButtonControl`	Program defined button
`CustomControl`	Program defined button
`EditControl`	
`LayoutControl`	Program defined combination of controls
`PopUpControl`	
`RadioControl`	
`SliderControl`	
`TextControl`	

Figure 2.1: The standard set of controls.

All interactive objects are defined by means of algebraic data types. As an example, to define the button control element in the table of Figure 2.1 one would write:

```
button = ButtonControl "Button" []
```

The constituents of this expression are the data constructor `ButtonControl`, applied to the string `"Button"`, and an empty list `[]` of control attributes. This is defined more concisely by the library type definition of a button control element:

```
:: ButtonControl ls pst
 = ButtonControl String [ControlAttribute *(ls,pst)]
```

It should be noted that the names of the type constructor and the data constructor are identical (`ButtonControl`). This convention is used throughout the object I/O library.

The type definition of the button control is parameterised with two type variables: *ls* and *pst*. These correspond to another fundamental characteristic of interactive objects: an interactive object can have *local state*, and also have an effect on a *process state*. The type of the local state is identified by the *ls* type parameter, while the type of the process state is identified by the *pst* type parameter. The *effect* of an interactive object that has a local state of type *ls* and a process state of type *pst* is defined by means of a function of type $(ls, pst) \to (ls, pst)$. Such a function is called a *callback function*. For most interactive objects, the callback function is an *attribute* of the object. Attributes are also defined by means of algebraic data types. For instance, among many other control attributes, one can find the callback function attribute of controls:

```
:: ControlAttribute st
 = ... | ControlFunction (st->st) | ...
```

Note that the pair of local state and process state constitute the *state* of an element. The *meaning* of attributing a control element with a callback function f is that when that element is selected by the user, and the current state is the value (l, p), then the new state will be $(f(l, p))$. In other words, a callback function defines a *state transition*.

Besides having a bag of interactive objects the object I/O library provides programmers *glue* to construct user interfaces. This glue serves two purposes: *(a)* from primitive objects one can construct new composite objects, and *(b)* it puts restrictions on what components are 'glue compatible'.

The object I/O library has one universal glue *:+:* that can be used to connect two interactive objects that operate on the same local state of type *ls* and process state of type *pst*. Its type definition is as follows:

```
:: :+:  t1 t2 ls pst
= (:+:) infixr 9 (t1 ls pst) (t2 ls pst)
```

In order to define what components are compatible to be glued *type constructor classes* are applied. The type constructor class `Controls` contains all control elements (see Figure 2.1) but also defines that only `Controls` members can be glued:

```
instance Controls ButtonControl,
                  CheckControl,
                 CompoundControl c   | Controls c,
                 CustomButtonControl,
                 CustomControl,
                 EditControl,
                 LayoutControl c     | Controls c,
                 PopUpControl,
                 RadioControl,
                 SliderControl,
                 TextControl,
                 :+: c1 c2           | Controls c1 & Controls c2
```

This specification says that `ButtonControl`s, `CheckControl`s, and so on, belong to the same type constructor class, namely `Controls`. Three of these instances (`CompoundControl`, `LayoutControl`, and `:+:`) have additional type constructor variables. The only permissable type constructor for these variables should be an element of the `Controls` class. This is expressed by putting a type constructor context restriction on the variables (the expression followed after `|`). Let `button` and `text` below define a button and a text control respectively:

```
button = ButtonControl "Button"    []
text   = TextControl   "Just text" []
```

then the following expressions are all legal `Controls` instances: `button :+: button`, `button :+: text`, `text :+: button`, and `text :+: text`.

The collection of interactive objects that is supported by the object I/O library is ordered in four categories, called *abstract devices*.

Menus: Menus provide the set of commands that are available to the user of an interactive program. A program can have an arbitrary number of menus. Menus can be hierarchical, i.e. they can contain menus (*sub menus*) which can contain sub menus as well. Menu items correspond with the menu commands of the program.

Receivers: Receivers are the basic components that interactive objects can use to communicate messages in a flexible way.

Timers: Timers are used by a program to be able to *softly* synchronise actions. A timer basically triggers a callback function every passing of a given time interval.

Windows and Dialogues: Windows provide the primary interface element to the user of a program. Windows can either be *dialogues* or general purpose *windows*. Windows and dialogues can contain arbitrary collections of controls. Analogous to menus, these control collections can be hierarchical, i.e. they can contain collections of controls (*compound controls*), and so on.

2.2 How to manage running interactive objects

In the previous section we have had a glimpse of how to define (compositions of) interactive objects. In the object I/O library every interactive object can be *created* and *destroyed* dynamically, but we prefer to call this *opening* and *closing* which sounds more peacefully. Once an interactive object is running the program will need to *modify* it in several ways. Examples are to enable and disable menu elements, and change the content of a window to reflect the state of the program, and so on. Closing an element is the ultimate modification of a running interactive object. So interactive objects have a *life-cycle* which consists of three consecutive phases: *opening*, *modification*, and *closing*. Below we discuss these phases.

2.2.1 Opening of interactive objects

For each abstract device a function is defined that will open an instance, given a definition. Again type constructor classes are used to control what elements are proper instances of each abstract device. As an example, dialogues are defined by the following type definition:

```
:: Dialog c ls pst
 = Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]
```

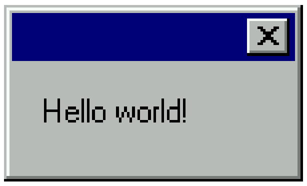
Following the naming convention of the object I/O library, the type variables `ls` and `pst` correspond with the local state of the dialogue and the process state of the program respectively. Callback functions are state transition functions of type $(ls, pst) \rightarrow (ls, pst)$, so the window attribute type constructor is parameterised with (ls, pst). The type constructor class `Dialogs` fixes the instances of dialogues. A (`Dialog c`) is a proper instance of this class, provided that `c` is a proper instance of the `Controls` type constructor class.

```
class Dialogs ddef where
  openDialog :: .ls !(ddef .ls (PSt .l)) !(PSt .l)
             -> (!ErrorReport,!PSt .l)
  ...

instance Dialogs (Dialog c) | Controls c
```

In Subsection 2.2.2 we will look more closely at the process state argument `PSt`.

The purpose of the *open* function of every abstract device is to map the *definition* of an interactive object to the *'physical'* graphical user interface object. It is this 'physical' object that can be manipulated by the user of a program (in case of windows, dialogues, and menus) or by the program itself (all abstract devices). The open function returns a value of type `ErrorReport` because it is possible that creating an interactive object fails (lack of system resources or logical inconsistencies). It is good programming practice to check for errors and provide proper feedback to the user.

Let's illustrate the use of the `openDialog` function. Suppose we want to open the following small and not very useful dialogue:



The program line that does this typically looks something like this:

```
    (error,new_process_state)
```

```
        = openDialog
              my_local_state
              (Dialog "" (TextControl "Hello world!" []) [])
              the_process_state
```

2.2.2 Modification of interactive objects

Once an interactive object has been opened and is in its running phase, it can be modified by the user and the program. For this purpose a running interactive object must be *stored* somewhere, and it must be *identified* by the program. Every running interactive object is stored in the *I/O state* of a program. In Section 2.1 we explained that every interactive object has access to a *local* state and a *process* state. The process state is a structured value defined by means of the record type PSt:

```
:: *PSt l               // The process state record type
 = {  ls :: !l          // The local process state
   ,  io :: !*IOSt l    // The I/O state
   }
:: *IOSt l              // IOSt is an abstract data type
```

A value of abstract type IOSt is created specifically for each interactive process by the object I/O system. In this special value the state of every running interactive object is stored. The other part of a process state, the ls field, can be defined freely by the programmer. In this field, the programmer can store all the data that is required by all interactive objects during the life-cycle of an interactive process. The example programs in this tutorial are usually very small. For such small programs for which no sensible 'logical' state has to be defined, the object I/O library provides the convenient type constructor Void:

```
:: Void = Void
```

The running state of every interactive object is stored in the IOSt. So, a function that modifies the running state of such an interactive object is usually of type $(IOSt\ l) \rightarrow (IOSt\ l)$. Some functions such as the abstract device open functions require the PSt record, and consequently have type $(PSt\ l) \rightarrow (PSt\ l)$. Because in general the IOSt will contain an arbitrary number of windows, dialogues, menus, timers, and receivers one also has to specify which particular interactive object one intends to modify. For this purpose interactive objects (and their component objects) can be identified by means of *Ids*. An Id is an abstract type that is generated by the object I/O system. An interactive object is identified by means of a specific Id by adding this Id to the *attribute list* of the corresponding object definition (see Section 2.1). As an example, for controls the corresponding attribute is:

```
:: ControlAttribute st
 = ... | ControlId Id | ...
```

The major part of the object I/O library defines the modification functions that the programmer can use to modify a running interactive object.

As an example, suppose we want to change the content of the text control of the "Hello world!" example to "Goodbye world!". To do this, we need to identify the text control. A control is identified uniquely by its personal `Id`. So we add a `ControlId` attribute to the attribute list of the text control. If `textid` identifies the text control then the corresponding program fragments look something like:

```
   (error,new_process_state)
    = openDialog
       my_local_state
       (Dialog ""
           (TextControl "Hello world!" [ControlId textid])
           []
       )
       the_process_state

   changeText process_state=:{io}
    = {process_state & io=setControlText textid "Goodbye world!" io}
```

2.2.3 Closing of interactive objects

As explained in the previous two subsections, once an interactive object has been opened it is in its running phase and it will remain in that state until it is explicitly closed by the program. Note that although it may seem to the user that he is able to close a window, it is actually the program that responds to a *user request* to really close a window. For all abstract devices and their components, there are close operations. A close operation will remove the 'physical' graphical user interface element and free system resources that were required to operate the interactive object properly. In addition, it is also removed from the `IOSt`. Closing an interactive object also closes its component objects, so closing a dialogue automatically closes all controls that are part of the dialogue.

2.3 How to start an interactive program

The starting point of every Clean program (interactive or not) is the `Start` function. The essence of an interactive program is that it is a function that can change the world. So for interactive programs the `Start` function must have type `*World` → `*World`. The typical appearance of the `Start` function of an interactive program looks something like:

```
Start :: *World -> *World
Start world
   = ... world
```

In the previous sections we have seen how to define interactive objects and get them running. The abstract device open functions require a `PSt` value, containing an `IOSt` value. These are created by the object I/O system using the function `startIO`.

```
startIO :: !DocumentInterface !.l !(ProcessInit (PSt .l))
                              ![ProcessAttribute (PSt .l)]
           !*World -> *World

:: ProcessInit pst :== IdFun pst
:: IdFun        st :==  st -> st
```

The details of this function will be explained in Chapter 11. Briefly, it creates an *initial* process state given the initial local process state of type *l*. This initial process state will contain a tailor-made `IOSt` value. In this way one switches from the world environment to the process state environment, and the interactive program can be *initialised*. This is done by the initialisation function (of type `(ProcessInit (PSt l))`). In this function the programmer can create the initial interactive objects that are required by the program.

After initialisation the interactive process will be evaluated. Evaluation means that the object I/O system takes care that the proper process state transition functions will be applied that are specified by the program. For instance, when the user presses a button or enters text in some text edit field, this generates low-level events. These events are interpreted by the object I/O system and mapped to *abstract events* such as “this button was pressed” and “this keyboard input was entered”. If the interactive object has a callback function (given as an attribute) then this function is applied to its current local state and process state. This returns a new local state and process state, with which the object I/O system continues.

The evaluation of an interactive process terminates as soon as it becomes *closed*. The only way for an interactive process to be closed is by means of the process modification function `closeProcess`:

```
closeProcess :: !(PSt .l) -> PSt .l
```

This function closes all currently running interactive objects of the interactive process and returns a process state that contains an *empty* `IOSt` value. All modification operations have no effect when applied to an empty `IOSt` value. If an interactive program consists of one interactive process, then the application of `closeProcess` suffices to terminate the evaluation of `startIO`. In Chapter 11 it is discussed how an interactive program can open and close interactive processes dynamically. Note that if an interactive program does not close all of its processes, it will run on forever.

2.4 My first Clean object I/O program

To complete the introduction we present the Clean object I/O version of the well known “Hello world!” program. Here it is:

```
module hello

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program creates a dialog that displays the "Hello world!" text.
//  ******************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO NDI Void initialise [] world
where
    initialise pst
        # (error,pst)       = openDialog Void hello pst
        | error<>NoError    = closeProcess pst
        | otherwise         = pst

    hello   = Dialog ""
```

```
(   TextControl "Hello world!" []
)
[   WindowClose (noLS closeProcess)
]
```

This program creates an interactive process which opens the same dialogue as shown earlier in Section 2.2.1. The singleton type `Void` introduced earlier is used to specify that this program has no interesting local process state (the second argument of `startIO`), and also no interesting local dialogue state (the first argument of `openDialog`).

The dialogue has one attribute, the callback function that should be applied in case the user wants to close the dialogue. In this case closing the dialogue will close the "Hello world!" program. So the callback function can simply be `closeProcess`. However, the type of a callback function of an interactive object also operates on a local state (which is `Void` in this case). To conveniently transform a function of type $(a \to b)$ into a function of type $(c, a) \to (c, b)$, the library function `noLS ::` $(a \to b)\ (c, a) \to (c, b)$ is applied.

Chapter 3

Global structure of the object I/O library

The *application programmer's interface* of the Clean object I/O library currently consists of 49 definition modules (they are all given in Appendix A). These modules contain everything you need to create interactive Clean programs with. *No other modules and no other symbols should be imported from the object I/O library.* Violation of this rule can result in error-prone applications at worst and non portability at least. In this chapter the module structure of the API is discussed. This will help you to find your way quickly in the object I/O library.

As a global naming convention, all definition modules have the prefix `Std`. The module `StdIO` is a convenience module that collects all modules that you normally need for graphical user interface programs. The module `StdTCP` is a convenience module that collects all modules required for TCP programming. The modules can be divided roughly into five major categories: *the abstract devices*, *interactive processes*, *drawing*, *channels* and *general.*

3.1 Abstract devices

Abstract devices have been introduced in Section 2.1 (page 13). These are the *menu*, *window*, *timer*, and *receiver* device. These devices occupy most of the modules and type definitions of the object I/O library. The following naming conventions have been employed for these modules:

- The names of the modules that contain type definitions to define abstract device instances have postfix `Def`. So *menu* definitions can be found in the module `Std`*Menu*`Def`, *receiver* definitions can be found in the module `Std`-*Receiver*`Def`, and so on. Although *controls* are not an abstract device, a definition module also exists for *controls*, namely `Std`*Control*`Def`.
- For each module that contains type definitions to define abstract device instances (so of the form `Std`*Object*`Def`) there also exists a module that defines basic access functions on its *attributes*. The corresponding name is formed as `Std`*Object*`Attribute`. So, *control* attribute access functions can be found in the module `Std`*Control*`Attribute`.
- Abstract device instances that consist of elements use type constructor classes to enumerate their elements. The corresponding type constructor classes and

standard instances are defined in the modules which names end with `Class`. So *menu elements* can be found in the module `Std`*MenuElement*`Class`. The *controls* can be found in `Std`*Control*`Class`.

- Receivers are non standard elements of some abstract device instances. Given the name *Object* of the parent device instance, you can find the type constructor class instance declarations in the modules which names are formed like `Std`*Object*`Receiver`. So, receiver instances of timers can be found in `Std`*Timer*`Receiver`.
- The operations on an object *Object* can be found in the module named `Std`*Object*. So *window* operations can be found in the module `Std`*Window*, *menu element* operations can be found in the module `Std`*MenuElement*, and so on.

Below we discuss the module structure of each of the abstract devices.

3.1.1 Menus

The API for menus and menu elements consists of six modules:

StdMenuDef
contains the type definitions of menus, pop up menus, and menu elements.

StdMenuAttribute
defines the access functions on menu (element) attributes.

StdMenuElementClass
defines the menu element instances for menus and pop up menus. These have different type constructor classes (`MenuElements` and `PopUpMenuElements` respectively) because pop up menus are not allowed to contain sub menus.

StdMenuReceiver
adds *receivers* as a proper instance of menu elements.

StdMenuElement
contains all operations on menu elements.

StdMenu
contains all operations on menus. Here also the `Menus` type constructor class is given together with its two instances `Menu` and `PopUpMenu`.

3.1.2 Windows

The API for windows, dialogues, and controls consists of eight modules:

StdWindowDef
contains the type definitions of windows and dialogues.

StdControlDef
contains the type definitions of controls. These can be placed in windows and dialogues.

StdWindowAttribute
defines basic access functions on window attributes.

StdControlAttribute
defines basic access functions on control attributes.

StdControlClass
defines the control instances for windows and dialogues. For windows and dialogues these belong to the `Controls` type constructor class.

StdControlReceiver
adds *receivers* as a proper instance of controls.

StdControl
contains all operations on controls.

StdWindow
contains all operations on windows and dialogues. Here also the `Windows` and `Dialogs` type constructor classes are given together with their instances `Window` and `Dialog`.

3.1.3 Timers

The API for timers consists of five modules:

StdTimerDef
contains the type definitions of timers and its elements.

StdTimerAttribute
defines basic access functions on timer attributes.

StdTimerElementClass
defines the timer elements instances. These are currently only glueing instances.

StdTimerReceiver
adds *receivers* as a proper instance of timer elements. This is actually the most interesting timer element.

StdTimer
contains all operations on timers.

3.1.4 Receivers

The API for receivers consists of three modules:

StdReceiverDef
contains the type definitions for uni-directional and bi-directional receivers.

StdReceiverAttribute
defines basic access functions on receiver attributes.

StdReceiver
contains all operations on receivers. Here you can also find the message passing functions.

3.2 Interactive processes

As stated in Section 2.3, every interactive program has to be opened as an interactive process. The API for interactive processes consists of three modules:

StdProcessDef
contains the type definitions for interactive processes.

StdProcessAttribute
defines basic access functions on process attributes.

StdProcess
contains all operations on processes including the functions mentioned in Chapter 2, `startIO` and `closeProcess`. The `startIO` function is actually a specialised version of the more general type constructor class `Processes` which allows you to create a topology of interactive processes.

3.3 Drawing

In graphical user interface applications graphics play an important role. Virtually every interface object has a visual representation that is drawn by the underlying platform. Drawing operations will be required by most applications to give the user visual feedback on the current documents that are being manipulated or the status of controls. The manipulation of text is also an issue in drawing information. Text can be presented in very different fonts, sizes, and variations. Drawing on screen and printing on paper is very analogous. With some care rendering can be abstracted from the output device.

The API for drawing consists of five modules:

StdPictureDef
contains the type definitions of drawable elements, fonts, colours, pen attributes, and some platform dependent constants.

StdPicture
contains all drawing operations on pictures. These are structured by means of the type constructor classes `Drawables` for drawing figures, and `Fillables` for filling figures. Picture attributes can be retrieved and modified.

StdBitmap
adds *bitmaps* as a proper instance of the `Drawables` type constructor class. Currently the set of bitmap operations is very limited.

StdPrint
defines very general printing functions. Printing is the same as drawing, except that the output is sent to a printer rather than some screen object.

StdPrintText
builds on `StdPrint` and offers some specialised functions to print text.

3.4 Channels

Internally, interactive processes that belong to the same interactive program can communicate arbitrary functional values by means of receivers (see Section 3.1.4).

To allow (interactive) Clean programs to communicate with other applications an extensive TCP/IP API is provided. *Channels* are the medium for communication.

The API for using channels consists of five modules:

StdChannels
: contains operations to send and receive on channels.

StdTCPDef
: contains various type definitions needed for event driven communication.

StdTCPChannels
: defines instances for most of the classes defined in `StdChannels`.

StdStringChannels
: defines instances to send and receive strings.

StdEventTCP
: contains functions for using event driven TCP.

3.5 General

Finally, there are a number of modules that are less easily categorised. These are the following eleven modules:

`StdClipboard`
: In this module *clipboard* operations are defined. Clipboard operations are defined in more detail in Chapter 12.

`StdFileSelect`
: In this module two functions are defined by which a user can open platform dependent directory browsing dialogues.

`StdId`
: In this module the *identification* value generating functions are defined. This is discussed in more detail in Chapter 4.

`StdIOBasic`, `StdIOCommon` **and** `StdKey`
: These modules provide a lot of type definitions and functions that are needed by many of the abstract device modules introduced in Section 3.1.

`StdMaybe`
: In this module a type is introduced that is very useful for providing optional results and optional arguments. It is used by many operations in the abstract device modules.

`StdPSt` **and** `StdPStClass`
: In these modules operations are collected on the process state that are not related to any abstract device. `StdPSt` contains several type constructor classes for file and time access. Their instances can be found in `StdPStClass`. Other frequently used functions in `StdPSt` are the 'lifting' functions defined on the process state. With these lifting functions one can easily transform for instance an `IOSt` transition function to a `PSt` transition function.

`StdSound`
: This is an experimental module introduced to provide access to sound files.

`StdTime`

In this module some time access operations can be found that can be used independently of the timer device.

Chapter 4

Object identification

Before we can really delve into the details of the object I/O library we first need to learn how to identify running interactive objects. As we have briefly discussed in Section 2.2.2, all objects can have an *identification attribute*. An identification attribute is a value of type `Id`. Because attributes are optional, the programmer is not forced to provide a value for this attribute. Objects without identification attribute can not be modified at run-time. If the program needs to modify an interactive object, it must have been provided with an identification attribute.

The type `Id` is an abstract data type, and you can import it via the module `StdId`. *All* `Ids` are generated by the system. The type constructor class `Ids` defines the creation functions. `Ids` can be created from the `World`, `IOSt`, and the `PSt` environment. Every new `Id` taken from these environments is guaranteed to be fresh with respect to the other `Ids` generated by any of these functions.

```
::  Id

class Ids env where
    openId   ::      !*env -> (! Id,       !*env)
    openIds  :: !Int !*env -> (![Id],      !*env)

    openRId  ::      !*env -> (! RId m,    !*env)
    openRIds :: !Int !*env -> (![RId m],   !*env)

    openR2Id ::      !*env -> (! R2Id m r, !*env)
    openR2Ids:: !Int !*env -> (![R2Id m r],!*env)

instance Ids World,
             IOSt .l,
             PSt  .l
```

As the `Ids` class definition shows, `Id` values are not the only identification values that are used in the object I/O system. Two special kinds of identification values of type (`RId m`) and (`R2Id m r`) are used to identify receivers that accept messages of type `m` in the first case, and in addition reply with a message of type `r` in the second case. Receivers are discussed in Chapter 10.

The purpose of having `Ids` is to unambiguously identify running interactive objects. When assigning `Ids` to interactive objects, a program must comply to the `Id` *assignment rule*: at all times during the life-time of an interactive program, an identification value (`Id`, `RId m`, or `R2Id m r`) must be bound to *at most one* running interactive object.

The abstract device open functions check whether the interactive object definition argument is valid with respect to the `Id` assignment rule. If this is not the case, the `ErrorReport` alternative `ErrorIdsInUse` is returned, and the interactive object will not be opened. If the `Id` assignment rule is not violated, and no other error occured, then the alternative `NoError` is returned.

For simple programs such as the "Hello world!" program in Section 2.4 no identification values have to be created. For somewhat longer programs that fit easily inside one module (such as most examples in this tutorial) it is usually most convenient to create the required identification values in the `Start` function and pass these to the *initialisation* function argument of `startIO` (Section 2.3). In the left-hand side of the initialisation function one can introduce names to these identification values that can then be used in all locally defined callback functions. So the outline of such a program is as follows:

```
module name_of_program

import StdEnv, StdIO                                // Import of standard libraries

Start :: *World -> *World
Start world
    # (ids,world)   = openIds n world               // Create n Ids
    = startIO SDI Void (initialise ids) [] world    // Pass these to initialise

initialise :: [Id] (PSt .l) -> PSt .l
initialise [id1,id2,...idn] pst                     // Introduce names for all n Ids
    = ...                                           // Create interactive objects
where
    ...
    callback (local_state,pst)                      // Local callback function
        = ... id1 ... id2 ...                       // can now use the Ids
```

The advantage of this scheme is that if good names are given to the `Id` values it is easier to make no mistakes.

Another scheme that is more useful in case it makes less sense to define all callback functions locally to the initialisation function is to put the `Id` values in a record in the local process state. Here, the record field names take care of naming the `Ids`. One can define a record `MyIds` that contains all `Id` entries (possibly hierarchically). If program states become more involved it is also a good idea to define them as a record because this makes it easier to change the program in the future. So, one can also introduce some local process state record `MyState` containing `MyIds`. Although not strictly necessary, it is also convenient to define a function that creates a `MyIds` record from any environment that belongs to the `Ids` type constructor class. Such a function has the overloaded type:

```
openMyIds :: !*env -> (!MyIds,!*env) | Ids env
```

This gives you more flexibility where to actually create the `MyIds` record because this function can be applied to any environment instance of the `Ids` type constructor class. The outline of a program using this scheme is as follows:

```
module name_of_program

import StdEnv, StdIO                                // Import of standard libraries

::  MyIds                                           // The record that stores Ids
```

```
 =  {   id1     :: Id
    ,   id2     :: Id
    ,   ...
    ,   idn     :: Id
    }
::  MyState                                     // The local process state
 =  {   myIds   :: MyIds                        // contains the MyIds record
    ,   ...
    }

openMyIds :: !*env -> (!MyIds,!*env) | Ids env      // openMyIds creates the Ids
openMyIds env
    # ([id1,id2...idn],world)   = openIds n world   // Create n Ids
    = ({id1=id1,id2=id2,...,idn=idn},env)           // Create MyIds record

Start :: *World -> *World
Start world
    # (myIds,world) = openMyIds world               // Create MyIds record
    # myState       = {...myIds=myIds...}           // in the local process state
    = startIO SDI myState initialise [] world

callback :: (LocalState,PSt MyState) -> (LocalState,PSt MyState)
callback (ls,pst=:{ls={myIds}})                     // Every callback function
    = ... myIds.id1 ... myIds.id2 ...               // can now use the Ids
```

Chapter 5

Drawing

In a graphical user interface most of the information that is presented to users is drawn, such as the shape of windows, dialogues, controls, but also handling of text. This central issue is handled in this chapter. All drawing functions require an environment of type `*Picture`. A `Picture` is created for each interactive object that can be drawn in. The life-cycle of a `Picture` environment is equal to the life-cycle of its parent object. Also for printing a `Picture` environment is used (printing is discussed in detail in Chapter 13).

A `Picture` defines a coordinate system for drawing operations (see figure 5.1).

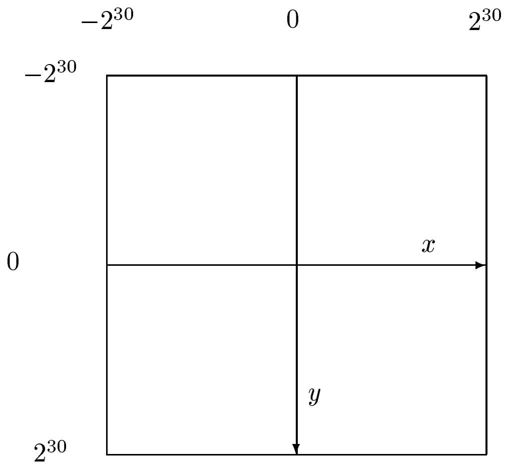


Figure 5.1: Picture coordinates.

X-axis coordinates increase from left to right, Y-axis coordinates increase from top to bottom. The range for both axes is defined by the macro `viewDomainRange` that can be found in module `StdIOCommon`:

```
viewDomainRange :== { corner1 = {x = 0-(2^30),y = 0-(2^30)}
                    , corner2 = {x =    2^30 ,y =    2^30 }
                    }
```

Drawing operations on a `Picture` use the coordinate system to define where objects should be drawn. The objects themselves are made up of the pixels, lying between the coordinates. Figure 5.2 zooms in on the coordinate system from zero and increasing. The pixels on x-coordinates 0, 5, 10,... and y-coordinates 0, 5, 10,... are displayed.

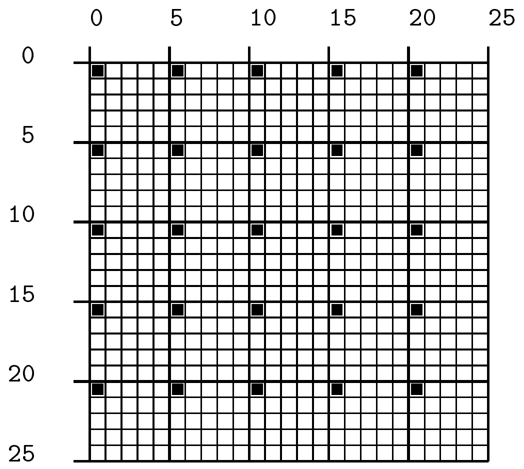


Figure 5.2: Picture coordinates and pixels

5.1 Pen attributes

Drawing a figure is done using a *pen*. The pen determines which pixels should be drawn and in what colour. Like most other elements in the object I/O library, pens have attributes. These are the following (they can be found in module `StdPictureDef`):

```
:: PenAttribute          // Default:
 = PenSize   Int         // 1
 | PenPos    Point2      // zero
 | PenColour Colour      // Black
 | PenBack   Colour      // White
 | PenFont   Font        // DefaultFont
```

`PenSize`
: This attribute defines the width and height of the drawing pen. The default value is 1, which means that drawing a point will fill an area of 1 pixel wide and 1 pixel high. Negative or zero values are always set to 1.

`PenPos`
: This attribute determines the current position of the drawing pen. Its default value is `zero`. The type definition of `Point2` is:

```
:: Point2 = {x::!Int,y::!Int}

instance zero Point2 where
   zero = {x=0,y=0}
```

`PenColour`
: This attribute determines the colour drawn pixels will have. The default value is black. The `Colour` data type is defined in module `StdPictureDef`:

```
:: Colour
   =   Black | DarkGrey | Grey | LightGrey | White
   |   Red   | Green    | Blue
   |   Cyan  | Magenta  | Yellow
   |   RGB RGBColour
:: RGBColour
   =   {r::!Int, g::!Int, b::!Int}
```

A colour can range between black and white (first five alternatives defining 100%, 75%, 50%, 25%, and 0% blackness), be one of red, green, blue, be one of cyan, magenta, yellow, or some custom defined combination of red, green, blue components. Currently the library does not support colour tables or palette management operations, so the use of RGB colours tends to be speculative.

`PenBack`
: This attribute determines the colour erased pixels will have. The default colour is white.

`PenFont`
: This attribute sets the current font that will be used when drawing text. Drawing text is not affected by the current width and height of the pen.

The current individual pen attributes can be set and retrieved by corresponding functions `setPen`*Attribute* and `getPen`*Attribute* respectively. It is also possible to set several pen attributes at once using `setPenAttributes` and retrieve all current pen attributes with `getPenAttributes`. These functions can all be found in `StdPicture`.

5.2 Drawing classes

The drawing operations are divided into three groups, ordered by means of type constructor classes:

Drawables
: draws (`draw` and `drawAt`) or erases (`undraw` and `undrawAt`) its instances. These are characters, strings, vectors, ovals, curves, boxes, rectangles, polygons, and bitmaps.

```
class Drawables figure where
    draw    ::         !figure !*Picture -> *Picture
    drawAt  :: !Point2 !figure !*Picture -> *Picture
    undraw  ::         !figure !*Picture -> *Picture
    undrawAt:: !Point2 !figure !*Picture -> *Picture
```

Fillables
: fills (`fill` and `fillAt`) or erases (`unfill` and `unfillAt`) its instances. These are ovals, curves, boxes, rectangles, and polygons.

```
class Fillables figure where
    fill    ::         !figure !*Picture -> *Picture
    fillAt  :: !Point2 !figure !*Picture -> *Picture
    unfill  ::         !figure !*Picture -> *Picture
    unfillAt:: !Point2 !figure !*Picture -> *Picture
```

Hilites

fills the interior of its instances in such a way that the current picture content remains visible. Its instances are boxes and rectangles.

```
class Hilites figure where
    hilite  ::         !figure !*Picture -> *Picture
    hiliteAt:: !Point2 !figure !*Picture -> *Picture
```

Each of these type constructor classes allows its elements to be drawn at the *current* pen position or at an *absolute* pen position. Because of this reason the data type definition of most of these elements do not specify their location. Exceptions are rectangles, lines, and points.

5.3 Font and text handling

When working with text you frequently will want to know the dimensions of the text for layout purposes or simply to calculate the size of an element containing that text. The dimensions of a piece of text depends on two parameters:

- The *font* is an abstract value that describes the shape of a text. The usual way to identify a font is by its name, point size, and style variations.
- The *drawing environment* (`Picture`) determines the actual size in terms of a resolution dependent unit. The resolution of the screen is usually a lot smaller than the resolution of a laser writer.

Because there is a great variance of available fonts and drawing environments per machine writing a program that handles fonts properly requires some care. Font operations are ordered in three groups.

The first group of font operations return information about the currently available fonts. The function `getFontNames` returns a list of the names of all available fonts. Given an element of this list, the functions `getFontStyles` and `getFontSizes` return for that particular font the available style variations and sizes. Because in modern font management systems for many fonts no restriction exists on the size, the function `getFontSizes` is also parameterised with two bounding size arguments. Their type definitions are:

```
getFontNames ::          !*Picture ->(![FontName], !*Picture)
getFontStyles::          !FontName
                         !*Picture ->(![FontStyle],!*Picture)
getFontSizes ::!Int !Int !FontName
                         !*Picture ->(![FontSize], !*Picture)
```

The second group of font operations opens fonts. The function `openFont` creates a value of type `Font` given a font definition. A font definition is a record of type `FontDef` and is defined as follows:

```
:: FontDef
   =   { fName   :: !FontName
       , fStyles :: ![FontStyle]
       , fSize   :: !FontSize
       }
```

Because there are so many different font systems and style variations both font names and style variations are of type `String`. If you want to be sure that you are selecting an existing font use the functions of the first group. In any case, if the font definition argument of `openFont` does not correspond with an installed font, then it returns a `False` Boolean and a font value that is supposed to be the closest matching font available. If it succeeds to find a matching font it will return a `True` Boolean and that font. The type of `openFont` is:

```
openFont :: !FontDef !*Picture -> (!(!Bool,!Font),!*Picture)
```

There are two functions that open the font that is used by default in a document window (`openDefaultFont`) and the font that is used by the system for dialogs, controls, window titles and so on (`openDialogFont`). These fonts are of course always available. Their types are:

```
openDefaultFont :: !*Picture -> (!Font,!*Picture)
openDialogFont  :: !*Picture -> (!Font,!*Picture)
```

The last group of font operations is related to font and text metrics. Font metrics are retrieved by the functions `getPenFontMetrics` and `getFontMetrics`. The first retrieves the metrics of the current pen `Font`, while the second retrieves the metrics of the argument `Font`.

```
getPenFontMetrics::       !*Picture -> (!FontMetrics,!*Picture)
getFontMetrics   :: !Font !*Picture -> (!FontMetrics,!*Picture)
```

The metrics of a font consists of three height related values (*leading*, *ascent*, and *descent*), illustrated in Figure 5.3 and one width related value.

The width related value (*max. width*) is the maximum width of all characters in that particular font. For *non-proportional* fonts such as `Courier` this implies that the width of all characters is identical, so "`iii`" is just as wide as "`mmm`". For *proportional* fonts such as this text the width of characters can vary a lot. Compare for instance the width of the text "iii" with "mmm". The metrics of a font are collected in a record of type `FontMetrics`:

```
:: FontMetrics
   =   { fAscent  :: !Int  // The ascent  of the font
       , fDescent :: !Int  // The descent of the font
       , fLeading :: !Int  // The leading of the font
       , fMaxWidth:: !Int  // The max. width of the font
       }
```

The final functions in the last group of font operations are used for calculating the width of a (list of) character(s), and a (list of) string(s). As with the font metrics functions, these functions also come in pairs, one using the current pen `Font`, and the other using the argument `Font`.

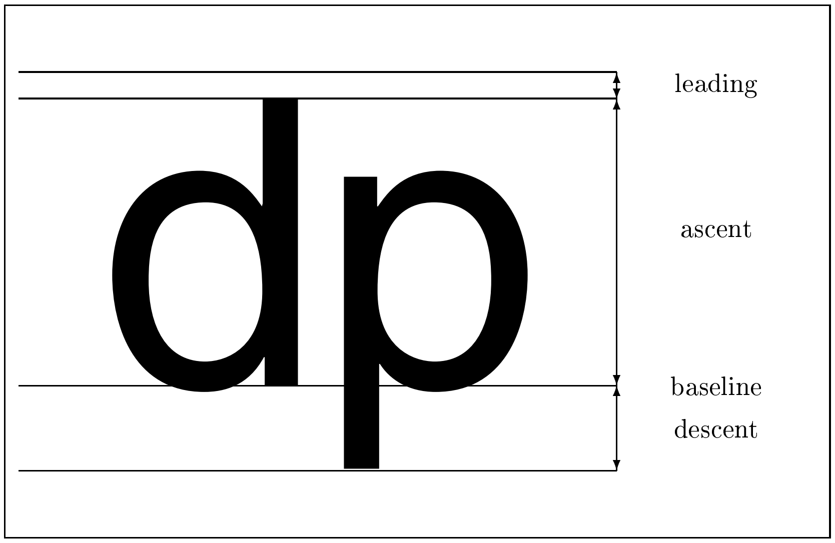


Figure 5.3: Font metrics.

```
getPenFontCharWidth   ::    ! Char    !*Picture ->(! Int, !*Picture)
getPenFontCharWidths  ::    ![Char]   !*Picture ->(![Int],!*Picture)
getPenFontStringWidth ::    ! String  !*Picture ->(! Int, !*Picture)
getPenFontStringWidths::    ![String] !*Picture ->(![Int],!*Picture)

getFontCharWidth   :: !Font ! Char    !*Picture ->(! Int, !*Picture)
getFontCharWidths  :: !Font ![Char]   !*Picture ->(![Int],!*Picture)
getFontStringWidth :: !Font ! String  !*Picture ->(! Int, !*Picture)
getFontStringWidths:: !Font ![String] !*Picture ->(![Int],!*Picture)
```

There is a subtle difference in calculating the width of *one character* versus the width of *one string*. The width of a character is determined by the character only. The width of a string can depend on the order of the characters it contains. A font system can take advantage of the fact that some adjacent characters can be placed more closely together to obtain a better looking result when drawing the string. This is called *kerning*. In the object I/O system, the programmer can rely on the fact that if a piece of text is drawn character by character then the character width function returns the correct width of the drawn character. If a piece of text is drawn by using a string, then the string width function returns the correct width of the drawn string.

Because there is wide variety of fonts available the `StdPictureDef` module provides a number of macros that help you make a program less dependent on the set of available fonts. The following macros provide a number of font definitions that are guaranteed to be available on the platform:

Font macros:	**Example:**
`SerifFontDef`	Garamond, Times
`SansSerifFontDef`	Helvetica
`SmallFontDef`	“This is a small text”
`NonProportionalFontDef`	`Courier`
`SymbolFontDef`	$\forall\ \exists\ \alpha\ \beta \Leftarrow \Rightarrow$

The following macros provide a number of standard font variations that are guaranteed to be available on the platform:

Style macros:	**Example:**
`ItalicsStyle`	*Madam, I'm Adam*
`BoldStyle`	**Madam, I'm Adam**
`UnderlinedStyle`	Madam, I'm Adam

5.4 Examples

In this section we give small examples of all of the drawable elements. Each of the examples is defined by a drawing function `example` of type $*Picture \rightarrow *Picture$. To actually get something on the screen one can use the following drawing framework program `drawingframe`. Figure 5.4 gives a snapshot of the framework program.

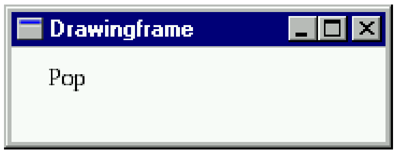


Figure 5.4: The framework window.

```
module drawingframe

//  **********************************************************************************
//  Clean tutorial example program.
//
//  This program defines a framework in which one can test drawing functions.
//  The program relies on a function, example, of type *Picture -> *Picture.
//  **********************************************************************************

import StdEnv,StdIO

Start :: *World -> *World
Start world
    = startIO SDI
              Void
              (snd o openWindow Void testwindow)
              [ProcessClose closeProcess]
              world

testwindow
    = Window "Test Drawing" NilLS
        [   WindowViewSize   size
        ,   WindowClose      (noLS closeProcess)
        ,   WindowLook       True (\_ _->example)
        ,   WindowViewDomain {corner1=origin,corner2=maxdomain}
        ]
where
    size        = {w=200,h=50}
    origin      = {x=(-20),y=(-20)}
    maxdomain   = {x=origin.x+size.w,y=origin.y+size.h}

// Here, example draws the string "Pop" at zero
example = drawAt zero "Pop"
```

5.4.1 Pen size and position

Given a pen position {x,y}, drawing a point, line, always occurs to the right and below the pen position. Figure 5.5 illustrates these cases: from left to right the following `example` functions are applied respectively:

```
example = drawPointAt zero o (setPenSize 1)
example = drawPointAt zero o (setPenSize 2)
example = drawPointAt zero o (setPenSize 3)
```

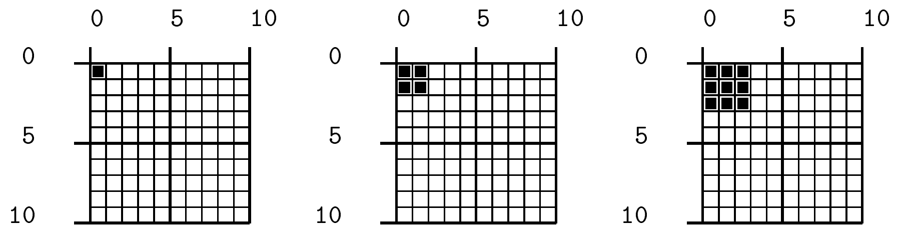


Figure 5.5: Drawing a point at zero with different pen sizes.

5.4.2 Drawing lines

The shape of a line is influenced by the `PenSize` attribute in the same way as the shape of points. Figure 5.6 shows the result of drawing lines with pen sizes 1, 2, and 3 respectively:

```
example = drawLine zero {x=5,y=5} o (setPenSize 1)
example = drawLine zero {x=5,y=5} o (setPenSize 2)
example = drawLine zero {x=5,y=5} o (setPenSize 3)
```

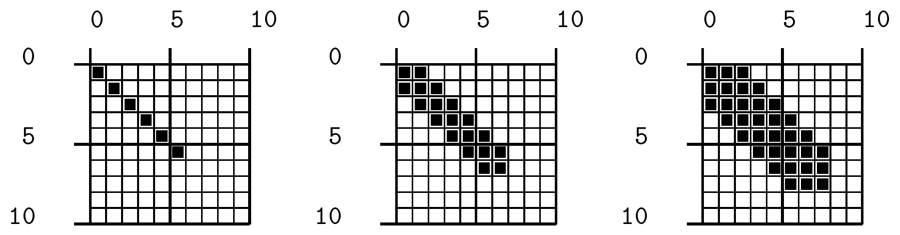


Figure 5.6: Drawing a line from zero to {x=5,y=5} with different pen sizes.

There are several ways to draw lines. The function `drawLineTo` draws a line from the current pen position to the argument point. If these happen to be equal, then the result is the same as `drawPointAt` with the same argument. The new pen position is the same as the target point. The function `drawLine` draws a line from the first argument point to the second argument point without changing the pen position.

Lines can also be drawn using the `Vector2` instance functions from the `Drawables` type constructor class. The function `draw` applied to a vector {vx,vy} draws a line from the current pen position `{x,y}` to the point `{x=x+vx,y=y+vy}`. The function `drawAt` applied to a point `{x,y}` and a vector `{vx,vy}` draws a line from `{x,y}` to the point `{x=x+vx,y=y+vy}`.

5.4.3 Drawing text

Given a pen position `{x,y}`, drawing a piece of text (`Char` or `String`) will always draw the text using the current `PenFont`. The shape of the drawn characters relies only on the font information, not on the current `PenSize`. Text can be drawn in any of the available colours. The baseline of the particular font determines the position of the first character, which is drawn with its left baseline starting at `{x,y}`. Figure 5.7 shows the result of the following `example` function:

```
example = drawAt zero "Pop"
```

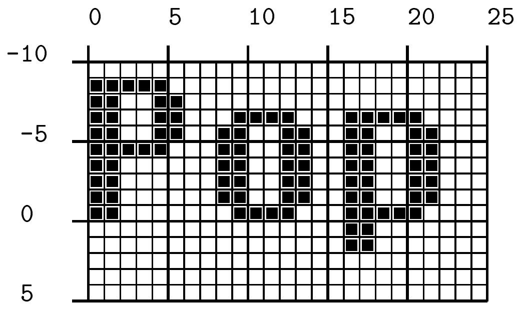


Figure 5.7: Drawing the text "Pop" at zero.

The new pen position is, in this case, `{x=24,y=0}`. One might expect the new pen position to be `{x=22,y=0}`, but usually the horizontal character space is also included. This facilitates drawing text character by character. But some caution should be taken. One might expect that the function

```
example = draw "p" o (draw "o") o (draw "P")
```

produces the same result, but this depends on the font (as explained in Section 5.3), so in general one should not assume that this is the case. The only certain way to know how much the pen position will change in case of text is by calculating the width of the *same* text, or by comparing the pen positions before and after drawing.

5.4.4 Drawing ovals

An `Oval` is a transformed unit circle defined by a horizontal radius, `oval_rx` and a vertical radius, `oval_ry`. For each point `{x,y}` on a unit circle, its corresponding point on the oval is given by `{x=x*oval_rx,y=y*oval_ry}`. The type definition of an `Oval` is:

```
:: Oval = {oval_rx::!Int, oval_ry::!Int}
```

Both radius values are always taken to be at least zero. If any of these values is negative, then zero is used instead. Ovals are drawn using the `Oval` instance functions from the `Drawables` type constructor class. The function `draw` uses the current pen position as the center of the oval. The function `drawAt` uses the argument `Point2` as the center of the oval. Drawing an oval does not change the pen position. In case one of the radius values is taken to be zero drawing the oval displays nothing. Figure 5.8 shows the result of drawing three ovals at `zero` defined as follows:

```
example = drawAt zero {oval_rx=5,oval_ry=3}
example = drawAt zero {oval_rx=5,oval_ry=5}
example = drawAt zero {oval_rx=3,oval_ry=5}
```

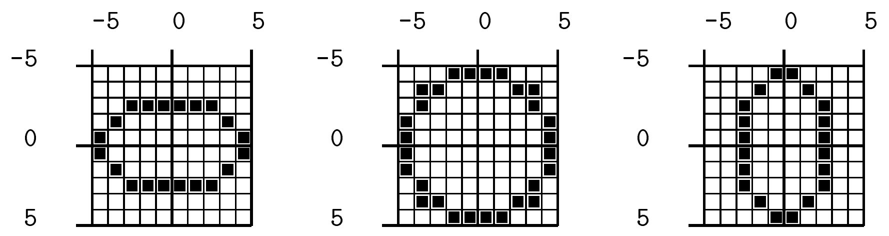


Figure 5.8: Three oval shapes drawn at zero.

The shape of an oval is also affected by the current `PenSize` attribute. Increasing the pen size does not increase the outline of the oval. The only pixels that are affected are inside the oval. Figure 5.9 shows the center oval of Figure 5.8 when drawn with pen size of 1, 2, and 3.

```
example = drawAt zero {oval_rx=5,oval_ry=5} o (setPenSize 1)
example = drawAt zero {oval_rx=5,oval_ry=5} o (setPenSize 2)
example = drawAt zero {oval_rx=5,oval_ry=5} o (setPenSize 3)
```

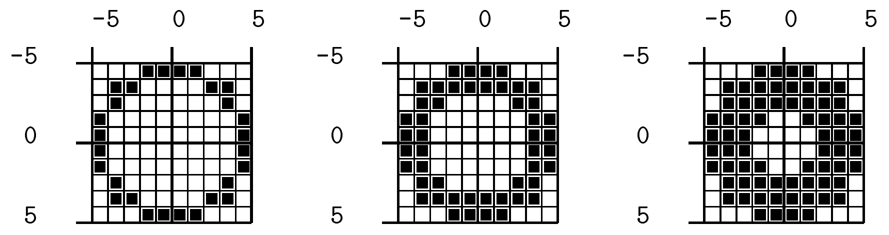


Figure 5.9: Three oval shapes drawn with increasing pen sizes.

`Oval`s are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the oval. Filling an oval includes its outline and its interior. Figure 5.10 shows the same three ovals as given in Figure 5.8, but now filled.

```
example = fillAt zero {oval_rx=5,oval_ry=3}
example = fillAt zero {oval_rx=5,oval_ry=5}
example = fillAt zero {oval_rx=3,oval_ry=5}
```

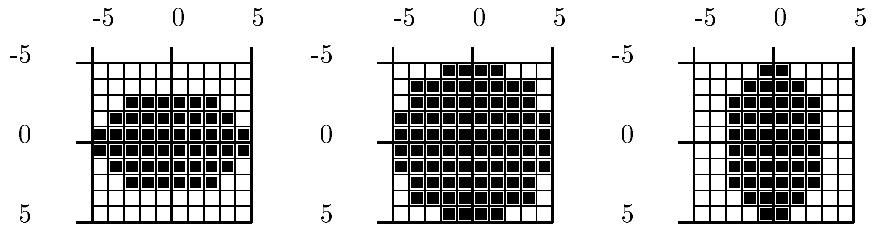


Figure 5.10: Three filled oval shapes.

5.4.5 Drawing curves

A `Curve` is a section of an `Oval`. A curve is defined by the source oval, `curve_oval`, a starting angle, `curve_from` and an ending angle, `curve_to`, both taken in radians, and the direction in which the section should be taken, `curve_clockwise` which is a Boolean value. The type definition of a `Curve` is:

```
:: Curve
   =   {   curve_oval      :: !Oval
       ,   curve_from      :: !Real
       ,   curve_to        :: !Real
       ,   curve_clockwise :: !Bool
       }
```

The start and end point of the section are again derived from the unit circle. Given an angle `alpha`, and a source oval defined by `{oval_rx, oval_ry}`, then the point on the curve (oval) corresponding with `alpha` is `{x=oval_rx*cos alpha, y=oval_ry*sin alpha}`. If `curve_clockwise` is `True` then the section is taken clockwise from the start point to the end point, otherwise it is taken counter clockwise. Figure 5.11 shows two sections of an `Oval`. In both cases the `curve_from` angle is $\frac{\pi}{6}$ and the `curve_to` angle is $\frac{3\pi}{2}$. The left section is taken counter clockwise, and the right section is taken clockwise. For your convenience, the value π is approximated in the module `StdPictureDef` by the macro `PI`.

```
curve   = { curve_oval      = {oval_rx=5,oval_ry=3}
          , curve_from      = PI/6.0
          , curve_to        = 3.0*PI/2.0
          , curve_clockwise = True
          }
example = drawAt zero  curve
example = drawAt zero {curve & curve_clockwise = False}
```

Figure 5.11 not only shows the curve sections that are taken from an oval, but also what happens when these sections are drawn *at* a specific position. In both cases the curves are drawn at `zero`. The starting point, indicated by the starting angle, is determined by the current pen position in case of the `draw` function of the `Drawables` type constructor class, and is determined by the `Point2` argument of the `drawAt` function of the `Drawables` type constructor class.

Drawing a `Curve` with varying `PenSizes` is the same as taking the section of the corresponding `Oval` drawn with that pen size. Figure 5.12 shows three times the same curve taken but drawn with pen sizes 1,2, and 3 respectively. The source oval

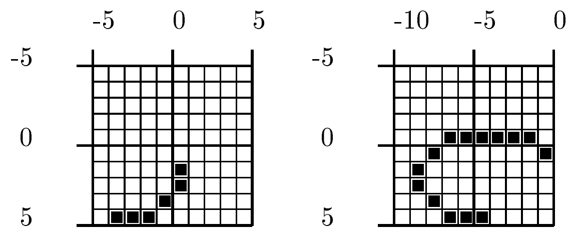


Figure 5.11: Two curves taken clockwise and counter clockwise.

is the same as the one drawn in Figure 5.9. The section is taken counter clockwise from $\frac{\pi}{4}$ to $1\frac{3}{4}\pi$.

```
oval    = { curve_oval      = {oval_rx=5,oval_ry=5}
          , curve_from      = PI/4.0
          , curve_to        = 1.75*PI
          , curve_clockwise = False
          }
example = drawAt zero oval o (setPenSize 1)
example = drawAt zero oval o (setPenSize 2)
example = drawAt zero oval o (setPenSize 3)
```

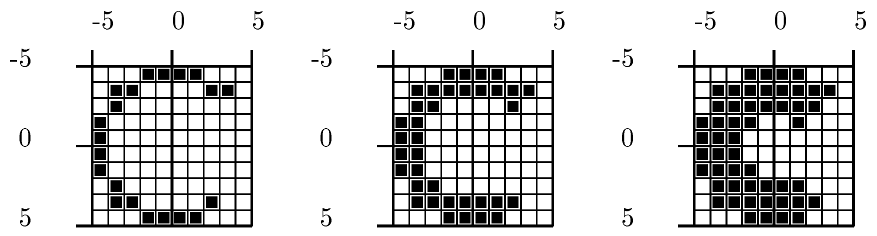


Figure 5.12: Three curves drawn with increasing pen sizes.

`Curve`s are also an instance of the `Fillables` type constructor class. When filling a curve, the interior formed by the drawn curve and two lines connecting the center of the source oval and the end points of the curve is filled. Figure 5.13 shows the two curves of Figure 5.11, but now using `fill` rather than `draw`.

```
curve   = { curve_oval      = {oval_rx=5,oval_ry=3}
          , curve_from      = PI/6.0
          , curve_to        = 3.0*PI/2.0
          , curve_clockwise = True
          }
example = fillAt zero  curve
example = fillAt zero {curve & curve_clockwise = False}
```

5.4.6 Drawing rectangles

A `Rectangle` is a shape of four connected lines that is defined by two diagonally oriented corner `Point2`s, `corner1` and `corner2`. The type definition of a `Rectangle` is as follows:

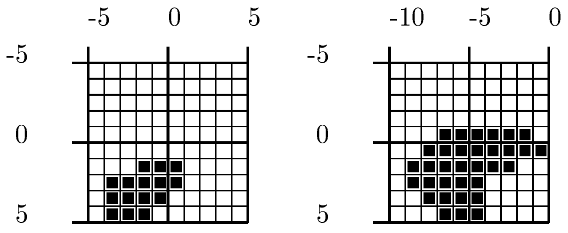


Figure 5.13: Two filled curves taken clockwise and counter clockwise.

```
:: Rectangle = {corner1::!Point2, corner2::!Point2}
```

The `Rectangle` type constructor is an instance of the `Drawables` type constructor class. The `drawAt` function is not very useful because it ignores its `Point2` argument and proceeds as `draw`. Any two `Point2`s are valid corner points of a `Rectangle`. In case a `Rectangle` has a zero width or zero height drawing that rectangle will show nothing. It does not matter in what order the two corner points are given. This is illustrated by the following `Rectangle` definitions, displayed in Figure 5.14.

```
example = draw {zero & corner2={x=10,y=6}}
example = draw {zero & corner2={x=6, y=6}}
example = draw {zero & corner1={x=10,y=6}}
```

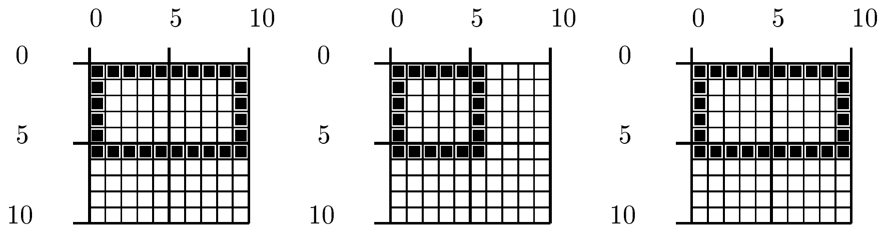


Figure 5.14: Three rectangle shapes.

The shape of a rectangle is also affected by the current `PenSize` attribute. Increasing the pen size does not increase the outline of the rectangle. The only pixels that are affected are inside the rectangle. Figure 5.15 shows the `Rectangle {corner1=zero, corner2={x=10,y=10}}` when drawn with pen size 1, 2, and 3.

```
example = draw {zero & corner2={x=10,y=10}} o (setPenSize 1)
example = draw {zero & corner2={x=10,y=10}} o (setPenSize 2)
example = draw {zero & corner2={x=10,y=10}} o (setPenSize 3)
```

`Rectangles` are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the rectangle. Filling a rectangle includes its outline and its interior. Figure 5.16 shows the same three rectangles as given in Figure 5.14, but now filled.

```
example = fill {zero & corner2={x=10,y=6}}
example = fill {zero & corner2={x=6, y=6}}
example = fill {zero & corner1={x=10,y=6}}
```

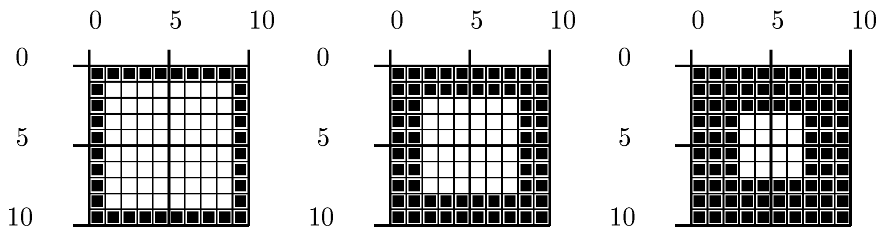


Figure 5.15: A rectangle drawn with increasing pen size.

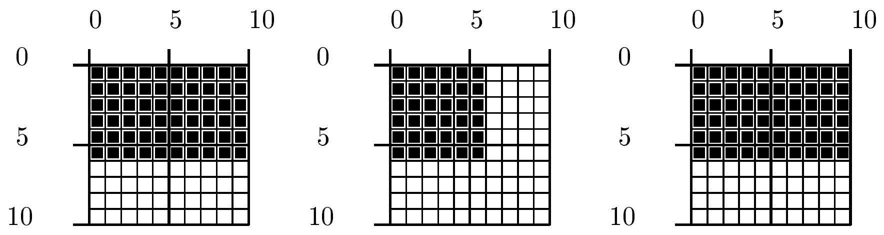


Figure 5.16: Three filled rectangles.

5.4.7 Drawing boxes

A `Box` is a `Rectangle` but without fixing a position. It is therefore only defined by a width, `box_w` and height, `box_h`. The type definition of a `Box` is:

```
:: Box = {box_w::!Int, box_h::!Int}
```

The position of a `Box` is determined by the drawing functions of the type constructor class `Drawables`. In case of `draw`, the current pen position is the base point. In case of `drawAt`, the `Point2` argument is the base point. Given this base point `base={x,y}`, and a box `{box_w,box_h}`, drawing the box is the same as drawing the rectangle `{corner1=base, corner2={x=x+box_w,y=y+box_h}}`. Any value for `box_w` or `box_h` is permitted (so also zero or negative values).

`Box`es are drawn and filled in the same way as `Rectangle`s are. So the effect of using different `PenSize`s is the same as well as filling boxes.

5.4.8 Drawing polygons

A `Polygon` is an object which shape is formed by a number of `Vector2`s, such as triangles, rectangles, but also more exotic shapes. The type definition of a `Polygon` is:

```
:: Polygon = {polygon_shape::![Vector2]}
```

A `Polygon` is always a closed shape. A shape `polygon_shape` is closed if the following equation holds:

```
foldr (+) zero polygon_shape = zero
```

The object I/O library will always close the `polygon_shape` if this is not the case, so you don't have to worry about this. Drawing a polygon of shape `polygon_shape` is simply drawing the closed list of vectors in sequence:

```
seq (map draw polygon_shape)
```

Similar to `Boxes`, `Polygons` do not specify their location. Again, this is determined by the drawing functions of the type constructor class `Drawables`. In case of `draw`, the base point is defined by the current pen position. In case of `drawAt`, the base point is defined by the `Point2` argument. Figure 5.17 shows three polygons defined by the following shapes:

```
example = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=(-4),vy=8}]}
example = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=0,   vy=8}
                                     ,{vx=(-8),vy=0}]}
example = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=(-8),vy=8}
                                     ,{vx=8,   vy=0}]}
```

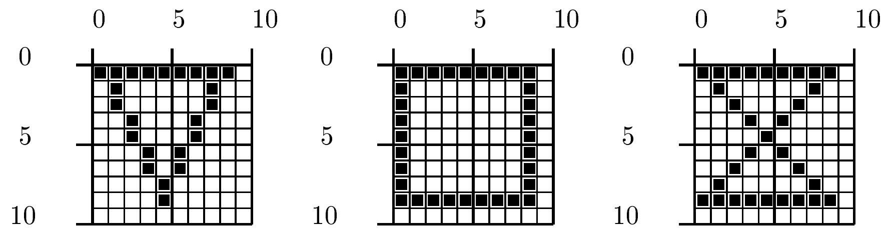


Figure 5.17: Three polygon shapes.

Because a polygon is a collection of vectors, its shape is affected by the current `PenSize` attribute. Figure 5.18 shows the three polygons of Figure 5.17 drawn with pen size 2.

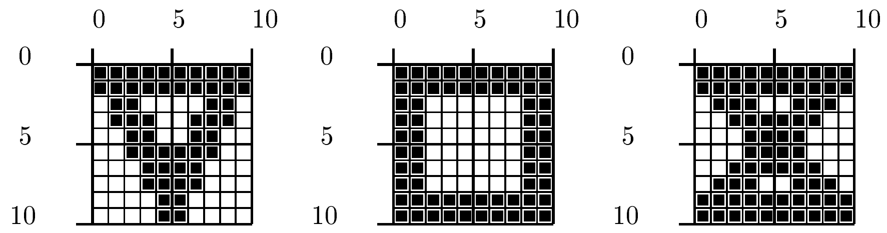


Figure 5.18: Three polygons drawn with pen size 2.

`Polygons` are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt` *fill* rather than draw the polygon. Filling a polygon includes its outline and its interior. Figure 5.19 shows the same three polygons as given in Figure 5.17, but now filled.

5.4.9 Drawing bitmaps

Using the drawing operations discussed so far one can produce images that have an 'algorithmic' nature: they consist of text, lines, curves, and polygons. Not every

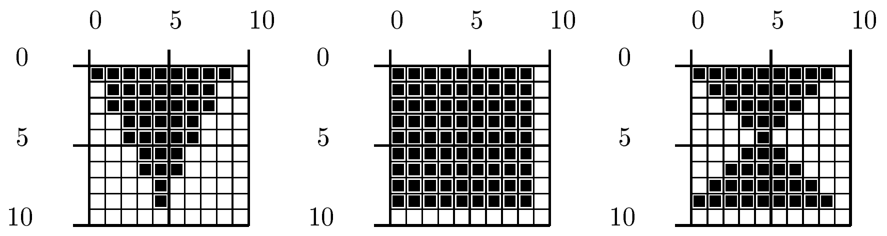


Figure 5.19: Three filled polygons.

image can be expressed (easily) in this way (consider for instance the image in Figure 5.20). To produce more complex images ***bitmaps*** are very useful. A bitmap is a prefabricated image of a certain size (in pixels) that is stored in the file system.

Figure 5.20: A non algorithmic image.

You can use your favourite drawing package and create and store images as a bitmap. The file format depends on the platform. Currently the following formats are supported:

Platform:	**Format:**
Macintosh	PICT
Windows(95/NT)	BMP

The bitmap operations can be found in module `StdBitmap` (Appendix A.1). Bitmaps can be read from file using the function `openBitmap`:

```
openBitmap::!{#Char} !*env -> (!Maybe Bitmap,!*env) | FileSystem env
```

Given the full path name of a bitmap file, `openBitmap` reads the bitmap in memory. If this is successful, `(Just bitmap)` is returned. Reasons for failure are illegal file name arguments, wrong file formats, lack of heap space (in case of Windows(95/NT)), or lack of extra memory (in case of Macintosh)[1].

A `Bitmap` is an abstract data type. `Bitmaps` can be drawn in any size you like. For this purpose you can change its size by using `resizeBitmap`. The only information one can retrieve from a `Bitmap` value is its size.

[1]In the current implementation bitmaps are not garbage collected. This puts a restriction on the number of bitmaps that can be used inside one application. In a future version bitmaps will be garbage collected.

```
resizeBitmap  :: !Size !Bitmap -> Bitmap
getBitmapSize ::       !Bitmap -> Size
```

`Bitmaps` are instances of the `Drawables` type constructor class. Given a current pen position `pos={x,y}` and a bitmap `bitmap` of size `{w,h}`, the functions `(draw bitmap)` and `(drawAt pos bitmap)` both place the bitmap exactly inside the rectangle `{corner1=pos, corner2={x=x+w,y=y+h}}`.

5.4.10 Drawing in XOR mode

For many programs it is sometimes useful to be able to temporarily draw figures over an existing drawing, and being able to remove them without affecting the source picture. Examples are flashing cursors, selection track boxes, and selection anchor points. For this purpose drawing in XOR mode is supported. The functions `appXorPicture` and `accXorPicture` handle this. Both apply their argument function to the argument picture in XOR mode. The latter function also allows its argument function to return a result.

```
appXorPicture:: !.(IdFun *Picture) !*Picture -> *Picture
accXorPicture:: !.(St *Picture .x) !*Picture -> (.x,!*Picture)
```

Drawing in XOR mode has the important property that drawing the same figure twice results in the same picture. Given a drawing function `f`, the following equations hold:

```
             (appXorPicture f) o (appXorPicture f) = id
     (snd o (accXorPicture f)) o (snd o (appXorPicture f)) = id
```

Let's explain what happens in the `Picture` when one uses XOR mode. Consider a source picture, `source`, shown left in Figure 5.21, which is a circle. Next to the source picture is the picture to be drawn in XOR mode, a fat rectangle, drawn by a drawing function `f`.

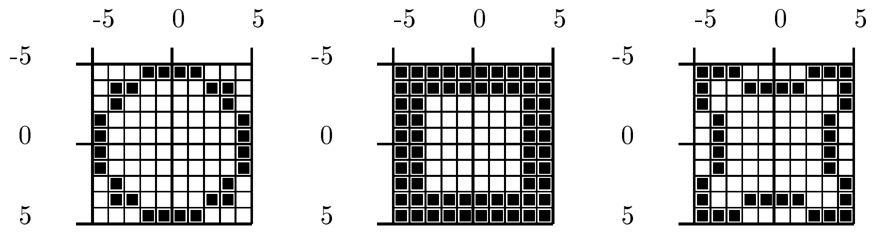


Figure 5.21: A source picture, the figure to be drawn in XOR mode, and the result.

If one interprets the white pixels of both pictures as `False`, and the black pixels of both pictures as `True` then the result of drawing `f` in XOR mode in `source` is the same as taking the Boolean exclusive or on all such interpreted pixels on the same coordinates. So all pixels that have the same colour become white (`False`), while all pixels of different colour become black (`True`). The result of this is shown in the right picture of Figure 5.21. Applying `f` once more in `xor` mode to this new picture yields a picture equal to `source`.

What happens when using more interesting colours than black and white is basically the same thing. In one way or another, the exclusive or is taken from the source picture and the drawing operations in such a way that repeating it gives the source picture again. What the colours of the 'xor-ed' picture are depends on the platform, and is not specified by the object I/O library.

5.4.11 Drawing in Hilite mode

Programs that want to indicate selections (for instance text segments in a word processor, or image components in a drawing program) can do this by drawing the selected area in *hilite* mode. For this purpose the type constructor class `Hilites` is used. Its type definition is:

```
class Hilites figure where
   hilite   ::         !figure !*Picture -> *Picture
   hiliteAt :: !Point2 !figure !*Picture -> *Picture
```

The instances of `Hilites` are `Box` and `Rectangle`. The pixels that are affected by `hilite` and `hiliteAt` are the same as for `fill` and `fillAt`. Drawing in hilite mode has the same property as drawing in XOR mode that drawing the same figure twice on a source picture leaves the source picture unchanged. Given a figure `figure`, the following equations hold:

```
(hilite   figure) o (hilite   figure) = id
(hiliteAt figure) o (hiliteAt figure) = id
```

The visual effect of hiliting these areas depends on the platform. On some platforms hiliting an area will change the colour of all pixels that have the background colour to a special hilite colour, ignoring all other pixels. Hiliting the area once more will revert the hilite colours back to the background colour. If a platform does not support hilite mode, the area will be drawn in XOR mode (shown in Figure 5.22). The source picture is the text "Pop" of Figure 5.7.

```
example = hilite {corner1={x=0,y=2},corner2={x=24,y=(-10)}}
        o (drawAt zero "Pop")
```

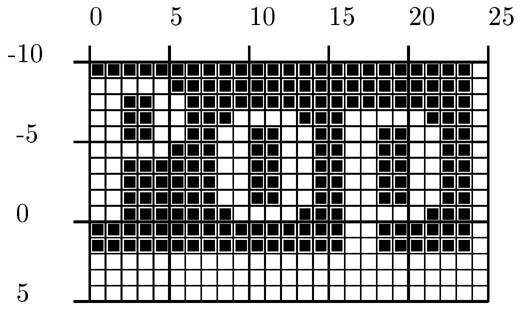


Figure 5.22: Hiliting a rectangular area using XOR mode.

5.4.12 Drawing in Clipping mode

Drawing in *clipping* mode is a powerful technique to create graphics that can not be drawn (or using much more complicated expressions) using only the drawing primitives discussed before. In clipping mode, the programmer specifies a *region* that works like a mask: drawing proceeds as described above, but only those pixels that are inside the clipping region are actually drawn.

A region is an abstract data type, `Region`. Regions are created by composing `Rectangles` and `Polygons`, using the type constructor class `toRegion`. When composing regions, using the list and `:^:` instance, the union of the argument regions is taken.

```
class toRegion area :: !area -> Region

::  PolygonAt
    =   {   polygon_pos :: !Point2
        ,   polygon     :: !Polygon
        }

instance toRegion Rectangle
instance toRegion PolygonAt
instance toRegion [r]         | toRegion r
instance toRegion (:^: r1 r2) | toRegion r1 & toRegion r2
```

Clipping is done using the functions `appClipPicture` and `accClipPicture`:

```
appClipPicture::!Region !.(IdFun *Picture) !*Picture->*Picture
accClipPicture::!Region !.(St *Picture .x) !*Picture->(.x,!*Picture)
```

Suppose we have the following drawing function, `f`, which draws a number of horizontal lines with a result as shown in Figure 5.23:

```
f = seq [drawLine {x=0,y=y} {x=9,y=y} \\ y<-[0,2..8]]
```

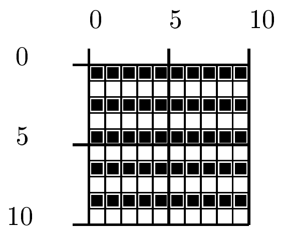


Figure 5.23: The source picture.

As clipping regions the polygons shown in Figure 5.17 are used. Figure 5.24 shows the result of the following clipping functions:

```
example = appClipPicture
            (toRegion { polygon_pos  =zero
                      , polygon_shape=[{vx=8,vy=0},{vx=(-4),vy=8}]
                      }
            ) f
example = appClipPicture
            (toRegion { polygon_pos  =zero
                      , polygon_shape=[{vx=8,vy=0},{vx=0,   vy=8}
```

```
                                                   ,{vx=(-8),vy=0}]
                       }
            ) f
example = appClipPicture
            (toRegion { polygon_pos  =zero
                       , polygon_shape=[{vx=8,vy=0},{vx=(-8),vy=8}
                                                   ,{vx=8,   vy=0}]
                       }
            ) f
```

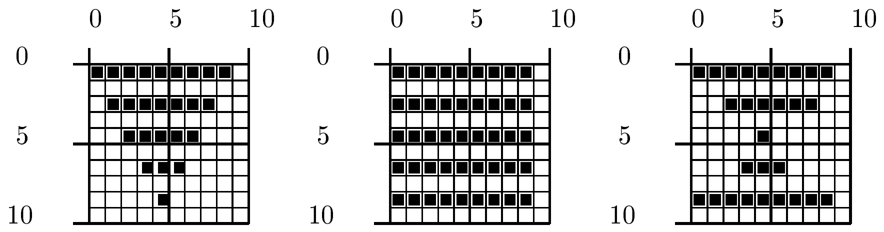


Figure 5.24: The clipped source picture.

Chapter 6

Windows and dialogues

In most graphical user interfaces the major top level interactive objects a user is confronted with are *windows* and *dialogues*. This is also the case for the object I/O library. Windows and dialogues share the same set of *controls* and many library functions. Dialogues are considered to be specialised windows. Depending on the platform, dialogues usually offer a special, enhanced, user interface. Another difference is that dialogues can be opened *modally*. In this mode the user is forced by the program to handle the dialogue completely before continuing with the program. To emphasize the similarity between windows and dialogues, their algebraic type definitions are almost identical (these can be found in module `StdWindowDef`):

```
:: Window c ls pst
 = Window Title (c ls pst) [WindowAttribute *(ls,pst)]
:: Dialog c ls pst
 = Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]
```

Before delving into details, we first introduce basic terminology for windows and dialogues in Section 6.1 and discuss their attributes in Section 6.2. Opening and closing of windows and dialogues is handled in Section 6.3. We then discuss the ways to handle the components windows and dialogues are made of in sections 6.4, 6.5, and 6.6. Handling keyboard and mouse input is presented in Section 6.7. Finally, this chapter is concluded with a treatment on modal dialogues in Section 6.8.

6.1 Basic terminology

The main purpose of a *window* is to present to the user a view on a document, graphically represented as an object of type `Picture`. `Picture`s have been discussed in Chapter 5. By using the mouse and keyboard, the user can manipulate the document. Controls in a window can add further manipulation functionality. The main purpose of a *dialogue* is to present to the user a structured way of passing information to perform actions. This structured communication is realised by means of controls.

6.1.1 Anatomy of windows and dialogues

Although from a user's perspective windows and dialogues appear to be 'solid' objects (Figure 6.1) it is illustrative to have a look at a window from a different perspective.

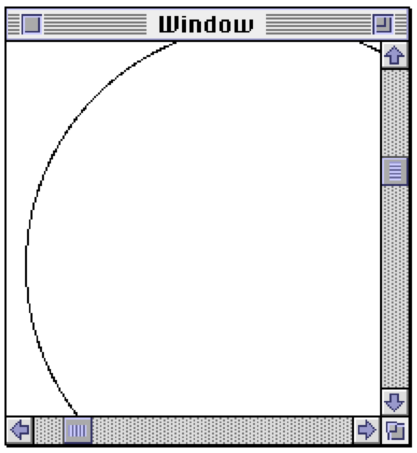


Figure 6.1: A window seen from the user perspective.

A window is composed of three layers, see Figure 6.2. The bottom layer, the *document layer*, is formed by the rendered document, the `Picture`. The middle layer, the *control layer*, contains all controls of the window. The top layer, the *window frame*, typically consists of a title bar, and window components to close and resize the window. The window frame can have any size and is restricted only by the screen size and a platform dependent minimum size. The window frame serves as a clipping area of the document layer. Windows usually contain scrollbars to help the user change the current view on the document layer. The current part of the document layer that is visible is called the *view frame*. The default drawing domain of the document layer `Picture` ranges from 0 to 2^{30} in both axes. This is in general to large for rendering the document. A window can limit the displayable range of a window by setting its *view domain*.

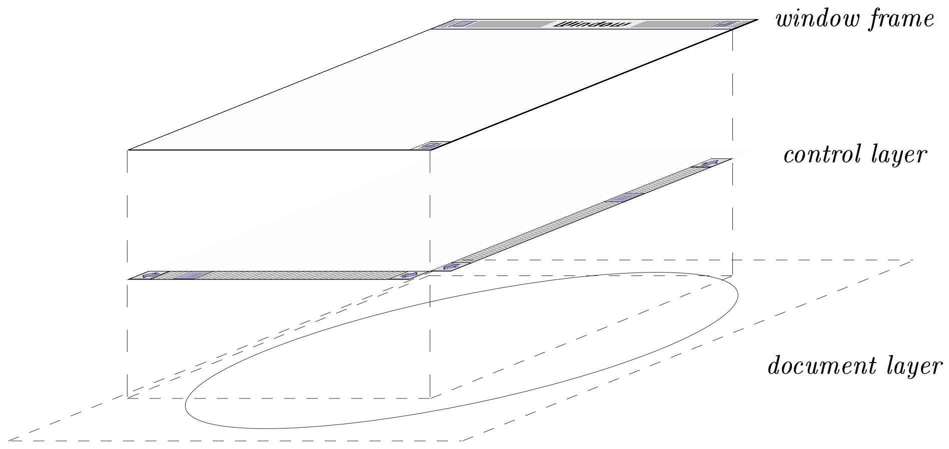


Figure 6.2: A different perspective at a window.

In contrast with windows, a dialogue is composed of only two layers, the control layer and the window frame, or also called the *dialogue frame*. Instead of a document layer, a dialogue has a platform dependent background. The program can not draw into nor navigate the background. The dialogue frame can not be resized by the user and is usually big enough to display the whole control layer.

For both windows and dialogues, the programmer has full control over the view frame, control layer, and document layer. Section 6.4 discusses how to handle the document layer. Handling the control layer is discussed in Section 6.5. The window and dialogue frame are controlled by the platform, see Section 6.6.

6.1.2 Stacking order

In general, a program can have an arbitrary number of windows and dialogues opened at the same time. These elements appear in a *stacking order*, seen by the application user in top to bottom order. For normal windows and dialogues the stacking order is not fixed. *Modal* dialogues however always appear topmost. Windows and modeless dialogues that are opened while (several) modal dialogues are open always appear below the bottom most modal dialogue.

6.1.3 Active window or dialogue

When a user works with a program, exactly one window or dialogue receives all keyboard and mouse input. This element is called the *active* window/dialogue and it has the *input focus*. With the exception of modal dialogues, the active window/dialogue does not necessarily occupy the top most stacking position.

6.2 Window and dialogue attributes

`WindowAttribute` is the type of window and dialogue attributes. The table below shows which attributes are valid for which element.

Window attributes		
For windows and dialogues:	**For windows only:**	**For dialogs only:**
`WindowActivate`	`WindowCursor`	`WindowCancel`
`WindowClose`	`WindowHScroll`	`WindowOk`
`WindowDeactivate`	`WindowKeyboard`	
`WindowHMargin`	`WindowLook`	
`WindowId`	`WindowMouse`	
`WindowIndex`	`WindowOrigin`	
`WindowInit`	`WindowPen`	
`WindowInitActive`	`WindowSelectState`	
`WindowItemSpace`	`WindowViewDomain`	
`WindowOuterSize`	`WindowVScroll`	
`WindowPos`		
`WindowViewSize`		
`WindowVMargin`		

6.2.1 Window and Dialog attributes

`WindowActivate` **and** `WindowDeactivate`

These attributes define the behaviour of the window in case the window becomes the active window (`WindowActivate`), and is no longer the active window (`WindowDeactivate`) respectively (see also 6.1.3). If no attribute is provided, this information will not be passed to the program.

`WindowClose`
: This attribute adds the platform dependent close facility to the window/dialogue. The associated function will be evaluated in case this control is triggered. Actually closing the window/dialogue is the responsibility of this function. In case no `WindowClose` attribute is provided, the window/dialogue can not be closed in that way.

`WindowHMargin` **and** `WindowVMargin`
: These attributes determine the left-right, and top-bottom margin of a window/dialogue respectively. Their default value in case of windows is zero, and platform dependent for dialogues.

`WindowId`
: This attribute identifies the window/dialogue to which it is associated. If you do not provide a `WindowId`, the object I/O system *open function* creates a fresh `Id` for the window/dialogue.

`WindowIndex`
: This attribute determines the initial stacking position of the window/dialogue (see also 6.1.2). If no `WindowIndex` attribute is provided, then the window/dialogue will be opened frontmost. Modal dialogues are always opened frontmost.

`WindowInit`
: This attribute defines an action that should be performed immediately after opening the window/dialogue. This is equivalent to the process initialisation action (see Section 2.3 and Chapter 11). If no `WindowInit` attribute is provided, no additional action is performed.

`WindowInitActive`
: This attribute defines which control should have the initial input focus when its parent window/dialogue is opened. If no `WindowInitActive` attribute is provided, the first keyboard sensitive control is selected.

`WindowItemSpace`
: This attribute determines the space between controls if no further offsets are provided in the layouts of controls. The default values are identical for windows and dialogues.

`WindowOuterSize` **and** `WindowViewSize`
: These two attributes determine the initial size of the window/dialogue. The first attribute defines the size in terms of the window/dialogue frame size. The second attribute defines the size in terms of the window/dialogue view frame size. If neither attribute is provided, then the system will derive a proper size. In case of dialogues the size is determined by the set of controls, margins, and item spaces. In case of windows the screen size is chosen.

`WindowPos`
: This attribute determines the initial position of the window/dialogue (see also Section 7.3). Relative `Ids` that occur in the `ItemPos` refer to other windows/dialogues. The object I/O system will always place a window/dialogue visibly on the current screen. If you do not provide a `WindowPos` to a window, then the window will be placed at the left top of the screen. If you do not provide a `WindowPos` to a dialogue, then the dialogue will be placed at a position conform the platform user interface (typically centered).

6.2.2 Window attributes

`WindowCursor`
: This attribute defines the shape of the cursor in case the mouse is over the window and not inside a control that may overrule this shape. In case no attribute is provided moving the mouse over the window will not change its shape.

`WindowHScroll` **and** `WindowVScroll`
: These attributes add a horizontal scrollbar and a vertical scrollbar to the window. If no attribute is given, no scrollbars are added.

`WindowKeyboard` **and** `WindowMouse`
: These attributes allow a window to respond to user actions with the mouse (`WindowMouse`) and keyboard (`WindowKeyboard`). If no attribute is provided, then this information will not be passed to the program. Both attributes can define an additional filter to ignore some input actions. If the `SelectState` of the window is `Unable` then neither function will obtain input.

`WindowLook`
: This attribute defines a function that, given the current `SelectState` of the window and information about which part of the window should be displayed, defines what the window should look like. The default look fills the window with the platform dependent background colour. (The `Look` function plays a similar role for `CustomButtonControls` (Section 7.1.3), `CustomControls` (Section 7.1.4), and `CompoundControls` (Section 7.1.11. It can also be used for printing (Chapter 13)).

  In addition to the `Look` function, the `WindowLook` attribute also has a boolean argument. If this attribute is `True`, then the object I/O system assumes that the content of the window is independent of the actual view frame orientation. In that case scrolling and changing the size of the window can be done more efficiently. If the attribute is `False`, then the `Look` function is always applied to the whole current view frame. See also Section 6.4.1 for more information on using this attribute.

`WindowOrigin`
: This attribute determines the initial position of the `ViewFrame`, the rectangular part of the `ViewDomain` that is currently visible in the window. This position is always verified to be within the given `ViewDomain`. If no `WindowOrigin` attribute is provided, then the left-top coordinates of the `ViewDomain` are used.

`WindowPen`
: This attribute determines the initial pen that is used to draw in the `Picture` of the window. If no `WindowPen` attribute is provided, then the default pen attributes are used (see also Section 5.1).

`WindowSelectState`
: This attribute defines whether the window accepts user keyboard or mouse input (`Able`) or not (`Unable`). The default value is `Able`. Note that although a window can be active, it can also be disabled. This only means that all input is ignored by the window.

`WindowViewDomain`
: This attribute defines the drawing coordinate system of the document layer (see also 6.1.1). Drawing operations outside this area will be clipped. If

no `ViewDomain` is provided, then the window will obtain the `ViewDomain` {`viewDomainRange & corner1=zero`}.

6.2.3 Dialogue attributes

`WindowCancel` **and** `WindowOk`

These attributes indicate which control should act conform the platform user interface 'cancel' control and 'confirm' control respectively. If such an attribute is not provided, then no control is selected.

6.3 Opening and closing of windows and dialogues

Windows and dialogues are opened using the appropriate instances of the respective type constructor classes `Windows` and `Dialogs` (module `StdWindow`, Appendix A.47). Windows and dialogues share the same set of controls. To enforce this by the type system, the instance declarations of windows and dialogues are provided with a type constructor class restriction on the type constructor variable `c`, which must be an instance of the `Controls` type constructor class.

```
class Windows wdef where
    openWindow :: .ls !(wdef .ls (PSt .l)) !(PSt .l)
                -> (!ErrorReport,!PSt .l)
    ...

class Dialogs ddef where
    openDialog :: .ls !(ddef .ls (PSt .l)) !(PSt .l)
                -> (!ErrorReport,!PSt .l)
    ...

instance Windows (Window c) | Controls c
instance Dialogs (Dialog c) | Controls c
```

To close windows and dialogues the function `closeWindow` has to be used. In many contexts one wants to close the *active* window or dialogue. For these cases the convenience function `closeActiveWindow` is provided.

```
closeWindow       :: !Id !(PSt .l) -> PSt .l
closeActiveWindow ::     !(PSt .l) -> PSt .l
```

A typical example of using `closeActiveWindow` is the callback function for the `WindowClose` attribute. If it is allowed to close the window or dialogue under all circumstances (nothing needs to be saved for instance) then the following attribute definition is sufficient:

```
WindowClose (noLS closeActiveWindow)
```

Note that `noLS` is used to obtain a function of the requested type $(.ls, PSt\ .l) \rightarrow (.ls, PSt\ .l)$.

6.4 Handling the document layer

The document layer of a window is used to present visual feedback to the user on the current status of the document that is being manipulated. Only windows have a document layer. Below we discuss two ways of drawing into the document layer. The first method, *indirect rendering* (Section 6.4.1), uses the look function, the second method, *direct rendering* (Section 6.4.2), draws directly into the document layer. Some programming pragmatics are discussed in Section 6.4.3. We conclude with an example in Section 6.4.4.

6.4.1 Indirect rendering

Two `WindowAttribute`s play a paramount role with respect to the document layer: `WindowViewDomain` and `WindowLook`. Let's have a closer look at them. Here are their type definitions (in module `StdWindowDef`):

```
::  WindowAttribute st
 =  ...
 |  WindowLook       Bool Look
 |  WindowViewDomain ViewDomain
 |  ...
```

The document layer is rendered using a `Picture`. As we have seen in Section 6.2, the default drawing range of a `Picture` is $(0, 2^{30})$ in both axes. This range can be changed by the `WindowViewDomain` attribute. It has a `ViewDomain` argument which is defined as a `Rectangle`:

```
::  Rectangle
 =  { corner1 :: !Point2, corner2 :: !Point2 }
::  Point2
 =  { x       :: !Int   , y       :: !Int    }
```

A `Rectangle` is a record consisting of the two diagonally opposite corner points of the new drawing range. The view domain can have any value that is between the `viewDomainRange` macro defined in `StdIOCommon`:

```
viewDomainRange :== { corner1 = {x = 0-(2^30),y = 0-(2^30)}
                    , corner2 = {x =    2^30 ,y =    2^30 }
                    }
```

The only illegal value is to have identical x or y coordinates for the corner points of a view domain. It causes your application to abort with the message *"Error in rule validateWindowDomain [windowvalidate]: Window has illegal ViewDomain argument"*.

The view domain defines the coordinates for your window to draw into. If a window has a view domain that is smaller than the available screen estate, it is possible that the size of a window exceeds the size of its view domain. So the view frame of a window may be larger than the view domain of the window. It is allowed to draw in the extra area as well. The coordinate system simply extends to the right and downwards. A window never displays an area that lies to the left or above the minimum view domain values.

The `Picture` of the document layer is rendered using the `WindowLook` attribute. This attribute has a `Look` function argument. It is defined as follows:

```
::  Look :== SelectState -> UpdateState -> *Picture -> *Picture
```

Whenever it is necessary to render (part of) the visible document layer, the object I/O system will apply the look function of the `WindowLook` attribute. It will be parameterised with the current `SelectState` of the window and detailed information about which part of the current view frame needs to be rendered. This information is presented by means of the `UpdateState` record:

```
::  UpdateState
 =  { oldFrame :: !ViewFrame
    , newFrame :: !ViewFrame
    , updArea  :: !UpdateArea
    }
::  ViewFrame  :== Rectangle
::  UpdateArea :== [ViewFrame]
```

The three fields of the `UpdateState` record contain the following information:

`oldFrame`
: If the size or orientation of the window was changed, then this field contains the view frame *before* that change. If this is not the cause, then this field contains the *current* view frame.

`newFrame`
: If the size or orientation of the window was changed, then this field contains the view frame *after* that change (so, the current view frame). If this is not the cause, then this field contains the *current* view frame.

`updArea`
: This field contains a list of `Rectangle`s (not necessarily disjoint) that define the parts of the visible *current* view frame that need to be drawn. This list always consists of at least one element. Each of its elements is always completely inside the current view frame. The remark that the rectangles need not necessarily be disjoint is important in case one draws in XOR, or *hilite* mode (see Sections 5.4.10 and 5.4.11). Drawing in overlapping areas using one of these modes results in undoing the intended effect.

Given the current `SelectState` of the window and the `UpdateState`, the look function is applied to the current `Picture` of the document layer. For this reason, the look function must be written in such a way that it works correctly for any argument `Picture`. You can not make any assumptions on the current content of the update area rectangles except that they are very unlikely to be correct.

In addition to the `Look` function, the `WindowLook` attribute has a `Bool` argument. If this value is `False`, then you have full control over drawing in a window. For instance, when the size of a window is decreased, the `UpdateState` passed to the `Look` will contain in its `oldFrame` the larger view frame before resizing, and in `newFrame` the decreased view frame after resizing. In addition, the `updArea` will consist of one rectangle only, equal to `newFrame`. In this way you can let the content of the image depend on the current window view frame. The following `Look` function draws a box the size of its new view frame and its two diagonal lines (see figure 6.3):

```
crossbox _ {oldFrame,newFrame} picture
```

```
    # picture = seq (map undraw (crossboxlines oldFrame)) picture
    # picture = seq (map draw   (crossboxlines newFrame)) picture
    = picture

crossboxlines {corner1=a,corner2=b}
    = [  {line_end1=p1,line_end2=p2}
      \\ (p1,p2)<-[(a,c),(a,d),(a,b),(b,c),(b,d),(c,d)]
      ]
where c = {b & y=a.y}; d = {a & y=b.y}
```

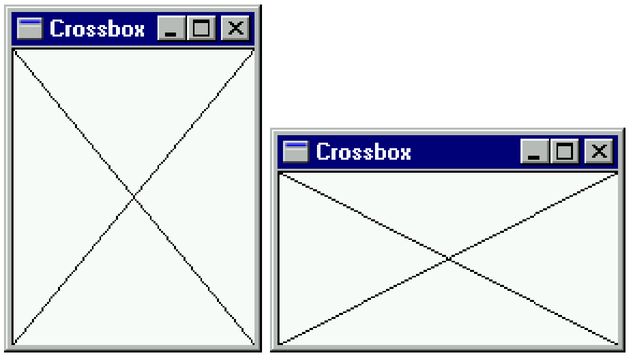


Figure 6.3: Letting the `Look` function follow the window size.

The purpose of the `WindowLook` attribute function is to describe the look of the *current state* of the document layer. If the document does not change, then this function always correctly renders the document. However, for most windows the state of the document changes during its life-cycle. The `WindowLook` attribute can be changed and retrieved using the `StdWindow` functions `setWindowLook` and `getWindowLook`:

```
setWindowLook :: !Id !Bool !(!Bool,!Look) !(IOSt .l) -> IOSt .l
getWindowLook :: !Id                      !(IOSt .l)
                    -> (!Maybe (Bool,Look),!IOSt .l)
```

`setWindowLook` changes the current `WindowLook` attribute of the window indicated by the `Id` argument with the new `(Bool,Look)` argument, provided this window is present and does not refer to a dialogue. If the first `Bool` argument is `False` then that's all. If it is `True`, then the look function will be applied to the window in the way described above.

6.4.2 Direct rendering

Instead of using only the `Look` function of a window, one can also draw directly in the `Picture` of the window's document layer. This is done using the functions `appWindowPicture` and `accWindowPicture` (`StdWindow`, Appendix A.47):

```
appWindowPicture:: !Id !.(IdFun *Picture) !(IOSt .l) -> IOSt .l
accWindowPicture:: !Id !.(St *Picture .x) !(IOSt .l)
                             -> (!Maybe .x,!IOSt .l)
```

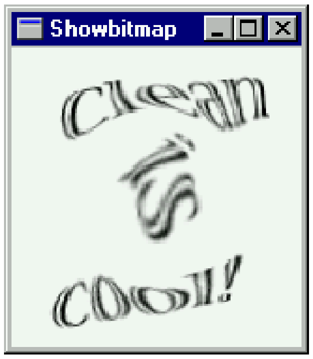


Figure 6.4: The bitmap program in action.

`appWindowPicture` applies the argument drawing function to the `Picture` in the document layer of the window indicated by the `Id` argument if this window is present and does not refer to a dialogue. `accWindowPicture` is the same, except that its argument function returns a result of some type. If the indicated window was indeed found, and the argument function returned a value `x`, then the result of `accWindowPicture` is `(Just x)`. In all other cases it is `Nothing`.

For some visual feedback such as drawing blinking cursors or track boxes this method is more suitable than the indirect way of using the `Look` function. The timer example in Section 9.1.1 shows a window in which the direct rendering method is applied.

6.4.3 Pragmatics

The `WindowLook` function is used by the object I/O system for all cases that the content of the window needs to rendered. Among others, causes are when the view frame, size, stacking order, or selectstate of a window changes. It is very annoying for the application user when these actions take to much time. Therefore it is worth your while to spend some effort in getting a good performance out of the list of drawing functions.

6.4.4 Example: displaying a bitmap

In this example we create a program that allows the user to select an arbitrary bitmap which will be displayed in a window (see Figure 6.4).

The first thing to do is to give the user the opportunity to select a bitmap. For this purpose the library function `selectInputFile` in module `StdFileSelect` (Appendix A.10) is provided. Applying this function opens the platform standard file selector dialogue. When a file has been selected by the user the return value contains the full pathname of the selected file. If no file has been selected, then `Nothing` is returned. This function belongs to the `FileSelectEnv` type constructor class which contains two other functions: `selectOutputFile` (which allows a user to select a (new) output file) and `selectDirectory` (which allows a user to select a directory).

```
class FileSelectEnv env where
  selectInputFile ::                !*env -> (!Maybe String,!*env)
  selectOutputFile:: !String !String !*env -> (!Maybe String,!*env)
  selectDirectory ::                !*env -> (!Maybe String,!*env)
```

The first part of the program looks as follows (if no file was selected the program simply terminates):

```
Start :: *World -> *World
Start world
  # (maybeFile,world) = selectInputFile world
  | isNothing maybeFile
    = world
```

Given a selected input file pathname, the function `openBitmap` (`StdBitmap`, Appendix A.1) can be used to read in the bitmap (see also Section 5.4.9).

```
openBitmap:: !{#Char} !*env-> (!Maybe Bitmap,!*env) | FileSystem env
```

`openBitmap` returns a bitmap if it could be read in successfully, otherwise it returns `Nothing` (in that case we also let the program terminate). This part of the program is as follows:

```
  # bitmapfile          = fromJust maybeFile
  # (maybeBitmap,world) = openBitmap bitmapfile world
  | isNothing maybeBitmap
    = world
```

Having successfully read in the bitmap, we can determine its size using the `StdBitmap` function `getBitmapSize`.

```
  # bitmap     = fromJust maybeBitmap
    bitmapsize = getBitmapSize bitmap
```

The window that displays the bitmap has the same initial size as the bitmap (setting the (`WindowViewSize bitmapsize`) attribute). In this example we use the indirect rendering method. This is done by setting the (`WindowLook True (\_ _-> drawAt zero bitmap`) attribute. Finally, when the user requests closing of the window, the application simply terminates. This is done by setting the (`WindowClose (noLS closeProcess)`) attribute. So we obtain the following window definition:

```
  window = Window "Bitmap" NilLS
             [ WindowViewSize  bitmapsize
             , WindowLook      True (\_ _->drawAt zero bitmap)
             , WindowClose     (noLS closeProcess)
             ]
```

The final step is to create an interactive process that contains the window by applying the `startIO` function (`StdProcess`, Appendix A.27). The only initialisation action of the process is to open the window. In case the user requests the process to terminate, the application obeys. This is done by setting the (`ProcessClose closeProcess`) attribute.

```
  = startIO SDI
            Void
            (snd o openWindow Void window)
            [ProcessClose closeProcess]
            world
```

This completes the program. Here is the complete program code.

```
module showbitmap

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program creates a window that displays a user selected bitmap.
//  Make sure that this application has sufficient heap or extra memory.
//  ******************************************************************************

import StdIO, StdEnv

Start :: *World -> *World
Start world
    # (maybeFile,world)     = selectInputFile world
    | isNothing maybeFile
        = world
    # bitmapfile            = fromJust maybeFile
    # (maybeBitmap,world)   = openBitmap bitmapfile world
    | isNothing maybeBitmap
        = world
    | otherwise
        # bitmap            = fromJust maybeBitmap
          bitmapsize        = getBitmapSize bitmap
          window            = Window "Bitmap" NilLS
                                 [   WindowViewSize  bitmapsize
                                 ,   WindowLook      True (\_ _->drawAt zero bitmap)
                                 ,   WindowClose     (noLS closeProcess)
                                 ]
        = startIO SDI
                  Void
                  (snd o openWindow Void window)
                  [ProcessClose closeProcess]
                  world
```

6.5 Handling the control layer

The control layer contains the controls of a window or dialogue. Windows and dialogues can have the same set of controls. As we have seen in Section 6.3, this has been made explicit by the type constructor class instance declarations of the respective type constructor classes.

The standard set of `Controls` instances is defined in the module `StdControlClass` (Appendix A.6). Many operations with respect to the control layer are identical to operations on controls that are element of `CompoundControls`. Controls and their operations are discussed in detail in Chapter 7.

6.6 Handling the window and dialogue frame

The window and dialogue frame are the ‘physical’ borders of windows and dialogues. A user can grab them using the mouse or some keyboard interface and drag them around, change the size (in case of windows), or dispose of them. The program has very limited influence on both the appearence and functionality of the frame. When *opening* a window or dialogue, the `WindowAttributes` are important. This is discussed in Section 6.6.1. User actions on the window or dialogue frame are handled by the program via the callback function mechanism. The program can also change frame properties. This is discussed in Section 6.6.2.

6.6.1 Opening a window or dialogue frame

The attributes of a window and dialogue definition that influence the window and dialogue frame are of course the `Title` and the following `WindowAttribute`s:

`WindowOuterSize` **and** `WindowViewSize`
: These attributes give the prefered size of the window or dialogue *frame* and the prefered size of the window or dialogue *view frame* respectively. By convention, if the attribute list of a window or dialogue contains both attributes, then only the first of these is valid.

`WindowClose`
: If this attribute is present, then the window or dialogue frame is provided with a platform dependent interface element to allow the user to request the program to close that window or dialogue.

`WindowHScroll` **and** `WindowVScroll`
: Although the horizontal and vertical scrollbar of a window are element of the control layer (Figure 6.2), their behaviour is intimately connected with the size of the view frame and therefore also with the size of the window and dialogue frame.

6.6.2 Changing a window and dialogue frame

The two most apparant changes to the window or dialogue frame are changes of orientation and size. In case of windows, these can be caused by the application user, depending on the attributes as explained in Section 6.6.1.

The program can also change the size of the frame for both windows and dialogues. For dialogues this can be done only indirectly by opening or closing controls. For windows this can also be done directly. If the program changes the view frame (either its size or orientation) then the layout of controls is changed in the same way as if the user had caused this change.

6.7 Handling keyboard and mouse input

Of all windows and dialogues that are in control by an application, at most one receives the keyboard and mouse input. This window is the *active window*.

Except when modal dialogues are open, the application user can always select one of the visible windows or dialogues to become the new active window. Windows and dialogues can be notified of these events by adding two `WindowAttribute`s: `WindowActivate` and `WindowDeactivate`. Both attributes are parameterised with a function that will be applied by the object I/O system as soon as that window becomes active or has become inactive respectively. It is guaranteed that the `WindowDeactivate` attribute function is applied before the `WindowActivate` function.

The underlying platform always gives visual clues to the application user about which window or dialogue is currently active. The program can retrieve this information using the `getActiveWindow` function (Appendix A.47). One should be aware that it is not correct to assume that the active window or dialogue has the topmost stack order position. As an example, one might try to get the `Id` of the active window by taking the `Id` from the first element of the result list of `getWindowStack`, but this only returns the top most window and not the active window.

The program can also activate windows and dialogues. This is done with the `setActiveWindow` function (Appendix A.47). Because modal dialogues are always front-most and the front-most modal dialogue is active, one can not activate a window or modeless dialogue while modal dialogues are open. Instead, `setActive-Window` restacks such a window or dialogue immediately behind the bottom-most modal dialogue *without* making it the active window.

The active window receives all keyboard and mouse input. If this window contains controls it can be the case that the input is channelled to one of these controls. That particular control then has the *input focus*. If no control has the input focus, then all input is handled by the active window. In case the active window is a dialogue its response to input is defined entirely by the underlying platform. In case of windows the program can customise the behaviour by adding a `WindowKeyboard` attribute for keyboard input (Section 6.7.1) and by adding a `WindowMouse` attribute for mouse input (Section 6.7.2). Both attributes have a filter function (`KeyboardStateFilter` and `MouseStateFilter` respectively) which is applied before the actual callback function is evaluated. Only if the filter returns `True` then the callback function is evaluated.

6.7.1 Keyboard input

Every keyboard sensitive interface object has a `KeyboardFunction` which is a process state transition function that receives, as a first argument, a value of type `KeyboardState`. This value represents one keyboard event. Keyboard events are always generated in sequences that are characterised by a value of type `KeyState` in the following order:

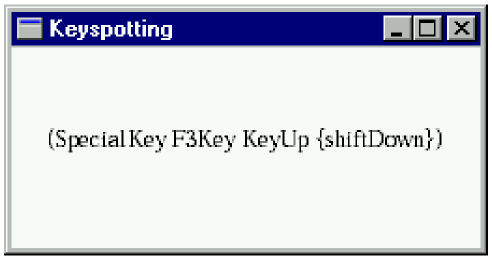
$$\texttt{(KeyDown False) (KeyDown True)}^{*}\ \texttt{KeyUp}$$

A keyboard event concerns either the input of an ASCII character (`CharKey` alternative) or a special key (`SpecialKey` alternative). The alternative `KeyLost` is generated whenever the sequence above was interrupted for some reason (for instance, when another window has become active).

```
::  KeyboardState
 =  CharKey    Char       KeyState
 |  SpecialKey SpecialKey KeyState Modifiers
 |  KeyLost
```

The `SpecialKeys` are imported via the module `StdKey` (Appendix A.15). Among others they define the function keys, arrow keys, page and line keys. The `Modifiers` type is defined in `StdIOCommon`. It is a record that refers to the state of the meta keys of the keyboard. Because some ASCII characters are generated using these meta keys they are not provided at the `CharKey` alternative. So pressing *shift 'a'* simply generates the value `(CharKey 'A' (KeyDown False))`.

The object I/O system guarantees that at all times only one keyboard alternative is being handled. Assume that a user is pressing the 'a' key on the keyboard. This generates a *character 'a' key down event* `(CharKey 'a' (KeyDown False))`, and then a sequence of *character 'a' repeat key events* `(CharKey 'a' (KeyDown True))`. If the user now also presses the 'b' key, the object I/O system inserts two virtual events that force the program to believe that the user first released the 'a' key with a *character 'a' key up event* `(CharKey 'a' KeyUp)`, and then pressed the 'b' key with a *character 'b' key down event* `(CharKey 'b' (KeyDown False))`. These are followed by *character 'b' repeat key events*.



(SpecialKey F3Key KeyUp {shiftDown})

Figure 6.5: The keyspotting program in action.

Example: keyspotting

To illustrate the use of keyboard handling we create a program that has a window in which the last keyboard input is displayed. Figure 6.5 presents a snapshot of the program.

To allow the window to track keyboard input, it must have the `WindowKeyboard` attribute. Because we intend to monitor every keyboard input the attribute's keyboard filter function argument must accept every `KeyboardState`. This can be defined conveniently by `(const True)`, using the `StdFunc` library function `const`. Of course the keyboard function must handle keyboard input, so the `SelectState` attribute is `Able`. The keyboard function, `spotting` is not interested in the local state of the window which can be ignored by using the lifting function `noLS1`. The keyboard function is parameterised with the `Id` of the window, `wid`. This gives us the following definition of the `WindowKeyboard` attribute:

```
    WindowKeyboard (const True) Able (noLS1 (spotting wid))
```

The keyboard function `spotting` uses the indirect rendering method (discussed in Section 6.4.1) to display the last keyboard input. It applies the `setWindowLook` function (`StdWindow`, Appendix A.47) to change the `Look` function of the window each time new keyboard input reaches the window. For this reason, `spotting` is parameterised with the `Id` of the window. In `StdIOCommon` for all type definitions that do not contain function types an instance is defined for `toString`. The type of `spotting` can be generalised so that it works for any second argument for which an instance of the overloaded function `toString` exists.

```
spotting :: Id x (PSt .l) -> PSt .l | toString x
spotting wid x pst
 = appPIO (setWindowLook wid True (False,look (toString x))) pst
```

The `Look` function of the window is parameterised with the string that should be displayed. It simply centers this string in the current view frame. The size of the view frame (a value of type `Rectangle`) can be determined using the `StdIOBasic` function `rectangleSize`. The size of the string, when drawn with the current pen, can be determined using the `StdPicture` function `getPenFontStringWidth` (see also Section 5.3).

```
look :: String SelectState UpdateState *Picture -> *Picture
look text _ {newFrame} picture
   # picture         = unfill newFrame picture
   # (width,picture) = getPenFontStringWidth text picture
   = drawAt {x=(w-width)/2,y=h/2} text picture
where
   {w,h}             = rectangleSize newFrame
```

Setting up `look` in this way has the advantage that the window's appearance adapts itself to its actual size.

The only things that need to be done to get a complete program is to create an `Id` value for the window, and start an interactive process that opens the window. This is shown below in the complete code of the keyspotting example.

```
module keyspotting

//  **************************************************************************
//  Clean tutorial example program.
//
//  This program monitors keyboard input that is sent to a Window.
//  **************************************************************************

import StdEnv,StdIO

Start :: *World -> *World
Start world
    # (wid,world)   = openId world
    # window        = Window "keyspotting" NilLS
                        [   WindowKeyboard  (const True) Able (noLS1 (spotting wid))
                        ,   WindowId        wid
                        ,   WindowClose     (noLS closeProcess)
                        ]
    = startIO SDI
              Void
              (snd o openWindow Void window)
              [ProcessClose closeProcess]
              world
where
    spotting :: Id x (PSt .l) -> PSt .l | toString x
    spotting wid x pst
        = appPIO (setWindowLook wid True (False,look (toString x))) pst

    look :: String SelectState UpdateState *Picture -> *Picture
    look text _ {newFrame} picture
        # picture           = unfill newFrame picture
        # (width,picture)   = getPenFontStringWidth text picture
        = drawAt {x=(w-width)/2,y=h/2} text picture
    where
        {w,h}               = rectangleSize newFrame
```

6.7.2 Mouse input

Every mouse sensitive interface object has a `MouseFunction` which is a process state transition function that receives, as a first argument, a value of type `MouseState`. This value represents one mouse event.

```
::  MouseState
    =   MouseMove   Point2 Modifiers
    |   MouseDown   Point2 Modifiers Int
    |   MouseDrag   Point2 Modifiers
    |   MouseUp     Point2 Modifiers
    |   MouseLost
```

The `Point2` and `Modifiers` types are defined in the modules `StdIOBasic` and `StdIOCommon` respectively. The `Point2` argument gives the position of the mouse in terms of the view domain coordinates of the interactive object that contains the specific `MouseFunction`. The `Modifiers` type constructor is a record that refers to the state of the meta keys of the keyboard that were pressed at the mouse event.

Mouse events are always generated in sequences that are characterised by the alternative constructor of the `MouseState` type constructor:

$$\{\texttt{MouseMove}\}^* \ \texttt{MouseDown} \ \{\texttt{MouseDrag}\}^* \ \texttt{MouseUp}$$

The `Int` argument of the `MouseDown` alternative gives the number of times the mouse was down within the *mouse double down time*. The mouse double down time is a platform dependent time interval that distinguishes two sequential mouse down events from a double click event. Although an integer is used for this count, its maximum value is usually three. If a mouse down event with count i has occured and a new mouse down event is generated within the mouse double down time, then the next mouse down event has count $i + 1$. If the next mouse down event is not generated within the mouse double down time, then the next mouse down event has count 1.

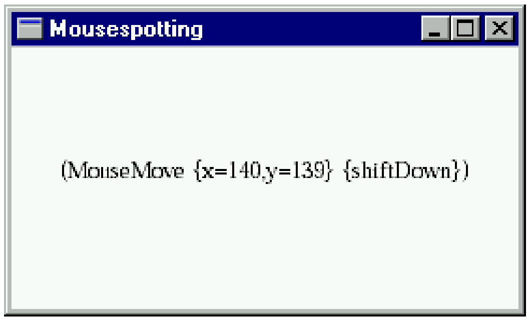
The `MouseLost` alternative is generated whenever for some reason the sequence above is interrupted (for instance when another window has become active).

The object I/O system guarantees that every `MouseFunction` of a mouse sensitive interface object is applied to a sequence of mouse events as characterised above. Assume that a certain window is active and the user is pressing the mouse. This generates first a *mouse down event* (`MouseDown` alternative), followed by a sequence of *mouse drag events* (`MouseDrag` alternative). If for some reason another window is being activated, the object I/O system inserts a virtual event that forces the program to believe that the user has released the mouse button with a *mouse up event* (`MouseUp` alternative). If the new window is also mouse sensitive, then its `MouseFunction` is applied to a new virtual event that forces the program to believe that the user has pressed the mouse again with a *mouse down event* (`MouseDown` alternative). These are followed again by *mouse drag event*s.

Example: mousespotting

To illustrate the use of mouse handling we create a program that monitors the mouse input of a window (see Figure 6.6). This program is almost identical to the keyspotting example in the Section 6.7.1. The only differences are the title of the window and the replacement of the `WindowKeyboard` attribute by a `WindowMouse` attribute. Here we take advantage of the fact that the `spotting` function is overloaded. For completeness, the mousespotting example is shown here.

```
module mousespotting

//  **********************************************************************************
//  Clean tutorial example program.
//
//  This program monitors mouse input that is sent to a Window.
//  **********************************************************************************

import StdEnv,StdIO

Start :: *World -> *World
Start world
    # (wid,world)    = openId world
    # window         = Window "mousespotting" NilLS
                          [   WindowMouse     (const True) Able (noLS1 (spotting wid))
```



(MouseMove {x=140,y=139} {shiftDown})

Figure 6.6: The mousespotting program in action.

```
                        ,   WindowId        wid
                        ,   WindowClose     (noLS closeProcess)
                        ]
    = startIO SDI
              Void
              (snd o openWindow Void window)
              [ProcessClose closeProcess]
              world
where
    spotting :: Id x (PSt .l) -> PSt .l | toString x
    spotting wid x pst
        = appPIO (setWindowLook wid True (False,look (toString x))) pst

    look :: String SelectState UpdateState *Picture -> *Picture
    look text _ {newFrame} picture
        # picture           = unfill newFrame picture
        # (width,picture)   = getPenFontStringWidth text picture
        = drawAt {x=(w-width)/2,y=h/2} text picture
    where
        {w,h}               = rectangleSize newFrame
```

6.8 Modal dialogues

Windows and dialogues have many aspects in common. One important difference, which was also mentioned in the introduction of this chapter, is that only dialogues can be opened *modally*. When a program opens a modal dialogue, the user is forced to completely handle the dialogue before any other operation can occur. When a modal dialogue is open, other modal dialogues can also be opened, but these must also be completely handled. In this way a stack of modal dialogues can be created. An example of such a situation is the behaviour of the platform dependent output file selector dialogue (which is opened by the `StdFileSelect` function `selectOutputFile`), see Figure 6.7. If the user selects an existing file, another modal dialogue is opened that asks the user if it is allright to overwrite that file. The user must answer this dialogue before the output file selector dialogue can be closed.

The type constructor class `Dialogs` contains an additional function to create dialogues in a modal way:

```
class Dialogs wdef where
    openDialog      :: .ls !(wdef .ls (PSt .l))    !(PSt .l)
                    ->  (  !ErrorReport,            !PSt .l)
    openModalDialog :: .ls !(wdef .ls (PSt .l))    !(PSt .l)
```

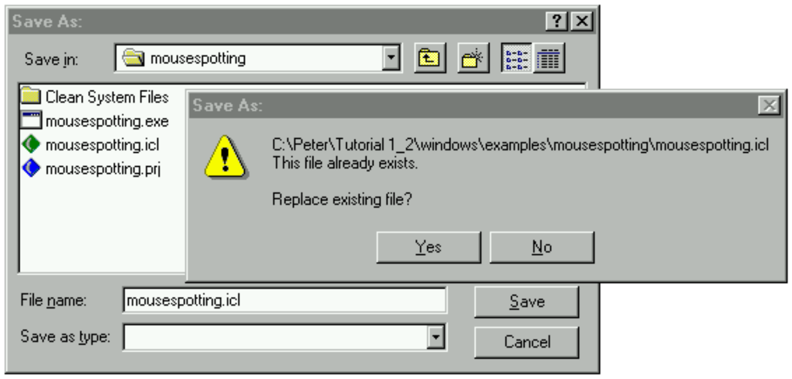


Figure 6.7: Two modal dialogues created by the `selectOutputFile` function.

```
                    ->  (!(!ErrorReport,!Maybe .ls),!PSt .l)
   ...
```

The arguments of `openModalDialog` and `openDialog` are identical: the dialogue has a local state of type `.ls` and it has some definition of type `(wdef .ls (PSt .l))`. Both functions return an `ErrorReport` to inform the program if the dialogue could be created. The major difference between these two class member functions is that `openDialog` opens its argument dialogue and terminates immediately. The function `openModalDialog` opens its argument dialogue and terminates only when it is closed. This situation can only be reached by applying `closeWindow` or `closeActiveWindow` (`StdWindow`, Appendix A.47) or by terminating the parent interactive process using `closeProcess` (`StdProcess`, Appendix A.27). After the modal dialogue has been closed, its final local state value is returned. This provides a comfortable way of passing additional information from the closed modal dialogue to the calling function, as we will see in the next section.

6.8.1 Example: a notice extension

In this example we show the use of modal dialogues by making new `Dialogs` class instances for *notices*. A notice is a very simple dialogue, containing only some line(s) of text, and a (number of) button(s) that close the notice. Notices can be used by programs to briefly inform the user about some situation. The confirmation dialogue in Figure 6.7 is an example of such a notice. We will first make the `notice` module which contains the new `Dialogs` instance in Section 6.8.1. Using this new interactive object we can rephrase the "Hello world" program (Section 2.4). This is done in Section 6.8.1.

The notice module

We want to make a special instance of the `Dialogs` type constructor class that specifies a notice. As said, a notice is a dialogue that contains some line(s) of text, and a (number of) button(s). As done for every object I/O element, a notice will be defined by means of an algebraic data type:

```
:: Notice ls pst
 = Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
```

```
:: NoticeButton st
 = NoticeButton String (IdFun st)
```

The first argument of a `Notice` is a list of text lines. These will be shown below each other. By using one additional `NoticeButton` argument we can make sure that a notice always consists of at least one notice button. This mandatory notice button is placed to the right of the notice. The notice buttons will appear from right to left. A notice button is defined by `NoticeButton`. It is parameterised with the title and a program defined function. This function is applied after the button has been pressed and the notice has been closed.

The `Notice` type constructor will become an instance of the `Dialogs` type constructor class. This would actually be sufficient to allow a program to create both non modal and modal notices, using the class member functions `openDialog` and `openModalDialog`. To simplify creation of notices even further, we want to make the following additional function with a simpler type:

```
openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
```

This specification is bundled in the notice definition module, which now looks as follows:

```
definition module notice

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program defines a new instance of the Dialogs class to create notices.
//  ********************************************************************************

import StdWindow

::  Notice ls pst
 =  Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
::  NoticeButton st
 =  NoticeButton String (IdFun st)

instance Dialogs Notice

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
```

Let's work out the notice implementation module. Because a notice is a specialised dialogue, it is sufficient to map a notice definition to a dialogue definition. This mapping is done by the function `noticeToDialog`. For programming convenience, we will convert the text lines into *text controls*, and the notice buttons into *button controls*. Although controls are not properly introduced yet, we hope this does not frustate your attempt to understand what's going on here.

Each notice text line is mapped to a text control. A text control is parameterised with a string, and can have a position attribute. All texts will be left-aligned. This is expressed by setting the (`ControlPos (Left,zero)`) attribute. For any list of strings `texts` this mapping can be conveniently expressed via a list comprehension (the `ListLS` data constructor is required for reasons explained in Chapter 7):

```
ListLS [TextControl text [ControlPos (Left,zero)] \\ text<-texts]
```

We want to place the lines together in a box without additional item space (the default item space and margin values of a dialogue are platform dependent). To make sure that we get our preferred values we override the dialogue item space and margin attributes with our own (using zero margins and an item space of three in both directions). This is done via a `LayoutControl`. The final mapping of the text lines results in:

```
texts‘= LayoutControl
        ( ListLS [  TextControl text [ControlPos (Left,zero)]
                 \\ text<-texts
                 ]
        )
        [ControlHMargin 0 0,ControlVMargin 0 0,ControlItemSpace 3 3]
```

Let's first introduce a function `noticebutton` that maps one notice button to a button control. When a notice button is selected, it should first close the notice (using `closeActiveWindow`), and then apply the argument function. This action is handled by the argument function of the `ControlFunction` attribute. We allow other attributes such as placement and so on to be appended to the button control by passing it as an additional argument of `noticebutton`.

```
noticebutton (NoticeButton text f) atts
   = ButtonControl text [ControlFunction f‘:atts]
where
   f‘ (lst,pst) = f (lst,closeActiveWindow pst)
```

Using `noticebutton` we can map the notice buttons to button controls. The mandatory notice button is mapped to the "confirm" control using the window attribute `WindowOk` (Section 6.2.1). We assume that `okid` refers to this `Id`. In addition, the mandatory button is right-aligned in the dialogue. So, if `ok` is the mandatory notice button, then it is mapped to the following button control:

```
ok‘ = noticebutton ok [ControlPos (Right,zero),ControlId okid]
```

The optional notice buttons appear to the left of the previous notice button. Again, we can use a list comprehension to do this mapping for a list of notice buttons `buttons`:

```
buttons‘ = ListLS
           [  noticebutton button [ControlPos (LeftOfPrev,zero)]
           \\ button<-buttons
           ]
```

Given a `Notice` value (`Notice texts ok buttons`), `noticeToDialog` creates a titleless `Dialog` and glues the mapped notice controls using the expression (`texts‘ :+: ok‘ :+: buttons‘`) (glueing controls is also discussed in the next chapter):

```
noticeToDialog okid (Notice texts ok buttons)
   = Dialog "" (texts‘:+:ok‘:+:buttons‘) [WindowOk okid]
```

Given the mapping function `noticeToDialog` it is now a trivial task to define the new instance declaration of the type constructor class `Dialogs`:

```
instance Dialogs Notice where
```

```
  openDialog ls notice pst
     # (okId,pst) = accPIO openId pst
     = openDialog ls (noticeToDialog okId notice) pst
  openModalDialog ls notice pst
     # (okId,pst) = accPIO openId pst
     = openModalDialog ls (noticeToDialog okId notice) pst
  getDialogType _
     = "Notice"
```

Also the definition of the convenience function `openNotice` now becomes trivial:

```
openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
openNotice notice pst
   = snd (openModalDialog undef notice pst)
```

For completeness, the `notice` implementation module is given here.

```
implementation module notice

//  *******************************************************************************
//  Clean tutorial example program.
//
//  This program defines a new instance of the Dialogs class to create notices.
//  *******************************************************************************

import StdTuple, StdMisc, StdFunc
import StdId, StdPSt, StdWindow

::  Notice ls pst
 =  Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
::  NoticeButton st
 =  NoticeButton String (IdFun st)

instance Dialogs Notice where
    openDialog ls notice pst
        # (okId,pst) = openId pst
        = openDialog ls (noticeToDialog okId notice) pst
    openModalDialog ls notice pst
        # (okId,pst) = openId pst
        = openModalDialog ls (noticeToDialog okId notice) pst
    getDialogType _
        = "Notice"

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
openNotice notice pst
    = snd (openModalDialog undef notice pst)

noticeToDialog :: Id !(Notice .ls (PSt .l))
    ->  Dialog  (:+: (LayoutControl (ListLS TextControl))
                (:+: ButtonControl  (ListLS ButtonControl)
                )) .ls (PSt .l)
noticeToDialog okid (Notice texts ok buttons)
    = Dialog "" (texts`:+:ok`:+:buttons`) [WindowOk okid]
where
    texts`   = LayoutControl
                (  ListLS [  TextControl text [ControlPos (Left,zero)]
                          \\ text<-texts
                          ]
                ) [ControlHMargin 0 0,ControlVMargin 0 0,ControlItemSpace 3 3]

    ok`      = noticebutton ok [ControlPos (Right,zero),ControlId okid]

    buttons` = ListLS
```

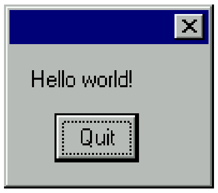


Figure 6.8: The hello world program, now using a notice.

```
                [  noticebutton button [ControlPos (LeftOfPrev,zero)]
                \\ button<-buttons
                ]

    noticebutton (NoticeButton text f) atts
        = ButtonControl text [ControlFunction f`:atts]
    where
        f` (lst,pst) = f (lst,closeActiveWindow pst)
```

Hello world revisited

We can now use our new notice instance to make another implementation of the "Hello world" program. It is very similar to the original version in Section 2.4 but instead of a `Dialog` it now uses a `Notice`. Figure 6.8 shows the notice. The program code is given below.

```
module usenotice

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program shows "Hello world!" using a Notice.
//  ********************************************************************************

import StdEnv, StdIO
import notice

Start :: *World -> *World
Start world
    = startIO NDI Void (openNotice hello) [] world
where
    hello   = Notice ["Hello world!"] (NoticeButton "Quit" (noLS closeProcess)) []
```

Chapter 7

Control handling

In the previous chapter windows and dialogues have been introduced. These top level interface elements can contain sets of *controls*, which are the subject of this chapter. Controls can be hierarchical, i.e. they can be composed of controls themselves. Using controls helps a program to provide a consistent and structured user interface. There are a lot of issues involved when working with controls.

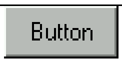
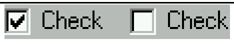
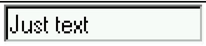
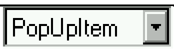
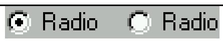
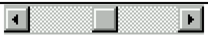
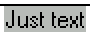
First of all we introduce each of the standard controls in Section 7.1. Then the glue is introduced to build larger control structures in Section 7.2. An important aspect of controls is to manage their layout, presented in Section 7.3. Related to layout is what should happen in case a window containing controls is resized. This is discussed in Section 7.4. Finally, Section 7.5 contains a number of examples that demonstrate the use of controls.

7.1 The standard controls

The data type definitions of the standard set of object I/O library controls is given in module `StdControlDef` (Appendix A.7). They can be divided into three groups:

Platform standard controls

These are the controls that exist on all platforms and that have a well-defined look and feel that is platform defined. Adding these elements to your application will give your program a standardised interface that experienced users feel comfortable with.

Control object:	**What does it look like:**
`ButtonControl`	
`CheckControl`	
`EditControl`	
`PopUpControl`	
`RadioControl`	
`SliderControl`	
`TextControl`	

Customised controls

Although the use of platform standard controls increases the familiarity of your application, sometimes you need to create controls with additional look and feel. For this purpose the library provides two customised controls. The look of the `CustomButtonControl` is program defined, but the library handles its feel as if it were a button. The `CustomControl` can be used when the program wants to define every aspect of its look and feel.

Hierarchical controls

One simple way of combining controls is to place them in a hierarchical control. Hierarchical controls always introduce a new layout scope. This functionality is provided by the `LayoutControl`. The `CompoundControl` is very analogous to a window in a control: it can have scrollbars, it has a `Picture` for its background, and so on.

Controls have an extensive set of attributes as one can see in the table below. The hierarchical controls have a number of special attributes. They are strongly related to window and dialogue attributes.

General control attributes:	**Hierarchical control attributes:**
`ControlActivate`	`ControlHMargin`
`ControlDeactivate`	`ControlHScroll`
`ControlFunction`	`ControlItemSpace`
`ControlHide`	`ControlLook`
`ControlId`	`ControlOrigin`
`ControlKeyboard`	`ControlOuterSize`
`ControlMinimumSize`	`ControlViewDomain`
`ControlModsFunction`	`ControlViewSize`
`ControlMouse`	`ControlVMargin`
`ControlPen`	`ControlVScroll`
`ControlPos`	
`ControlResize`	
`ControlSelectState`	
`ControlTip`	
`ControlWidth`	

We defer the explanation of the hierarchical control attributes until Section 7.1.11. Let's have a look at the attributes in the left column of the table.

`ControlActivate` **and** `ControlDeactivate`

In every interactive program that has windows and dialogues, there is one window or dialogue that receives the user input, the active window. If this window or dialogue contains controls, then the user input is further directed to that particular control that has the *input focus*. When a control obtains the input focus, the function associated with its `ControlActivate` attribute is evaluated. When a control looses the input focus, the same thing happens with its `ControlDeactivate` attribute function.

`ControlFunction` **and** `ControlModsFunction`

These two attributes are the primary callback function attributes of controls. The first function is evaluated whenever the associated control is selected by the user. The same happens for the second function, except that it is also provided with the modifier keys that have been pressed at the moment the user selected the control. If these two attributes both appear in an attribute list, then the first of the two is selected.

`ControlHide`
: This attribute defines that the control is initially invisible. It does occupy space. By default controls are visible when element of a window or dialogue. Of course, when some parent object is invisible, then all child controls are invisible as well.

`ControlId`
: This attribute identifies the control to which it is associated. If you do not provide an `Id`, the control can not be modified by the program (as explained in Chapter 4).

`ControlKeyboard` **and** `ControlMouse`
: These attributes add keyboard and mouse handling callback functions to the associated control. For most platform standard controls these attributes are ignored because the platform already defines their response to these inputs.

`ControlMinimumSize`
: This attribute defines the minimum size of the control. This value is relevant in case of resizing (see Section 7.4). If no `ControlMinimumSize` is provided, the default value zero is chosen.

`ControlPen`
: When set, this attribute defines the initial pen attributes that the look function of customised controls uses for drawing. Pen attributes were introduced in Section 5.1. If not present, the default pen values are used.

`ControlPos`
: This attribute determines the layout position (see Section 7.3) of the associated control. By default, a control is placed right next to the previous control (if it happens to be the first control, it will be positioned at the left-top).

`ControlResize`
: This attribute defines that the control is *resizeable*. A control is resizable if it responds to user or program resize actions of a parent object. By default controls do not resize on these events. See Section 7.4 about the resizing behaviour of controls.

`ControlSelectState`
: This attribute defines whether the control accepts user input (`Able`) or not (`Unable`). Most platform standard controls give some visual clue to the user about their current select state. By default the `SelectState` of controls is `Able`.

`ControlTip`
: The tip attribute provides the user of your application with a textual explanation of the functionality of the control in a platform dependent way. This explanation should be very brief (a few words only).

`ControlWidth`
: For all platform standard controls their height is determined by their demanded attributes. The program can influence the width of such controls by setting the `ControlWidth` attribute. The argument of this attribute is a value of type `ControlWidth` which has the following type definition:

```
:: ControlWidth
 = PixelWidth   Int
 | TextWidth    String
 | ContentWidth String
```

The `PixelWidth` alternative defines the width in terms of the number of pixels. The `TextWidth` alternative specifies that the control should have the width exactly identical to the width of the argument string when drawn in the dialog font. If the associated control is text oriented, then the alternative `ContentWidth` specifies that the control should have the width as if it would contain the argument string.

In the remaining part of this section we introduce each of the standard controls. We first show the non hierarchical controls in alphabetical order, and end with the hierarchical controls. Of each control we give the set of types you must use to define it. Of the general control attributes it is indicated which are valid. This information can also be retrieved from the object I/O library. In the definition module `StdControlAttribute` of each control there is a predicate function that defines which attributes are valid. Finally, a simple example of a control definition is given.

7.1.1 The ButtonControl

The *button control* represents an action that should occur given the current state of the window or dialogue. The definition of a button control is as follows:

```
::  ButtonControl ls pst
 =  ButtonControl String [ControlAttribute *(ls,pst)]
```

A button control has a title, given by a string. The string should not contain control characters because they will generally produce rubbish on the screen. The & character can be used to add keyboard interface to the user, depending on the platform. The escape sequence is &&. Valid control attributes are:

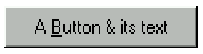
ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
ControlActivate		ControlMouse	
ControlDeactivate		ControlPen	
ControlFunction	√	ControlPos	√
ControlHide	√	ControlResize	
ControlId	√	ControlSelectState	√
ControlKeyboard		ControlTip	√
ControlMinimumSize		ControlWidth	√
ControlModsFunction	√		

The initial size of a button control is determined by its initial text line and `ControlWidth` attribute. When the button is selected by the user the first of `ControlFunction` and `ControlModsFunction` in the attribute list is evaluated. If the name of the button control is modified to a new text line, then its size is not changed.

A button control can be the confirm or cancel button of a window or dialogue (see Section 6.2.1). It must have the `ControlId` attribute set. The `Id` value must then be given to the `WindowOk` or `WindowCancel` attribute respectively.

Here is a `ButtonControl` with a title that contains newlines:

```
buttoncontrol
  = ButtonControl "A &Button && its text" []
```



7.1.2 The CheckControl

A *check control* is a group of check control items of which an arbitrary number can be selected. All alternatives are visible. The definition of a check control is as follows:

```
::  CheckControl ls pst
 =  CheckControl [CheckControlItem *(ls,pst)] RowsOrColumns
                 [ControlAttribute *(ls,pst)]
::  RowsOrColumns
 =  Rows    Int
 |  Columns Int
::  CheckControlItem st
:== (String,Maybe ControlWidth,MarkState,IdFun st)
```

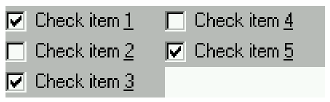
The `RowsOrColumns` argument indicates the local layout of the items. Rowwise layout is given by the `Rows` alternative, and columnwise layout by the `Columns` alternative. Of each check item its title, width, initial mark state, and callback function is specified. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`		`ControlMouse`	
`ControlDeactivate`		`ControlPen`	
`ControlFunction`		`ControlPos`	✓
`ControlHide`	✓	`ControlResize`	
`ControlId`	✓	`ControlSelectState`	✓
`ControlKeyboard`		`ControlTip`	✓
`ControlMinimumSize`		`ControlWidth`	
`ControlModsFunction`			

When a check control item is selected its mark state will be toggled (from `Mark` to `NoMark` and vice versa). No other check control items are affected. The corresponding callback function is then evaluated.

Here is an example of a `CheckControl` that consists of five items, placed in two columns. Each odd numbered item has a check mark.

```
 checkcontrol
   = CheckControl
     [  ( "Check item &"+++toString i
        , Nothing
        , if (isOdd i) Mark NoMark
        , id
        )
     \\ i<-[1..5]
     ]  (Columns 2) []
```



7.1.3 The CustomButtonControl

A *custom button control* is a control that *feels* like a button control, but which *look* is customised by the program. The definition of a custom button control is as follows:

```
:: CustomButtonControl ls pst
 = CustomButtonControl Size Look [ControlAttribute *(ls,pst)]
```

```
:: Size        = {w::!Int,h::!Int}
:: SelectState = Able | Unable
:: UpdateState = { oldFrame :: !ViewFrame
                 , newFrame :: !ViewFrame
                 , updArea  :: !UpdateArea
                 }
:: ViewFrame  :== Rectangle
:: UpdateArea :== [ViewFrame]
:: Look       :== SelectState -> UpdateState -> *Picture -> *Picture
```

Both the initial size and look of a custom button control are defined by the program. The look of a custom button control is identical to the look of a window as discussed in Section 6.4.1. The `UpdateState` argument contains zero based rectangles of the same size as the custom button control itself. Every custom button control has a `*Picture` environment to which the look drawing functions are applied. Valid control attributes are:

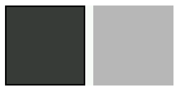
ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`		`ControlMouse`	
`ControlDeactivate`		`ControlPen`	√
`ControlFunction`	√	`ControlPos`	√
`ControlHide`	√	`ControlResize`	√
`ControlId`	√	`ControlSelectState`	√
`ControlKeyboard`		`ControlTip`	√
`ControlMinimumSize`	√	`ControlWidth`	
`ControlModsFunction`	√		

When the custom button control is selected by the user the first of `ControlFunction` and `ControlModsFunction` in its attribute list is evaluated.

A button control can be the confirm or cancel button of a window or dialogue (see Section 6.2.1). It must have the `ControlId` attribute set. The `Id` value must then be given to the `WindowOk` or `WindowCancel` attribute respectively.

The look of this `CustomButtonControl` depends on its `SelectState`. The picture on the left shows the custom button control in `Able` state, the picture on the right in `Unable` state.

```
customButtoncontrol
  = CustomButtonControl {w=50,h=50} look []
where
  look Able {newFrame} picture
    # picture = setPenColour DarkGrey picture
    # picture = fill         newFrame picture
    # picture = setPenColour Black    picture
    # picture = draw         newFrame picture
    = picture
  look Unable {newFrame} picture
    # picture = setPenColour LightGrey picture
    # picture = fill newFrame picture
    = picture
```



7.1.4 The CustomControl

A *custom control* is a control of which both the look and feel are program defined. The definition of a custom control is as follows:

```
:: CustomControl ls pst
 = CustomControl Size Look [ControlAttribute *(ls,pst)]
:: Size        = {w::!Int,h::!Int}
:: SelectState = Able | Unable
:: UpdateState = { oldFrame :: !ViewFrame
                 , newFrame :: !ViewFrame
                 , updArea  :: !UpdateArea
                 }
:: ViewFrame  :== Rectangle
:: UpdateArea :== [ViewFrame]
:: Look       :== SelectState -> UpdateState -> *Picture -> *Picture
```

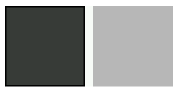
Both the initial size and look of a custom control are defined by the program. The look of a custom control is identical to the look of a window as discussed in Section 6.4.1. The `UpdateState` argument contains zero based rectangles of the same size as the custom control itself. Every custom control has a `*Picture` environment to which the look drawing functions are applied. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`	√	`ControlMouse`	√
`ControlDeactivate`	√	`ControlPen`	√
`ControlFunction`		`ControlPos`	√
`ControlHide`	√	`ControlResize`	√
`ControlId`	√	`ControlSelectState`	√
`ControlKeyboard`	√	`ControlTip`	√
`ControlMinimumSize`	√	`ControlWidth`	
`ControlModsFunction`	√		

The *feel* of a custom control is defined by its mouse and keyboard callback functions. If the user selects the custom control with the mouse, then the mouse callback function handles all input. If the custom control has the input focus, and the user is typing, then the keyboard callback function handles the keyboard input.

The look of this `CustomControl` depends on its `SelectState`. The picture on the left shows the custom control in `Able` state, the picture on the right in `Unable` state.

```
custombuttoncontrol
  = CustomControl {w=50,h=50} look []
where
  look Able {newFrame} picture
    # picture = setPenColour DarkGrey picture
    # picture = fill         newFrame picture
    # picture = setPenColour Black    picture
    # picture = draw         newFrame picture
    = picture
  look Unable {newFrame} picture
    # picture = setPenColour LightGrey picture
    # picture = fill newFrame picture
    = picture
```



7.1.5 The EditControl

The *edit control* is used to provide the user with an interface to edit (typically small amounts of) textual data. The definition of an edit control is as follows:

```
::  EditControl ls pst
 =  EditControl String ControlWidth NrLines
                [ControlAttribute *(ls,pst)]
::  NrLines  :== Int
```

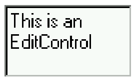
An edit control initially displays some text line. If the textline contains newlines then these are interpreted as line breaks. The edit control has an initial interior width (defined by the `ControlWidth` argument) and shows an integral number of lines. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`	$\surd$	`ControlMouse`	
`ControlDeactivate`	$\surd$	`ControlPen`	
`ControlFunction`		`ControlPos`	$\surd$
`ControlHide`	$\surd$	`ControlResize`	$\surd$
`ControlId`	$\surd$	`ControlSelectState`	$\surd$
`ControlKeyboard`	$\surd$	`ControlTip`	$\surd$
`ControlMinimumSize`		`ControlWidth`	
`ControlModsFunction`			

If the `SelectState` of an edit control is `Unable`, the user can not type in text. The program can keep track of the inserted text by setting the *ControlKeyboard* attribute. For each typed key the keyboard function is evaluated. If the edit control has the `ControlId` attribute set, then its content can be retrieved using the `getControlText` function of module `StdControl` (Appendix A.4).

Here is an example of an `EditControl` with interior width 80 pixels and a height of three text lines.

```
editcontrol
  = EditControl "This is an EditControl"
                (PixelWidth 80) 3 []
```



7.1.6 The PopUpControl

A *pop up control* is a group of pop up control items of which exactly one item is selected. The items of a pop up control are presented in a pop up menu. Usually only the currently selected item is displayed. For this reason pop up controls consume much less space than the functionally equivalent radio controls (discussed in Section 7.1.7). The definition of a pop up control is as follows:

```
::  PopUpControl ls pst
 =  PopUpControl [PopUpControlItem *(ls,pst)] Index
                [ControlAttribute *(ls,pst)]
::  PopUpControlItem st :== (String,IdFun st)
::  IdFun            st :== st -> st
```

The initially selected item is indicated by the `Index` value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`	√	`ControlMouse`	
`ControlDeactivate`	√	`ControlPen`	
`ControlFunction`		`ControlPos`	√
`ControlHide`	√	`ControlResize`	
`ControlId`	√	`ControlSelectState`	√
`ControlKeyboard`		`ControlTip`	√
`ControlMinimumSize`		`ControlWidth`	√
`ControlModsFunction`			

When a pop up control item is selected the previously selected item is unchecked and the new item becomes the selected item. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected item is selected.

Here is an example of a `PopUpControl` that consists of five items. The first item is the initially selected item.

```
popupcontrol
  = PopUpControl
    [  ("PopUpItem "+++toString i,id)
    \\ i<-[1..5]
    ]  1 []
```

7.1.7 The RadioControl

A *radio control* is a group of radio control items of which exactly one item is selected. All alternatives are visible. The radio control is functionally equivalent with the pop up control (Section 7.1.6), but it consumes more space. The definition of a radio control is almost identical to that of a check control (Section 7.1.2):

```
::  RadioControl ls pst
 =  RadioControl [RadioControlItem *(ls,pst)] RowsOrColumns Index
                 [ControlAttribute *(ls,pst)]
::  RowsOrColumns
 =  Rows    Int
 |  Columns Int
::  RadioControlItem st
:== (String,Maybe ControlWidth,IdFun st)
::  IdFun st :== st -> st
::  Index    :== Int
```

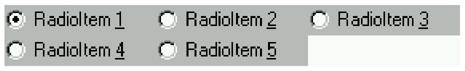
The `RowsOrColumns` argument indicates the local layout of the items. Rowwise layout is given by the `Rows` alternative, and columnwise layout by the `Columns` alternative. The initially selected item is indicated by the `Index` value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Of each radio item its title, width, and callback function is specified. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`		`ControlMouse`	
`ControlDeactivate`		`ControlPen`	
`ControlFunction`		`ControlPos`	√
`ControlHide`	√	`ControlResize`	
`ControlId`	√	`ControlSelectState`	√
`ControlKeyboard`		`ControlTip`	√
`ControlMinimumSize`		`ControlWidth`	
`ControlModsFunction`			

When a radio control item is selected the previously selected radio control item will be unselected, and the new radio control item gets the selection mark. The corresponding callback function is then evaluated. The callback function is also evaluated when choosing the currently selected item.

Here is an example of a `RadioControl` that consists of five items, placed in two rows. The first item is initially selected.

```
radiocontrol
 = RadioControl
   [  ("RadioItem &"+++toString i
      ,Nothing
      ,id
      )
   \\ i<-[1..5]
   ] (Rows 2) 1 []
```



7.1.8 The SliderControl

Slider controls are used to select a value within a range of values. The definition of a slider control is as follows:

```
::  SliderControl ls pst
 =  SliderControl Direction ControlWidth SliderState
                 (SliderAction     *(ls,pst))
                 [ControlAttribute *(ls,pst)]
::  Direction       = Horizontal | Vertical
::  SliderState     = { sliderMin  :: !Int
                      , sliderMax  :: !Int
                      , sliderThumb:: !Int
                      }
::  SliderAction st :== SliderMove -> st -> st
::  SliderMove      =  SliderIncSmall | SliderDecSmall
                    |  SliderIncLarge | SliderDecLarge
                    |  SliderThumb Int
```

A slider control can be given a horizontal direction (the `Horizontal` alternative of `Direction`) or a vertical direction (the `Vertical` alternative of `Direction`). In this direction it can have a certain length, which is given by the `ControlWidth` attribute.

The value range of a slider control is defined by the `SliderState` record. The initial slider state determines the integer range: `sliderMin` gives the minimum value, `sliderMax` gives the maximum value. In case these values are given in the wrong order, they will be ordered properly. The initially chosen value is given by the `sliderThumb` value. This value must be inclusively between `sliderMin` and `sliderMax`. If a value smaller than the minimum range is given, then it is set to the minimum. If a value larger than the maximum range is given, then it is set to the maximum.

Valid control attributes for the `SliderControl` are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`		`ControlMouse`	
`ControlDeactivate`		`ControlPen`	
`ControlFunction`		`ControlPos`	√
`ControlHide`	√	`ControlResize`	√
`ControlId`	√	`ControlSelectState`	√
`ControlKeyboard`		`ControlTip`	√
`ControlMinimumSize`		`ControlWidth`	
`ControlModsFunction`			

A slider control typically has five regions that can be selected by the user. These regions are shown in Figure 7.1

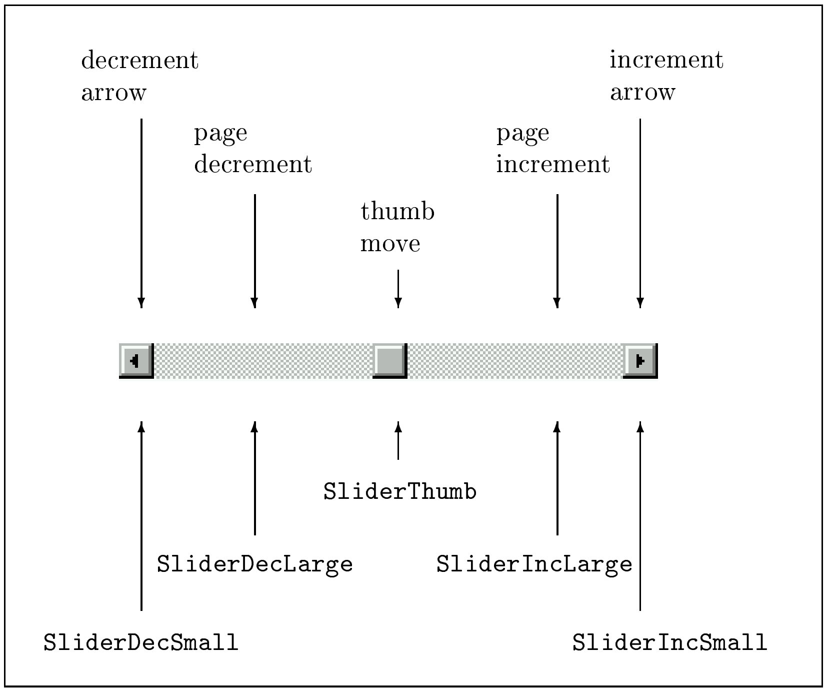


Figure 7.1: The regions of the `SliderControl`.

When the user is working with the slider control, its callback function is evaluated. The algebraic data type `SliderMove` has an alternative constructor for each of the regions of the slider control:

`SliderDecSmall`	*decrement arrow*
`SliderIncSmall`	*increment arrow*
`SliderDecLarge`	*page down region*
`SliderIncLarge`	*page up region*
`SliderThumb`	*thumb move*

The program can decide what to do with this information. It is the responsibility of the programmer that the application responds the way the user expects.

Here is an example of a `SliderControl` that is orientied horizontally and has a length of 200 pixels. It results in the slider control shown in Figure 7.1.

```
slidercontrol
  = SliderControl Horizontal 200
              {sliderMin=(-100),sliderMax=100,sliderThumb=0}
              (\_ st->st) []
```

7.1.9 The TextControl

A *text control* displays one line of text that can not be changed by the user. The definition of a text control is as follows:

```
::  TextControl ls pst
 =  TextControl String [ControlAttribute *(ls,pst)]
```

Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
ControlActivate		ControlMouse	
ControlDeactivate		ControlPen	
ControlFunction		ControlPos	√
ControlHide	√	ControlResize	
ControlId	√	ControlSelectState	
ControlKeyboard		ControlTip	√
ControlMinimumSize		ControlWidth	√
ControlModsFunction			

The initial size of a text control is determined by its initial text line and its `ControlWidth` attribute. Text controls are not resizeable, and they will also not change in size in case their text line is modified by the program.

Here is an example of a text control.

```
textcontrol
  = TextControl "This is a TextControl" []
```

This is a TextControl

7.1.10 The LayoutControl

A *layout control* is a control that contains other controls. It introduces a new *layout scope*: i.e. controls inside it are positioned relative to the bounds of the layout control. The definition of a layout control is as follows:

```
::  LayoutControl  c ls pst
 =  LayoutControl (c ls pst) [ControlAttribute *(ls,pst)]

instance Controls (LayoutControl c) | Controls c
```

As you can see from the type definition, the type variable `c` appears on a type constructor position. The instance declaration of `LayoutControl` of the `Controls` type constructor class adds the restriction that `c` must be a `Controls` instance itself. Valid control attributes are:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`		`ControlOrigin`	
`ControlDeactivate`		`ControlOuterSize`	√
`ControlFunction`		`ControlPen`	
`ControlHide`	√	`ControlPos`	√
`ControlHMargin`	√	`ControlResize`	√
`ControlHScroll`		`ControlSelectState`	√
`ControlId`	√	`ControlTip`	
`ControlItemSpace`	√	`ControlViewDomain`	
`ControlKeyboard`		`ControlViewSize`	√
`ControlLook`		`ControlVMargin`	√
`ControlMinimumSize`	√	`ControlVScroll`	
`ControlModsFunction`		`ControlWidth`	
`ControlMouse`			

The size of a layout control, if not provided as a `ControlOuterSize` or `ControlViewSize` attribute, is derived by the system from its control elements (see Section 7.3 for more information on the layout of controls). Related to its size and element layout are a number of attributes. The `ControlMinimumSize` attribute determines the minimum size of the layout control (see resizing controls, Section 7.4), the `ControlResize` attribute controls the resize behaviour (see also Section 7.4), the `ControlItemSpace`, `ControlHMargin`, and `ControlVMargin` attributes define the distance between the elements themselves and the horizontal and vertical distance of the elements to the border of the layout control respectively. If a layout control does not specify any of these attributes, it obtains the same attribute values as its parent object.

7.1.11 The CompoundControl

The *compound control* is a control that contains other controls. It introduces a new *layout scope* (see the `LayoutControl` in Section 7.1.10). The type definition of a compound control is almost identical to that of a `LayoutControl`:

```
::  CompoundControl  c ls pst
 =  CompoundControl (c ls pst) [ControlAttribute *(ls,pst)]

instance Controls (CompoundControl c) | Controls c
```

Despite the equivalence of their type definitions, a compound control provides programmers with a lot more functionality than its cousin layout control. This becomes clear when you have a look at its valid control attributes:

ControlAttribute:	**Valid:**	**ControlAttribute:**	**Valid:**
`ControlActivate`	$\surd$	`ControlOrigin`	$\surd$
`ControlDeactivate`	$\surd$	`ControlOuterSize`	$\surd$
`ControlFunction`		`ControlPen`	$\surd$
`ControlHide`	$\surd$	`ControlPos`	$\surd$
`ControlHMargin`	$\surd$	`ControlResize`	$\surd$
`ControlHScroll`	$\surd$	`ControlSelectState`	$\surd$
`ControlId`	$\surd$	`ControlTip`	$\surd$
`ControlItemSpace`	$\surd$	`ControlViewDomain`	$\surd$
`ControlKeyboard`	$\surd$	`ControlViewSize`	$\surd$
`ControlLook`	$\surd$	`ControlVMargin`	$\surd$
`ControlMinimumSize`	$\surd$	`ControlVScroll`	$\surd$
`ControlModsFunction`		`ControlWidth`	
`ControlMouse`	$\surd$		

The size of a compound control, if not provided as a `ControlOuterSize` or `Control-ViewSize` attribute, is derived by the system from its control elements (see Section 7.3 for more information on the layout of controls). Related to its size and element layout are a number of attributes. The `ControlMinimumSize` attribute determines the minimum size of the compound control (see resizing controls, Section 7.4), the `ControlResize` attribute controls the resize behaviour (see also Section 7.4), the `ControlItemSpace`, `ControlHMargin`, and `ControlVMargin` attributes define the distance between the elements themselves and the horizontal and vertical distance of the elements to the border of the compound control respectively. If a compound control does not specify any of these attributes, it obtains the same attribute values as its parent object.

The compound control anatomy is the same as that of a window (Section 6.1.1). It consists of the same three layers as a window except that we call its top layer the *compound frame* rather than window frame. The compound frame has no title nor features like resize controls and so on.

Analogous to windows, compound controls have a view domain. As explained in Section 6.1.1, a view domain defines a finite area in which can be drawn (Chapter 5), it can also be used as an area to place controls (Section 7.3). Also for compound controls scrolling attributes can be added: `ControlHScroll` and `ControlVScroll`. These attributes control the current view frame orientation of the compound control. The left top point of the view frame that is currently visible is called the *origin*. This value can be set initially with the `ControlOrigin` attribute. The `Control-Look` attribute is used by the object I/O library to render the document layer of the compound control in exactly the same way as is being done for windows.

The feel of a compound control is of course partially determined by its element controls. If the `ControlMouse` (`ControlKeyboard`) attribute is given, then the compound control can handle all mouse (keyboard) events that are directed to it.

7.2 Control glue

In the previous section the standard set of controls has been discussed. This list does not cover all controls class instances. In the library module `StdControlClass` (Appendix A.6) a number of additional instances are defined, namely the type constructors `:+:`, `ListLS`, `NilLS`, and `AddLS`, `NewLS` (their definition can be found in module `StdIOBasic`, Appendix A.13). These additional instances are required to *glue* controls. They are treated below.

7.2.1 `:+:`

The most common constructor to glue controls is `:+:`. Its type constructor definition and `Controls` class instance declaration are as follows:

```
::  :+: t1 t2 ls cs
 = (:+:) infixr 9 (t1 ls cs) (t2 ls cs)

instance Controls ((:+:) c1 c2) | Controls c1 & Controls c2
```

Assume we have two `Controls` instances `c1` and `c2`, working on the same local state of type `ls` and context state of type `cs`. Now the glued expression `c1:+:c2` is also a `Controls` instance working on the same local state and context state. Because `:+:` is right associative, the expression `c1:+:c2:+:c3` should be read as `c1:+:(c2:+:c3)`.

As an example, reconsider the button control definition `buttoncontrol` in Section 7.1.1, and the edit control definition `editcontrol` in Section 7.1.5:

```
buttoncontrol
  = ButtonControl "A &Button && its text" []
editcontrol
  = EditControl "This is an EditControl" (PixelWidth 80) 3 []
```

Now the following expressions are legal combinations: `(buttoncontrol :+: edit-control)` and `(editcontrol :+: buttoncontrol)`.

7.2.2 `ListLS` and `NilLS`

The `:+:` type constructor is not always the most appropriate glue. When constructing sets of controls of the *same type*, it is much more convenient to use lists and list comprehensions. When constructing an unknown number of controls, it is even impossible to use `:+:`. Again, lists provide more flexibility. List-like glue is provided by the type constructor `ListLS`. The type constructor `NilLS` is a shorthand for `ListLS []`. It can also be conveniently used to state that a hierarchical control, window, or dialogue has no controls. Here are the definitions:

```
::  ListLS t ls cs = ListLS [t ls cs]
::  NilLS    ls cs = NilLS

instance Controls (ListLS c) | Controls c
instance Controls NilLS
```

Given a list of `Controls` instances `xs = [`$c_1 \ldots c_n$`]`, working on the same local state of type `ls` and context state `cs`, the expression `ListLS xs` is also a `Controls` instance working on the same local state and context state.

As an example, assume that you have a list of text labels, `labels`, and that you want to create a set of controls in the following way: for each label in `labels`, create a text control with content that label, and right to that text control an empty edit control that has a width of 100 pixels. The edit controls have to be positioned at the same x-coordinate. The method has to work for any list length and of course handle labels of different length. One additional nasty property is that text controls and edit controls can have different heights (platform dependent).

Here's how to do it. First you determine the longest label size:

```
maxlength = maxList (map size labels)
```

Given this value, you select the longest label from `labels`:

```
maxlabel  = hd (filter (\l->size l==maxlength) labels)
```

Now you can use `ListLS` to create an arbitrary number of text controls glued with edit controls, using a list comprehension:

```
ListLS
[  TextControl label [ControlWidth (ContentWidth maxlabel)
                     ,ControlPos   (Left,zero)
                     ]
   :+:
   EditControl "" (PixelWidth 100) 1 []
\\ label<-labels
]
```

When applied to the following value of `labels`:

```
labels = [ "A short label"
         , "A fairly long and verbose label"
         , "Another lengthy label"
         ]
```

and put it in a dialogue, the result is as in Figure 7.2.

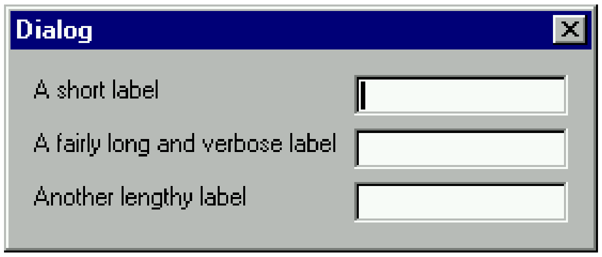


Figure 7.2: Glueing with list comprehensions.

7.2.3 `AddLS` and `NewLS`

The previously discussed glueing type constructors always glue controls that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, `AddLS` and `NewLS`.

Given a `Controls` instance `c1` that works on a local state of type `ls` and a context state of type `cs`, one can add another `Controls` instance `c2` that works on an *extended* local state of type `(new,ls)` and the same context state of type `cs`. Let `x` be a value of type `new`, then this is done by the expression `c1 :+: {addLS=x, addDef=c2}`.

```
:: AddLS t ls cs = E. .new: { addLS::new, addDef::t *(new,ls) cs }

instance Controls (AddLS c) | Controls c
```

Given a `Controls` instance `c1` that works on a local state of type `ls` and a context state of type `cs`, one can add another `Controls` instance `c2` that works on a *new* local state of type `new` and the same context state of type `cs`. Let `x` be a value of type `new`, then this is done by the expression `c1 :+: {newLS=x, newDef=c2}`.

```
:: NewLS t ls cs = E. .new: { newLS::new, newDef::t new cs }

instance Controls (NewLS c) | Controls c
```

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification.

As an example of glueing controls in this way, let's implement a manually incrementable counter control. To do this we fill a `LayoutControl` with three other controls. To display the current count value an `Unable` `EditControl` is used. Two `ButtonControl`s are used to decrement and increment the counter.

```
counter displayid
  = {newLS =count0
    ,newDef=LayoutControl
            (   EditControl (toString count0) (ContentWidth "000") 1
                                  [ControlSelectState Unable
                                  ,ControlPos         (Left,zero)
                                  ,ControlId          displayid
                                  ]
            :+: ButtonControl "-" [ControlFunction (count (-1))]
            :+: ButtonControl "+" [ControlFunction (count 1)]
            )   [ControlHMargin   0 0
                ,ControlVMargin   0 0
                ,ControlItemSpace 0 0
                ]
    }
where
  count0 = 0

  count :: Int (Int,PSt .l) -> (Int,PSt .l)
  count dx (count,pst=:{io})
    = ( count+dx
      , appPIO (setControlText displayid (toString (count+dx))) pst
      )
```

Figure 7.3 shows the counter after some user manipulations that resulted in the counter value `-15`.

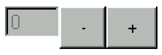


Figure 7.3: The counter control.

7.3 Control layout

The object I/O library offers the programmer an expressive layout mechanism to define the layout of controls. As we have seen before, controls can be element of windows, dialogues, layout and compound controls. The layout rules that are discussed in this section apply to each of these cases. Some layout rules refer to the view domain and view frame of the parent object. In case of dialogues and layout controls these two are identical: they are zero based rectangles with a size equal to the interior size of the parent object.

As we have seen, every control can have a `ControlPos` attribute. This attribute is defined as follows:

```
::  ControlAttribute st
 =  ... | ControlPos ItemPos | ...
::  ItemPos
:== (ItemLoc,ItemOffset)
::  ItemLoc
 =  Fix
 |  LeftTop    | RightTop    | LeftBottom | RightBottom
 |  Left       | Center      | Right
 |  LeftOf Id  | RightTo Id  | Above Id   | Below Id
 |  LeftOfPrev | RightToPrev | AbovePrev  | BelowPrev
::  ItemOffset
 =  NoOffset
 |  OffsetVector Vector2
 |  OffsetFun    ParentIndex OffsetFun
::  ParentIndex
:== Int
::  OffsetFun
:== (ViewDomain,Point2) -> Vector2

instance zero ItemOffset
```

The layout position of a control consists of two values: an `ItemLoc` value and an `ItemOffset` value. The `ItemLoc` actually determines the location of the control, the `ItemOffset` value adds an offset to this position (which of course influences the position of other controls). The `ItemLoc` values can be divided into four groups:

Fixed position

This is only the `Fix` alternative of `ItemLoc`. `Fix` places the left top corner of the associated control at the left top value of the *view domain* of its parent object. The `ItemOffset` determines the actual position in the view domain, and is given in the coordinate system of the view domain of the parent object.

You can use this layout attribute when you want to 'pin' a control on the document layer of a parent object, rather than its control layer (see Section 6.1.1). For dialogues and layout controls, that do not have a document layer, this layout attribute is identical to the boundary aligned attribute `LeftTop`. In a window or compound control, scrolling the view frame will also scroll all controls that have a fixed position.

Boundary aligned

These are the alternatives `LeftTop`, `RightTop`, `LeftBottom` and `RightBottom`. These attributes place their associated control at the left-top, right-top, left-bottom, and right-bottom respectively of the current *view frame* of the parent object.

Line aligned

These are the alternatives `Left`, `Center`, and `Right`. These attributes place their associated control below all previous line aligned controls in the vertical position, and left-aligned, centered, or right-aligned respectively with respect to the current *view frame* of the parent object in the horizontal direction.

Relative position
These are the alternatives `LeftOf`, `RightTo`, `Above`, `Below`, and `LeftOfPrev`, `RightToPrev`, `AbovePrev`, and `BelowPrev`. The first four alternatives must be parameterised with the `Id` of a control that is element of the same parent object, otherwise a runtime error will occur. The latter four alternatives can be defined in terms of the first four but have the advantage that you do not have to create `Id` values for controls that you only want to relatively place other controls to. Placing controls relatively to other controls must construct a *tree* of related controls: cyclic references are not allowed and result also in a runtime error.

Controls that have a relative position layoutform a *layout tree* with one *layout root* control. The layout attribute of the layout root control determines the layout positions of the whole layout tree. If it is at a fixed position, the layout tree obtains a fixed position. If it is boundary aligned, the layout tree is aligned at the same boundary. If it is line aligned, the layout tree is line aligned.

Except for the *first* layout root control the default layout attribute for controls is `(RightToPrev,zero)`. For the first layout root control the default attribute is `(Left,zero)`. Consequently, the default layout order is from left to right in one single line.

Controls are allowed to overlap partially or completely. This is particularly useful in case of combinations of hidden and visible controls when at all times only one is visible. It allows the program to change the control structure in an easy way by hiding and showing controls.

In the remaining part of this section a number of examples are given to illustrate control layout. In each of the examples we assume that the controls that are being laid out are placed in a parent object with a view *domain* and view *frame* as given in Figure 7.4. The x axis (the horizontal arrow) and y axis (the vertical arrow) intersect at the coordinate `zero`.

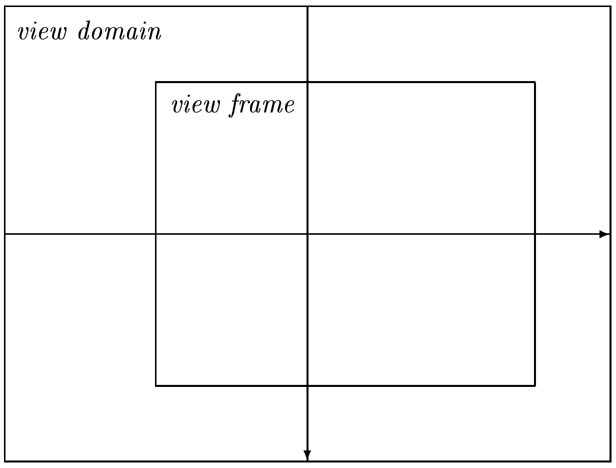


Figure 7.4: View domain and view frame.

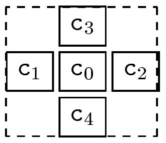


Figure 7.5: A layout tree of five controls.

Controls are being displayed as boxes. Figure 7.5 shows the control configuration that is used in the examples. It consists of five equally sized controls, $c_0 \ldots c_4$. The layout root control is c_0. The controls $c_1 \ldots c_4$ have a relative layout to c_0 with layout attributes `LeftOf`, `RightTo`, `Above`, and `Below` respectively. The offsets are `zero`.

7.3.1 Layout at fixed position

Controls and layout trees that have a `Fix` layout attribute are being placed relative to the view domain of the parent object. So their visibility depends on the current orientation of the parent view frame (recall that the view frame clips everything that is outside of it). Figure 7.6 shows the control configuration of Figure 7.5 when the layout root control has the attribute `(Fix,zero)`.

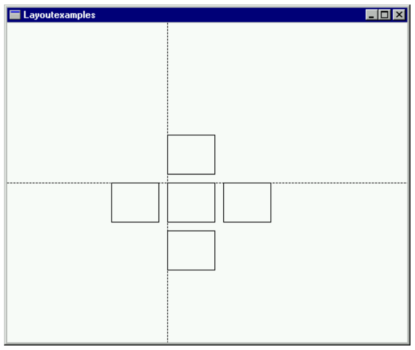


Figure 7.6: The layout tree at `(Fix,zero)`.

7.3.2 Layout at view frame boundary

Placing a control at a boundary aligned position ensures that the control is always visible, because its position is relative to the view frame of its parent object (provided that the view frame is large enough). If this control is the layout root, then the visibility of the controls in its layout tree depend on their relative position. For instance, if a layout root control is placed at the left- top of the view frame (using `LeftTop`), then relatively placed controls at its left and above it are invisible, assuming all controls have zero offsets.

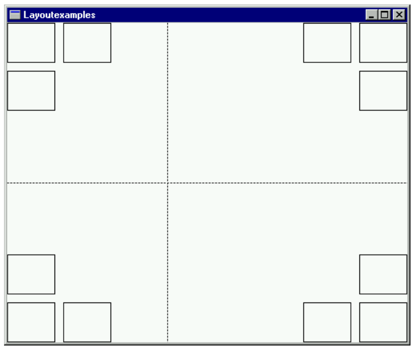


Figure 7.7: The layout tree at `LeftTop`, `RightTop`, `LeftBottom`, and `RightBottom`.

This is illustrated in Figure 7.7. It shows the positions of the control configuration of Figure 7.5 when positioned at every corner of the view frame using `LeftTop`, `RightTop`, `LeftBottom`, and `RightBottom` respectively with `zero` offsets.

7.3.3 Layout in lines

Laying out controls and layout trees in lines is similar to writing characters in an English piece of text: each next control is placed right next to the previous control until a new line is started. A new line starts below the previous line. Lines can be *left* aligned, *centered*, or *right* aligned. The layout attributes `Left`, `Center`, and `Right` introduce a new line and its alignment. The first line starts at the top of the view frame. If the view frame is not large enough to hold all the lines, then the controls in these lines will not be visible.

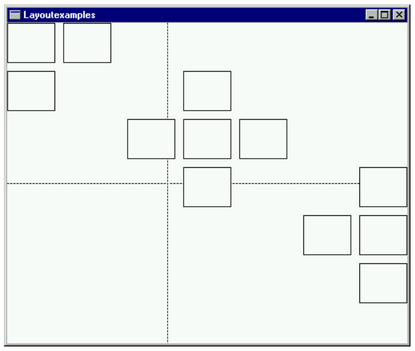


Figure 7.8: The layout tree at `Left`, `Center`, and `Right`.

This is illustrated in Figure 7.8. It shows the positions of the control configuration of Figure 7.5 when positioned at `Left`, `Center`, and `Right` respectively, using `zero` offsets.

7.3.4 Layout offsets

So far we have used `zero` offsets in the layout attribute examples. The layout position of a control is changed by an offset vector value `v = {vx,vy}` as follows: first, the layout position of the control is calculated as explained above, using a `zero` offset. Now assume that this results in the *exact* location `pos = {x,y}`. Then the real position of the control is `{x=x+vx, y=y+vy}`. Figure 7.9 illustrates this. Given two controls $\texttt{c}_0$ and $\texttt{c}_1$ it shows the result of placing $\texttt{c}_1$ at `RightTo` control $\texttt{c}_0$ with an offset value `v = {vx,vy}`. The dashed box shows the location of $\texttt{c}_1$ using a `zero` offset.

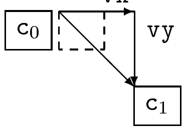


Figure 7.9: Laying out controls using an offset vector.

7.3.5 Layout relative to the previous control

As explained earlier in this section, the default layout attribute of a control is `(RightToPrev,zero)`. The other layout attributes that refer to the previous control are `LeftOfPrev`, `AbovePrev`, and `BelowPrev`. In this section we explain what the previous control is.

Section 7.2 introduced the glue to create control structures. The best way to look at such a control structure is to have a look at its *numbered* graph structure. Consider the following expression: `(a:+:b:+:c)` with `a`, `b`, and `c` standard `Controls` class instances as introduced in Section 7.1. Figure 7.10 shows the graph structure (recall that `:+:` is right associative).

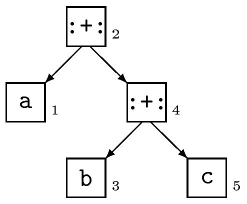


Figure 7.10: The numbered graph of `(a:+:b:+:c)`.

Each node in the graph has an index. If a node is a glue node, then first number the left sub tree, then the node itself, then the right sub tree. If a node is a standard `Controls` class instance, then number it. The nodes of the sub tree of a `CompoundControl` are not numbered. Proceeding in this way, one obtains the index figures at each of the nodes in Figure 7.10. If a node in the graph with index i represents one of the standard `Controls` class instances then its *previous* control is represented by that node in the graph that has the highest index less than i and represents also one of the standard `Controls` class instances. So the previous control of `c` is not `:+:`$_4$ but `b`$_3$ because we assumed that `b` is an instance of the standard `Controls` class. Analogously, the previous element of `b` is neither one of the two `:+:` nodes, but `a`$_1$. Finally, `a` has no previous control.

7.4 Resizing controls

The object I/O system has a simple mechanism to let controls respond to resize actions of their parent object. If a control wants to respond to resize events, it should have a `ControlResize` attribute. It is defined as follows:

```
::  ControlAttribute st
 =  ... | ControlResize ControlResizeFunction | ...
::  ControlResizeFunction
:== Size -> Size -> Size -> Size
```

The control resize function is applied to the current *outer size* of the associated control, the old *view frame size* of its parent object, and the new *view frame size* of its parent object. The result size is supposed to be its new *outer size*. This calculation is performed for all controls that are part of the object that is being resized. If a hierarchical control has a resize function, and its new size is different from its old size, then this computation continues recursively, otherwise the layout of its elements is not recalculated. Given the new sizes of the controls, the layout is recalculated and adjusted accordingly. The effect of this strategy is that the relative layout of controls is never changed.

As an example, consider one wants to have a `CompoundControl` that always displays three `CustomControls` next to each other at the top of its view frame. The `CompoundControl` takes care that its view frame width is always dividable by three, using its own `ControlResize` function `compoundresize`. Its `ControlLook` function draws a rectangle fitting its current view frame.

```
compound = CompoundControl
             (ListLS [custom,custom,custom])
             [ControlResize     compoundresize
             ,ControlViewSize   compoundsize
             ,ControlLook       True (\_ {newFrame}->draw newFrame)
             ,CompoundHMargin   0 0
             ,CompoundVMargin   0 0
             ,CompoundItemSpace 0 0
             ]
compoundsize = {w=60,h=75}
compoundresize _ _ newparentsize=:{w}
             = {newparentsize & w=w/3*3}
```

The `CustomControls` resize their widths according to the new width of their parent control, using the `ControlResize` function `customresize`. For their look function we reuse the look function mentioned in Section 6.4.1, page 57. This function draws a rectangle fitting its current view frame and its two diagonals.

```
custom
 = CustomControl {w=compoundsize.w/3,h=6} crossbox
                 [ControlResize customresize]
customresize customsize _ {w}
 = {customsize & w=w/3}
```

Figure 7.11 shows what happens with the controls when the parent object is resized. At the left the initial state of the `CompoundControl` and its `CustomControls` is displayed. As explained in Section 7.3, the three custom controls form a layout tree with the layout root control having the layout attribute `(Left,zero)` and the other `CustomControls` `(RightToPrev,zero)`. In the middle, the `CompoundControl` is resized to the right and bottom. This resize action causes first recalculation of the size of the `CompoundControl`, using `compoundresize`. Because this value differs from the old size, recalculation continues for each `CustomControl`. The final result is shown at the right.

7.5 Examples

Here are a number of additional examples of using controls. The key spotting and mouse spotting examples already handled in the previous chapter are revisited.

7.5.1 Keyspotting revisited

In this example we extend the keyspotting example in Section 6.7.1 with the possibility to monitor also the keyboard input of controls. We create a `Window` that

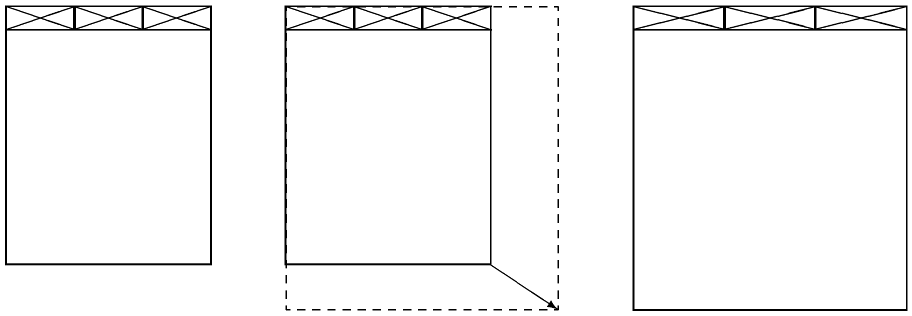
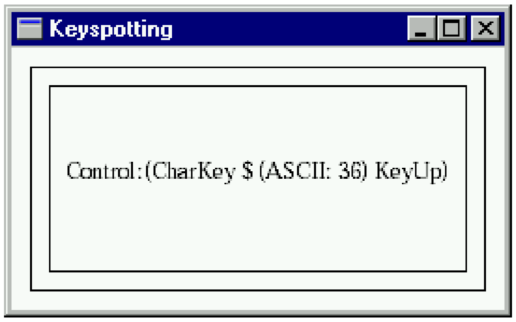


Figure 7.11: Resizing a `CompoundControl` with three `CustomControls`.



Control:(CharKey $ (ASCII: 36) KeyUp)

Figure 7.12: The keyspotting program in action.

contains a `CompoundControl` that contains a `CustomControl`. Before we discuss each of the components below, we have a look at the way they handle keyboard input. A screenshot of the program is given in Figure 7.12.

Each of the components is keyboard sensitive and uses the same `KeyboardFunction`, `spotting`. This function is almost identical to the one presented in Section 6.7.1. The only difference is that the keyboard input is now shown in the `CustomControl` instead of the window. For this purpose `spotting` is parameterised with the `Id` of the `CustomControl`. It is also parameterised with a string that states who currently has the input focus. Now `spotting` changes the `Look` function of the `CustomControl` and forces its update (by using a `True` boolean).

```
spotting cid who x pst
    = appPIO (setControlLook cid True (True,look text)) pst
where
    text = who+++":"+++toString x
```

The `Look` function, `look`, of the `CustomControl` is almost identical to the original look function: except that it centers its argument string it also draws a rectangle around it, so that we can easily see where the components are.

```
look text _ {newFrame} picture
    # picture        = unfill newFrame picture
    # picture        = draw   newFrame picture
    # (width,picture)= getPenFontStringWidth text picture
    = drawAt {x=(w-width)/2,y=h/2} text picture
where
```

```
    {w,h}           = rectangleSize newFrame
```

The `CustomControl`, `custom`, displays which component is currently receiving what keyboard input. The control is identified by `cid`. It parameterises its `KeyboardFunction` `spotting` with its `Id` and the string `"Control"`. Its initial look draws a box around itself. The control is resizable, specified by adding the `ControlResize` attribute. Whenever the parent object is resized, `custom` changes its size in exactly the same amount.

```
custom = CustomControl customsize (look "")
          [ ControlKeyboard (const True) Able
                            (noLS1 (spotting cid "Control"))
          , ControlId       cid
          , ControlResize   resize
          ]
resize oldCSize oldParentSize newParentSize
       = { w = oldCSize.w+newParentSize.w-oldParentSize.w
         , h = oldCSize.h+newParentSize.h-oldParentSize.h
         }
```

The `CompoundControl`, compound, contains only `custom`. It parameterises its `KeyboardFunction` `spotting` with the `Id` of `custom` and the string `"Compound"`. Its initial view frame size is chosen such that it is large enough to display `custom` completely. It conveniently uses the `look` function to draw a box around itself. The compound control is also resizable, and uses the same resize function as `custom`.

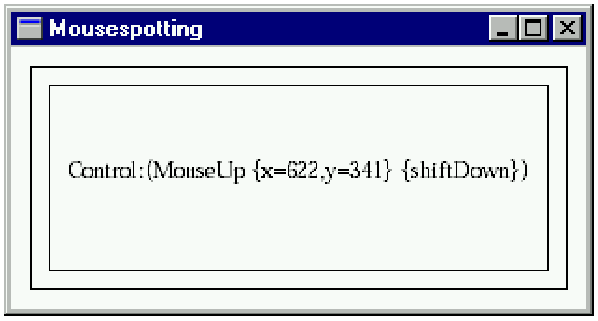
```
compound = CompoundControl custom
            [ ControlKeyboard (const True) Able
                              (noLS1 (spotting cid "Compound"))
            , ControlViewSize {w=customsize.w+2*margin
                              ,h=customsize.h+2*margin
                              }
            , ControlResize   resize
            , ControlLook     True (look "")
            ]
```

Finally, the `Window`, `window`, contains only `compound`. It parameterises its `spotting` function with the `Id` of `custom` and the string `"Window"`. Its initial size is chosen such that it is large enough to display `compound` completely. For this purpose also the margin attributes are set. Termination of the program is taken care of by having the program quit when the user closes the window.

```
window = Window "keyspotting" compound
          [ WindowKeyboard  (const True) Able
                            (noLS1 (spotting cid "Window"))
          , WindowViewSize  {w=customsize.w+4*margin
                            ,h=customsize.h+4*margin
                            }
          , WindowHMargin   margin margin
          , WindowVMargin   margin margin
          , WindowClose     (noLS closeProcess)
          ]
```

Remaining details that need to be defined are the actual creation of the interactive program and its window. For completeness, we include the program code of `keyspotting` below.

```
module keyspotting

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program monitors keyboard input that is sent to a Window which consists
//  of a CompoundControl which consists of a CustomControl.
//  ******************************************************************************

import StdEnv,StdIO

Start :: *World -> *World
Start world
    # (cid,world)   = openId world
    # custom        = CustomControl customsize (look "")
                        [   ControlKeyboard (const True) Able
                                            (noLS1 (spotting cid "Control"))
                        ,   ControlId       cid
                        ,   ControlResize   resize
                        ]
    # compound      = CompoundControl custom
                        [   ControlKeyboard (const True) Able
                                            (noLS1 (spotting cid "Compound"))
                        ,   ControlViewSize {w=customsize.w+2*margin
                                            ,h=customsize.h+2*margin
                                            }
                        ,   ControlResize   resize
                        ,   ControlLook     True (look "")
                        ]
    # window        = Window "keyspotting" compound
                        [   WindowKeyboard  (const True) Able
                                            (noLS1 (spotting cid "Window"))
                        ,   WindowViewSize  {w=customsize.w+4*margin
                                            ,h=customsize.h+4*margin
                                            }
                        ,   WindowHMargin   margin margin
                        ,   WindowVMargin   margin margin
                        ,   WindowClose     (noLS closeProcess)
                        ]
    = startIO SDI Void (snd o openWindow Void window)
                        [ProcessClose closeProcess] world
where
    customsize  = {w=550,h=100}
    margin      = 10
    resize oldCSize oldParentSize newParentSize
        = { w = oldCSize.w+newParentSize.w-oldParentSize.w
          , h = oldCSize.h+newParentSize.h-oldParentSize.h
          }
    spotting cid who x pst
        = appPIO (setControlLook cid True (True,look text)) pst
    where
        text    = who+++":"+++toString x
    look text _ {newFrame} picture
        # picture           = unfill newFrame picture
        # picture           = draw   newFrame picture
        # (width,picture)   = getPenFontStringWidth text picture
        = drawAt {x=(w-width)/2,y=h/2} text picture
    where
        {w,h}               = rectangleSize newFrame
```



Control:(MouseUp {x=622,y=341} {shiftDown})

Figure 7.13: The mousespotting program in action.

7.5.2 Mousespotting revisited

In this example we extend the mouse spotting example of Section 6.7.2. Just like then, the revised mouse spotting example is almost identical to the revised keyspotting example discussed above. The only differences are the title of the window and the replacement of the keyboard attributes by mouse attributes. A screenshot of the program is given in Figure 7.13. For completeness, we show the code of the revised mouse spotting example below.

```
module mousespotting

//  **********************************************************************************
//  Clean tutorial example program.
//
//  This program monitors mouse input that is sent to a Window which consists
//  of a CompoundControl which consists of CustomControl.
//  **********************************************************************************

import StdEnv,StdIO

Start :: *World -> *World
Start world
    # (cid,world)   = openId world
    # custom        = CustomControl customsize (look "")
                        [   ControlMouse    (const True) Able
                                            (noLS1 (spotting cid "Control"))
                        ,   ControlId       cid
                        ,   ControlResize   resize
                        ]
    # compound      = CompoundControl custom
                        [   ControlMouse    (const True) Able
                                            (noLS1 (spotting cid "Compound"))
                        ,   ControlViewSize {w=customsize.w+2*margin
                                            ,h=customsize.h+2*margin
                                            }
                        ,   ControlResize   resize
                        ,   ControlLook     True (look "")
                        ]
    # window        = Window "mousespotting" compound
                        [   WindowMouse     (const True) Able
                                            (noLS1 (spotting cid "Window"))
                        ,   WindowViewSize  {w=customsize.w+4*margin
                                            ,h=customsize.h+4*margin
                                            }
                        ,   WindowHMargin   margin margin
                        ,   WindowVMargin   margin margin
                        ,   WindowClose     (noLS closeProcess)
                        ]
    = startIO SDI Void (snd o openWindow Void window)
                        [ProcessClose closeProcess] world
where
    customsize  = {w=550,h=100}
    margin      = 10
    resize oldCSize oldParentSize newParentSize
        = { w = oldCSize.w+newParentSize.w-oldParentSize.w
          , h = oldCSize.h+newParentSize.h-oldParentSize.h
          }
    spotting cid who x pst
        = appPIO (setControlLook cid True (True,look text)) pst
    where
        text    = who+++":"+++toString x
    look text _ {newFrame} picture
        # picture           = unfill newFrame picture
        # picture           = draw   newFrame picture
        # (width,picture)   = getPenFontStringWidth text picture
        = drawAt {x=(w-width)/2,y=h/2} text picture
    where
        {w,h}               = rectangleSize newFrame
```

Chapter 8

Menus

Many interactive applications allow a user to manipulate a number of documents. As we saw in the previous chapter, the user can issue these manipulations by means of the keyboard and mouse. Another common source of manipulations is by issueing *commands* to the application. This is where *menus* come in. Menus help a program to structure the set of available commands. To the user of an application, the use of menus provides a consistent and easily browsable graphical display of the set of available commands. For these reasons it is recommended to use menus in a program.

In Section 8.1 we introduce the standard set of menu definitions that are at a programmers disposal. There are two top-level menus, the common menu, and the pop up menu. Then the glue is introduced to create larger menu structures in Section 8.2. One special menu is available for interactive processes that have the multiple document interface (MDI) attribute, the *windows* menu. This menu enumerates the currently open and visible windows of that process and give some commands to organise them. This is treated in Section 8.3. Menus provide a consistent graphical interface to users. To enhance the consistency, a number of programming conventions have evolved. These are discussed in Section 8.4.

8.1 Menus and menu elements

Menus and menu elements can be defined by means of the type definitions in module `StdMenuDef` (Appendix A.19). There are two top-level menu objects: the *menu* that can be found in the *menu bar* of an application, and the *pop up menu* that can be opened at any place in the application area (usually in a window). Menu objects are instances of the type constructor class `Menus`. This class and the instance declarations can be found in `StdMenu`.

```
class Menus mdef where
    openMenu :: .ls !(mdef .ls (PSt .l)) !(PSt .l)
                         -> (!ErrorReport,!PSt .l)
    ...

instance Menus (Menu      m) | MenuElements      m
instance Menus (PopUpMenu m) | PopUpMenuElements m
```

Menus and pop up menus share most of the menu elements. The only exception is that pop up menus are not allowed to have *sub menus*. This restriction is enforced by the two separate menu element classes for the two menus, `MenuElements` and `PopUpMenuElements` respectively.

Menus and their elements share the same set of *menu attributes*. Before we introduce each of the menu components, we will focus on the attributes first.

8.1.1 The menu attributes

The menu attributes are used by both menus and menu elements. They are the following:

```
:: MenuAttribute    st
 = MenuId           Id
 | MenuSelectState  SelectState
 | MenuIndex        Int
 | MenuInit         (IdFun st)
 | MenuFunction     (IdFun st)
 | MenuMarkState    MarkState
 | MenuModsFunction (ModifiersFunction st)
 | MenuShortKey     Char
```

`MenuId`
: This attribute identifies the menu or menu element to which it is associated. If you do not provide a `MenuId` the menu (element) can not be modified by the program.

`MenuSelectState`
: This attribute defines whether the menu (element) can be used by the user (`Able`) or not (`Unable`). Usually this will affect the look of the menu (element). The default value is `Able`.

`MenuIndex`
: This attribute defines the index position of a menu. Index positions range from one (for the first menu) upto the number of menus. A negative or zero `MenuIndex` attribute value will place the menu in front of all current menus. A `MenuIndex` attribute value that is larger than the current number of menus will place the menu behind all current menus. Other `MenuIndex` attribute values place the menu *behind* the menu with that index value.

`MenuInit`
: This attribute defines an action that should be performed immediately after opening the menu. This is equivalent to the window initialisation action (see Section 6.2.1). If no `MenuInit` attribute is provided, no additional action is performed.

`MenuFunction` **and** `MenuModsFunction`
: These two attributes add callback functions to menu elements that are evaluated when the menu element to which they are associated is selected by the user. The difference between these two attributes is that the former is simply evaluated whenever the menu element is selected, and that the latter provides the callback function with the modifier keys that have been pressed at the moment of selecting the menu element. In a `MenuAttribute` list, the first of these two attributes is chosen.

`MenuMarkState`
: This attribute adds a check mark symbol (`Mark`) or leaves it out (`NoMark`) to a menu element. The default value is `NoMark`.

`MenuShortKey`

This attribute defines a character that can be used by user to select the menu element to which the character is associated by means of the keyboard. The menu element can be selected by pressing that character and some special, platform dependent modifier key.

The object I/O library also supports the standard Windows keyboard interface of user selection of menus and menu elements. In this interface, a menu (element) can be selected by the keyboard by marking one of the characters in its title. A character is marked by prefixing it with '&'. The escape sequence is '&&'.

8.1.2 The Menu

A *menu* is a top level interface element that contains a group of related commands. The menu appears in the menu bar of the interactive process. The definition of a menu is as follows:

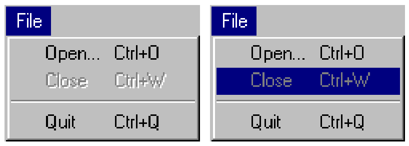
```
:: Menu m ls pst = Menu Title (m ls pst) [MenuAttribute *(ls,pst)]

instance Menus (Menu m) | MenuElements m
```

The usual appearance of a menu is by its title, which is displayed in the menu bar. The user can browse through its commands by mouse or by a platform dependent keyboard interface. Valid menu attributes are:

MenuAttribute:	**Valid:**	**MenuAttribute:**	**Valid:**
`MenuFunction`		`MenuMarkState`	
`MenuId`	√	`MenuModsFunction`	
`MenuIndex`	√	`MenuSelectState`	√
`MenuInit`	√	`MenuShortKey`	

Example The left picture shows the menu when not selected by the user, the right picture when selected by the user.

```
menu
  = Menu "File"
      (   MenuItem "Open..." [MenuShortKey    'o']
      :+: MenuItem "Close"   [MenuSelectState Unable
                             ,MenuShortKey    'w'
                             ]
      :+: MenuSeparator      []
      :+: MenuItem "Quit"    [MenuShortKey    'q']
      )   []
```



8.1.3 The PopUpMenu

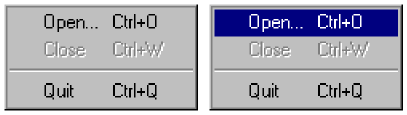
A *pop up menu* is a top level interface element that contains a group of related commands. It can appear at any place in an interactive process. Its definition is actually a restricted version of that of a menu:

```
:: PopUpMenu m ls pst = PopUpMenu (m ls pst)

instance Menus (PopUpMenu m) | PopUpMenuElements m
```

Pop up menus have no title and no attributes.

Example Here is an example of a pop up menu, containing the same elements as the menu example above. The image to the left shows the menu when opened, the image to the right when the user selects an item.

```
menu
  = PopUpMenu
      (   MenuItem "Open..." [MenuShortKey    'o']
      :+: MenuItem "Close"   [MenuSelectState Unable
                             ,MenuShortKey    'w'
                             ]
      :+: MenuSeparator      []
      :+: MenuItem "Quit"    [MenuShortKey    'q']
      )
```



8.1.4 The MenuItem

The *menu item* is the standard element that refers to a command. The definition of a menu item is as follows:

```
:: MenuItem ls pst = MenuItem Title [MenuAttribute *(ls,pst)]

instance MenuElements      MenuItem
instance PopUpMenuElements MenuItem
```

The title of a menu element is displayed as a member of its parent menu. When the user selects the menu item its `Menu(Mods)Function` attribute is evaluated if the menu item, and all of its parent menus are `Able`. The appearance of the menu item reflects this state. Valid menu item attributes are:

MenuAttribute:	**Valid:**	**MenuAttribute:**	**Valid:**
`MenuFunction`	√	`MenuMarkState`	√
`MenuId`	√	`MenuModsFunction`	√
`MenuIndex`		`MenuSelectState`	√
`MenuInit`		`MenuShortKey`	√

An example of a menu item is given in the previous sections 8.1.2 and 8.1.3.

8.1.5 The MenuSeparator

The *menu separator* is a menu element that is only used to provide users with visual clues that separate groups of related menu elements within a parent menu. Graphically, a menu separator usually inserts some vertical space and a horizontal divider within a menu. Menu separators have no further functionality. The definition of a menu separator is as follows:

```
:: MenuSeparator ls pst = MenuSeparator [MenuAttribute *(ls,pst)]

instance MenuElements      MenuSeparator
instance PopUpMenuElements MenuSeparator
```

Because menu separators have no other purpose than providing some 'white space' between menu elements, the only valid menu separator attribute is the `MenuId` (which can be used to remove a separator dynamically):

MenuAttribute:	**Valid:**	**MenuAttribute:**	**Valid:**
`MenuFunction`		`MenuMarkState`	
`MenuId`	√	`MenuModsFunction`	
`MenuIndex`		`MenuSelectState`	
`MenuInit`		`MenuShortKey`	

In the menu examples in the sections 8.1.2 and 8.1.3 menu separators have been used.

8.1.6 The RadioMenu

A *radio menu* element is a group of menu items of which exactly one menu item is selected. All alternatives are visible. The definition of a radio menu is as follows:

```
:: RadioMenu ls pst =   RadioMenu [MenuRadioItem *(ls,pst)] Index
                                  [MenuAttribute *(ls,pst)]
:: MenuRadioItem st :== (Title,Maybe Id,Maybe Char,IdFun st)

instance MenuElements      RadioMenu
instance PopUpMenuElements RadioMenu
```

The initially selected item is indicated by the `Index` argument. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by $1 \ldots n$. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid radio menu attributes are:

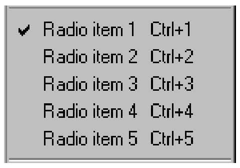
MenuAttribute:	**Valid:**	**MenuAttribute:**	**Valid:**
`MenuFunction`		`MenuMarkState`	
`MenuId`	√	`MenuModsFunction`	
`MenuIndex`		`MenuSelectState`	√
`MenuInit`		`MenuShortKey`	

When an item of the radio menu is selected the previously selected radio menu item will be unchecked, and the new radio menu item gets the check mark. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected radio menu item is selected.

Example A radio menu and its initial look (it is instructive to compare this with the radio *control* example at page 82).

```
radiomenu
```

```
    = RadioMenu
      [  ("Radio item "+++toString i,Nothing,Just (iChar i),id)
      \\ i<-[1..5]
      ]  1 []
  where
    iChar i = toChar (toInt '1'+i-1)
```



8.1.7 The SubMenu

The *sub menu* is a menu element that contains other menu elements. So it is a menu within a menu, and can of course contain sub menus as well. The definition of a sub menu is therefore very similar to that of a menu:

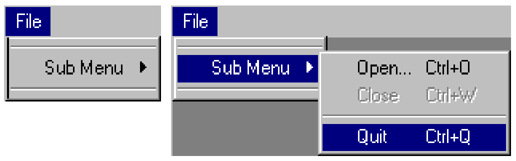
```
:: SubMenu m ls pst
 = SubMenu Title (m ls pst) [MenuAttribute *(ls,pst)]

instance MenuElements (SubMenu m) | MenuElements m
```

The usual appearance of a sub menu is by its title. This title appears in the menu that contains the sub menu. On most system some additional visual feedback is provided to inform the user that this item is a sub menu. The user can browse through its elements by mouse or by a platform dependent keyboard interface. Valid sub menu attributes are:

MenuAttribute:	**Valid:**	**MenuAttribute:**	**Valid:**
MenuFunction MenuId MenuIndex MenuInit	√	MenuMarkState MenuModsFunction MenuSelectState MenuShortKey	√

Example This sub menu has the same definition as the menu at page 103 except that it has a SubMenu data constructor rather than the Menu data constructor and a different title. The left picture shows the sub menu when not selected by the user, the right picture when selected by the user.

```
submenu
  = SubMenu "Sub Menu"
      (   MenuItem "Open..." [MenuShortKey    'o']
      :+: MenuItem "Close"   [MenuSelectState Unable
                             ,MenuShortKey    'w'
                             ]
      :+: MenuSeparator      []
      :+: MenuItem "Quit"    [MenuShortKey    'q']
      )   []
```



8.2 Menu glue

In the previous section the standard set of menus and menu elements has been discussed. This list is not complete. In the library module `StdMenuElementClass` (Appendix A.21) a number of additional instances are defined, namely the type constructors `:+:`, `ListLS`, `NilLS`, and `AddLS`, `NewLS` (their definition can be found in Appendix A.14). These additional instances are required to *glue* menu elements. They are treated below.

8.2.1 :+:

The most common constructor to glue menu elements is `:+:`. Its type constructor definition and `MenuElements` class instance declaration are as follows:

```
::  :+: t1 t2 ls cs
 = (:+:) infixr 9 (t1 ls cs) (t2 ls cs)

instance MenuElements      ((:+:) m1 m2) | MenuElements      m1
                                         & MenuElements      m2
instance PopUpMenuElements ((:+:) m1 m2) | PopUpMenuElements m1
                                         & PopUpMenuElements m2
```

Assume that we have two `MenuElements` (or `PopUpMenuElements`) instances `m1` and `m2`, working on the same local state of type `ls` and context state `cs`. Now the glued expression `m1:+:m2` is also a `MenuElements` (`PopUpMenuElements`) instance working on the same local state and context state. Because `:+:` is right associative, the expression `m1 :+: m2 :+: m3` should be read as `m1 :+: (m2 :+: m3)`.

As an example, consider the following two menu elements:

```
item      = MenuItem "Hello there!" []
separator = MenuSeparator           []
```

Now the expressions `(item :+: separator)` and `(separator :+: item)` are legal combinations.

8.2.2 ListLS and NilLS

The `:+:` type constructor is not always the most appropriate glue. When constructing sets of menu elements of the *same type*, it is much more convenient to use lists and list comprehensions. When constructing an unknown number of menu elements, it is even impossible to use `:+:`. Again, lists provide more flexibility. List-like glue is provided by the type constructor `ListLS`. The type constructor `NilLS` is a shorthand for `ListLS []`. It can also be conveniently used to state that a sub menu or top level menu has no menu elements. Here are the definitions.

```
::  ListLS t ls cs = ListLS [t ls cs]
::  NilLS    ls cs = NilLS

instance MenuElements      (ListLS m) | MenuElements      m
instance MenuElements      NilLS
instance PopUpMenuElements (ListLS m) | PopUpMenuElements m
instance PopUpMenuElements NilLS
```

Given a list of `MenuElements` (`PopUpMenuElements`) instances `ms = [`$\mathtt{m}_1$ `...`$\mathtt{m}_n$`]`, working on the same local state of type `ls` and context state `cs`, then the expression `ListLS ms` is also a `MenuElements` (`PopUpMenuElements`) instance working on the same local state and context state.

8.2.3 `AddLS` and `NewLS`

The previously discussed glueing type constructors always glue menu elements that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, `AddLS` and `NewLS`.

```
:: AddLS t ls cs = E. .new: { addLS::new, addDef::t *(new,ls) cs }

instance MenuElements      (AddLS m) | MenuElements      m
instance PopUpMenuElements (AddLS m) | PopUpMenuElements m
```

Given a `MenuElements` (`PopUpMenuElements`) instance `m1` that works on a local state of type `ls` and a context state of type `cs`, one can add another `MenuElements` (`PopUpMenuElements`) instance `m2` that works on an *extended* local state of type `(new,ls)` and the same context state of type `cs`. Let `x` be a value of type `new`, then this is done by the expression `m1 :+: {addLS=x, addDef=m2}`.

```
:: NewLS t ls cs = E. .new: { newLS::new, newDef::t new cs }

instance MenuElements      (NewLS m) | MenuElements      m
instance PopUpMenuElements (NewLS m) | PopUpMenuElements m
```

Given a `MenuElements` (`PopUpMenuElements`) instance `m1` that works on a local state of type `ls` and a context state of type `cs`, one can add another `MenuElements` (`PopUpMenuElements`) instance `m2` that works on a *new* local state of type `new` and the same context state of type `cs`. Let `x` be a value of type `new`, then this is done by the expression `m1 :+: {newLS=x, newDef=m2}`.

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification. Section 8.5 contains an example of the use of `NewLS`.

8.3 The Windows menu

For programs using interactive processes that have the multiple document interface (for MDI processes, see Section 11.1) one special menu is added by the object I/O system to the menu system of such an interactive process. This is the *Windows menu.* This menu contains a number of commands to arrange the current set of visible windows, and a list of all window titles. The set of arrange commands is platform dependent, but contains at least the following three commands:

Cascade:
This command arranges all windows to have equal size. They will be placed in diagonal order: from the left top to the right bottom of the process window. The windows will always be shown completely inside the process window. If there are to many windows, the arrangements starts over again for the remaining windows, which will therefore overlap the other windows.

Tile Horizontally:
This command arranges all windows in rows without overlapping. It is tried to give all windows the same size.

Tile Vertically:
This command arranges all windows in columns without overlapping. It is tried to give all windows the same size.

Separated from these commands by a `MenuSeparator` is the list of open windows in lexicographical order on the window title. The currently active window is checked. Selecting any of these windows activates that window. (This causes the previously active window to become deactivated, and the new window to become active.)

8.4 Menu conventions

The use of menus provides application users with a consistent and uniform access to the available set of commands. In this section we discuss a number of conventions that are usually followed to increase the level of consistency.

8.4.1 Subsetting the available commands

In general when using an interactive application, the application will move through several states. In each state a particular subset of the complete set of available commands will be applicable to the user while the remaining commands should not be selected. A well designed application should make this clear to the user by *subsetting* the available commands.

The easiest way to subset commands is by *disabling* and *enabling* the menu elements that should be unselectable and selectable respectively. For this purpose the functions `enableMenuElements` and `disableMenuElements` (module `StdMenuElement`, Appendix A.20) are available to enable and disable individual menu elements. Complete menus can be enabled and disabled using the functions `enableMenus` and `disableMenus` (module `StdMenu`, Appendix A.17). This module also contains two functions to enable and disable the whole current set of menus: `enableMenuSystem` and `disableMenuSystem`.

Another way to subset commands is to have at all times only those menus and menu elements open that should be available. A disadvantage of this technique is that it can become complicated quickly, and that it leaves the user disoriented when the set of commands changes frequently.

8.4.2 Command conventions

In this section we discuss some frequently used conventions in applications with respect to commands.

Clipboard commands

Applications that support the use of the clipboard (Chapter 12) to *cut*, *copy*, and *paste* private and external data usually are found in an `"Edit"` menu. Conventions are:

Cut:
: This command should be enabled only if the application is in a state that an object has been selected that can be transfered to the clipboard. Issueing this command should remove that object from its context and place it in the clipboard. Its name should be `"Cut"` and it should have the shortkey attribute `'x'`.

Copy:
: This command should be enabled only if the application is in a state that an object has been selected that can be transfered to the clipboard. Issueing this command should place it in the clipboard, but not remove it from its context. Its name should be `"Copy"` and it should have the shortkey attribute `'c'`.

Paste:
: This command should be enabled only if the clipboard contains an object that can be currently incorporated in the application. Issueing this command should read the clipboard and put that object in the application. Its name should be `"Paste"` and it should have the shortkey attribute `'v'`.

Undo command

Applications that allow users to manipulate documents by sequences of commands can support an *undo* command. The undo command can also be undone by a *redo* command. These commands are usually found in an `"Edit"` menu. Conventions are:

Undo:
: This command should be enabled only if the user has issued a sequence of commands that can be undone. The number of undoable commands depends on the sophistication of the application. The name of this command is `"Undo"` and it has the shortkey attribute `'z'`.

Redo:
: This command should be enabled only if a (sequence of) undo command has been issued. At each selection it restores the changes of the undo command. The name of this command is `"Redo"` and it has the shortkey attribute `'y'`.

Document commands

The document commands are frequently found commands to create new documents, open existing documents, and save and close open documents. These commands are usually found in a `"File"` menu. Conventions are:

New:
: This command should be enabled only if the application can modify a new document. Issueing this command should create or reuse a window containing the new document. Its name should be `"New"` and it should have the shortkey attribute `'n'`.

Open:
This command should be enabled only if the application can modify an additional, existing document. Issueing this command should give the user the opportunity to search for a file that will be opened by the application. For this purpose the `StdFileSelect` function `selectInputFile` can be used (Appendix A.10). The name of the command should be `"Open..."` and it should have the shortkey attribute `'o'`.

Close:
This command should be enabled only if the application has an open document window or dialogue. Issueing this command should close the currently active window or dialogue. It is good programming practice to check if the document has been recently saved. If this is not the case, then the user should be asked if the document should be saved before closing. The name of this command should be `"Close"` and it should have the shortkey attribute `'w'`.

Save:
This command should be enabled only if the currently active document window version differs from a (possibly not present) file version. Issueing this command should save the current state of the document in the active window to file. If there is no file associated yet, then the application should first ask for a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.10). The name of the command should be `"Save"` and it should have the shortkey attribute `'s'`.

Save As:
This command should be enabled only if the application has an open document window. Issueing this command should give the user the possibility to browse the file system and provide a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.10). The name of the command should be `"Save As..."`

Quit or Exit command

Users can leave an application using the *quit* or *exit* command. A user should always be allowed to leave the application. It is good programming practice to check if there are any unsaved documents in the application. If this is the case then the user should be asked if these documents should be saved before closing. The name of the quit command is usually `"Quit"` (with shortkey attribute value `'q'`) or `"Exit"`. This command is usually found in a `"File"` menu.

8.5 Example: a small menu system

In this section we show a program that creates an interactive process with a multiple document interface. It can serve as a framework for writing your own multiple document interface programs. The interactive process contains one menu, *File*, which contains three commands: a command to open a new window, *New*; a command to close the active window, *Close*; and a command to leave the application, titled *Quit*. The *New* and *Quit* commands are always available. The *Close* command will be subsetted using enabling and disabling (as described in Section 8.4.1). We start with the commands, and proceed with the menu and process definition.

The *New* command is a `MenuItem` with a local state of type `Int`. The local state is used to give every new window a new title by appending the current value to the string "Window". The initial value is one. Here is the definition of the command:

```
 { newLS =1
 , newDef=MenuItem "&New" [MenuShortKey 'n',MenuFunction new]
 }
```

The callback function of the *New* command, new, uses the local integer state. For each successfully created window the value is incremented. In addition, the *Close* command is enabled. For now, assume that the *Close* command is identified by the Id value closeid. If the window can not be created a notice is opened, using the notice library developed in Section 6.8.1.

```
new :: (Int,PSt .l) -> (Int,PSt .l)
new (i,pst)
    # (error,pst) = openWindow Void window pst
    | error<>NoError
        # notice  = Notice ["MDI could not open new window"]
                       (NoticeButton "Ok" id)
                       []
        = (i,openNotice notice pst)
    | otherwise
        = (i+1,appPIO (enableMenuElements [closeid]) pst)
```

The *Close* command is identified by the Id closeid. The application starts with no windows, so its initial SelectState is Unable:

```
 MenuItem "&Close" [MenuShortKey 'w',MenuFunction (noLS close)
                   ,MenuId closeid,  MenuSelectState Unable
                   ]
```

The callback function close closes the active window with the StdWindow function closeActiveWindow. It is then checked if there still are open windows. If this is not the case, then close should disable the *Close* command. The function get-WindowsStack returns the list of Ids of all currently open windows. So if this list is empty, then there are no more windows open. We get the following definition:

```
close :: (PSt .l) -> PSt .l
close pst
    # pst          = closeActiveWindow pst
    # (rest,pst)   = accPIO getWindowsStack pst
    | isEmpty rest = appPIO (disableMenuElements [closeid]) pst
    | otherwise    = pst
```

Because we are not interested in the window details, we keep its implementation very simple. It contains no controls, which expressed by the value NilLS. If the user requests to close the window, the *Close* command callback function close is evaluated. The look of the window draws the current view frame rectangle and the lines between the diagonally opposite corner points.

```
window = Window ("Window "+++toString i)
           NilLS
           [ WindowClose    (noLS close)
           , WindowViewSize {w=300,h=300}
```

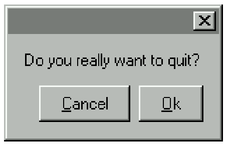


Figure 8.1: The termination notice.

```
           , WindowLook     True look
           ]

look :: SelectState UpdateState *Picture -> *Picture
look _ {newFrame=frame=:{corner1,corner2}} picture
  # picture = unfill frame picture
  # picture = draw frame picture
  # picture = drawLine corner1 corner2 picture
  = drawLine {corner1 & x=corner2.x} {corner2 & x=corner1.x} picture
```

The *Quit* command has a straightforward definition:

```
 MenuItem "&Quit" [MenuShortKey 'q',MenuFunction (noLS quit)]
```

Its callback function `quit` asks the user to confirm this choice. For this purpose we use another notice. If the user confirms the choice, then the application is terminated, otherwise nothing happens. Figure 8.1 shows the notice.

```
quit :: (PSt .l) -> PSt .l
quit pst
    = openNotice notice pst
where
    notice = Notice ["Do you really want to quit?"]
               (NoticeButton "&Ok" (noLS closeProcess))
               [NoticeButton "&Cancel" id]
```

The definition of the *File* menu simply glues the definitions of the commands as described above into one menu definition.

```
menu
 = Menu "&File"
   ( { newLS =1
     , newDef=MenuItem "&New" [MenuShortKey 'n',MenuFunction new]
     }
   :+: MenuItem "&Close" [MenuShortKey 'w',MenuFunction (noLS close)
                         ,MenuId closeid,  MenuSelectState Unable
                         ]
   :+: MenuSeparator     []
   :+: MenuItem "&Quit"  [MenuShortKey 'q',MenuFunction (noLS quit)]
   )   []
```

The remaining details that need to be arranged are the creation of the `Id` value `closeid`, the creation of the menu, and the creation of the parent interactive process of the menu. We conveniently reuse the `quit` function to handle user requests to close the interactive process by setting the (`ProcessClose quit`) attribute. Here is the full program code.

```
module MDI

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program creates a Multiple Document Interface process with a Window menu.
//  ******************************************************************************

import StdEnv, StdIO
import notice

Start :: *World -> *World
Start world
    # (id,world)    = openId world
    = startIO MDI Void (initialise id) [ProcessClose quit] world

quit :: (PSt .l) -> PSt .l
quit pst
    = openNotice notice pst
where
    notice  = Notice ["Do you really want to quit?"]
                (NoticeButton "&Ok"     (noLS closeProcess))
                [NoticeButton "&Cancel" id]

initialise :: Id (PSt .l) -> PSt .l
initialise closeid pst
    # (error,pst)   = openMenu 0 menu pst
    | error<>NoError= abort "MDI could not open File Menu"
    | otherwise     = pst
where
    menu= Menu "&File"
            ( { newLS =1
              , newDef=MenuItem "&New"  [MenuShortKey 'n',MenuFunction new]
              }
            :+: MenuItem "&Close"       [MenuShortKey 'w',MenuFunction (noLS close)
                                        ,MenuId closeid,  MenuSelectState Unable
                                        ]
            :+: MenuSeparator           []
            :+: MenuItem "&Quit"        [MenuShortKey 'q',MenuFunction (noLS quit)]
            )   []

    new :: (Int,PSt .l) -> (Int,PSt .l)
    new (i,pst)
        # (error,pst)   = openWindow Void window pst
        | error<>NoError
            # notice    = Notice ["MDI could not open new window"]
                            (NoticeButton "Ok" id)
                            []
            = (i,openNotice notice pst)
        | otherwise
            = (i+1,appPIO (enableMenuElements [closeid]) pst)
    where
        window  = Window ("Window "+++toString i)
                    NilLS
                    [   WindowClose     (noLS close)
                    ,   WindowViewSize  {w=300,h=300}
                    ,   WindowLook      True look
                    ]

        look :: SelectState UpdateState *Picture -> *Picture
        look _ {newFrame=frame=:{corner1,corner2}} picture
            # picture   = unfill frame picture
            # picture   = draw frame picture
            # picture   = drawLine corner1 corner2 picture
            = drawLine {corner1 & x=corner2.x} {corner2 & x=corner1.x} picture

    close :: (PSt .l) -> PSt .l
close pst
    # pst           = closeActiveWindow pst
    # (rest,pst)    = accPIO getWindowsStack pst
    | isEmpty rest  = appPIO (disableMenuElements [closeid]) pst
    | otherwise     = pst
```

Chapter 9

Timers

Timers provide interactive programs with a tool to let actions occur at regular time intervals. These actions respond to *timer events*, and so they can be properly defined as callback functions. Typical examples of timer applications are blinking cursors, clocks, and time-out mechanisms.

The definition types of timers can be found in module `StdTimerDef`, Appendix A.44. The main type definitions are as follows:

```
::  Timer t ls pst
 =  Timer TimerInterval (t ls pst) [TimerAttribute *(ls,pst)]
::  TimerInterval
:== Int

::  TimerAttribute   st
 =  TimerFunction    (TimerFunction st)
 |  TimerId          Id
 |  TimerInit        (IdFun st)
 |  TimerSelectState SelectState
::  TimerFunction    st
:== NrOfIntervals -> st -> st
```

A `TimerInterval` is an integer value that must be at least zero. The time unit is platform dependent and is defined by the function `ticksPerSecond` (module `StdSystem`, Appendix A.37).

The `TimerAttributes` are the following:

`TimerFunction:`
: This attribute is the callback function that is evaluated when a timer event is handled. Its first argument is the number of whole timer intervals that have elapsed since its previous evaluation, so this value is at least 1. If the timer interval is zero, then this number is always 1. Enabling a timer from a disabled state resets the last evaluation time.

`TimerId:`
: This attribute identifies the timer. If you do not provide a `TimerId` then timer can not be modified or closed by the program (except by closing the parent interactive process).

`TimerInit:`
: This attribute defines an action that should be performed immediately after

opening the timer. This is equivalent to the window and menu initialisation actions (sections 6.2.1 and 8.1.1). If no `TimerInit` attribute is provided, no additional action is performed.

`TimerSelectState:`
: This attribute defines whether the timer will respond to timer events (`Able`) or not (`Unable`). The default value is `Able`.

The `Timer` type constructor is parameterised with a type constructor variable. Analogous to menus that contain menu elements, timers can contain *timer elements*. The instances must be member of the `TimerElements` class. This is expressed by the `Timers` timer *creation* member function, `openTimer` which can be found in module `StdTimer` (Appendix A.42):

```
class Timers tdef where
  openTimer :: .ls !(tdef .ls (PSt .l)) !(PSt .l)
                         -> (!ErrorReport,!PSt .l)

instance Timers (Timer t) | TimerElements t
```

Currently, the instances of the `TimerElements` class are *receivers* and the usual *glueing* type constructors that we have already encountered for controls (Section 7.2) and menu elements (Section 8.2), namely `:+:`, `ListLS`, `NilLS`, and `AddLS`, `NewLS`. The receiver instances are declared in module `StdTimerReceiver` (Appendix A.46). Receivers are handled in Chapter 10. The glueing constructors are declared in the same module `StdTimerElementClass` that contains the `TimerElements` class (Appendix A.45).

9.1 Examples

In this section we give some examples to illustrate the use of timers.

9.1.1 Expanding circles

In this example we create a program that uses a timer to continuously draw a number of expanding concentric circles in a window. It also opens a menu containing only the quit command. Figure 9.1 shows the program in action. We discuss these object I/O components in reverse order (menu, window, timer).

The menu definition is very simple, as it contains only one command to terminate the example program. Recall that to terminate an interactive process the `StdProcess` function `closeProcess` must be used. The `StdIOBasic` function `noLS` suitably turns `closeProcess` into the desired type. Here is the menu definition.

```
mdef = Menu "&Circles"
         (MenuItem "&Quit" [MenuFunction (noLS closeProcess)
                           ,MenuShortKey 'q'
                           ]
         ) []
```

The window is also simple. The initial view frame size of the window in which the circles are to be drawn is `windowEdge` by `windowEdge` (200 in the example). To

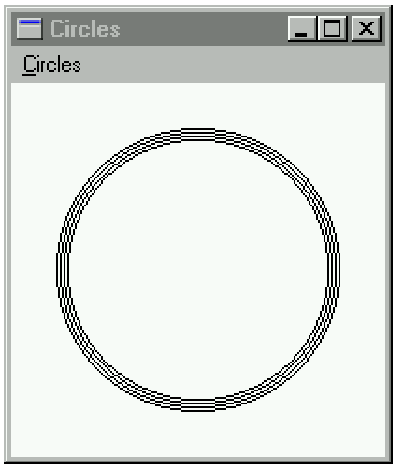


Figure 9.1: The circles program in action.

ease drawing, we take care that the view domain of the window has the origin `zero` exactly in the center of the view domain. This can be done conveniently by defining the `WindowViewDomain` attribute to be:

```
viewDomain
 = { corner1 = {x= ~windowEdge/2,y= ~windowEdge/2}
   , corner2 = {x=  windowEdge/2,y=  windowEdge/2}
   }
```

The window does not contain controls (expressed by the `Controls` type instance `NilLS`). It is identified by the `Id` value `wid`. The window definition is as follows:

```
wdef = Window "Circles" NilLS
         [WindowId         wid
         ,WindowViewSize   (rectangleSize viewDomain)
         ,WindowViewDomain viewDomain
         ]
```

The timer draws a number of concentric circles that have an increasing radius. For this purpose it uses a local state of the following type and initial value:

```
:: TimerState
 = { nrCircles    :: Int
   , equiDistance :: Int
   , minRadius    :: Int
   }
initTimerState = { nrCircles=4, equiDistance=2, minRadius=0 }
```

The `nrCircles` field contains the number of circles that are drawn. The `equi-Distance` field is the difference of radius between two neighbouring circles. The `minRadius` field keeps track of the radius of the smallest visible circle.

The timer interval is set to a twentieth of a second (`ticksPerSecond/20`). The timer contains no timer elements. Its definition is as follows:

```
tdef = Timer (ticksPerSecond/20) NilLS [TimerFunction timer]
```

The timer function `timer` will be evaluated by the object I/O system every twentieth of a second (if possible). The timer function actually ignores the number of elapsed intervals and simply draws the next sequence of circles. There are two cases to distinguish:

- If the smallest circle still fits entirely inside the window (tested by `minRadius < windowEdge/2`), then `timer` *undraws* the smallest circle and *draws* the new circle which should have a radius equal to `minRadius + nrCircles * equiDistance`. Finally, the `minRadius` field is changed to reflect the fact that the smallest visible circle now has radius `minRadius + equiDistance`.
- If the smallest circle does not fit entirely inside the window, then `timer` completely *unfills* the window. By setting the new local `TimerState` back to `initTimerState` the circles are drawn again from the center.

```
timer _ (lst=:{nrCircles,equiDistance,minRadius},pst)
| minRadius<windowEdge/2
  # lst      = {lst & minRadius=minRadius+equiDistance}
    newRadius= minRadius+nrCircles*equiDistance
  # pst      = appPIO (appWindowPicture wid
                      ( draw   {oval_rx=newRadius,oval_ry=newRadius}
                      o undraw {oval_rx=minRadius,oval_ry=minRadius}
                      )) pst
  = (lst,pst)
| otherwise
  # pst = appPIO (appWindowPicture wid (unfill viewDomain)) pst
  = (initTimerState,pst)
```

The last details that remain to be defined are the actual opening of the menu, window, and timer, the opening of the interactive process, and the creation of `wid`. For completeness we show the complete program code.

```
module circles

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program creates a window that displays growing concentric circles.
//  For this purpose it uses a timer.
//  ******************************************************************************

import StdEnv, StdIO

::  TimerState
    =   {   nrCircles       :: Int
        ,   equiDistance    :: Int
        ,   minRadius       :: Int
        }

Start :: *World -> *World
Start world
    = circles (openId world)

circles :: (Id,*World) -> *World
circles (wid,world)
    = startIO SDI
              Void
              (snd o seqList [openWindow Void           wdef
                             ,openMenu   Void           mdef
```

```
                              ,openTimer  initTimerState tdef
                              ]
               ) []
               world
where
    windowEdge       = 200
    viewDomain       = { corner1={x= ~windowEdge/2,y= ~windowEdge/2}
                       , corner2={x=  windowEdge/2,y=  windowEdge/2}
                       }
    wdef             = Window "Circles" NilLS
                         [   WindowId         wid
                         ,   WindowViewSize   (rectangleSize viewDomain)
                         ,   WindowViewDomain viewDomain
                         ]
    mdef             = Menu "&Circles"
                         (   MenuItem "&Quit" [MenuFunction (noLS closeProcess)
                                              ,MenuShortKey 'q'
                                              ]
                         )   []
    tdef             = Timer (ticksPerSecond/20) NilLS
                         [   TimerFunction   timer
                         ]
    initTimerState   = { nrCircles   = 4
                       , equiDistance= 2
                       , minRadius   = 0
                       }
    timer _ (lst=:{nrCircles,equiDistance,minRadius},pst)
        | minRadius<windowEdge/2
            # lst         = {lst & minRadius=minRadius+equiDistance}
              newRadius   = minRadius+nrCircles*equiDistance
            # pst         = appPIO (appWindowPicture wid
                                   ( draw   {oval_rx=newRadius,oval_ry=newRadius}
                                   o undraw {oval_rx=minRadius,oval_ry=minRadius}
                                   )) pst
            = (lst,pst)
        | otherwise
            # pst = appPIO (appWindowPicture wid (unfill viewDomain)) pst
            = (initTimerState,pst)
```

9.1.2 Internal clock

In this example we create a program that uses three timers to track the elapsed time since program startup. The timers track the elapsed seconds, minutes, and hours respectively. A dialogue is used to provide visual feedback. Figure 9.2 shows the application in action. We first have a look at the dialogue, and then the three timers.

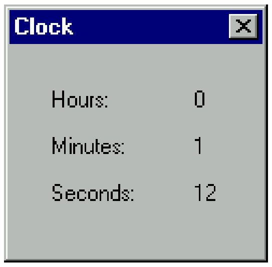


Figure 9.2: The timing program in action.

The dialogue `ddef` simply uses `TextControls` to display the hours, minutes, and seconds. These controls are identified by the `Id` values `hoursId`, `minutesId`, and secondsId respectively. LayoutControls are used to get the layout in the desired way. Observe the use of list comprehensions and the ListLS Controls class instance. In this example closing the dialogue also terminates the application. Here is the dialogue definition:

```
ddef
= Dialog "Clock"
  (   LayoutControl
  (   ListLS [TextControl text [ControlPos (Left,zero)]
             \\ text<-["Hours:","Minutes:","Seconds:"]
             ]
  )   []
  :+: LayoutControl
  (   ListLS [TextControl "0"  [ControlPos (Left,zero)
                               ,ControlId  id
                               ,ControlWidth (ContentWidth "00")
                               ]
             \\ id<-[hoursId,minutesId,secondsId]
             ]
  )   []
  )   [WindowClose (noLS closeProcess)]
```

Because the operation of each of the three timers is very similar, we use one function tdef to define them and parameterise it with their respective TimerIntervals, dt. The timers do not contain any timer elements. Each timer has a local integer state (initially zero) that keeps the current number of evaluated time units modulo their maximum number, maxunit. The timer function tick uses the NrOfIntervals parameter. Given the current number of evaluated time units in its local integer state, each timer function adds the NrOfIntervals value nrElapsed to it. Depending on dt, the proper modulo value is taken (using maxunit). This new value is the new local state and is also drawn in the dialogue by setting the text of the corresponding text control.

```
tdef dt
    = Timer dt NilLS [TimerFunction tick]
where
    tick nrElapsed (time,pst)
        # time = (time+nrElapsed) mod maxunit
        = (time,appPIO (setControlText id (toString time)) pst)
    (id,maxunit) = if (dt==second) (secondsId,60)
                  (if (dt==minute) (minutesId,60)
                                   (hoursId,  24))
```

The last details that remain to be defined are the actual opening of the three timers, the dialogue, the opening of the interactive process, and the creation of the proper Ids. For completeness we show the complete program code.

```
module clock

//  **********************************************************************************
//  Clean tutorial example program.
//
//  This program creates a window that tracks the elapsed time since startup.
//  For this purpose it uses three timers to track the seconds, minutes, and hours
//  separately.
// ****************************************************************************

import StdEnv,StdIO

::  DialogIds
    =   {   secondsId   :: Id
        ,   minutesId   :: Id
        ,   hoursId     :: Id
        }

second  :== ticksPerSecond
minute  :== 60*second
hour    :== 60*minute

openDialogIds :: *env -> (DialogIds,*env) | Ids env
openDialogIds env
    # ([secondsid,minutesid,hoursid:_],env) = openIds 3 env
    = ({ secondsId=secondsid, minutesId=minutesid, hoursId=hoursid },env)

Start :: *World -> *World
Start world
    # (dialogIds,world) = openDialogIds world
    = startIO NDI Void (initialise dialogIds) [] world

initialise :: DialogIds (PSt .l) -> (PSt .l)
initialise {secondsId,minutesId,hoursId} pst
    # (errors,pst)  = seqList [  openTimer 0 (tdef timerinfo)
                              \\ timerinfo<-[second,minute,hour]
                              ]  pst
    | any ((<>) NoError) errors
        = closeProcess pst
    # (error,pst)   = openDialog Void ddef pst
    | error<>NoError
        = closeProcess pst
    | otherwise
        = pst
where
    tdef dt
        = Timer dt NilLS [TimerFunction tick]
    where
        tick nrElapsed (time,pst)
            # time  = (time+nrElapsed) mod maxunit
            = (time,appPIO (setControlText id (toString time)) pst)

        (id,maxunit) = if (dt==second) (secondsId,60)
                      (if (dt==minute) (minutesId,60)
                                       (hoursId,  24))

    ddef= Dialog "Clock"
            (   LayoutControl
            (   ListLS  [  TextControl text [ControlPos (Left,zero)]
                        \\ text<-["Hours:","Minutes:","Seconds:"]
                        ]
            )   []
            :+: LayoutControl
            (   ListLS  [  TextControl "0"  [ControlPos (Left,zero)
                                            ,ControlId  id
                                            ,ControlWidth (ContentWidth "00")
                                            ]
                        \\ id<-[hoursId,minutesId,secondsId]
                        ]
            )   []
            )   [WindowClose (noLS closeProcess)]
```

Chapter 10

Receivers

All the interactive object I/O components discussed so far have in common that the events to which they respond are *abstract*. In this context abstract means that it is not specified in detail what *concrete* events cause a specific callback function to be evaluated. For instance, a callback function associated with a `MenuItem` (Section 8.1.4) is evaluated when it has been selected by the user. How this selection takes place is not specified.

In this section we discuss an interactive object I/O component that responds to program defined events, or rather *messages*. This component is the *receiver*. It plays an important role in the construction of interactive components. There are *no restrictions* on the kind of messages that can be *sent* or *received*, provided that they are type correct. The latter is obtained by using special identification values for receivers, the *receiver ids* (Chapter 4).

There are also receivers that are able to receive data from other programs via a network. These receivers are not topic of this chapter but of Chapter 14.

We will first have a look at the definition of receivers in Section 10.1. Receivers can be opened as top level object I/O components, but also as elements of windows, dialogues, menus, and timers. This is discussed in Section 10.2. Knowing how to define and open receivers, we show which functions are available to send messages in Section 10.3. Section 10.4 contains a number of examples to demonstrate the use of receivers.

10.1 Receiver definitions

There are two kinds of receivers. *Uni-directional* receivers respond only to messages. *Bi-directional* receivers respond to messages and also reply with a message. The types needed to define receivers can be found in the module `StdReceiverDef`, Appendix A.34.

A uni-directional receiver that responds to messages of type `msg` and that has a local state of type `ls` and process state of type `pst`) is defined by an expression of type (`Receiver msg ls pst`):

```
:: Receiver msg ls pst
 = Receiver (RId msg) (ReceiverFunction msg *(lst,pst))
                      [ReceiverAttribute    *(lst,pst)]
:: ReceiverFunction msg st
```

```
:== msg -> st -> st
```

A bi-directional receiver that responds to messages of type `msg` and returns a response message of type `resp` and that has a local state of type `ls` and process state `pst`) is defined by an expression of type (`Receiver2 msg resp ls pst`):

```
:: Receiver2 msg resp ls pst
 = Receiver2 (R2Id msg resp) (Receiver2Function msg resp *(ls,pst))
                             [ReceiverAttribute          *(ls,pst)]
:: Receiver2Function msg resp st
:== msg -> st -> (resp,st)
```

The set of receiver attributes is currently limited to the `ReceiverSelectState` which default value is `Able`.

The receiver attributes are defined as follows:

```
::  ReceiverAttribute          st
 =  ReceiverInit               (IdFun st)
 |  ReceiverSelectState        SelectState
 |  ReceiverConnectedReceivers [Id]
```

The `ReceiverInit` attribute defines an action that should be performed immediately after opening the receiver. This is equivalent to the window, menu, and timer initialisation actions (sections 6.2.1, 8.1.1, and 9). If no `ReceiverInit` attribute is provided, no additional action is performed. The `ReceiverSelectState` attribute indicates whether the receiver responds (`Able`) or ignores (`Unable`) messages. Its default value is `Able`. The `ReceiverConnectedReceivers` attribute is discussed in Section 14.

10.2 Receiver creation

Receivers can be opened as top level interface elements. This is done in the usual, overloaded way. Module `StdReceiver`, Appendix A.32) declares the type constructor class `Receivers` with the top level receiver creation member function `open-Receiver`. Uni-directional and bi-directional receivers are proper instances of this class.

```
class Receivers rdef where
  openReceiver :: .ls !*(*rdef .ls (PSt .l)) !(PSt .l)
                             -> (!ErrorReport,!PSt .l)
  ...

instance Receivers (Receiver  msg)
instance Receivers (Receiver2 msg resp)
```

Receivers can also be opened as elements of windows, dialogues, menus, and timers. So, for instance in a window one can not only add the set of controls as discussed in Chapter 7 but also receivers. This is accomplished by declaring receivers to be instances of the `Controls` type constructor class in module `StdControlReceiver` (Appendix A.8). Receivers are declared to be instances of *menu elements* in module `StdMenuReceiver` (Appendix A.22), and *timer elements* in module `StdTimer-Receiver` (Appendix A.46).

10.3 Message passing

In contrast with the previously discussed object I/O components, receivers *must* have an identification value. The *message passing* functions require this identification value to ensure that a message of the correct type is sent. The message passing functions can be found in module `StdReceiver`, Appendix A.32.

All message passing functions return a report about the message passing action. This report is an algebraic type `SendReport`, which has the following alternatives:

```
::  SendReport
 =  SendOk
 |  SendUnknownReceiver
 |  SendUnableReceiver
 |  SendDeadlock
 |  OtherSendReport !String
```

For all functions, the alternative value `SendOk` is returned in case message passing was successful. The alternative `SendUnknownReceiver` is returned in case the indicated receiver is not open at the moment of sending the message. The other `SendReport` alternatives are discussed below.

We start with message passing to uni-directional receivers in Section 10.3.1. Bi-directional message passing is discussed in Section 10.3.2.

10.3.1 Uni-directional message passing

There are two functions a programmer can use to send a message to a uni-directional receiver: `asyncSend` and `syncSend` which have the same function types.

```
asyncSend :: !(RId msg) msg !(PSt .l) -> (!SendReport,!PSt .l)
syncSend  :: !(RId msg) msg !(PSt .l) -> (!SendReport,!PSt .l)
```

Using `asyncSend`, a message is placed at the end of the asynchronous message queue of the indicated receiver and will, at some point, be handled by that receiver. It is unspecified *when* that event occurs. Asynchronous messages that are sent in sequence will be evaluated in that sequence. Asynchronous message passing only fails in case the indicated receiver is not open, as discussed above. You should observe that successfully queueing an asynchronous message does not guarantee that the message will be handled. The receiver might be disabled or closed before all of its messages have been handled.

If you want to enforce a receiver to handle a message, you should use `syncSend`. This function does not place the message in the message queue of the indicated receiver, but instead applies the receiver function of that receiver to the message. This implies that `syncSend` has to *switch context* because the indicated receiver is part of another object I/O component. Another consequence is that synchronous messages may overtake asynchronous messages. Synchronous messages that are sent in sequence will be evaluated in that order.

Sending a synchronous message fails in case the indicated receiver does not exist. If the indicated receiver does exist, but its `ReceiverSelectState` attribute is `Unable`, then the message is also not handled. In this case, the `SendReport` alternative `SendUnableReceiver` is returned. Finally, even if the indicated receiver exists and is `Able` communication can still fail. This is possible when the indicated receiver itself is involved in a synchronous message passing transaction and waiting for termination. If this transaction involves the original receiver, then a deadlock situation is detected. In that case, `syncSend` also fails and returns with `SendDeadlock`.

Examples of uni-directional message passing are given in Section 10.4. The first example, in Section 10.4.1, demonstrates the use of asynchronous message passing, while the second example, in Section 10.4.2, demonstrates synchronous message passing.

10.3.2 Bi-directional message passing

Bi-directional message passing is *synchronous*. A message is sent using the function `syncSend2`:

```
syncSend2 :: !(R2Id msg resp) msg  !(PSt .l)
    -> (!(!SendReport,!Maybe resp), !PSt .l)
```

Analogous to `syncSend`, `syncSend2` locates the indicated bi-directional receiver and applies the argument message immediately to the corresponding receiver function. If this receiver could be found and happens to be `Able`, evaluation of the receiver function will yield a response message `resp`. In this case the `SendReport` result of `syncSend2` is `SendOk`, and the response value is returned as `(Just resp)`. In all exceptional cases, there is no response value and `Nothing` is returned. To evaluate the receiver function, `syncSend2` has to switch context as well.

Bi-directional receivers can be used to retrieve local encapsulated data. Example 10.4.3 demonstrates this.

10.4 Examples

In this section we give some examples to illustrate the use of receivers.

10.4.1 Talk windows

In this example we create a program that uses receivers to send keyboard input from one window to another window, and vice versa. This results in a talk like application (although it is not very useful as a talk application because it runs on one computer). Figure 10.1 shows the application in action.

The initialisation action of the talk program, defined by the function `initialise`, first creates a menu that has only one quit menu item. This command ensures that we can always terminate the application. Then two `RId` values are generated that will be used to identify the two receivers. The function `openTalkWindow` is then applied twice to create the two windows. Its parameters are the name of the window and the two receiver ids. The first receiver id parameter identifies the private receiver, while the second identifies the other receiver.

```
initialise :: (PSt .l) -> PSt .l
initialise pst
  # menu       = Menu "&Talk"
                 (MenuItem "&Quit" [MenuShortKey 'q'
                                   ,MenuFunction (noLS closeProcess)
                                   ]
```

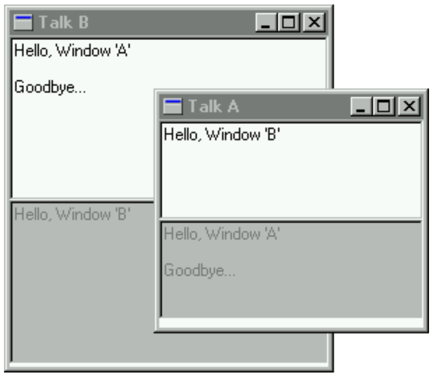


Figure 10.1: The talk program in action.

```
                 ) []
  # (error,pst)= openMenu undef menu pst
  | error<>NoError
               = abort "talk could not open menu."
  | otherwise
     # (a,pst) = accPIO openRId pst
     # (b,pst) = accPIO openRId pst
     # pst     = openTalkWindow "A" a b pst
     # pst     = openTalkWindow "B" b a pst
     = pst
```

openTalkWindow creates a window in which the user can type text, using an edit control, and see the messages of the other talk window, also by means of an edit control. The messages of the other talk window are of course handled by a receiver.

The first edit control is identified by an Id value inId. The user keyboard input is handled by function input. This function is parameterised with the Id of the edit control and the RId of the receiver of the other window. Because we only want to send new input to the other window receiver, the keyboard filter function input-filter filters out key up events. The edit control shares the window view frame real estate with the other edit control. When the window is being resized, it adopts its width to the new width of the window and consumes half the height. This is expressed by the function resizeHalfHeight _ _ {w,h} = {w=w,h=h/2}.

```
input = EditControl "" (PixelWidth (hmm 50.0)) 5
         [ ControlId       inId
         , ControlKeyboard inputfilter Able (noLS1 (input inId you))
         , ControlResize   resizeHalfHeight
         ]
inputfilter :: KeyboardState -> Bool
inputfilter keystate = getKeyboardStateKeyState keystate<>KeyUp
```

The keyboard input function input first gets the current state of its parent window, using the StdControl library function getParentWindow. From this value the current content of its edit control can be retrieved, using the function getControl-Text. This new content, which is a String, is being sent asynchronously to the other receiver.

```
input :: Id (RId String) KeyboardState (PSt .l) -> PSt .l
input inId you _ pst
    # (Just wst,pst) = accPIO (getParentWindow inId) pst
    # text           = fromJust (snd (getControlText inId wst))
    = snd (asyncSend you text pst)
```

The second edit control is used to display the received text from the other window receiver. Because it is used only for output its `SelectState` is `Unable`. To be able to change its text content it is identified by the `Id` value `outId`. Its position is below the first edit control. Its resize behaviour is identical to that of the first edit control. Here is its definition:

```
output = EditControl "" (PixelWidth (hmm 50.0)) 5
           [ ControlId          outId
           , ControlPos         (BelowPrev,zero)
           , ControlSelectState Unable
           , ControlResize      resizeHalfHeight
           ]
```

The final component of the window is the receiver that obtains the new string content of the other window input edit control. Its receiver function `receive` is parameterised with the `Id` of the output edit control. For every string message received from the other talk window, it replaces the current content of the output edit control with the received text. This is done using the function `setControl-Text`. The function `setEditControlCursor` makes sure that the end of the text is visible.

```
receiver = Receiver me (noLS1 (receive outId)) []

receive :: Id String (PSt .l) -> PSt .l
receive outId text pst=:{io}
    # io = setControlText outId text io
    # io = setEditControlCursor outId (size text) io
    = {pst & io=io}
```

These are the three components of the window. The only task of `openTalkWindow` is to create the two `Id` values that identify the input and output edit controls, and open the window containing the two edit controls and the receiver control. We also want the initial window view frame size to be equal to the width and added height of the two edit controls. Because the height of the edit controls is given indirectly (by specifying that it should be as high as five text lines) we have to ask the system to calculate the size of the edit controls. This is done by the `StdControl` function `controlSize`. Its first argument is the `Controls` instance which size has to be calculated. The second argument, a boolean, must be true if the controls are supposed to be opened in a window and false if they are to be opened in a dialogue. This value is required because of the different platform margins and item spaces. If you want to provide your own horizontal and vertical margins and item spaces then this can be done in the following three arguments. Choosing `Nothing` results in the default values. The result size of `controlSize` can now be used correctly as the initial view frame size of the window. Here is the definition of `openTalkWindow`.

```
openTalkWindow:: String (RId String) (RId String) (PSt .l) -> PSt .l
openTalkWindow name me you pst
  # (inId, pst)= accPIO openId pst
  # (outId,pst)= accPIO openId pst
  # input      = EditControl "" (PixelWidth (hmm 50.0)) 5
                  [ ControlId       inId
                  , ControlKeyboard inputfilter Able
                                    (noLS1 (input inId you))
                  , ControlResize   resizeHalfHeight
                  ]
  # output     = EditControl "" (PixelWidth (hmm 50.0)) 5
                  [ ControlId          outId
                  , ControlPos         (BelowPrev,zero)
                  , ControlSelectState Unable
                  , ControlResize      resizeHalfHeight
                  ]
  # (size,pst) = controlSize (input:+:output)
                  True Nothing Nothing Nothing pst
  # receiver   = Receiver me (noLS1 (receive outId)) []
  # wdef       = Window ("Talk "+++name) (input:+:output:+:receiver)
                  [ WindowViewSize size ]
  # (error,pst)= openWindow undef wdef pst
  | error<>NoError= abort "talk could not open window."
  | otherwise     = pst
```

For completeness the whole program is shown here.

```
module talk

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program creates two windows that communicate with each other using message
//  passing. Text that has been typed in one window is being sent to the other, and
//  vice versa.
//  ****************************************************************************

import  StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO MDI Void initialise [] world
where
    initialise :: (PSt .l) -> PSt .l
    initialise pst
        # menu          = Menu "&Talk"
                            (   MenuItem "&Quit" [  MenuShortKey 'q'
                                                 ,  MenuFunction (noLS closeProcess)
                                                 ]
                            )   []
        # (error,pst)   = openMenu undef menu pst
        | error<>NoError
            = abort "talk could not open menu."
        | otherwise
            # (a,pst)   = accPIO openRId pst
            # (b,pst)   = accPIO openRId pst
            # pst       = openTalkWindow "A" a b pst
            # pst       = openTalkWindow "B" b a pst
            = pst

openTalkWindow :: String (RId String) (RId String) (PSt .l) -> PSt .l
openTalkWindow name me you pst
    # (inId, pst)   = accPIO openId pst
    # (outId,pst)   = accPIO openId pst
    # input         = EditControl "" (PixelWidth (hmm 50.0)) 5
```

```
                        [   ControlId           inId
                        ,   ControlKeyboard     inputfilter Able
                                                (noLS1 (input inId you))
                        ,   ControlResize       resizeHalfHeight
                        ]
    # output        = EditControl "" (PixelWidth (hmm 50.0)) 5
                        [   ControlId           outId
                        ,   ControlPos          (BelowPrev,zero)
                        ,   ControlSelectState  Unable
                        ,   ControlResize       resizeHalfHeight
                        ]
    # (size,pst)    = controlSize (input:+:output) True Nothing Nothing Nothing pst
    # receiver      = Receiver me (noLS1 (receive outId)) []
    # wdef          = Window ("Talk "+++name) (input:+:output:+:receiver)
                        [   WindowViewSize  size
                        ]
    # (error,pst)   = openWindow undef wdef pst
    | error<>NoError= abort "talk could not open window."
    | otherwise     = pst
where
    inputfilter :: KeyboardState -> Bool
    inputfilter keystate
        = getKeyboardStateKeyState keystate<>KeyUp

    input :: Id (RId String) KeyboardState (PSt .l) -> PSt .l
    input inId you _ pst
        # (Just wst,pst)    = accPIO (getParentWindow inId) pst
        # text              = fromJust (snd (getControlText inId wst))
        = snd (asyncSend you text pst)

    receive :: Id String (PSt .l) -> PSt .l
    receive outId text pst=:{io}
        # io    = setControlText outId text io
        # io    = setEditControlCursor outId (size text) io
        = {pst & io=io}

    resizeHalfHeight :: Size Size Size -> Size
    resizeHalfHeight _ _ {w,h} = {w=w,h=h/2}
```

10.4.2 Resetting the counter

In this example we extend the example counter in Section 7.2.3 (page 89) with a means to reset the counter to zero. We proceed in a bottom-up style: the counter control is a layout control extended with a receiver that, when it receives a message, will reset the counter to zero. The counter control encapsulates its local counter state. Out of the local state scope of the counter control we add button control that, when selected, sends a message to the receiver component of the counter control. The whole is being placed in a dialogue. Figure 10.2 gives a snapshot of the program.

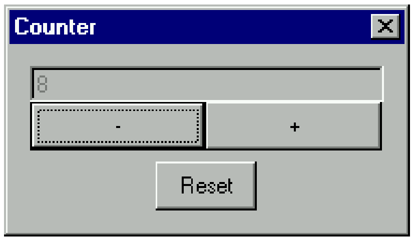


Figure 10.2: The counter control with reset button.

The main component is of course the counter control, defined by `counter`. It encapsulates an integer local state with initial value `initcount`. We have changed its definition from the original one shown on page 89 to give another example of using control layout. It is also different in that it contains a receiver.

```
counter
  = { newLS  = initcount
    , newDef = LayoutControl
               (   EditControl (toString initcount)
                             (PixelWidth (hmm 50.0)) 1
                             [ControlSelectState Unable
                             ,ControlId          displayid
                             ]
               :+: ButtonControl "-"
                             [ControlFunction (count (-1))
                             ,ControlWidth    (PixelWidth (hmm 25.0))
                             ,ControlPos      (BelowPrev,zero)
                             ]
               :+: ButtonControl "+"
                             [ControlFunction (count   1 )
                             ,ControlWidth    (PixelWidth (hmm 25.0))
                             ]
               :+: Receiver resetid reset []
               )   [ControlPos       (Center,zero)
                   ,ControlHMargin   0 0
                   ,ControlVMargin   0 0
                   ,ControlItemSpace 0 0
                   ]
    }
```

The only purpose of the receiver is to reset the current local counter value to `initcount` and show this by changing the text of the edit control. Note that the receiver function `reset` is not interested at all in the message.

```
reset :: m (Int,PSt .l) -> (Int,PSt .l)
reset _ (_,pst)
  = ( initcount
    , appPIO (setControlText displayid (toString initcount)) pst
    )
```

The reset button is a straightforward `ButtonControl`. It is centered below the counter control. Its `ControlFunction` argument only has to send a message synchronously to the receiver component of the counter control. Because this component does not care about the message, the button function can be as bold to send the `StdMisc` library function `undef`. This function, when evaluated, aborts the application. This demonstrates that message passing is truely lazy in the message argument.

```
resetbutton
  = ButtonControl "Reset"
      [ControlFunction (noLS (snd o syncSend resetid undef))
      ,ControlPos      (Center,zero)
      ]
```

The final details of the program are to generate the proper identification values and to create the initial process and dialogue. For completeness, the program code is given here.

```
module counterreset

//  **************************************************************************
//  Clean tutorial example program.
//
//  This program defines a Controls component that implements a manually settable
//  counter. A receiver is used to add a reset option.
//  **************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO NDI Void initialise [] world
where
    initialise pst
        # (displayid,pst)   = accPIO openId  pst
        # (resetid,  pst)   = accPIO openRId pst
        # (error,    pst)   = openDialog Void (dialog displayid resetid) pst
        | error<>NoError
           = abort "counter could not open Dialog."
        | otherwise
           = pst

dialog displayid resetid
    = Dialog "Counter" (counter :+: resetbutton) [WindowClose (noLS closeProcess)]
where
    counter
        = { newLS  = initcount
          , newDef = LayoutControl
                     (   EditControl (toString initcount) (PixelWidth (hmm 50.0)) 1
                                       [ControlSelectState Unable
                                       ,ControlId          displayid
                                       ]
                     :+: ButtonControl "-" [ControlFunction  (count (-1))
                                       ,ControlWidth     (PixelWidth (hmm 25.0))
                                       ,ControlPos       (BelowPrev,zero)
                                       ]
                     :+: ButtonControl "+" [ControlFunction  (count   1 )
                                       ,ControlWidth     (PixelWidth (hmm 25.0))
                                       ]
                     :+: Receiver resetid reset []
                     )   [ControlPos       (Center,zero)
                         ,ControlHMargin   0 0
                         ,ControlVMargin   0 0
                         ,ControlItemSpace 0 0
                         ]
          }
    where
        initcount  = 0

        count :: Int (Int,PSt .l) -> (Int,PSt .l)
        count dx (count,pst)
            = (count+dx,appPIO (setControlText displayid (toString (count+dx))) pst)

        reset :: m (Int,PSt .l) -> (Int,PSt .l)
        reset _ (_,pst)
            = (initcount,appPIO (setControlText displayid (toString initcount)) pst)

    resetbutton
        = ButtonControl "Reset"
            [ControlFunction (noLS (snd o syncSend resetid undef))
            ,ControlPos      (Center,zero)
          ]
```

10.4.3 Reading the counter

In this example we extend the counter example once more and add a dialogue that reads the local counter value. To be able to do this a bi-directional receiver is added to the counter. Figure 10.3 gives a snapshot of the program.

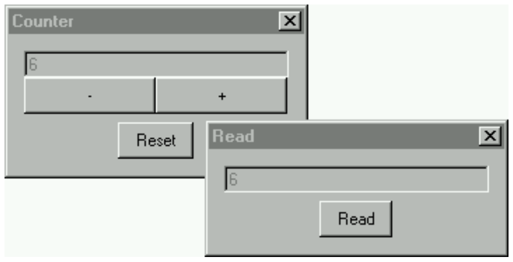


Figure 10.3: Reading the counter control with reset button.

The first change is to extend the counter component with a bi-directional receiver component defined by the expression `(Receiver2 readid read [])`, where `readid` is a `R2Id` identification value. The receiver function `read` is very straightforward: it returns the current local counter value without changing anything (also in this case `read` is not interested in the input message type):

```
read :: m (Int,PSt .l) -> (Int,(Int,PSt .l))
read _ (count,pst) = (count,(count,pst))
```

From the type of this function we can derive that the type of the `R2Id` value of its associated receiver must be `(R2Id m Int)`.

The initialisation action opens another dialogue defined by `display` that obtains and shows the counter value on request. This request can be done by a user by pressing the button control labeled `"Read"`. Again, an `Unable` edit control is used to display the read value.

```
display displayid readid
  = Dialog "Read"
    (   EditControl   "" (PixelWidth (hmm 50.0)) 1
                         [ControlSelectState Unable
                         ,ControlId          displayid
                         ]
    :+: ButtonControl "Read" [ControlFunction    (noLS read)
                         ,ControlPos         (Center,zero)
                         ]
    )   [ WindowClose (noLS closeProcess) ]
```

When the `"Read"` button has been selected, its callback function `read` is evaluated. It sends a message to the bi-directional receiver component of the counter control using `syncSend2`. If this action fails it aborts the program. Although this situation will never occur, this check has been added for program hygiene. In a successfull communication, `value` will be `(Just count)`, and it is this value that is then displayed in the display edit control of the dialogue.

```
read pst
    # ((error,value),pst)= syncSend2 readid undef pst
    | error<>SendOk
        = abort "could not read counter value"
    | otherwise
        = appPIO (setControlText displayid
                    (toString (fromJust value))
                  ) pst
```

The initialisation action creates the necessary identification values and the two dialogues. Here is the complete program code.

```
module counterread

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program defines a Controls component that implements a manually settable
//  counter.
//  A bi-directional receiver is added to give external access to the counter value.
//  ******************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO NDI Void initialise [] world
where
    initialise pst
        # (displayid,pst) = accPIO openId   pst
        # (resetid,  pst) = accPIO openRId  pst
        # (readid,   pst) = accPIO openR2Id pst
        # (error,    pst) = openDialog Void (dialog displayid resetid readid) pst
        | error<>NoError
            = abort "counter could not open counter dialog."
        # (displayid,pst) = accPIO openId   pst
        # (error,    pst) = openDialog Void (display displayid readid) pst
        | error<>NoError
            = abort "counter could not open display dialog."
        | otherwise
            = pst

dialog displayid resetid readid
    = Dialog "Counter" (counter:+:resetbutton) [WindowClose (noLS closeProcess)]
where
    counter
        = { newLS  = initcount
          , newDef = LayoutControl
                     (   EditControl (toString initcount) (PixelWidth (hmm 50.0)) 1
                                     [ControlSelectState Unable
                                     ,ControlId       displayid
                                     ]
                     :+: ButtonControl "-" [ControlFunction (count (-1))
                                     ,ControlWidth    (PixelWidth (hmm 25.0))
                                     ,ControlPos      (BelowPrev,zero)
                                     ]
                     :+: ButtonControl "+" [ControlFunction (count   1 )
                                     ,ControlWidth    (PixelWidth (hmm 25.0))
                                     ]
                     :+: Receiver  resetid reset []
                     :+: Receiver2 readid  read  []
                     )   [ControlPos       (Center,zero)
                         ,ControlHMargin   0 0
                         ,ControlVMargin   0 0
                         ,ControlItemSpace 0 0
```

```
                       ]
          }
    where
        initcount = 0

        count :: Int (Int,PSt .l) -> (Int,PSt .l)
        count dx (count,pst)
            = (count+dx,appPIO (setControlText displayid (toString (count+dx))) pst)

        reset :: m (Int,PSt .l) -> (Int,PSt .l)
        reset _ (_,pst)
            = (initcount,appPIO (setControlText displayid (toString initcount)) pst)

        read :: m (Int,PSt .l) -> (Int,(Int,PSt .l))
        read _ (count,pst) = (count,(count,pst))

    resetbutton
        = ButtonControl "Reset"
            [ControlFunction (noLS (snd o syncSend resetid undef))
            ,ControlPos      (Center,zero)
            ]

display displayid readid
    = Dialog "Read"
        (   EditControl   "" (PixelWidth (hmm 50.0)) 1
                             [ControlSelectState Unable
                             ,ControlId          displayid
                             ]
        :+: ButtonControl "Read" [ControlFunction    (noLS read)
                             ,ControlPos         (Center,zero)
                             ]
        )   [ WindowClose (noLS closeProcess) ]
where
    read pst
        # ((error,value),pst)   = syncSend2 readid undef pst
        | error<>SendOk
            = abort "could not read counter value"
        | otherwise
            = appPIO (setControlText displayid (toString (fromJust value))) pst
```

Chapter 11

Interactive processes

The examples discussed so far used the `StdProcess` library function `startIO`. This function creates one interactive process that engages in some graphical user interface actions with a user and then terminates, using the `StdProcess` function `closeProcess`. In this chapter we show how an interactive process can spawn new interactive processes that will run *interleaved*. It is also possible to create a number of processes at once.

We start the discussion by looking at the ways to define interactive processes in Section 11.1. This is followed by the functions to open interactive processes. Finally we give some examples to illustrate their application.

11.1 Defining interactive processes

All interactive processes are defined as an instance of the following type definition (module `StdProcessDef`, see Appendix A.29):

```
::  Process
 =  E. .l: Process
                DocumentInterface
                l
                (ProcessInit      (PSt l))
                [ProcessAttribute (PSt l)]
::  DocumentInterface
 =  NDI | SDI | MDI
```

The `DocumentInterface` argument of an interactive process specifies its kind of *document interface.* The document interface concept is based on the idea that an interactive process provides the user with an interface to manipulate documents (even if they don't look like documents at all). Depending on the number of documents a process offers (zero, one, or many) the library provides the process with a platform dependent infrastructure. Three kinds of document interfaces are specified:

No Document Interface (NDI)
An interactive process with this document interface does not present a document to a user. It has no menus and no windows. It is typically used for 'background' interactive processes or very simple programs that have only a

dialogue. The interactive process is allowed to open dialogues, timers, and receivers.

Single Document Interface (SDI)
An interactive process with this document interface presents exactly one document at a time to the user. Usually all interactive components are presented within one window frame. The interactive process is allowed to open dialogues, timers, and receivers. At any time, at most one window can be open. It is allowed to close a window and then open a new one.

Multiple Document Interface (MDI)
An interactive process with this document interface can present an arbitrary number of documents to the user.

If some callback function or initialisation function attempts to open an interactive object that violates its document interface, then the `ErrorReport` result of that particular object opening function is `ErrorViolateDI`. When defining an interactive process, you should choose the most 'minimal' document interface that fits that process.

The interactive process type constructor introduces an initial local process state of some type `l` and encapsulates it using existential quantification. When the interactive process is created, the object I/O system creates the `PSt` record with for its `ls` field the value of this argument. Because of the limited sizes of the example programs so far, the value we have used so far was `Void`. In addition to the local process state value, the object I/O system creates an initial, empty I/O state of type `IOSt`.

Given the initial process state record, the object I/O system lets the interactive process initialise itself by applying the third argument of the `Process` type constructor which has type `(ProcessInit (PSt l))` .

```
::  ProcessInit pst :== IdFun pst
::  IdFun       st  :== st -> st
```

When this function has been evaluated, and the interactive process has not been terminated by the `closeProcess` function, the interactive process is in its *running phase*. It should be noted that it is unspecified *when* the initialisation of the process takes place: it is guaranteed only that this function is the first action on the process state.

The final argument of the `Process` type constructor are its attributes. These are the following:

```
::  ProcessAttribute st
 =  ProcessActivate     (IdFun st)
 |  ProcessDeactivate   (IdFun st)
 |  ProcessClose        (IdFun st)
 |  ProcessOpenFiles    (ProcessOpenFilesFunction    st)
 |  ProcessWindowPos    ItemPos
 |  ProcessWindowSize   Size
 |  ProcessWindowResize (ProcessWindowResizeFunction st)
 |  ProcessToolbar      [ToolbarItem st]
 |  ProcessNoWindowMenu
```

The meaning of these attributes are:

`ProcessActivate` **and** `ProcessDeactivate`
: These two attributes correspond closely to the `WindowActivate` and `Window-Deactivate` attributes. Recall that keyboard and mouse input is always directed to the active window. The parent interactive process that contains this window is the *active process*. If the input focus is moved to a window that is not owned by the interactive process then the `ProcessDeactivate` attribute function is evaluated to inform the program that the interactive process has become inactive. If an inactive process obtains the input focus, its `Process-Activate` attribute function is evaluated to inform the program that it has become active again.

`ProcessClose`
: This attribute corresponds closely to the `WindowClose` attribute. If for some reason the interactive process is requested to be closed, its `ProcessClose` attribute function is evaluated. It can take the opportunity to save data to disk and to ask the user if the process can be closed safely. It is however the responsibility of the program to terminate the process. It is good programming practice to always include this attribute because the user of your program expects to be able to close your application in this way.

`ProcessOpenFiles`
: Associating this attribute to an interactive process means that the interactive process can respond to user requests to open a number of files. The callback function has the somewhat verbose type `ProcessOpenFilesFunction`:

```
:: ProcessOpenFilesFunction st :== [String] -> st -> st
```

The `[String]` argument contains the full pathnames of the files to be opened. It is good programming practice to include this attribute if your application allows users to open files because it will let your application behave in the platform conform way.

`ProcessWindowPos`, `ProcessWindowSize`, **and** `ProcessWindowResize`
: *In the current implementation these attributes have no effect.* When they will, it will be the following: the *process window* is the root window in which all top-level user interface elements are created of an interactive process are located. On a Macintosh the process window is simply the screen. On the Windows platform it is also the screen in case of NDI processes. For SDI and MDI processes it is a window frame that can be resized by the user. For the latter case, the `-Pos` and `-Size` attribute specify the initial position and size in terms of the screen. In addition, if the user resizes the process window (by actually resizing the window or by changing the screen resolution), and a `ProcessWindowResize` attribute has been specified, then a function of type `ProcessWindowResizeFunction` is evaluated:

```
:: ProcessWindowResizeFunction st
:== Size -> Size -> st -> st
```

The first `Size` argument is the size of the process window before the resize occurred, and the second argument is the size afterwards.

`ProcessToolbar`
: The *process toolbar* is a combination of menus and controls. The purpose of the toolbar is to present to the user of your application a list of the most frequently used commands, also called *tools* in this context. These commands are

supposed to be visualised via easily recognisable icons, currently represented by means of bitmaps. An optional string argument provides a brief textual description of the tool, in a similar way as the `ControlTip` attributes do (discussed in Section 7.1). It is also possible to include a separation between groups of related tools by using the `ToolbarSeparator` alternative.

```
::  ToolbarItem st
 =  ToolbarItem Bitmap (Maybe String) (IdFun st)
 |  ToolbarSeparator
```

The current implementation does not support enabling and disabling of tools, nor does it allow the callback function to receive modifier keys.

`ProcessNoWindowMenu`
This attribute is valid only for MDI processes. In the default case every MDI process has a special *Windows* menu (discussed in Section 8.3). If this attribute is set, then this menu is not added to the menu system of the interactive process.

11.2 Interactive process creation

The basic function to create individual processes from a `World` environment is `startIO`, which we have encountered many times in the preceding chapters. For completeness we repeat its type definition:

```
startIO :: !DocumentInterface
           !.l
           !(ProcessInit      (PSt .l))
           ![ProcessAttribute (PSt .l)]
           !*World
        -> *World
```

Its arguments are identical to the arguments of the `Process` type constructor given in Section 11.1. The function `startIO` creates an interactive process with the same arguments and terminates only when this process and its child processes have terminated. It is a specialised function which uses the more general overloaded function `startProcesses` of the type constructor class `Processes`.

```
class Processes pdef where
    startProcesses :: !pdef !*World   -> *World
    openProcesses  :: !pdef !(PSt .l) -> PSt .l
```

The type constructor `Process` is of course an instance of this class. The actual implementation of `startIO` is nothing but a shorthand for this instance:

```
startIO :: !DocumentInterface !.l !(ProcessInit (PSt .l))
                              ![ProcessAttribute (PSt .l)]
           !*World -> *World
startIO documentInterface local init atts world
    = startProcesses
        (Process documentInterface
                 local
                 init
                 (if (documentInterface==MDI)
                     [ProcessNoWindowMenu:atts]
                     atts
                 )
        ) world
```

The other Processes member function openProcesses also creates an interactive process, but now in the context of an interactive process itself. Interactive processes can not use startIO because the World environment is not available from the PSt environment. When applied to an interactive process definition, openProcesses adds an initial version of that interactive process to the process state administration. At some point in time that interactive process will be initialised and joins the game. So process creation is asynchronous.

In addition to the Process type constructor instance, you can also create a list of interactive processes using the above Processes class member functions. This is stated concisely with the following type class instance declaration in module StdProcess:

```
instance Processes [pdef] | Processes pdef
```

So, if $p_1 \ldots p_n$ are Processes instances, then the expression (openProcesses $[p_1..p_n]$) is a process state transition function that creates these interactive process instances.

There is no special parent-child relationship between an interactive process and the interactive processes that it creates. For instance, termination of one interactive process (using closeProcess) has no consequence for the other processes. Indeed, the only entity that can terminate an interactive process is the interactive process itself (or, using brute force from the Operating System). This is one of the reasons why it is good programming practice to always add the ProcessClose attribute to every of your process definitions to allow graceful termination of your program.

11.3 Examples

In this section we give some examples of the use of interactive processes.

11.3.1 Talk revisited

In this example we have a new look at the talk example of Section 10.4.1. In that version, the program created one interactive process, using startIO, which opened the two talk windows. In the new version instead of a talk window we create a SDI process. Receivers are still used to send the user typed messages to each of the talk windows. The menu is now created for both processes. Below we discuss the differences. Figure 11.1 shows the program in action.

The initialisation of the new talk program is of course different. Instead of the function openTalkWindow that opened a talk window we now define a function talk that defines a talk process which contains the talk window, menu, and receiver. This implies that the RId values have to be created beforehand. We obtain the following Start rule:

```
Start :: *World -> *World
```

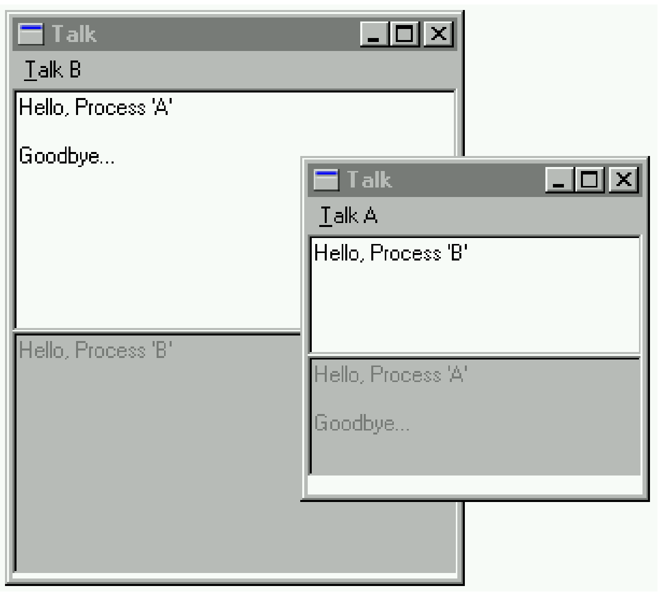


Figure 11.1: The talk program in action.

```
Start world
    # (a,    world) = openRId world
    # (b,    world) = openRId world
    # (talkA,world) = talk "A" a b world
    # (talkB,world) = talk "B" b a world
    = startProcesses [talkA,talkB] world
```

The menu definition is now moved inside the talk process. It is almost identical to the old version. The only difference is termination of the application. In the old version termination was no issue because there was only one interactive process with two talk windows. In the new version, closing one talk process won't close the other. Instead, before closing its parent process, the `quit` function sends a new message alternative `Quit` to the other process to request termination. So the message type is changed into:

```
::  Message = NewLine String | Quit
```

and, accordingly, the menu function `quit` into:

```
quit :: (RId Message) (PSt .l) -> PSt .l
quit you pst = closeProcess (snd (syncSend you Quit pst))
```

Of course the definition of the receiver function `receive` now handles two alternatives. When receiving `(NewLine text)`, it proceeds as before, and when it receives `Quit` it only needs to terminate its parent process, knowing that the requesting process will terminate itself.

```
receive :: Id Message (PSt .l) -> PSt .l
receive outId (NewLine text) pst=:{io}
    = {pst & io=setEditControlCursor outId (size text) (
                  setControlText       outId text io)
      }
receive _ Quit pst
    = closeProcess pst
```

These are the major differences. Here is the complete program.

```
module talk

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program creates two interactive processes that communicate via message
//  passing.
//  In a future distributed version this program can be used as a graphical talk
//  application.
//
//  ****************************************************************************

import  StdEnv, StdIO

::  Message             // The message type:
    =   NewLine String  //  transmit a line of text
    |   Quit            //  request termination

Start :: *World -> *World
Start world
    # (a,    world) = openRId world
    # (b,    world) = openRId world
    # (talkA,world) = talk "A" a b world
    # (talkB,world) = talk "B" b a world
    = startProcesses [talkA,talkB] world

talk :: String (RId Message) (RId Message) *env -> (Process,*env) | Ids env
talk name me you env
    # (outId,env)   = openId env
    # (inId, env)   = openId env
    # input         = EditControl   "" (PixelWidth (hmm 50.0)) 5
                        [   ControlId           inId
                        ,   ControlKeyboard     inputfilter Able
                                                (noLS1 (input inId you))
                        ,   ControlResize       resizeHalfHeight
                        ]
    # output        = EditControl   "" (PixelWidth (hmm 50.0)) 5
                        [   ControlId           outId
                        ,   ControlPos          (BelowPrev,zero)
                        ,   ControlSelectState  Unable
                        ,   ControlResize       resizeHalfHeight
                        ]
    # receiver      = Receiver me (noLS1 (receive outId)) []
    # talkwindow    = Window ("Talk "+++name) (input:+:output:+:receiver)
                        [   WindowViewSize  {w=hmm 50.0,h=120}
                        ]
    # menu          = Menu ("&Talk "+++name)
                        (   MenuItem "&Quit"
                            [   MenuShortKey 'q',MenuFunction (noLS (quit you))]
                        )   []
    = ( Process SDI
                Void
                (snd o seqList [openWindow Void talkwindow,openMenu Void menu])
                [ProcessClose (quit you)]
      , env
      )
where
    inputfilter :: KeyboardState -> Bool
    inputfilter keystate
    = getKeyboardStateKeyState keystate<>KeyUp

input :: Id (RId Message) KeyboardState (PSt .l) -> PSt .l
input inId you _ pst
    # (Just window,pst) = accPIO (getParentWindow inId) pst
    # text              = fromJust (snd (getControlText inId window))
    = snd (asyncSend you (NewLine text) pst)

receive :: Id Message (PSt .l) -> PSt .l
receive outId (NewLine text) pst=:{io}
    = {pst & io=setEditControlCursor outId (size text) (
                setControlText       outId text io)
      }
receive _ Quit pst
    = closeProcess pst

quit :: (RId Message) (PSt .l) -> PSt .l
quit you pst
    = closeProcess (snd (syncSend you Quit pst))

resizeHalfHeight :: Size Size Size -> Size
resizeHalfHeight _ _ {w,h} = {w=w,h=h/2}
```

11.3.2 Clock revisited

In this example we are going to turn the clock example of Section 9.1.2 into a stopwatch component that can be added in an arbitrary interactive process. The stopwatch commands will be to *reset* timing, *pause* timing, *continue* timing, and *close* the stopwatch component. All stopwatch definitions are placed in the module `stopwatch.icl`. The function `stopwatch` defines the stopwatch component. A main module, `usestopwatch.icl` that opens and controls the stopwatch is also defined. This module defines an interactive process with a menu to control the stopwatch component. Figure 11.2 gives a snapshot of the program in action. We first look at the stopwatch and then at the main program.

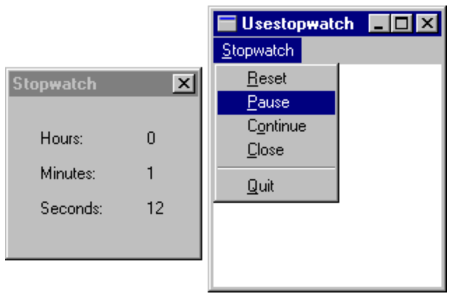


Figure 11.2: The stopwatch and control processes.

The stopwatch component

The original clock program created an interactive process with three timers and a dialogue. Each of the three timers changes a *local* state that keeps track of the elapsed seconds, minutes, and hours. This situation is schematised in Figure 11.3.

The stopwatch process is controlled by sending messages to a 'gateway' receiver. The timers are extended with a receiver component that handles the commands *reset*, *pause*, and *continue*. These commands will be sent to them by the gateway

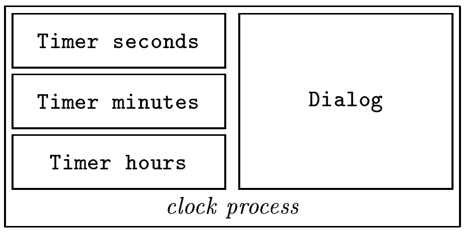


Figure 11.3: The structure of the clock process.

receiver. This gateway receiver handles the *close* command. Finally, the stopwatch process creates the same dialogue as the original clock program. The stopwatch process is schematised in Figure 11.4.

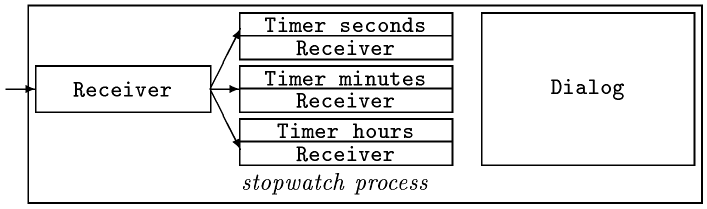


Figure 11.4: The structure of the stopwatch process.

So the new components are the gateway receiver and the receiver components of the timers. We first look at the gateway receiver and then timer receivers. Both receivers accept messages of the following type:

```
::  StopwatchCommands = Reset
                      | Pause
                      | Continue
                      | Close
```

The alternatives of the algebraic type `StopwatchCommands` correspond of course with the stopwatch commands *reset*, *pause*, *continue*, and *close*. The gateway receiver function `receive`, on receiving the `Close` message simply terminates the interactive process by applying `closeProcess` to its process state. Every other message is routed to the timer receiver components which are identified by the list `timerinfos`. Here is the function definition of `receive`:

```
receive :: StopwatchCommands (PSt .l) -> PSt .l
receive Close pst
 = closeProcess pst
receive msg pst
 = snd (seqList [syncSend timerRId msg\\{timerRId}<-timerinfos] pst)
```

The timer receiver components receive only the `StopwatchMessage` alternatives `Reset`, `Pause`, and `Continue`. In the clock example, the timer was parameterised with its timer interval. In this example, we need to add parameters to identify the timer (because it is going to be reset) and its receiver component (because receivers must have an identification value). So the definition of a stopwatch timer now is:

```
tdef :: TimerInfo
     -> Timer (Receiver StopwatchCommands) Int (PSt .l)
tdef {timerId,timerRId,timerInterval}
 = Timer timerInterval (Receiver timerRId receive [])
     [ TimerId       timerId
     , TimerFunction tick
     ]
```

The *reset* command should set the timer back to zero. One might suppose that it is sufficient to change only the value of the local state to zero, but that is not completely true. Resetting the stopwatch can occur at any moment. At that moment the timer should be synchronised with its local state. This can be done by first *disabling* and then *enabling* the timer. For this purpose the `StdTimer` functions `disableTimer` and `enableTimer` should be used. The reason that it works is because `enableTimer`, when applied to a disabled timer, synchronises the timer with the moment of evaluation. It does nothing in case the indicated timer was already enabled. Finally, on receiving the `Reset` message, the timer receiver component must set the corresponding text field of the dialogue to zero. This gives the following definition of the `Reset` alternative.

```
receive Reset (time,pst=:{io})
    # io = disableTimer timerId io
    # io = enableTimer  timerId io
    # io = setControlText textid "00" io
    = (0,{pst & io=io})
```

The *pause* command should halt the timer until further notice (either *reset* or *continue*). This is easily done by disabling the timer:

```
receive Pause (time,pst=:{io})
    = (time,{pst & io=disableTimer timerId io})
```

The *continue* command should let the timer continue from where it was paused. This is easily done by enabling the timer:

```
receive Continue (time,pst=:{io})
    = (time,{pst & io=enableTimer timerId io})
```

The final details of the stopwatch component are to create the proper identification values for the timers and their receiver components, and to export its definition as an interactive process. This is done by the function `stopwatch`. The stopwatch process is defined as a process group with no interesting local or public process state (using the ubiquitous `Void` singleton type constructor). Its initialisation action first creates the required `Ids` for the dialogue, and then the parameters required for the timers. Then initialisation proceeds as described above.

```
stopwatch :: (RId StopwatchCommands) -> Process
stopwatch rid
    = Process NDI Void initialise` []
where
    initialise` pst
        # (dialogIds, pst) = accPIO openDialogIds  pst
        # (timerInfos,pst) = accPIO openTimerInfos pst
        = initialise rid dialogIds timerInfos pst
```

For completeness, the definition module and implementation module of the stopwatch component are given below.

```
definition module stopwatch

//  **************************************************************************
//  Clean tutorial example program.
//
//  This module exports the types and functions needed to incorporate a stopwatch
//  component.
//  **************************************************************************

import StdIO

::  StopwatchCommands
    =   Reset
    |   Pause
    |   Continue
    |   Close

stopwatch :: (RId StopwatchCommands) -> Process
```

```
implementation module stopwatch

//  **************************************************************************
//  Clean tutorial example program.
//
//  This program defines a stopwatch process component.
//  It uses three timers to track the seconds, minutes, and hours separately.
//  Message passing is used to reset, pause, and continue timing.
//  The current time is displayed using a dialogue.
//  **************************************************************************

import StdEnv,StdIO

::  DialogIds
    =   {   secondsId       :: Id
        ,   minutesId       :: Id
        ,   hoursId         :: Id
        }
::  TimerInfo
    =   {   timerId         :: Id
        ,   timerRId        :: RId StopwatchCommands
        ,   timerInterval   :: TimerInterval
        }
::  StopwatchCommands
    =   Reset
    |   Pause
    |   Continue
    |   Close

second  :== ticksPerSecond
minute  :== 60*second
hour    :== 60*minute

openDialogIds :: *env -> (DialogIds,*env) | Ids env
openDialogIds env
    # ([secondsid,minutesid,hoursid:_],env) = openIds 3 env
    = ({ secondsId=secondsid,minutesId=minutesid,hoursId=hoursid }, env)

openTimerInfos :: *env -> ([TimerInfo],*env) | Ids env
openTimerInfos env
    # (tids,env)  = openIds  3 env
    # (rids,env)  = openRIds 3 env
    # intervals   = [second,minute,hour]
    = ( [  {timerId=tid,timerRId=rid,timerInterval=i}
        \\ tid<-tids & rid<-rids & i<-intervals
        ]
      , env
      )

stopwatch :: (RId StopwatchCommands) -> Process
stopwatch rid
    = Process NDI Void initialise` []
where
    initialise` pst
        # (dialogIds, pst) = accPIO openDialogIds  pst
        # (timerInfos,pst) = accPIO openTimerInfos pst
        = initialise rid dialogIds timerInfos pst

initialise :: (RId StopwatchCommands) DialogIds [TimerInfo] (PSt .l) -> PSt .l
initialise rid {secondsId,minutesId,hoursId} timerinfos pst
    # (errors,pst)  = seqList [  openTimer 0 (tdef timerinfo)
                              \\ timerinfo<-timerinfos
                              ]  pst
    | any ((<>) NoError) errors
        = closeProcess pst
    # (error,pst)   = openDialog Void ddef pst
    | error<>NoError
        = closeProcess pst
    # (error,pst)   = openReceiver Void rdef pst
    | error<>NoError
        = closeProcess pst
    | otherwise
        = pst
where
    tdef {timerId,timerRId,timerInterval}
        = Timer timerInterval (Receiver timerRId receive [])
            [   TimerId         timerId
            ,   TimerFunction   tick
            ]
    where
        tick nrElapsed (time,pst=:{io})
            # time  = (time+nrElapsed) mod maxunit
            # io    = setControlText textid (toString time) io
            = (time,{pst & io=io})

        receive Reset (time,pst=:{io})
            # io    = disableTimer timerId io
            # io    = enableTimer  timerId io
            # io    = setControlText textid "00" io
            = (0,{pst & io=io})
        receive Pause (time,pst=:{io})
            = (time,{pst & io=disableTimer timerId io})
        receive Continue (time,pst=:{io})
            = (time,{pst & io=enableTimer timerId io})

        (textid,maxunit)    = if (timerInterval==second) (secondsId,60)
                             (if (timerInterval==minute) (minutesId,60)
                                                         (hoursId,  24))

    ddef    = Dialog "Stopwatch"
                (   LayoutControl
                (   ListLS  [  TextControl text [ControlPos (Left,zero)]
                            \\ text<-["Hours:","Minutes:","Seconds:"]
                            ]
                )   []
                :+: LayoutControl
                (   ListLS  [  TextControl "0"  [ControlPos (Left,zero)
                                                ,ControlId  id
                                                ,ControlWidth (ContentWidth "00")
                                                ]
                            \\ id<-[hoursId,minutesId,secondsId]
                            ]
                )   []
                )   [ WindowClose (noLS closeProcess) ]
    rdef    = Receiver rid (noLS1 receive) []
    where
        receive Close pst
            = closeProcess pst
        receive msg pst
            = snd (seqList [syncSend timerRId msg \\ {timerRId}<-timerinfos] pst)
```

Using the stopwatch

Given the stopwatch component module `stopwatch`, we can create a program that opens, uses, and closes the stopwatch. The program will be as simple as possible. Its first action is to create the `(RId StopwatchCommands)` value, so that it can be given conveniently to the initialisation function of the interactive process. As this process will only have a menu, it is a single document interface process.

```
Start :: *World -> *World
Start world
    # (stopwatchid,world) = openRId world
    = startIO SDI Void (initialise stopwatchid) [] world
```

The initialisation function of the interactive process first creates the menu, `mdef` and then the stopwatch.

```
initialise :: (RId StopwatchCommands) (PSt .l) -> PSt .l
initialise stopwatchid pst
    # (error,pst)    = openMenu Void mdef pst
    | error<>NoError = closeProcess pst
    | otherwise      = openProcesses (stopwatch stopwatchid) pst
```

The menu triggers the stopwatch commands *reset*, *pause*, *continue*, and *close*. In addition, a *quit* command is added to terminate the whole program.

```
mdef
 = Menu "&Stopwatch"
   (   MenuItem "&Reset"    [MenuFunction (noLS (send Reset))]
   :+: MenuItem "&Pause"    [MenuFunction (noLS (send Pause))]
   :+: MenuItem "C&ontinue" [MenuFunction (noLS (send Continue))]
   :+: MenuItem "&Close"    [MenuFunction (noLS (send Close))]
   :+: MenuSeparator        []
   :+: MenuItem "&Quit"     [MenuFunction (noLS (closeProcess o
                                                 (send Close)))]
   )   []
```

Each of the stopwatch command menu functions is defined by the `send` function which is parameterised with the corresponding `StopwatchCommands` message alternative. Its purpose is to send its argument message to the gateway receiver of the stopwatch process, identified by `stopwatchid`. If this fails it emits a system beep. Here is its definition:

```
send msg pst
    # (error,pst)   = syncSend stopwatchid msg pst
    | error<>SendOk = appPIO beep pst
    | otherwise     = pst
```

Here is the complete code of the main program.

```
module usestopwatch

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program creates a simple program that uses the stopwatch process.
//  The program only has a menu to open the stopwatch and control it.
//  ****************************************************************************

import StdEnv, StdIO
import stopwatch

Start :: *World -> *World
Start world
    # (stopwatchid,world) = openRId world
    = startIO SDI Void (initialise stopwatchid) [] world

initialise :: (RId StopwatchCommands) (PSt .l) -> PSt .l
initialise stopwatchid pst
    # (error,pst)   = openMenu Void mdef pst
    | error<>NoError= closeProcess pst
    | otherwise     = openProcesses (stopwatch stopwatchid) pst
where
    mdef    = Menu "&Stopwatch"
                (   MenuItem "&Reset"    [MenuFunction (noLS (send Reset))]
                :+: MenuItem "&Pause"    [MenuFunction (noLS (send Pause))]
                :+: MenuItem "C&ontinue" [MenuFunction (noLS (send Continue))]
                :+: MenuItem "&Close"    [MenuFunction (noLS (send Close))]
                :+: MenuSeparator        []
                :+: MenuItem "&Quit"     [MenuFunction (noLS (closeProcess o
                                                              (send Close)))]
                ) []
    send msg pst
        # (error,pst)   = syncSend stopwatchid msg pst
        | error<>SendOk = appPIO beep pst
        | otherwise     = pst
```

Chapter 12

Clipboard handling

The *clipboard* is a simple communication metaphor that can be used between and within applications. An application can write some data to the clipboard (typically text or pictures) which can be read at a later point of time by the same or another application. This mechanism is supported by the functions in the definition module `StdClipboard`. At the moment only text can be handled. Because we intend to incorporate pictures as well in the near future the current version is set up in such a way that it can be extended upward compatibly. For this purpose an abstract data type, `ClipboardItem`, is defined. Two overloaded functions from the type constructor class `Clipboard` take care that data types can be converted to and from `ClipboardItem`s:

```
::  ClipboardItem

class Clipboard item where
    toClipboard   :: !item          -> ClipboardItem
    fromClipboard :: !ClipboardItem -> Maybe item

instance Clipboard {#Char}
```

By convention applications should provide several 'popular' data formats for the same clipboard content in descending order of accuracy. For instance, a text processor can first store its private format for the laid out text including font and style information, followed by an ASCII version of the same text, followed by a picture of the laid out text. For this reason the function `setClipboard` that writes the clipboard is not applied to one single clipboard data item but a list of them. The previous content will be erased. Programs are supposed to provide only one data item of each format, so `setClipboard` removes all duplicate formats from this list. Note that providing `setClipboard` with an empty list erases the clipboard.

```
setClipboard :: ![ClipboardItem] !(PSt .l) -> PSt .l
```

The function that reads the clipboard, `getClipboard`, returns the list of the current clipboard content in descending order of accuracy. With the conversion function `fromClipboard` an application can determine easily if some data item is present that it can handle.

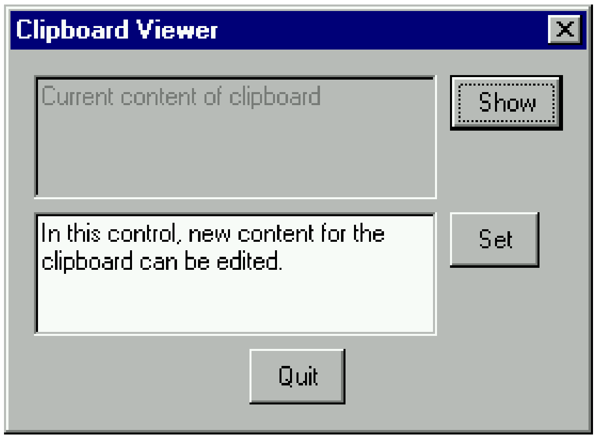
```
getClipboard :: !(PSt .l) -> (![ClipboardItem],!PSt .l)
```

Finally, because reading in a complete clipboard can be time-consuming or space-consuming the function `clipboardHasChanged` is provided that checks whether the clipboard has been updated since the last time the program checked it.

```
clipboardHasChanged :: !(PSt .l) -> (!Bool,!PSt .l)
```

12.1 Example: a clipboard editor

To illustrate the use of the clipboard, we construct a small program that shows the current content of the clipboard and that can write some text to the clipboard (see Figure 12.1). It will create only one dialogue with two text fields. In the first text field, the *show* field, the content of the clipboard can be *loaded* by pressing the button to its right. In the second text field, the *set* field, the content of the clipboard can be *stored* by pressing the button to its right.



In this control, new content for the clipboard can be edited.

Figure 12.1: The clipboard program.

The show text field and its activating button can be defined as follows:

```
showclip
  = EditControl   "" width nrlines [ControlSelectState Unable
                                   ,ControlId  showid
                                   ,ControlPos (Left,zero)
                                   ]
    :+:
    ButtonControl "Show" [ControlFunction (noLS show)]
```

The show text field is an edit text control that will not respond to keyboard input (because its `SelectState` attribute is `Unable`). It is identified by some `Id` of value `showid`. It will have some `width` and a height defined by the number of lines `nrlines`. The "Show" button, when selected, must read the content of the clipboard and figure out if there was a text clipboard item. It then sets the text of the show text field to the loaded clipboard content (an empty string if nothing was found). This action can be defined as follows:

```
show pst
    # (content,pst) = getClipboard pst
    # text          = getString content
    = appPIO (setControlText showid text) pst

getString [clip:clips]
    | isNothing item= getString clips
    | otherwise     = fromJust item
where
    item            = fromClipboard clip
getString []
    = ""
```

The set text field and its activating button are defined as follows:

```
setclip
  = EditControl   "" width nrlines [ControlId  setid
                                   ,ControlPos (Left,zero)
                                   ]
    :+:
    ButtonControl "Set"  [ControlFunction (noLS set)]
```

The set text field is an edit text control which accepts keyboard input. It is identified by some `Id` of value `setid`. It has the same dimensions as the show text control. The “Set” button, when selected, must get the content of the set text control and write this to the clipboard. This action is defined as follows:

```
set pst
   # (wst,pst)= accPIO (getParentWindow showid) pst
   # text     = fromJust (snd (getControlText setid (fromJust wst)))
   = setClipboard [toClipboard text] pst
```

The definition of the clipboard viewing dialogue simply summaries these elements and adds a “Quit” button to terminate the program:

```
clipview = Dialog "Clipboard Viewer"
                  (showclip :+: setclip :+: quit)
                  []
quit     = ButtonControl "Quit"
                  [ControlFunction (noLS closeProcess)
                  ,ControlPos      (Center,zero)
                  ]
```

The last details that remain to be defined are the opening of the interactive program. Here is the complete program code.

```
module clipboardview

//  **************************************************************************
//  Clean tutorial example program.
//
//  This program creates a dialogue to display and change the current content
//  of the clipboard.
//  **************************************************************************

import StdEnv, StdIO
```

```
Start :: *World -> *World
Start world
    # (showid,world) = openId world
    # (setid, world) = openId world
    = startProcesses (Process NDI Void (initialise showid setid) []) world

initialise showid setid pst
    # (error,pst)   = openDialog Void clipview pst
    | error<>NoError= closeProcess pst
    | otherwise     = pst
where
    clipview= Dialog "Clipboard Viewer" ( showclip :+: setclip :+: quit ) []
    showclip= EditControl "" width nrlines [ ControlSelectState Unable
                                           , ControlId          showid
                                           , ControlPos         (Left,zero)
                                           ]
              :+:
              ButtonControl "Show"         [ ControlFunction    (noLS show) ]
    setclip = EditControl "" width nrlines [ ControlId          setid
                                           , ControlPos         (Left,zero)
                                           ]
              :+:
              ButtonControl "Set"          [ ControlFunction    (noLS set) ]
    quit    = ButtonControl "Quit"         [ ControlFunction    (noLS closeProcess)
                                           , ControlPos         (Center,zero)
                                           ]
    width   = PixelWidth (hmm 50.0)
    nrlines = 4

    show pst
        # (content,pst) = getClipboard pst
        # text          = getString content
        = appPIO (setControlText showid text) pst

    set pst
        # (wst,pst)     = accPIO (getParentWindow showid) pst
        # text          = fromJust (snd (getControlText setid (fromJust wst)))
        = setClipboard [toClipboard text] pst

    getString [clip:clips]
        | isNothing item= getString clips
        | otherwise     = fromJust item
    where
        item            = fromClipboard clip
    getString []
        = ""
```

Chapter 13

Printing

In this chapter we introduce two modules that enable Clean programs to print: `StdPrint` (Appendix A.25) and `StdPrintText` (Appendix A.26). At first we will discuss the dialogues the user is confronted with when he wants to print something (Section 13.1). The `StdPrint` module can be used to print any arbitrary output. It is illustrated in the Sections 13.2 to 13.5. The `StdPrintText` module can be used when only text should be printed. Its use is explained in Section 13.6.

13.1 The user interface for printing

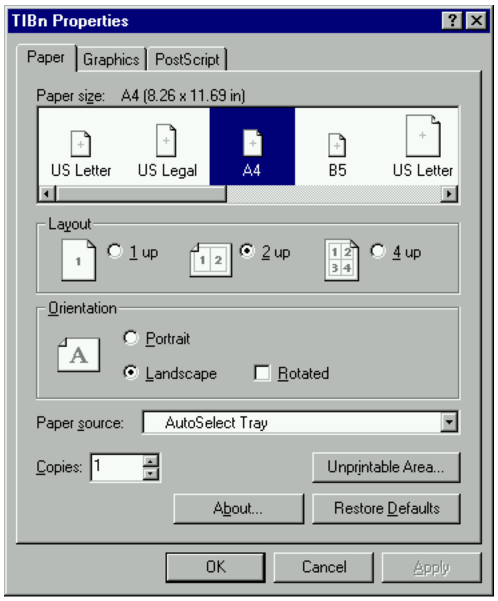
When a user wants to get some printed output, he has to specify several parameters for printing. These parameters can be divided into two groups: the *print setup* parameters determine the printer and the paper format that are used and the page orientation (portrait or landscape). The *print job* parameters include the number of copies and the range of pages to print. According to these two sets there are two dialogues to specify these parameters and two Clean types for objects that store this information. The dialogues are the *print setup dialogue* (Figure 13.1) and the *print job dialogue* (Figure 13.2). The types are the abstract data type `PrintSetup` and the `JobInfo` record (see Section 13.2). The print job dialogue launches the print job if the user does not cancel. In most applications the user has to answer this dialogue just before printing, but using the print setup dialogue is optional.

There is always a default `PrintSetup` which is returned by the `defaultPrintSetup` function. Such a `PrintSetup` can be edited with the print setup dialogue when it is applied to the `printSetupDialog` function. These functions have the following signatures:

```
::  PrintSetup

defaultPrintSetup :: !*env -> (!PrintSetup, !*env)
                  |  FileEnv env
printSetupDialog  :: !PrintSetup
                     !*env -> (!PrintSetup, !*env)
                  |  PrintEnvironments env
```

The `env` parameter can be the `World`,`Files`, or the `PSt`. For the `defaultPrint-Setup` function the `env` parameter can also be the `IOSt`.

On Macintosh computers the print setup dialogue and the print job dialogue are



Paper size: A4 (8.26 x 11.69 in)

Layout: 1 up [✓ 2 up] 4 up

Orientation: Portrait [✓ Landscape] Rotated

Paper source: AutoSelect Tray

Copies: 1

Figure 13.1: A print setup dialogue

typically reachable from two different menu items. On Windows(95/NT) the print setup dialogue can usually only be reached by clicking on a button in the print job dialogue. The `StdSystem` module function `printSetupTypical` can be used to determine whether a separate menu item for the print setup dialogue is typical on the used platform.

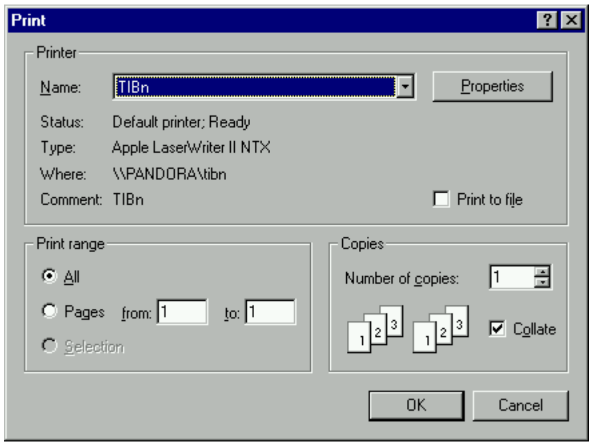
13.2 The print function

The `print` function in the `StdPrint` module has the following signature:

```
print ::  !Bool !Bool
          .(PrintInfo !*Picture -> ([IdFun *Picture],!*Picture))
          !PrintSetup !*env
      -> (!PrintSetup,!*env)
      |  PrintEnvironments env

::  PrintInfo
    =   {   printSetup::  PrintSetup
        ,   jobInfo    ::  JobInfo
        }
::  JobInfo
    =   {   range      ::  !(!Int,!Int)
        ,   copies     ::  !Int
        }
```

The parameters of `(print doDialog emulateScreen pages printSetup env)` have the following meaning:

`doDialog`


Print

Printer

Name: TIBn

Status: Default printer; Ready

Type: Apple LaserWriter II NTX

Where: \\PANDORA\tibn

Comment: TIBn

☐ Print to file

Print range

● All

○ Pages from: 1 to: 1

○ Selection

Copies

Number of copies: 1

☑ Collate

Figure 13.2: A print job dialogue

If True, the print job dialogue will pop up, otherwise printing will happen in the default way.

`emulateScreen`
Iff True, the screen resolution will be emulated, see Section 13.3.

`pages`
This function defines the printed output. It generates a list of drawing functions. Each drawing function corresponds to one page. The function can access a `PrintInfo` record and a `Picture`. The `Picture` parameter can be used to retrieve information like font metrics or string widths from a printer `Picture` that will have the same properties as the `Picture` environment used for printing the pages.

`printSetup`
The print setup. As mentioned before, the user can change this print setup from the print job dialogue under Windows(95/NT).

`env`
A print environment can be `Files`, `World` or `PSt`. If the print environment is the `PSt`, then the print function can update the contents of windows during printing (to be more exact: between the rendering of two pages). This is important on the Windows(95/NT) platforms, since printing can take some time. On the Macintosh platform printing is a blocking operation, so no window updating is necessary. If the print environment is either `Files` or the `World`, then no window updating will happen, and, on the Windows(95/NT) platform, no cancel dialogue for printing will pop up.

The `PrintInfo` record contains information that will be needed for generating the list of drawing functions. The fields of the `JobInfo` part of this record have the following meaning:

`range`
This field contains the numbers of the first and last page which the user has chosen via the print dialogue. If the user has chosen "all pages", then the first page will be 1, and the last page will be 9999.

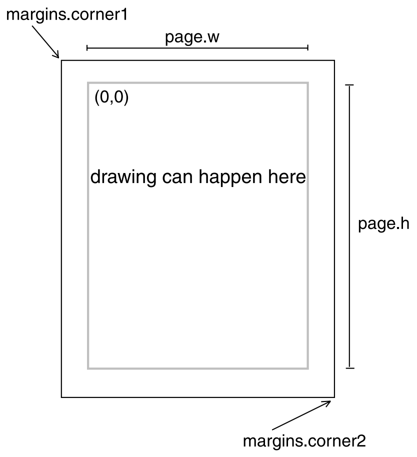


Figure 13.3: Page measurements

`copies`
: These are the number of copies to generate. This will not necessarily be equal to the number of copies as specified in the print job dialogue. Some printer drivers take care of producing the appropriate number of copies themselves. In that case the value of this field will be 1.

The `PrintInfo` record also contains a `printSetup` field. This print setup will be used for printing. The `PrintSetup` value can be passed to the `getPageDimensions` function:

```
getPageDimensions      ::  !PrintSetup !Bool -> PageDimensions

::  PageDimensions
    =   {   page       :: !Size
        ,   margins    :: !Rectangle
        ,   resolution :: !(!Int,!Int)
        }
```

The `PageDimensions` record fields have the following meaning:

`page`
: This field contains the size of the drawable area of a sheet in pixels, see Figure 13.3. The `print` function sets the origin of such a `Picture` to `zero`. So any drawing in a rectangle from `zero` to {`x=page.w-1, y=page.h-1`} will be visible on the printed paper.

`margins`
: This field contains information about the size of a whole sheet of paper. This is not only the area in which the printer can draw, but also the margins of the paper. The x-axis and y-axis coordinate values of `corner1` are negative, while the x-axis and y-axis coordinate values of `corner2` are greater than the corresponding values of the `page` field, see Figure 13.3.

`resolution`
: This field contains the horizontal and vertical resolution of the printer in *dpi* (dots per inch).

These measurements of a printer picture depend on whether the screen resolution is emulated or not. Only if the Boolean parameter of the `getPageDimensions` function is True, then the values for a printer picture that emulates the screen resolution are returned.

The `print` function returns the `PrintSetup` that is stored in the `PrintInfo` record.

Here is an easy example program, `printExample1`, that uses the `print` function:

```
module printExample1

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program demonstrates the use of the "print" function.
//  It prints two pages, one with the text "Hello Printer", and one that informs
//  about the printer resolution.
//  ****************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
    # (defaultPS, world) = defaultPrintSetup world
    = snd (print True False pages defaultPS world)
where
    pages :: PrintInfo *Picture -> ([IdFun *Picture],*Picture)
    pages {printSetup, jobInfo={range=(first,last), copies}} picture
      # {resolution=(xRes,_)}
                     = getPageDimensions printSetup False
      # twoPages     = [ drawAt { x=100, y=100 } "Hello Printer"
                       , drawAt { x=100, y=100 } (   "Horizontal Resolution: "
                                                 +++ toString xRes
                                                 +++ " dpi."
                                                 )
                       ]
      # oneCopy      = twoPages % (first-1,last-1)
      = ( flatten (repeatn copies oneCopy), picture)
```

On the Macintosh platform, a Clean program needs “extra memory” for printing. In the “Application Options” dialogue of the CleanIDE, this value should be set to about 200K, otherwise the program may crash.

Running `printExample1` will pop up the print job dialogue. Let’s assume that the user chooses to print only one copy of the first two pages. These values will be passed to the `pages` function, via its first argument of type `PrintInfo`. So the values of `first` and `copies` are one, and the value of `last` is two. The list of drawing functions, which is returned by the `pages` function is then equal to `two-Pages`. It contains two drawing functions, so two pages will be printed. The first printed page will contain the text “Hello Printer”, and the second page will contain the printer resolution.

If the user chooses to print only the first page, then the value of `last` is also one. The `%` operator selects only the first element of `bothPages` and so only one page is printed. If the user chooses to print all pages, then the value of `last` is 9999. The `%` operator then selects both pages, which will subsequently be printed. The application of `repeatn` ensures that the right number of copies is generated.

In general, printing is done in the same way as drawing on the screen, namely by applying drawing functions on a `Picture` environment. In the rest of this chapter printer `Pictures` and screen `Pictures` are distinguished. There are two differences between these two kinds of `Pictures`:

1. Functions which hilite parts of a `Picture`, or perform a XOR operation, will

have no effect on a printer `Picture`.

2. Bitmapped fonts cannot be drawn on both kinds of `Picture`s with the same sizes. This is because the pixel resolutions of both output devices are, in general, different. TrueType fonts work well for both kinds.

13.3 Reusing drawing functions

In many cases an application should be able to draw a certain content on printer `Picture`s as well as on screen `Picture`s. Ideally, such a program should use the same code for both output devices. But a problem arises because in the object I/O library *two* different units of measurement are used for drawing, namely *points* and *pixels* (see also Chapter 5).

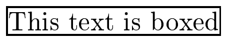
The physical extent and position of a graphical object on the paper or on the screen always depends on `Int` values, which are passed to a drawing function. These values are interpreted either as point values or as pixel values. Point values occur only when the size of a font is specified. In all other cases pixel values are used, e.g. in `drawAt {x=100,y=100}`.

A point is approximately $\frac{1}{72}$ of an inch. A pixel has of course also a physical extent, which can be measured in inches. But unlike a point, the size of a pixel varies from output device to output device: the size of a printer pixel can be about 3 to 4 times smaller than the size of a screen pixel. This implies that the result of applying a drawing function on a printer `Picture` will result in smaller images compared with screen `Picture`s. Only when text is drawn, this shrinking does not happen, because the size of the used font is specified in points, a resolution independent specification of physical extent.

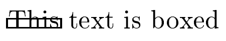
Consider for example the following function:

```
myDrawfunction picture
    # picture = drawAt {x=20,y=80} string picture
    # picture = drawAt {x=18,y=68} box    picture
    = picture
where
    string    = "This text is boxed"
    box       = {box_w=120,box_h=15}
```

could cause a screen to contain the following:



but the printed output could look like this:



The reason is that the upper code uses constants to express pixel values. There a two ways to deal with this problem: *implicit* and *explicit* scaling.

Implicit scaling

Setting the second Boolean argument of the `print` function to `True` enables implicit scaling. This causes printer `Picture`s to emulate the screen resolution. This will suppress the shrinking effect. This approach has the advantage that it's very easy to write a function that can be used for drawing on both a screen and printer `Picture`. But not everything can be printed very well by using this option. For instance lines on the paper can not be thinner than on the screen. Further problems arise when the printer resolution is not a whole multiple of the screen resolution. This causes rounding errors in the emulation process. If the ratio between printer and screen resolution is for instance 3.5, then the emulation of line drawing with a pen size of one screen pixel will result in a line on the paper that is only 3 printer pixels wide. The following drawing function draws five hundred horizontal lines:

```
      seq [drawAt {x=0,y=y} {vx=500,vy=0} \\ y<-[0..500]]
```

On paper space appears between the lines, but not on the screen.

Explicit scaling

In most cases using implicit scaling is preferrable because it leads to simple programs. But if the disadvantages of implicit scaling are not acceptable, the application has to perform the scaling itself. One way to obtain this is to avoid constant pixel values. Consider for instance a program that draws a string on a certain place in a window or sheet. The position of the string could be specified by using values that are related to string widths or font metrics. Since string width and font metrics always depend on a `Picture`, they will reflect the resolution of the output device. The upper `myDrawfunction` on page 162 could be rewritten to:

```
myDrawfunction picture
  # (metrics,picture)= getPenFontMetrics picture
  # (mwidth,descent) = (metrics.fMaxWidth,metrics.fDescent)
  # height           = fontLineHeight metrics
  # (swidth,picture) = getPenFontStringWidth string picture
  # box     = {box_w=swidth+6*mwidth/5,box_h=0-height}
  # picture = drawAt {x=20,            y=80}      string picture
  # picture = drawAt {x=20-mwidth/5,y=80+descent} box picture
  = picture
where
  string    = "This text is boxed"
```

The constants in this example are not pixel values, because they are *multiplied* with pixel values like `height`. This example also shows that writing resolution independent drawing functions results in more complex code.

Another way to perform scaling is to use the printer and screen resolution explicitly. Then the code fragment on page 162 could be rewritten to:

```
myDrawfunction (sclNom,sclDenom) picture
  # picture = drawAt {x=scl 20,y=scl 80} string picture
  # picture = drawAt {x=scl 18,y=scl 68} box    picture
  = picture
where
  string    = "This text is boxed"
  box       = {box_w=scl 120,box_h=scl 15}
  scl x     = sclNom*x/sclDenom
```

For printing, `sclNom` should be the printer resolution, and `sclDenom` should be the resolution of the screen. For drawing on the screen, both values should be one. The `StdPicture` module function `getResolution` will return the horizontal and vertical resolution of a screen or printer picture:

```
getResolution :: !*Picture -> (!(!Int,!Int),!*Picture)
```

The `accScreenPicture` function (see Appendix A.30) can be handy in this context. The printer resolution can of course also be found in the `PageDimensions` record.

13.4 The printUpdateFunction function

The `printUpdateFunction` function in `StdPrint` is a good example of using implicit scaling. This function offers a very easy way to print the contents of a window. It has the following signature:

```
printUpdateFunction :: !Bool
                       (UpdateState -> *Picture -> *Picture)
                       [Rectangle]
                       !PrintSetup !*env
                    -> (!PrintSetup,!*env)
                    |  PrintEnvironments env
```

The first and the last two parameters are identical to the first and last two parameters of the `print` function. Also the result has the same semantics as shown with the `print` function. Let's assume that an application has opened a window. The contents of the window is defined by its `WindowLook` attribute. This attribute contains a `Look` function which updates the contents of a window everytime this is required (see also Section 6.4.1). The `Look` is defined as:

```
::  Look :== SelectState -> UpdateState -> *Picture -> *Picture
```

If such a `Look` function is curried with a `SelectState`, it can be passed to the `printUpdateFunction` function as the second argument.

The list of rectangles parameter specifies the parts of the view domain that should be printed. If such a rectangle is too big for one sheet of paper, it will be distributed on several pages.

13.5 The printPagePerPage function

The third function in the `StdPrint` module is the `printPagePerPage` function. The `printUpdateFunction` and the `print` functions are both specialisations of this function. This function performs a state transition function on a polymorphic state. The state transition function itself has to be passed to the `printPagePerPage` function. With each application of this state transition function the contents of the actually printed page is drawn. Furtheron this state transition function returns a Boolean value which indicates whether there are further pages to print. So it will be applied repeatedly, until this value becomes `True`.

The function `printPagePerPage` has the following signature (slightly changed for readability):

```
printPagePerPage
 :: !Bool !Bool
    .unq
    (.unq PrintInfo *Picture  -> ((Bool,Point2),(.state,*Picture)))
    (       (.state,*Picture) -> ((Bool,Point2),(.state,*Picture)))
    !PrintSetup !*env
 -> (Alternative .unq .state,!*env)
 | PrintEnvironments env

:: Alternative unq state = Cancelled unq | StartedPrinting state

instance PrintEnvironments Files, World, (PSt .l)
```

The arguments of

```
printPagePerPage doDialog emulateScreen unq prepare stateTrans env
```

have the following meaning:

`doDialog`, `emulateScreen`, `printSetup`, **and** `env`
: Analogous to those in the `print` function.

`unq`
: If a unique object *unq*, e.g. a text file, has to be accessed and passed back by `printPagePerPage` it should be passed via this parameter. If the user cancels printing via the print job dialogue, (`Cancelled` *unq*) will be returned. Otherwise the `prepare` function is applied to this value.

`prepare`
: Calculates the initial state. In order to do so it can access the `PrintInfo` record and a printer `Picture`. Drawing in that `Picture` will have no effect on the printed output. The `prepare` also returns a Boolean and a `Point2` value. The Boolean value is `False` iff there is another page to print. The `Point2` will be the origin of the next page, if there is one.

`stateTrans`
: This argument is the state transition function that generates the pages to be printed. Just like the `prepare` function, each application of `stateTrans` returns a Boolean and a `Point2` value with the same meaning. If the user did not cancel printing via the cancel dialogue, the `printPagePerPage` function will return the final `state` in the `StartedPrinting` alternative constructor of `Alternative`. Since this `state` is polymorphic, any arbitrary value can be returned, e.g. the `PrintInfo` record.

13.6 Printing text

The module `StdPrintText` (Appendix A.26) offers functions to print text. These functions are overloaded in the argument that specifies the text to print. In this way the source of the text can be an arbitrary data structure, e.g. a text file or a list of characters.

The `printText1` function has the following signature:

```
printText1 :: !Bool    !WrapMode
              !FontDef !Int
              !*charStream
              !PrintSetup !*env
           -> ((!*charStream,!PrintSetup),!*env)
           |  CharStreams charStream & PrintEnvironments env

class CharStreams cs where
    getChar    :: !*cs -> (!Bool,!Char,!*cs)
    savePos    :: !*cs -> *cs
    restorePos :: !*cs -> *cs
    eos        :: !*cs -> (!Bool,!*cs)

::  WrapMode   :== Int

NoWrap         :== 0
LeftJustify    :== 1
RightJustify   :== 2
```

This function simply prints the text contained in the `charStream` object.

The first and the two last parameters are identical to the first and last two parameters of the `print` function. Also the returned `PrintSetup` and print environment have their usual semantics. The other parameters have the following meaning:

`WrapMode`
: This parameter determines how lines are handled that are too long to fit on the paper. `NoWrap` suppresses wrapping, while `LeftJustify` and `RightJustify` wrap long lines and adjust the rest of the line to the left and right margin respectively.

`FontDef`
: This parameter determines the font of the text.

`Int`
: This parameter controls the tab width, measured in the number of space characters.

A `CharStreams` instance contains the text to be printed. Its functionality is analogous to the behaviour of a text file. So a `CharStreams` value contains a finite sequence of characters and a position pointer that points to the actual character. Applying `getChar` retrieves this actual character and increases the position pointer. The Boolean return value is True iff this operation was successful. Characters can be read sequentially, until the `eos` (end of stream) function returns `True`. The actual position of a `CharStreams` value can be saved with the `savePos` function. The `restorePos` function resets the position pointer to the previously saved position.

The following example illustrates how to print a text that is stored in a list of characters. Therefore the `CharStreams` class is instanciated with a proper type `ListCharStream`.

```
module printCharList

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program demonstrates the use of the function printText1.
//  It instantiates the CharStreams class with the ListCharStream type. Objects
//  of type ListCharstream contain a list of characters, which should be printed.
//  ******************************************************************************

import StdEnv, StdIO

::  *ListCharStream
    =   {   list    :: [Char]
        ,   savedPos:: [Char]
        }

instance CharStreams ListCharStream where
    getChar sc=:{list}
        = ( not empty, if empty ' ' (hd list), {sc & list=if empty list (tl list)})
    where
        empty   = isEmpty list
    savePos sc
        = { sc & savedPos=sc.list }
    restorePos sc
        = { sc & list=sc.savedPos }
    eos sc=:{list}
        = (isEmpty list,sc)

fontDef = { fName="Courier New", fStyles=[], fSize=9 }

Start world
    # (defaultPS, world) = defaultPrintSetup world
    = printText1 True NoWrap fontDef 4
              { list=['Hello again, printer'], savedPos=[] }
              defaultPS
              world
```

The `StdPrintText` module instanciates the `CharStreams` class with the `*FileCharStream` type. The following example illustrates how to print a text file, and how to use the functions `fileToCharStream` and `charStreamToFile`.

```
module printFile

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program demonstrates printing a text file by using the function printText1.
//  ******************************************************************************

import StdEnv, StdIO

fileName = "printFile.icl"
fontDef = { fName="Courier New", fStyles=[], fSize=9 }

Start world
    # (ok,file,world)       = fopen fileName FReadData world
    | not ok
        = abort ("file "+++fileName+++" not found")
    # (defaultPS,world)     = defaultPrintSetup world
    # ((charStream,_),world)= printText1 True NoWrap fontDef 4
                                (fileToCharStream file)
                                defaultPS world
    # (ok,world)            = fclose (charStreamToFile charStream) world
    | not ok
        = abort "can't close file"
    | otherwise
        = world
```

The `printText2` function allows you to print text with a header on each page. This function expects the same parameters as `printText1`, but takes in addition two strings. The first string will appear on the left corner of each header, and the second string, concatenated with the current page number, will appear on the right side of each header. The previous example can be altered as follows:

```
module printFileWithHeader

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program demonstrates printing a text file by using the function printText2.
//  ****************************************************************************

import StdEnv, StdIO

fileName = "printFileWithHeader.icl"
fontDef  = { fName="Courier New", fStyles=[], fSize=9 }

Start world
    # (ok,file,world)       = fopen fileName FReadData world
    | not ok
       = abort ("file "+++fileName+++" not found")
    # (defaultPS,world)     = defaultPrintSetup world
    # ((charStream,_),world)= printText2 "This file is printed with Clean" "page "
                               True NoWrap fontDef 4
                               (fileToCharStream file)
                               defaultPS world
    # (ok,world)            = fclose (charStreamToFile charStream) world
    | not ok
       = abort "can't close file"
    | otherwise
       = world
```

Finally, the function `printText3` permits full control over the look of a header and/or trailer (or footer) on each page. Its signature is:

```
printText3
    :: !Bool !WrapMode !FontDef !Int
       (PrintInfo    *Picture -> (userInfo,(Int,Int),*Picture))
       (userInfo Int *Picture -> *Picture)
       !*charStream !PrintSetup  !*env
 -> (!(!*charStream,!PrintSetup),!*env)
 |  CharStreams charStream & PrintEnvironments printEnv
```

The fifth and sixth parameter of

```
printText3 doDialog wrapMode fontParams spacesPerTab
           textRangeFunc
           eachPageDrawFunc
           charStream printSetup env
```

have the following meaning:

`textRangeFunc`
: This function takes a `PrintInfo` record and a printer `Picture` and returns a triple. The first result can be any data of arbitrary type. This data will be passed by `printText3` to `eachPageDrawFunc`. The second triple result is a pair `(top,bottom)`, where `top` $<$ `bottom` (`printText3` aborts if this is not the case). The printed text will appear between these y-coordinates only, leaving you room to print a header and a trailer for each page.

`eachPageDrawFunc`

This function is responsible for drawing a header and trailer. As parameters it takes the data produced by `textRangeFunc`, the actual page number, and a `Picture`. This function will be called from `printText3` for each page. It should return the `Picture` in which the header and/or trailer for the current page are drawn. Drawing is not restricted to any particular area on the printer `Picture`.

Chapter 14

TCP

In this chapter we cover Clean's TCP interface. Blocking is an inherent problem with network I/O: a function *blocks* when it can not be reduced further until some external condition changes. A program can not do anything while a function blocks[1]. An example of blocking is a program that tries to receive some data. Such a program could block until the data arrives.

It makes a difference whether a Clean program engages in event driven I/O or not. Programs engage in event driven I/O by evaluation of functions such as `startIO` or `startProcesses` (described in Chapter 11). Because the TCP interface can be used on the `World` environment (*World* programs) and the GUI environments `PSt` and `IOSt` (*GUI* programs) we will distinguish between these kinds of programs in this chapter. For *World* programs blocking is not a real problem. This is not true for GUI programs: they should not block, or at least not block a "long" time. The "$\frac{1}{10}$ second rule" says that a GUI program should not take longer than $\frac{1}{10}$ second to process an event. Hence a GUI program should not block longer than this time (if the programmer respects that rule). The reason for this rule is that the user of an GUI program should have the impression that the program responds immediately to his actions. Furtheron, window update events should also be handled by the program promptly. Failing to do so can make your desktop windows look really ugly and uninformative.

After a short introduction to TCP in Section 14.1 some basic ideas are discussed in Section 14.2. Section 14.3 is about the blocking approach which is applicable in *World* programs. The non blocking approach for GUI programs is discussed in Section 14.4.

14.1 Introduction to TCP

TCP (Transmission Control Protocol) offers the possibility to transfer data between programs which are running on different computers via a network. TCP is a connection oriented protocol. Before data can be transferred a connection between two running programs has to be established. A TCP connection is always a duplex connection. This means that both sides can send and receive data when they are connected. After all data has been sent and received, the connection has to be teard down.

[1]The Clean compiler produces *sequential* code. This is the reason why a program can not do anything else while it is blocking. If the Clean compiler would be able to produce non sequential code, blocking would be much less of a problem.

When establishing a connection between two programs, one of them plays the role of the *server*, and the other the *client*. The client has to know the *address* of the server. The address consists of two parts: an *IP address* and a *port number*. The IP address uniquely identifies every computer on the internet. An IP address is a 32 bit quantity, which is often written down in “dotted decimal form” like e.g. “131.174.33.11”. It is possible to establish different connections to one computer at the same time. The port number discriminates these connections. The port number is a 16 bit quantity. For some services some port numbers are reserved. For example, “HTTP” is reachable via the port number 80 and “telnet” via port number 23.

When a connection is established, the following happens: first the server has to *listen* on a certain port. Listening on a port means being prepared for receiving connection requests from clients. The client knows both this port number and the IP address of the computer, where the server is running. So he issues a connection request, which will reach the server via the internet. If the server accepts the connection request, the connection is established. Now data can be transferred. The sent data will reach the other side without being duplicated or altered in its chronological order.

A service called DNS (Dynamic Name System) helps us human beings to avoid having to remind IP addresses. The DNS service translates alphanumerical aliases such as “www.cs.kun.nl” into IP adresses.

14.2 Basic ideas

In Clean’s TCP interface *channels* play an important role. Intuitively a channel is like a pipe: what you put in on one end comes out on the other end after a while. We do not transport material goods like water or gas with our pipes, but information or rather *messages*. It is well possible that many messages reside within a channel at the same time while they are on their way to the other end. But it is not possible that they come out in another order than they were put in (we do not support out of band data). The most important things you can do with a channel are of course sending and receiving.

In our approach each channel type has a message type. This means, that we can only send and receive messages with that message type on a certain channel. But of course it is possible to define several channel types. For instance the following channel type is defined:

```
::  *TCP_RCharStream :== TCP_RCharStream_ Char
```

The `TCP_RCharStream` type is a channel type with the message type `Char`. You don’t have to know how `TCP_RCharStream_` is defined (note the underscore). In general a channel type is a type constructor with one argument, its message type. If you want to receive messages on such a channel you can apply the receive function:

```
receive :: !*(*ch .a) !*env -> (!.a,!*(*ch .a),!*env)
        |  ChannelEnv env & Receive ch
```

To help you understand this weird definition, let’s give an example how to specialise this type. It is necessary to know that `TCP_CharStream` is an instance of the `Receive` class, and that a `ChannelEnv` can be a `World`, `IOSt` or `PSt` environment. The type of `receive` is defined as general as possible. The following function definition has a type that is more restrictive than it could be:

```
receive2 :: TCP_RCharStream *World -> (Char,TCP_RCharStream,*World)
receive2 ch env = receive ch env
```

The type of `receive2` is obtained by uniform substitution of `!*(*ch .a)` with `TCP_RCharStream` and `!.a` with `Char`. From the type of `receive2` we see that we can receive a `Char`, if we apply the `receive2` function on a `TCP_RCharStream` in a `World`. Channel types are uniquely attributed.

14.3 Blocking TCP in *World* programs

Since this section is about blocking use of TCP, the described methods are applicable for *World* programs. Section 14.4 discusses non blocking use of TCP in *GUI* programs.

The library modules that are relevant for this section are `StdChannels`, `StdTCPDef` and `StdTCPChannels` (see Appendixes A.2, A.40 and A.39). The module `StdTCP` imports all these modules similar to the `StdEnv` and `StdIO` modules.

In general, a TCP session is divided into three phases: the connection establishment phase, the data transfer phase, and the disconnect phase. These three phases will be covered in the Sections 14.3.1 to 14.3.3. In these sections small example functions are described. These functions are combined to give an example for a client and a server program in Section 14.3.4. The technique of multiplexing is discussed in Section 14.3.5 which includes a fancy chat server example program. Section 14.3.6 introduces some further channels that can be used.

14.3.1 Establishing a connection

We show the process of establishing a connection first for client programs, and then for server programs.

The client program has to know the IP address and port number of the server program. There is a function that uses the DNS to get an IP Address:

```
lookupIPAddress :: !String !*env -> (!Maybe IPAddress,!*env)
                |  ChannelEnv env
```

The `String` that is passed can be an alphanumerical internet address such as `martinpc.cs.kun.nl`, but also a dotted decimal notation such as `131.174.33.11`. To establish a new connection, the client simply calls the `connectTCP_MT` function:

```
connectTCP_MT     :: !(Maybe !Timeout) !(!IPAddress,!Port)    !*env
                  -> (!TimeoutReport,!Maybe TCP_DuplexChannel,!*env)
                  |  ChannelEnv env

::  Port          :== Int
::  Timeout       :== Int
::  TimeoutReport =   TR_Expired
                  |   TR_Success
                  |   TR_NoSuccess
```

The optional `Timeout` value is measured in (platform dependent) ticks. The `TimeoutReport` informs about success or failure of the attempt to connect. Iff this value is `TR_Success` then `Just` a `TCP_DuplexChannel` is returned. A `TCP_DuplexChannel` is a record that contains two channels: one TCP channel to *send* and one to *receive*:

```
::  *TCP_DuplexChannel
    :== DuplexChannel *TCP_SChannel_ *TCP_RChannel_ ByteSeq

::  DuplexChannel sChannel rChannel a
    =   {   sChannel ::  sChannel a
        ,   rChannel ::  rChannel a
        }
::  *TCP_SChannel    :== TCP_SChannel_ ByteSeq
::  *TCP_RChannel    :== TCP_RChannel_ ByteSeq
```

The TCP_SChannel and the TCP_RChannel types are channel types. The type of the messages that can be received is the abstract data type ByteSeq. This type resembles sequences of bytes. So with one message a sequence of bytes can be sent or received (see Appendix A.40).

Example

We define a function that looks up an IP address and tries to connect to that machine on port 2000. For simplicity we let the program abort if any of the operations fails. If no failure occurs, the result is a TCP_DuplexChannel.

```
clientConnect :: !*World -> (!TCP_DuplexChannel,!*World)
clientConnect world
  # (mbIPAddr,world)= lookupIPAddress "martinpc.cs.kun.nl" world
  | isNothing mbIPAddr
    = abort "DNS lookup failed"
  # ipAddr          = fromJust mbIPAddr
  # (tReport,mbDuplex,world)
                    = connectTCP_MT Nothing (ipAddr,2000) world
  | tReport<>TR_Success
    = abort "can't connect to port 2000"
  | otherwise
    # duplexChannel = fromJust mbDuplex
    = (duplexChannel,world)
```

Now we discuss the process of establishing a connection for a server program. The server has to call the openTCP_Listener function to listen on a given port:

```
openTCP_Listener  ::  !Port !*env
                  ->  (!OkBool,!Maybe TCP_Listener,!*env)
                  |   ChannelEnv env

::  *TCP_Listener :== TCP_Listener_ (IPAddress,TCP_DuplexChannel)
::  OkBool        :== Bool
```

The Port parameter should be the desired port number. Iff listening on the given port succeeds, then the OkBool result will be True and the Maybe result will be Just a TCP_Listener. If another program on the same machine already listens on the given port, this attempt will fail. A TCP_Listener is a channel on which connection requests from remote clients can be received. So the server does this by applying the receive function on the TCP_Listener:

```
receive :: !*(*ch .a) !*env -> (!.a,!*(*ch .a),!*env)
        |  ChannelEnv env & Receive ch
```

From the type definition of TCP_Listener you can see that the message type a is a pair that consists of an IPAddress and a TCP_DuplexChannel. Receiving a message of such a type establishes a new connection. The IPAddress is the IP address of the remote client that wants to connect. The TCP_DuplexChannel is the new connection itself.

So a TCP_Listener is a *channel* on which *channels* can be received. With one receive on a TCP_Listener a TCP_SChannel (to send) *and* a TCP_RChannel (to receive) are obtained. (If you are familiar with the sockets API you will recognise that a receive on a TCP_Listener resembles the accept call.)

Example

The following function listens on port 2000 and accepts one connection:

```
serverConnect :: !*World -> (!TCP_DuplexChannel,!*World)
serverConnect world
    # (ok,mbListener,world) = openTCP_Listener 2000 world
    | not ok
       = abort "can't open Listener on port 2000"
    # listener              = fromJust mbListener
    # ((_,duplexChannel),listener,world)
                            = receive listener world
    # world                 = closeRChannel listener world
    = (duplexChannel,world)
```

The closeRChannel function is discussed in Section 14.3.3.

14.3.2 Basic operations on channels

In the previous section we came in contact with three channel types: the TCP_-Listener, the TCP_SChannel, and the TCP_RChannel. These types are instances of type constructor classes that are defined in module StdChannels (Appendix A.2). The TCP_Listener and the TCP_RChannel are instances of the Receive class:

```
class Receive ch where
    receive_MT :: !(Maybe !Timeout)          !*(*ch .a)  !*env
               -> (!TimeoutReport,!Maybe !.a,!*(*ch .a), !*env)
                                             | ChannelEnv  env
    available  ::                            !*(*ch .a)  !*env
               -> (!Bool,                    !*(*ch .a), !*env)
                                             | ChannelEnv  env
    eom        ::                            !*(*ch .a)  !*env
               -> (!Bool,                    !*(*ch .a), !*env)
                                             | ChannelEnv  env
    ...
```

Each of these functions gets and returns a channel and an environment. "MT" is a shorthand for *m*aybe *t*imeout. The receive_MT function tries to receive a message of type a on the channel. This function might block until the message arrives. A timeout value can be given to limit this time. The timeout value is given in (platform dependent) ticks. The returned TimeoutReport indicates whether the receive was succesful (TR_Success), whether the timeout expired (TR_Expired), or whether the other side teard down the connection (TR_NoSuccess). The available function polls on the channel to check if a message is currently available. The `eom` (*e*nd *o*f *m*essages) function returns `True` iff it is sure that `available` will never become `True` anymore. To understand these functions, the following *model* of receive channels is handy:

> A receive channel consists of a message queue and is always in one of three states: the *idle*, *available*, or *eom* state, as shown in Figure 14.1.

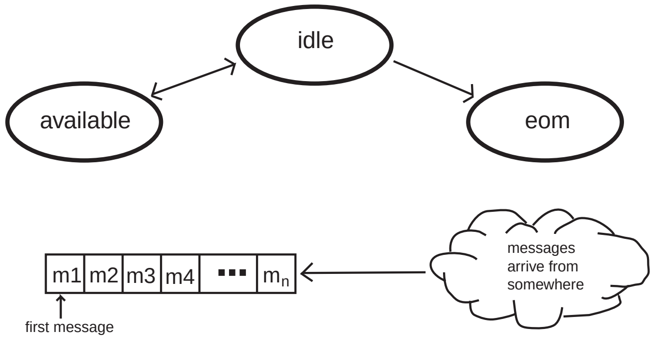


Figure 14.1: Model of a receive channel

> Iff the receive channel is in the *available* state, then there is a message in the message queue. Iff the channel is in the *idle* or *eom* state, then the message queue is empty. If a message arrives, then the state can change from *idle* to *available*, but if the channel is in the *eom* state, no message will arrive anymore. In this case the state does not change anymore. This happens when the other side tears down the connection.

To ease programming there are some specialisations of the functions above:

```
receive  ::      !*(*ch .a) !*env -> (! .a, !*(*ch .a),!*env)
         |  ChannelEnv env & Receive ch
nreceive :: !Int !*(*ch .a) !*env -> (![.a],!*(*ch .a),!*env)
         |  ChannelEnv env & Receive ch
```

The `receive` function receives one message from the channel and `(nreceive n)` receives `n` messages. These functions are only partially defined. If the receive channel state is *eom* or becomes *eom* while the function blocks, then the program will abort! These functions should only be used if it is certain that the specified amount of data will be received.

Example

The following function receives a byte sequence and compares it with a given string. If the strings are different, the program aborts.

```
serverReceive :: String TCP_RChannel *World
              ->       (TCP_RChannel,*World)
serverReceive expectedMessage rChannel world
    # (message,rChannel,world) = receive rChannel world
    | toString message<>expectedMessage
        = abort "received wrong message"
    | otherwise
        = (rChannel, world)
```

A special role is played by channels on which channels can be received, currently only the `TCP_Listener`. If a message is `available` on such a channel it might be possible that a subsequent `receive_MT` will return `Nothing`. This happens when a client tries to connect to a server, but disconnects again before the connection is accepted (received) from the server. Furtheron `eom` will never get `True` for this kind of channels.

On all other channels it holds that if `available` is True, receiving on that channel will not block and will return `Just` a message.

Before we discuss the basic operations on send channels, we introduce a model for these channels, see Figure 14.2.

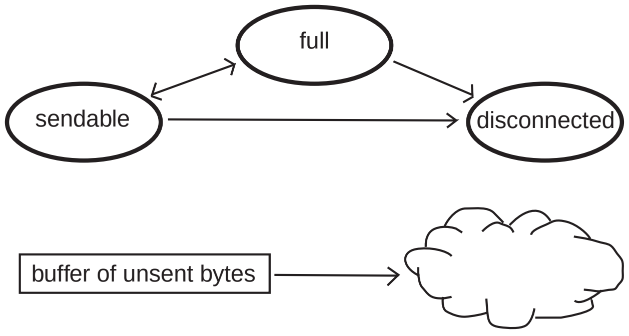


Figure 14.2: Model of a send channel

> A send channel has also three states: *sendable*, *full*, and *disconnected*. Furtheron there is a buffer of unsent bytes. If the channel is in the *disconnected* state, then no sending of data is possible. This happens when the remote side tears down the connection. The *disconnected* state is the final state (similar to the *eom* state for receive channels).

Sending on a channel can be blocking. Since this is not as obvious as in the receive case, here is a scenario where sending becomes blocking: let's assume a TCP connection, where data is sent only in one direction, from a "producer" program to a "consumer" program. Let's also assume that the producer sends much faster than the consumer, e.g. the producer sends one MByte per second but the consumer receives only one KByte per second, because he doesn't want to receive faster. It is obvious, that this does not work forever. At a certain point in time all buffers in the two participating computers and in the computers between them get full. The producer has to be stopped in his overproduction then, which means, that the producer's sending operation will block.

This explains why there are *sendable* and *full* states, and the message queue. Iff a channel is in the *full* state, then trying to send will not pump any data into the internet. Instead, the data will be queued locally in the channel's internal buffer. Iff the channel state changes from *full* to *sendable* then it's reasonable to flush this buffer. The buffer is a part of the Clean heap and does not have a limited size (apart from the usual memory limitations).

Now let's have a look at the basic operations on send channels:

```
class Send ch where
    send_MT       :: !(Maybe !Timeout) !.a   !*(*ch .a)  !*env
                  -> (!TimeoutReport, !Int,  !*(*ch .a), !*env)
```

```
                                          | ChannelEnv  env
    nsend_MT      :: !(Maybe !Timeout) ![.a] !*(*ch .a)  !*env
                  -> (!TimeoutReport, !Int,  !*(*ch .a), !*env)
                                          | ChannelEnv  env
    flushBuffer_MT:: !(Maybe !Timeout)       !*(*ch .a)  !*env
                  -> (!TimeoutReport, !Int,  !*(*ch .a), !*env)
                                          | ChannelEnv  env
    disconnected  ::                         !*(*ch .a)  !*env
                  -> (!Bool,                 !*(*ch .a), !*env)
                                          | ChannelEnv  env
    bufferSize    ::                         !*(*ch .a)
                  -> (!Int,                  !*(*ch .a))
    ...
```

The `send_MT` function sends a message and the `nsend_MT` function sends a list of messages. Both functions take a `(Maybe Timeout)` argument. If not all of the data could be sent within the timeout period, the rest will be queued in the send channels internal buffer. This internal buffer will not be sent automatically, it can be sent (flushed) with the `flushBuffer_MT` function. This function tries to send as much as possible from the internal buffer. The `Int` value that is returned by these three functions is the number of sent bytes. If the timeout expires before everything could be sent, then the returned `TimeoutReport` will be `TR_Expired`. If the send channel's state changed to *disconnected* during the send operation the `TimeoutReport` will be `TR_NoSucces`. Only if all data was sent, this value will be `TR_Success`. The size of the internal buffer in bytes is returned by the `bufferSize` function. The `disconnected` function is used to poll whether the channel is in the disconnected state.

The following functions are provided to avoid having to specify timeouts for sending. They are specialisations of the functions in the `Send` class.

```
send  :: !.a   !*(*ch .a) !*env -> (!*(*ch .a),!*env)
      |  ChannelEnv env & Send ch
nsend :: ![.a] !*(*ch .a) !*env -> (!*(*ch .a),!*env)
      |   ChannelEnv env & nsend_MT ch
```

Example

The following function sends the message "hello server".

```
clientSend :: TCP_SChannel *World -> (TCP_SChannel,*World)
clientSend sChannel world
    = send (toByteSeq "hello server") sChannel world
```

The `MaxSize` class allows to limit the size of the received byte sequences for receive channels. For further details see the `StdChannels` module.

Any object whose type is an instance of the `toString` class can be converted into a `ByteSeq`:

```
toByteSeq :: x -> ByteSeq | toString x
```

And vice versa: `ByteSeq`s can be converted into strings, because they are an instance of `toString`.

14.3.3 Tearing down a connection

Using TCP there are basically two ways to tear down a connection: *graceful* and *abortive* disconnects. The abortive disconnect happens when the `abortConnection` function is applied on a send channel. If the channel should be closed gracefully, then the `closeChannel_MT` function should be applied. Both functions are member of the `Send` class:

```
class Send ch where
    closeChannel_MT :: !(Maybe !Timeout) !*(*ch .a)  !*env
                    -> (!TimeoutReport,!Int,         !*env)
                                         | ChannelEnv  env
    abortConnection ::                   !*(*ch .a)  !*env
                    ->                               !*env
                                         | ChannelEnv  env
    ...
```

The `closeChannel_MT` function takes a `(Maybe Timeout)` argument because it tries to send the channel's internal buffer before closing the channel. This function has also a specialized version without a timeout:

```
closeChannel :: !*(*ch .a) !*env -> !*env | ChannelEnv env & Send ch
```

The internal buffer will not be sent when the `abortConnection` function is used. Furthermore, it is possible that data that is already sent will not reach the other side, when this function is used.

To close receive channels the `closeRChannel` function should be used:

```
class closeRChannel ch :: !*(*ch .a) !*env -> !*env | ChannelEnv env
```

This function will not block.

14.3.4 Putting it together

The example functions that were introduced in Section 14.3.1 and 14.3.2 are now assembled together into two programs: `client` and `server`. The client attempts to connect to the server and sends the message "hello server". The server checks if the received message matches that text. Both sides close the connection gracefully afterwards. It is assumed that the server runs on a machine called `martinpc.cs.kun.nl`.

Here is the client program:

```
module client

//  ****************************************************************************
//  Clean tutorial example program.
//
//  This program implements a minimal TCP session (client).
//  ****************************************************************************

import StdEnv, StdIO, StdTCP

clientConnect :: !*World -> (!TCP_DuplexChannel,!*World)
clientConnect world
    # (mbIPAddr,world)          = lookupIPAddress "martinpc.cs.kun.nl" world
    | isNothing mbIPAddr
```

```
        = abort "DNS lookup failed"
    # ipAddr                    = fromJust mbIPAddr
    # (tReport,mbDuplex,world)  = connectTCP_MT Nothing (ipAddr,2000) world
    | tReport<>TR_Success
        = abort "can't connect to port 2000"
    | otherwise
        # duplexChannel         = fromJust mbDuplex
        = (duplexChannel,world)

clientSend :: TCP_SChannel *World -> (TCP_SChannel,*World)
clientSend sChannel world = send (toByteSeq "hello server") sChannel world

Start world
    # ({sChannel,rChannel},world)   = clientConnect world
    # (sChannel,world)              = clientSend    sChannel world
    # world                         = closeChannel  sChannel world
    # world                         = closeRChannel rChannel world
    = world
```

And here is the server:

```
module server

//  ******************************************************************************
//  Clean tutorial example program.
//
//  This program implements a minimal TCP session (server).
//  ******************************************************************************

import StdEnv, StdIO, StdTCP

serverConnect :: !*World -> (!TCP_DuplexChannel,!*World)
serverConnect world
    # (ok,mbListener,world)                 = openTCP_Listener 2000 world
    | not ok
        = abort "can't open Listener on port 2000"
    | otherwise
        # listener                          = fromJust mbListener
        # ((_,duplexChannel),listener,world)= receive listener world
        # world                             = closeRChannel listener world
        = (duplexChannel,world)

serverReceive :: String TCP_RChannel *World -> (TCP_RChannel,*World)
serverReceive expectedMessage rChannel world
    # (message,rChannel,world)              = receive rChannel world
    | toString message<>expectedMessage     = abort "received wrong message"
    | otherwise                             = (rChannel, world)

Start world
    # ({sChannel,rChannel},world)   = serverConnect world
    # (rChannel,world)              = serverReceive "hello server" rChannel world
    # world                         = closeChannel  sChannel world
    # world                         = closeRChannel rChannel world
    = world
```

14.3.5 Multiplexing

The `selectChannel_MT` function allows you to determine on which channel in a set of channels operations can be used in a non blocking way. This is very useful to determine that particular channel in a set for which data has arrived first. It's signature is:

```
selectChannel_MT :: !(Maybe !Timeout)
```

```
                   !*r_channels  !*s_channels  !*World
                -> (![(!Int, !SelectResult)],
                   !*r_channels, !*s_channels, !*World)
                |  SelectReceive r_channels & SelectSend s_channels

::  SelectResult
    =   SR_Available
    |   SR_EOM
    |   SR_Sendable
    |   SR_Disconnected
```

The `r_channels` parameter can be a set of lists of receive channels. These lists can be combined with the `:^:` constructor (Appendix A.14). For instance, the following is defined:

```
::  *TCP_RChannels = TCP_RChannels [TCP_RChannel]
::  *TCP_Listeners = TCP_Listeners [TCP_Listener]
```

If `tcpRChs` is a list of `TCP_RChannels` and `tcpLists` is a list of `TCP_Listeners` then the following expression is a valid value for the `r_channels` parameter:

```
TCP_RChannels tcpRChs :^: TCP_Listeners tcpLists
```

The `:^:` constructor allows us to combine lists of channels of different types. Similar sets of send channels can be created for the `s_channels` argument. To specify an empty set of channels, it's also possible to pass `Void` as a value for `r_channels` or `s_channels`.

If the timeout expires, the function will return an empty list. Otherwise the `Int` part of each result pair will identify one of the channels out of `r_channels` or `s_channels` for which the `SelectResult` holds. Suppose for instance that both lists `tcpRChs` and `tcpLists` contain two elements. In the fragment:

```
  # r_channels = TCP_RChannels tcpRChs :^: TCP_Listeners tcpLists
  # ([(who,what):_],r_channels,_,world)
              = selectChannel_MT Nothing r_channels Void world
  # (TCP_RChannels tcpRChs :^: TCP_Listeners tcpLists)
              = r_channels
```

a set of four receive channels is passed via the `r_channels` parameter to the select function. So the `who` value will be inbetween zero and three. Let's assume here for simplicity that the list that is returned by the `selectChannel_MT` function only contains one element. If at first some data would have arrived for the first channel in the `tcpRChs` list, then `who` equals zero, for the other element of that list `who` equals one. Since the `TCP_Listeners` data constructor is the right argument of `:^:`, the `who` value would be two or three respectively, if at first a connection request would have arrived for one of the listeners. In all these cases the value for `what` would have been `SR_Available`. But if at first one of the `TCP_RChannels` would get into the *eom* state, then the result would be `SR_EOM`. Since only receive channels are passed to the `selectChannel_MT` function, the `SelectResult` result can only be `SR_Available` or `SR_EOM`. The `SR_Sendable` and the `SR_Disconnected` values can only be returned if some send channels are part of the `s_channels` parameter. As an example we pass a list `tcpSChs` of `TCP_SendChannels` to the `selectChannel_MT` function.

```
# ( [(who,what):_],_,TCP_SChannels tcpSChs,world )
     = selectChannel_MT Nothing Void (TCP_SChannels tcpSChs) world
```

The `who` value will identify one of the send channels in the list (`tcpSChs!!who`). Since no receive channels were passed to the `selectChannel_MT` function, the `what` value can only be `SR_Sendable` or `SR_Disconnected`. These values indicate that the channel is in the *sendable* or *disconnected* state respectively.

Of course it is also possible to pass receive channels and send channels to `selectChannel_MT`. The `SelectResult` result tells you whether a receive channel or a send channel is identified by the `Int` result. The numbering of channels starts with zero for receive channels and for send channels. If more than one channel could be selected by `selectChannel_MT`, then priority is given from left to right.

As an example program we'll discuss a server for a *chat* application. A chat client program (see Figure 14.3) will ask its user for a nickname and the address of the host where the server is running. After connecting to that server the client application will pop up a window with two edit controls. In the upper field the user can type some text which will be broadcasted via the server to all other people who are currently connected to the server. The client program is not a topic here, because it is a *GUI* program, but it is incorporated in the examples part of this tutorial.

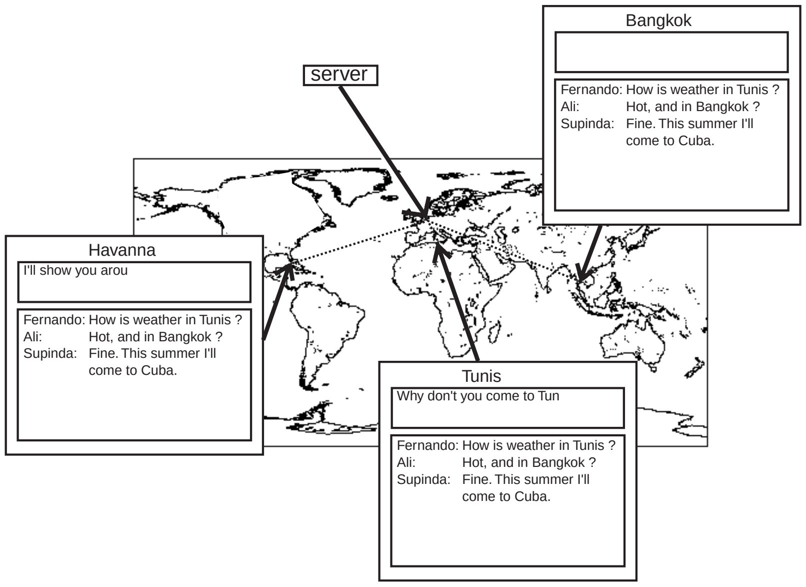

Bangkok
server
Fernando: How is weather in Tunis ?
Ali: Hot, and in Bangkok ?
Supinda: Fine. This summer I'll come to Cuba.
Havanna
I'll show you arou
Fernando: How is weather in Tunis ?
Ali: Hot, and in Bangkok ?
Supinda: Fine. This summer I'll come to Cuba.
Tunis
Why don't you come to Tun
Fernando: How is weather in Tunis ?
Ali: Hot, and in Bangkok ?
Supinda: Fine. This summer I'll come to Cuba.

Figure 14.3: Chatting via the internet

The server program first opens a listener on the port 2000. It uses a list of `ChanInfo` records. Each element of this list corresponds to one connection to a client. Apart from the two channels for the communication, the nickname for the connection is stored in a `ChanInfo` record. The set of receive channels that is passed to the `selectChannel_MT` function consists of the listener and the receive channels for each open connection. There are three cases to handle: a new connection is made, data has been sent, a connection is closed.

```
module chatServer

//  ********************************************************************************
//  Clean tutorial example program.
```

```
//
//  This program demonstrates the usage of the selectChannel_MT function
//  ****************************************************************************

import  StdEnv, StdTCP, StdIO

chatPort    :== 2000

::  *ChanInfo
    =   {   sndChan :: TCP_SChannel
        ,   rcvChan :: TCP_RChannel
        ,   nickname:: String
        }

Start :: !*World -> *World
Start world
    # (ok,mbListener,world) = openTCP_Listener chatPort world
    | not ok
        = abort ("chatServer: can't listen on port "+++toString chatPort)
    | otherwise
        # (console,world)   = stdio world
        = loop (fromJust mbListener) [] console world

loop :: !TCP_Listener ![ChanInfo] !*File !*World -> *World
loop listener channels console world
    # (sChans,rChans,nicknames)     = unzip3 channels
    # glue                          = TCP_Listeners [listener]
                                        :^:
                                      TCP_RChannels rChans
    # ([(who,what):_],glue,_,world) = selectChannel_MT Nothing glue Void world
    # (TCP_Listeners [listener:_]) :^: (TCP_RChannels rChans)
                                    = glue
    # channels                      = zip3 sChans rChans nicknames

    | who==0                    // Case 1: someone wants to join the chatroom
        # (tReport,mbNewMember,listener,world)
                            = receive_MT (Just 0) listener world
        | tReport<>TR_Success   // The potential new member changed his mind
            = loop listener channels console world
        # (_,{sChannel,rChannel})
                            = fromJust mbNewMember
        # (byteSeq,rChannel,world)
                            = receive rChannel world
        # nickname          = toString byteSeq
        # message           = "*** "+++nickname+++" joined the group."
        # console           = fwrites (message+++"\n") console
        # channel           = {sndChan=sChannel,rcvChan=rChannel,nickname=nickname}
        # channels          = [channel:channels]
        # (channels,world)  = broadcastString message channels [] world
        | nickname%(0,3)=="quit"
            = quit listener channels world
        // otherwise
            = loop listener channels console world

    | what==SR_Available    // Case 2: somebody has something to say
        # (channel=:{rcvChan, nickname},channels)
                            = selectList (who-1) channels
        # (byteSeq,rcvChan,world)
                            = receive rcvChan world
        # message           = toString byteSeq
        # channels          = channels++[{channel & rcvChan=rcvChan}]
        # (channels,world)  = broadcastString (nickname+++": "+++message)
                                channels [] world
        = loop listener channels console world

    | what==SR_EOM          // Case 3: somebody leaves the group
        # ({sndChan,rcvChan,nickname},channels)
```

```
                           = selectList (who-1) channels
        # message          = "*** "+++nickname+++" left the group"
        # console          = fwrites (message+++"\n") console
        # (channels,world) = broadcastString message channels [] world
        # world            = seq [closeChannel sndChan,closeRChannel rcvChan] world
        = loop listener channels console world

broadcastString :: !String ![ChanInfo] ![ChanInfo] !*World -> ([ChanInfo],!*World)
broadcastString string [] accu world
    = (reverse accu, world)
broadcastString string [channel=:{sndChan}:channels] accu world
    # (sndChan,world)      = send (toByteSeq string) sndChan world
    = broadcastString string channels [{channel & sndChan=sndChan}:accu] world

selectList :: !Int [.a] -> (!.a,![.a])
selectList n l
    # (left,[element:right]) = splitAt n l
    = (element, left++right)

quit listener channels world
    = closeChannels channels (closeRChannel listener world)

closeChannels [] world
    = world
closeChannels [{sndChan, rcvChan}: channels] world
    # world = closeChannel  sndChan world
    # world = closeRChannel rcvChan world
    = closeChannels channels world

unzip3 :: ![!ChanInfo] -> (![TCP_SChannel], ![TCP_RChannel], ![String])
unzip3 []
    = ([],[],[])
unzip3 [{sndChan, rcvChan, nickname}:t]
    # (a,b,c) = unzip3 t
    = ([sndChan:a], [rcvChan:b], [nickname:c])

zip3 :: ![TCP_SChannel] ![TCP_RChannel] ![String] -> [!ChanInfo]
zip3 [] [] []
    = []
zip3 [sndChan:a] [rcvChan:b] [nickname:c]
    = [{sndChan=sndChan, rcvChan=rcvChan, nickname=nickname} : zip3 a b c]
```

Server processes typically run in the background. Macintosh users should be aware that the MacOS was not designed as a multitasking operating system. Hence it is not a default property of programs to be able to run in the background. But the desired behaviour can be obtained if the 'Can Background' bit of the 'SIZE' resource of the executable is set. The program that allows to edit these resources of files is called 'ResEdit'.

14.3.6 More channels

Sometimes it is very handy if every single byte can be sent or received atomically. For this purpose we implemented the following two channels:

```
::  *TCP_SCharStream :== TCP_SCharStream_ Char
::  *TCP_RCharStream :== TCP_RCharStream_ Char
```

On a `TCP_S(R)CharStream` characters can be sent (received). The functions `toSCharStream` and `toRCharStream` convert `TCP_S(R)Channels` into `TCP_S(R)CharStreams`. The `nsend` and `nreceive` functions are handy to use with this kind of channels. The following function takes a `TCP_SChannel`, converts it into a character stream, and sends the characters “hello partner”.

```
s :: !TCP_SChannel !*World -> (!TCP_SCharStream, !*World)
s tcp_SChannel world
    # sCharStream = toSCharStream tcp_SChannel
    = nsend ['hello partner'] sCharStream world
```

If `rCharStream` is the corresponding `TCP_RCharStream` and `world` the `World`, then the other side could receive these thirteen characters with the following function application:

```
    nreceive 13 rCharStream world
```

Another kind of channels are the so called *string channels*. One handicap in using raw TCP is that the `ByteSeq` packets are not atomic. That means, that sending two byte sequences with sizes of for instance 10 and 14 bytes could be received on the other side as one byte sequence with a size of 24 bytes. String channels use their own protocol, which is built on top of the TCP protocol. If a string is sent on a string channel, then at first a representation of the size of that string is sent, and afterwards the contents. In this way it is possible to send empty strings as well as strings with sizes of several megabytes. On the corresponding receive channel these strings will be received as a whole. String channels should communicate only with other string channels, since these channels use their very own protocol. It is possible to use the `setMaxSize` function of the `MaxSize` class to limit the size of receivable strings. In this way an application can be protected against bogus programs, which claim to send strings that are to huge to fit in the memory of the used computer.

The definitions and instanciations for string channels can be found in the module `StdStringChannels` (Appendix A.36). We only show here the following:

```
::  *StringSChannel :== StringSChannel_ String
::  *StringRChannel :== StringRChannel_ String
```

The character streams and the string channels can be passed to the `selectChannel_MT` function. The naming convention is that the data constructors for the objects that are passed to the `selectChannel_MT` function end with an additional "s". For example, it is possible to pass `TCP_RCharStreams` and `StringRChannels` to the `selectChannel_MT` function:

```
  selectChannel_MT
          Nothing
          (TCP_RCharStreams [ch0,ch1] :^: StringRChannels [ch2,ch3])
          Void
          world
```

14.4 Non Blocking TCP in *GUI* Programs

As said before, *GUI* programs should not block for a "long" time. To explain how this condition can be solved we take a look at a very similar method of handling input, namely keyboard input. In *World* programs we simply can write something like:

```
  # (line, console) = freadline console
```

The program would block until the user presses the "Enter" key. In a *GUI* program we should not use the `freadline` function to get keyboard input. We have seen in Section 6.7.1 that we should specify a callback function as a part of the `WindowKeyboard` attribute. When the user presses a key, the runtime system will look up this callback function (which is stored in the `PSt`) and apply it to the key code and the `PSt`. The callback function performs the necessary actions on the program state as a reaction on the user's keyboard input.

With TCP connections similar things happen. One obvious difference is that a callback function for TCP is not specified as a part of a `WindowAttribute`. Instead it is specified as a `ReceiverFunction` which is a part of a receiver definition (see Chapter 10). To receive data we have to open a receiver. If via a TCP connection some data arrives, the `ReceiverFunction` of a receiver will be applied to this data. But not only arrival of data will cause a `ReceiverFunction` to be applied. In general, if a channel's state changes, the runtime system will generate a certain *event* on which a `ReceiverFunction` will be applied. There are two events for receive channels:

```
::  ReceiveMsg m = Received m | EOM
```

and two events for send channels:

```
::  SendEvent = Sendable | Disconnected
```

A `(ReceiveMsg m)` event informs the application that the message `m` has arrived. The `EOM` event informs about closure of the channel. The two `SendEvents` inform the application that the state of a send channel changed to sendable or disconnected.

Another underlying idea is the following: when receivers are opened, receive channels are *eaten* but send channels are not *eaten*!

Eating has to do with the uniqueness typing system of Clean. A function eats an object if the type of that object is uniquely attributed, and not returned.

Example

Let's have a look at the following two functions:

```
sum_eating :: *{Int} -> Int
sum_eating {[0]=a0,[1]=a1} = a0+a1

sum_not_eating :: *{Int} -> (Int, *{Int})
sum_not_eating a=:{[0]=a0,[1]=a1} = (a0+a1,a)
```

`sum_eating` eats its unique array argument, but `sum_not_eating` doesn't. As a consequence, the array can not be used anymore if it is passed to `sum_eating`. The following application would be rejected by the type system:

```
Start = sum_eating a + a.[0]
where
    a = {47,11}
```

Value `a` is not unique on the right hand of the equal sign, because it is used twice. It was tried to use `a` in the expression `a.[0]`, although `sum_eating` has eaten `a`. Fortunately we can calculate the desired sum with the following rule:

```
Start
    # (s1,a) = sum_not_eating a
    = s1 + a.[0]
where
    a = {47,11}
```

As a consequence, it is impossible to apply any function to a receive channel, after a receiver was opened for such a channel.

Let's examine, how we can open receivers for receive channels.

For each receive channel there is an algebraic receiver definition type that is used for opening a receiver. An object of such a type is passed to the `openReceiver` function, as shown in Chapter 10. For `TCP_RChannel`s there is a `TCP_Receiver`:

```
::  *TCP_Receiver ls pst
 =   TCP_Receiver
         Id TCP_RChannel
         (ReceiverFunction (ReceiveMsg ByteSeq) *(ls,pst))
         [ReceiverAttribute                     *(ls,pst)]

::  ReceiverFunction m st :== m -> st -> st
```

To open such a receiver, an `Id`, a `TCP_RChannel`, a `ReceiverFunction`, and `ReceiverAttribute`s have to be specified. The `Id` can be used to disable or close the receiver. If data has arrived on the channel, the `ReceiverFunction` will be called with the `Received` alternative. If the connection is teard down by the remote peer, then the `ReceiverFunction` will be called with the `EOM` alternative. Similar definitions are the `TCP_ListenerReceiver` (for `TCP_Listener`s), the `TCP_CharReceiver` (for receiving character by character) and the `StringChannelReceiver` (for `StringRChannel`s) (see Appendixes A.40 and A.36).

Example

We show a function `f` that opens a receiver for string channels. The `ReceiverFunction` `rcvFun` stores the received strings in a file which is also passed to `f`. The file is put in the local state of the receiver. When the `EOM` event occurs, the receiver will be closed by the runtime system after evaluation of the `EOM` alternative. It is not necessary to close such a receiver explicitly.

```
f :: StringRChannel *File (PSt .l) -> PSt .l
f stringChannel file pst
    # (rId,  pst)    = openRId pst
    # (error,pst)    = openReceiver
                           file
                           (StringChannelReceiver
                                rId
                                stringChannel rcvFun []
                           ) pst
    | error<>NoError = abort "an error occurred"
    | otherwise      = pst
where
    rcvFun (Received string) (file,pst)
        = (fwrites string file, pst)
    rcvFun EOM (file, pst)
        = (undef, snd (fclose file pst))
```

Now lets turn our attention to receivers for send channels. This kind of receiver is called `SendNotifier`. To open a `SendNotifier` we use the function `openSendNotifier` which does not eat its channel argument:

```
openSendNotifier :: .ls !(SendNotifier *(*ch .a) .ls (PSt .l))
                                                    !(PSt .l)
                    -> (!ErrorReport,!*(*ch .a),   ! PSt .l)
                    |  accSChannel ch & Send ch

::  SendNotifier sChannel ls pst
 =  SendNotifier
        sChannel
        (ReceiverFunction SendEvent *(ls,pst))
        [ReceiverAttribute          *(ls,pst)]
```

As you can see no `Id` is used. Indeed, an `Id` is not needed because the only thing we want to do with a `SendNotifier` is to close it. This is done automatically when we close the corresponding send channel.

Because the send channel will not be eaten, it has to be stored somewhere in the program state.

To send some data we can apply the following functions, which are defined in the `StdChannels` module. These functions simply call their `MT` counterpart with a timeout of zero ("`NB`" is a shorthand for "non blocking"):

```
send_NB        :: !.a !*(*ch .a) !*env -> (!*(*ch .a),!*env)
               | ChannelEnv env & Send ch
flushBuffer_NB ::     !*(*ch .a) !*env -> (!*(*ch .a),!*env)
               | ChannelEnv env & Send ch
```

Typically a *GUI* program uses the `send_NB` function to send data in a non blocking way. If not all of the data can be sent immediately because the send channel's state changed to full, then `send_NB` stores the unsent data in the send channel's internal buffer. When the flow conditions permit sending, the runtime system applies the `SendNotifier`'s `ReceiverFunction` on the `Sendable` event. The `ReceiverFunction` should then try to flush the internal buffer by using the `flushBuffer_NB` function.

As an example we will discuss a server program that accepts one connection and echoes the incoming data. To do this it opens two receivers in its initialisation function: for the receive channel a receiver that handles the incoming data, and for the send channel a send notifier. The send notifier allows the application to perform flow control. It should be possible that the remote side wants to receive the echoed data much slower than it wants to send. The echo server monitors the size of the internal buffer of the send channel. If this buffer size is greater than zero, then the receiver which receives the incoming data will be disabled. It will be enabled only when due to a `Sendable` event the buffer can be flushed again.

```
module echoServer

//  ********************************************************************************
//  Clean tutorial example program.
//
//  This program demonstrates the usage of functions for event driven TCP.
//  It listens on port 7, accepts a connection, and echoes the input.
//  ********************************************************************************
```

```
import StdEnv, StdTCP, StdIO

echoPort :== 7

::  *State
    =   { duplex:: TCP_DuplexChannel    // The channel
        , eom   :: Bool                 // EOM occurred on receive channel
        }

Start :: *World -> *World
Start world
    # (_,mbListener,world)          = openTCP_Listener echoPort world
    # ((_,duplex),listener,world)   = receive (fromJust mbListener) world
    # world                         = closeRChannel listener world
    = startIO NDI {duplex=duplex,eom=False} initialise [] world

/*  initialise - the function to initialise the PSt.
*/
initialise :: (PSt State) -> PSt State
initialise pst=:{ls=ls=:{duplex={rChannel,sChannel}},io}
    # (tcpRcvId,io)        = openId io
    # pst                  = {pst & ls={ls & duplex={rChannel=undef,sChannel=undef}}
                                  , io=io
                             }
//  Open a receiver for the receive channel
    # (error1,pst)         = openReceiver tcpRcvId
                                (TCP_Receiver tcpRcvId rChannel rcvFun []) pst
//  Open a receiver for the send channel
    # (error2,sChannel,pst)= openSendNotifier tcpRcvId
                                (SendNotifier sChannel sndFun []) pst
    | error1<>NoError || error2<>NoError
        = abort "error: can't open receiver"
    | otherwise
        = {pst & ls={ls & duplex={rChannel=undef,sChannel=sChannel}}}

/*  rcvFun - the callback function for the receive channels receiver.
*/
rcvFun :: (ReceiveMsg ByteSeq) (Id,PSt State) -> (Id,PSt State)
rcvFun (Received byteSeq) (tcpRcvId,pst=:{ls=ls=:{duplex=dc=:{sChannel}},io})
    # (sChannel,io)         = send_NB byteSeq sChannel io
    # (buffSize,sChannel)   = bufferSize sChannel
    # ls                    = {ls & duplex={dc & sChannel=sChannel}}
    | buffSize==0
        = (tcpRcvId,{pst & ls=ls,io=io})
    | otherwise             // Disable this receiver if the send channel is full
        # io                = disableReceivers [tcpRcvId] io
        = (tcpRcvId,{pst & ls=ls,io=io})
rcvFun EOM (tcpRcvId,pst=:{ls=ls=:{duplex=dc=:{sChannel}},io})
    # (buffSize,sChannel)   = bufferSize sChannel
    # pst                   = {pst & ls={ls & duplex = {dc & sChannel=sChannel}
                                            , eom    = True
                                        }
                                  , io=io
                              }
    | buffSize==0           // All data has been sent, so close program
        = (tcpRcvId,closeProcess (close pst))
    | otherwise
        = (tcpRcvId,pst)

/*  sndFun - the callback function for the send channels receiver.
*/
sndFun :: SendEvent (Id,PSt State) -> (Id,PSt State)
sndFun Sendable (tcpRcvId,pst=:{ls=ls=:{duplex=dc=:{sChannel},eom},io})
    # (sChannel,io)         = flushBuffer_NB sChannel io
    # (buffSize,sChannel)   = bufferSize sChannel
    # pst                   = {pst & ls={ls & duplex={dc & sChannel=sChannel}}
```

```
                                 , io=io
                             }
//  Reenable the receive channel's receiver if the send channel is still sendable.
    # pst                  = case (buffSize,eom) of
                                (0,False) -> appPIO (enableReceivers [tcpRcvId]) pst
                                (0,True ) -> close pst
                                _         -> pst
    = (tcpRcvId,pst)
sndFun Disconnected (ls,pst)
    = (ls,closeProcess pst)

close :: (PSt State) -> PSt State
close pst=:{ls=ls=:{duplex},io}
    # io    = closeChannel duplex.sChannel io
    = {pst & ls={ls & duplex={duplex & sChannel=undef}},io=io}
```

Is it forbidden to use blocking functions in *GUI* programs? This question arises because using blocking functions usually results in nicer programs. The answer is: of course it is not forbidden. The only problem that arises when using blocking functions is that the program will simply do not anything else while it blocks. In particular it will not update the contents of windows. The programmer has to estimate himself whether this is acceptable or not. Furthermore it depends on many factors whether the functions will block or not. For example there is no problem to send data in a blocking way to a program that is known to receive fast enough.

Appendix A

I/O library

A.1 StdBitmap

```
definition module StdBitmap


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdBitmap contains functions for reading bitmap files and drawing bitmaps.
//  ****************************************************************************


import  StdMaybe
from    StdFile     import FileSystem
from    osbitmap    import Bitmap
import  StdPicture
export  FileSystem World


openBitmap      :: !{#Char} !*env -> (!Maybe Bitmap,!*env)  | FileSystem env
/*  openBitmap reads in a bitmap from file.
    The String argument must be the file name of the bitmap.
    If the bitmap could be read, then (Just bitmap) is returned, otherwise Nothing
    is returned.
*/

getBitmapSize   :: !Bitmap -> Size
/*  getBitmapSize returns the size of the given bitmap.
    In case the bitmap is the result of an erroneous openBitmap, then the size is
    zero.
*/

resizeBitmap :: !Size !Bitmap -> Bitmap
/*  zooms or stretches a bitmap. The second argument is the size
    of the resulting bitmap
*/

instance Drawables Bitmap
/*  draw       bitmap
        draws the given bitmap with its left top at the current pen position.
    drawAt pos bitmap
        draws the given bitmap with its left top at the given pen position.
    undraw(At)
        equals unfill(At) the box {box_w=w,box_h=h} with {w,h} the size of the
        bitmap.
*/
```

A.2 StdChannels

```
definition module StdChannels

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdChannels defines operations on channels
//  ********************************************************************************

from    StdMaybe        import  Maybe
from    StdOverloaded   import  ==, toString
from    tcp             import  ChannelEnv

instance ChannelEnv World


//  ********************************************************************************
//  receive channels
//  ********************************************************************************

class Receive ch where
    receive_MT  ::  !(Maybe !Timeout)           !*(*ch .a)  !*env
                ->  (!TimeoutReport, !Maybe !.a,!*(*ch .a), !*env)
                                                | ChannelEnv  env
    receiveUpTo ::  !Int                        !*(*ch .a)  !*env
                ->  (![.a],                     !*(*ch .a), !*env)
                                                | ChannelEnv  env
    available   ::                              !*(*ch .a)  !*env
                ->  (!Bool,                     !*(*ch .a), !*env)
                                                | ChannelEnv  env
    eom         ::                              !*(*ch .a)  !*env
                ->  (!Bool,                     !*(*ch .a), !*env)
                                                | ChannelEnv  env
/*  receive_MT
        tries to receive on a channel. This function will block until data can be
        received, eom becomes true, or the timeout expires.
    receiveUpTo max ch env
        receives messages on a channel until available becomes False or max
        messages have been received.
    available
        polls on a channel whether some data is ready to be received. If the
        returned Boolean is True, then a following receive_MT will not block and
        return TR_Success.
    eom ("end of messages")
        polls on a channel whether data can't be received anymore.
*/

class closeRChannel ch  :: !*(*ch .a) !*env -> !*env    | ChannelEnv  env
//  Closes the channel


//  ********************************************************************************
//  send channels
//  ********************************************************************************

class Send ch where
    send_MT         ::  !(Maybe !Timeout) !.a   !*(*ch .a)  !*env
                    ->  (!TimeoutReport, !Int,  !*(*ch .a), !*env)
                                                | ChannelEnv  env
    nsend_MT        ::  !(Maybe !Timeout) ![.a] !*(*ch .a)  !*env
                    ->  (!TimeoutReport, !Int,  !*(*ch .a), !*env)
                                                | ChannelEnv  env
    flushBuffer_MT  ::  !(Maybe !Timeout)       !*(*ch .a)  !*env
                    ->  (!TimeoutReport, !Int,  !*(*ch .a), !*env)
                                                | ChannelEnv  env
    closeChannel_MT ::  !(Maybe !Timeout)       !*(*ch .a)  !*env
```

```
                   ->  (!TimeoutReport, !Int,              !*env)
                                               | ChannelEnv  env
    abortConnection ::                         !*(*ch .a)  !*env
                   ->                                      !*env
                                               | ChannelEnv  env
    disconnected    ::                         !*(*ch .a)  !*env
                   ->  (!Bool,                 !*(*ch .a), !*env)
                                               | ChannelEnv  env
    bufferSize      ::                         !*(*ch .a)
                   ->  (!Int,                  !*(*ch .a))
/*  send_MT mbTimeout a ch env
        adds the data a to the channels internal buffer and tries to send this
        buffer.
    nsend_MT mbTimeout l ch env
        adds the data l to the channels internal buffer and tries to send this
        buffer.
    flushBuffer_MT
        tries to send the channels internal buffer.
    closeSChannel_MT
        first tries to send the channels internal buffer and then closes the
        channel.
    abortConnection
        will cause an abortive disconnect (sent data can be lost).
    disconnected
        polls on a channel, whether data can't be sent anymore. If the returned
        Boolean is True, then a following send_MT will not block and return
        TR_NoSuccess.
    bufferSize
        returns the size of the channels internal buffer in bytes.

    The integer value that is returned by send_MT, nsend_MT, flushBuffer_MT, and
    closeSChannel_MT is the number of sent bytes.
*/


//  ******************************************************************************
//  miscellaneous
//  ******************************************************************************

class MaxSize ch where
    setMaxSize      ::  !Int !*(*ch .a) -> *(*ch .a)
    getMaxSize      ::       !*(*ch .a) -> (!Int, !*(*ch .a))
    clearMaxSize    ::       !*(*ch .a) -> *(*ch .a)
//  Set, get, or clear the maximum size of the data that can be received

::  DuplexChannel sChannel rChannel a
    =   {   sChannel:: sChannel a
        ,   rChannel:: rChannel a
        }

::  TimeoutReport
    =   TR_Expired
    |   TR_Success
    |   TR_NoSuccess

::  Timeout         :== Int     // timeout in ticks

::  ReceiveMsg m    =   Received m
                    |   EOM
                    //  Receiving "EOM" automatically closes the receiver
::  SendEvent       =   Sendable
                    |   Disconnected
                    //  Receiving "Disconnected" automatically closes the receiver

instance ==         TimeoutReport
instance toString   TimeoutReport
```

```
//  ******************************************************************************
//  derived functions
//  ******************************************************************************

nreceive_MT :: !(Maybe !Timeout) !Int !*(*ch .a) !*env
            -> (!TimeoutReport, ![.a],!*(*ch .a),!*env)
            |  Receive ch & ChannelEnv env
/*  nreceive_MT mbTimeout n ch env
        tries to call receive_MT n times. If the result is (tReport, l, ch2, env2),
        then the following holds:
            tReport==TR_Succes      <=> length l==n
            tReport==TR_NoSuccess    => length l<n
*/

/*  The following two receive functions call their "_MT" counterpart with no
    timeout. If the data can't be received because eom became True the function will
    abort.
*/
receive         ::          !*(*ch .a)  !*env
                -> (!.a,    !*(*ch .a), !*env)
                |   ChannelEnv env & Receive ch
nreceive        ::  !Int    !*(*ch .a)  !*env
                ->  (![.a], !*(*ch .a), !*env)
                |   ChannelEnv env & Receive ch

/*  The following three send functions call their "_MT" counterpart with no timeout.
*/
send            :: !.a          !*(*ch .a)  !*env
                ->             (!*(*ch .a), !*env)
                |   ChannelEnv env & Send ch
nsend           :: ![.a]        !*(*ch .a)  !*env
                ->             (!*(*ch .a), !*env)
                |   ChannelEnv env & nsend_MT ch
closeChannel    ::              !*(*ch .a)  !*env
                ->                          !*env
                |   ChannelEnv env & Send ch

/*  The following two send functions call their "_MT" counterpart with timeout == 0.
    "NB" is a shorthand for "non blocking"
*/
send_NB         :: !.a          !*(*ch .a)  !*env
                ->             (!*(*ch .a), !*env)
                |   ChannelEnv env & Send ch
flushBuffer_NB  ::              !*(*ch .a)  !*env
                ->             (!*(*ch .a), !*env)
                |   ChannelEnv env & Send ch
```

A.3 StdClipboard

```
definition module StdClipboard


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdClipboard specifies all functions on the clipboard.
//  ********************************************************************************


import  StdMaybe
from    iostate import  PSt, IOSt


//  Clipboard data items:

::  ClipboardItem

class Clipboard item where
    toClipboard     :: !item            -> ClipboardItem
    fromClipboard   :: !ClipboardItem   -> Maybe item
/*  toClipboard
        makes an item transferable to the clipboard.
    fromClipboard
        attempts to retrieve an item of the instance type from the clipboard item.
        If this fails, the result is Nothing, otherwise it is (Just item).
*/

instance Clipboard {#Char}


//  Access to the current content of the clipboard:

setClipboard :: ![ClipboardItem] !(PSt .l) -> PSt .l
getClipboard :: !(PSt .l) -> (![ClipboardItem],!PSt .l)
/*  setClipboard
        replaces the current content of the clipboard with the argument list.
        Of the list only the first occurence of a ClipboardItem of the same type
        will be stored in the clipboard.
        Note that setClipboard [] erases the clipboard.
    getClipboard
        gets the current content of the clipboard without changing the content.
*/


clipboardHasChanged :: !(PSt .l) -> (!Bool,!PSt .l)
/*  clipboardHasChanged holds if the current content of the clipboard is different
    from the last access to the clipboard.
*/
```

A.4 StdControl

```
definition module StdControl


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdControl specifies all control operations.
//  ********************************************************************************


import  StdControlClass


controlSize             :: !(cdef .ls (PSt .l)) !Bool
                           !(Maybe (Int,Int)) !(Maybe (Int,Int)) !(Maybe (Int,Int))
                           !(PSt .l)
                  -> (!Size,!PSt .l) | Controls cdef
/*  controlSize calculates the size of the given control definition as it would be
        opened as an element of a window/dialog.
    The Boolean argument determines whether a window (True) or a dialog (False) is
        intended.
    The Maybe arguments are the prefered horizontal margins, vertical margins, and
        item spaces (see also the (Window/Control)(H/V)Margin and
        (Window/Control)ItemSpace attributes). If Nothing is specified, their
        default values with respect to the window/dialog are used.
*/


/*  Functions that change the set of controls in windows/dialogues.
*/

openControls            :: !Id .ls (cdef .ls (PSt .l)) !(PSt .l)
                                         -> (!ErrorReport,!PSt .l)
                                         |  Controls cdef
openCompoundControls    :: !Id .ls (cdef .ls (PSt .l)) !(PSt .l)
                                         -> (!ErrorReport,!PSt .l)
                                         |  Controls cdef
openPopUpControlItems   :: !Id !Index ![PopUpControlItem (PSt .l)]
                                         !(IOSt .l) -> IOSt .l
/*  openControls
        adds the given controls argument to the indicated window or dialog.
    openCompoundControls
        adds the given controls argument to the indicated compound control.
    openPopUpControlItems
        adds the PopUpControlItems to the indicated PopUpControl behind the item at
        the given index position (counted from 1).
    The window/dialog is not resized.
    These functions have no effect in case the indicated window/dialog/compound
    control could not be found (ErrorUnknownObject) or if controls are opened with
    duplicate Ids (ErrorIdsInUse).
*/

closeControls           :: !Id [Id] !Bool   !(IOSt .l) -> IOSt .l
/*  closeControls removes the indicated controls (second argument) from the
    indicated window (first argument) and recalculates the layout iff the Boolean
    argument is True.
*/

closeAllControls        :: !Id !(IOSt .l) -> IOSt .l
/*  closeAllControls removes all controls from the indicated window.
*/

closePopUpControlItems  :: !Id ![Index] !(IOSt .l) -> IOSt .l
/*  closePopUpControlItems closes PopUpControlItems by their Index position of the
    indicated PopUpControl.
    If the currently checked element of a PopUpControl is closed, the first
    remaining element of that PopUpControl will be checked.
*/


setControlPos           :: !Id ![(Id,ItemPos)]  !(IOSt .l) -> (!Bool,!IOSt .l)
/*  setControlPos changes the current layout position of the indicated controls to
    their new positions.
    If there are relatively laynout controls, then their layout also changes. The
    window is not resized.
    The Boolean result is False iff the window is unknown.
*/


/*  Functions that change the state of controls.
    Only those Id arguments that refer to controls within the same interactive
    process are used to change the corresponding controls.
*/

showControls            :: ![Id]                        !(IOSt .l) -> IOSt .l
showControl             :: ! Id                         !(IOSt .l) -> IOSt .l
hideControls            :: ![Id]                        !(IOSt .l) -> IOSt .l
hideControl             :: ! Id                         !(IOSt .l) -> IOSt .l
/*  (show/hide)Control(s) makes the indicated control(s) visible/invisible.
    Hiding a control overrides the visibility of its elements, which become
        invisible.
    Showing a hidden control re-establishes the visibility state of its elements.
*/

enableControls          :: ![Id]                        !(IOSt .l) -> IOSt .l
enableControl           :: ! Id                         !(IOSt .l) -> IOSt .l
disableControls         :: ![Id]                        !(IOSt .l) -> IOSt .l
disableControl          :: ! Id                         !(IOSt .l) -> IOSt .l
/*  (en/dis)ableControl(s) (en/dis)ables the indicated control(s).
    Disabling a control overrides the SelectStates of its elements, which become
        unselectable.
    Enabling a disabled control re-establishes the SelectStates of its elements.
*/

markCheckControlItems   :: !Id ![Index]                 !(IOSt .l) -> IOSt .l
unmarkCheckControlItems :: !Id ![Index]                 !(IOSt .l) -> IOSt .l
/*  (unm/m)arkCheckControlItems unmarks/marks the indicated check items of the given
    CheckControl. Indices range from 1 to the number of check items. Illegal indices
    are ignored.
*/

selectRadioControlItem  :: !Id  !Index                  !(IOSt .l) -> IOSt .l
/*  selectRadioControlItem marks the indicated radio item of a RadioControl, causing
    the mark of the previously marked radio item to disappear. The item is given by
    the Id of the RadioControl and its index position (counted from 1).
*/

selectPopUpControlItem  :: !Id  !Index                  !(IOSt .l) -> IOSt .l
/*  selectPopUpControlItem marks the indicated popup item of a PopUpControl, causing
    the mark of the previously marked popup item to disappear. The item is given by
    the Id of the PopUpControl and its index position (counted from 1).
*/

moveControlViewFrame    :: !Id Vector2                  !(IOSt .l) -> IOSt .l
/*  moveControlViewFrame moves the orientation of the CompoundControl over the given
    vector, and updates the control if necessary. The control frame is not moved
    outside the ViewDomain of the control. MoveControlViewFrame has no effect if the
    indicated control has no ControlDomain attribute.
*/

setControlViewDomain    :: !Id ViewDomain               !(IOSt .l) -> IOSt .l
/*  setControlViewDomain sets the view domain of the indicated CompoundControl as
```

```
    given. The control view frame is moved such that a maximum portion of the view
    domain is visible. The control is not resized.
    In case of unknown Ids, or non CompoundControls, setControlViewDomain has no
    effect.
*/

setControlScrollFunction:: !Id Direction ScrollFunction !(IOSt .l) -> IOSt .l
/*  setControlScrollFunction set the ScrollFunction of the indicated CompoundControl
    in the given Direction if it has one.
    In all other cases, setControlScrollFunction has no effect.
*/

setControlTexts         :: ![(Id,String)]               !(IOSt .l) -> IOSt .l
setControlText          :: !Id !String                  !(IOSt .l) -> IOSt .l
/*  setControlText(s) sets the text of the indicated (Text/Edit/Button)Control(s).
    If the indicated control is a (Text/Button)Control, then AltKey are interpreted
    by the system.
    If the indicated control is an EditControl, then the text is taken as it is.
*/

setEditControlCursor    :: !Id !Int                     !(IOSt .l) -> IOSt .l
/*  setEditControlCursor sets the cursor at position @2 of the current content of
    the EditControl.
    In case @2<0, then the cursor is set at the start of the current content.
    In case @2>size content, then the cursor is set at the end of the current
    content.
*/

setControlLooks         :: ![(Id, Bool,(Bool,Look))]    !(IOSt .l) -> IOSt .l
setControlLook          ::   !Id !Bool (Bool,Look)      !(IOSt .l) -> IOSt .l
/*  setControlLook(s) sets the (render,look) attribute of the indicated
    (Custom(Button)/Compound)Control(s). If this concerns a transparant
    CompoundControl then it becomes non-transparant.
    An indicated control is only redrawn if the first Boolean is True.
*/

setSliderStates         :: ![(Id, IdFun SliderState)]   !(IOSt .l) -> IOSt .l
setSliderState          ::   !Id (IdFun SliderState)    !(IOSt .l) -> IOSt .l
setSliderThumbs         :: ![(Id,Int)]                  !(IOSt .l) -> IOSt .l
setSliderThumb          ::   !Id Int                    !(IOSt .l) -> IOSt .l
/*  setSliderState(s)
        applies the function to the current SliderState of the indicated
        SliderControl(s) and redraws the settings if necessary.
    setSliderThumb(s)
        sets the new thumb value of the indicated SliderControl(s) and redraws the
        settings if necessary.
*/

appControlPicture       :: !Id !.(IdFun *Picture)       !(IOSt .l) -> IOSt .l
accControlPicture       :: !Id !.(St *Picture .x)       !(IOSt .l)
                                        -> (!Maybe .x,!IOSt .l)
/*  (app/acc)ControlPicture applies the given drawing function to the Picture of
    the indicated (Custom(Button)/Compound)Control. If the CompoundControl is
    transparant, or the indicated control could not be found then this operation
    has no effect. In that case, accControlPicture also returns Nothing.
*/


updateControl           :: !Id !(Maybe ViewFrame)       !(IOSt .l) -> IOSt .l
/*  updateControl applies the Look attribute function of the indicated
    (Compound/Custom(Button))Control.
    The Look attribute function is applied to the following arguments:
    The current SelectState of the control, and
    the UpdateState argument
        {oldFrame=viewframe,newFrame=viewframe,updArea=[frame]}
    where viewframe is the current ViewFrame of the control;
    and frame depends on the optional ViewFrame argument:
        in case of (Just rectangle):
            the intersection of viewframe and rectangle.
        in case of Nothing:
            viewframe.
    updateControl has no effect in case of unknown controls, or if the indicated
    control is not a (Compound/Custom(Button))Control, or the optional viewframe
    argument is an empty rectangle.
*/


/*  Access functions on WState. To read the state of a control, a WState is
    required which can be obtained by the getWindow function. The WState value
    represents the state of a window or dialogue at that particular moment.
*/

::  WState

getWindow               :: !Id !(IOSt .l) -> (!Maybe WState, !IOSt .l)
getParentWindow         :: !Id !(IOSt .l) -> (!Maybe WState, !IOSt .l)
/*  getWindow returns a read-only WState for the indicated window.
        In case the indicated window does not exist Nothing is returned.
    getParentWindow returns a read-only WState for the parent window/dialogue
        of the indicated control. In case the Id does not correspond with a
        control, Nothing is returned.
*/

getControlTypes         ::      !WState -> [(ControlType,Maybe Id)]
getCompoundTypes        :: !Id  !WState -> [(ControlType,Maybe Id)]
/*  getControlTypes
        yields the list of ControlTypes of the component controls of this window.
    getCompoundTypes
        yields the list of ControlTypes of the component controls of this
        CompoundControl.
    For both functions (Just id) is yielded if the component control has a
    (ControlId id) attribute, and Nothing otherwise. Component controls are not
    collected recursively through CompoundControls.
    If the indicated CompoundControl is not a CompoundControl, then [] is yielded.
*/

/*  Functions that return the current state of controls.
    For each access there is one singular and one plural version. In case of the
    plural version the result list is of equal length as the argument Id list. Each
    result list element corresponds in order with the argument Id list.
    In both versions the first Boolean result is False in case of invalid Ids (if so
    dummy values are returned - see comment).
    Important: controls with no ControlId attribute, or illegal ids, can not be
    found in the WState!
*/


getControlLayouts       :: ![Id] !WState -> [(Bool,(Maybe ItemPos,Vector2))]
getControlLayout        :: ! Id  !WState ->  (Bool,(Maybe ItemPos,Vector2))
/*  getControlLayout(s) yields (Just ControlPos) if the indicated control had a
    ControlPos attribute and Nothing otherwise. The Vector2 offset is the exact
    current location of the indicated control (LeftTop,OffsetVector offset).
*/

getControlViewSizes     :: ![Id] !WState -> [(Bool,Size)]
getControlViewSize      :: ! Id  !WState ->  (Bool,Size)
getControlOuterSizes    :: ![Id] !WState -> [(Bool,Size)]
getControlOuterSize     :: ! Id  !WState ->  (Bool,Size)
/*  getControlViewSize(s) yields the current ViewFrame size of the indicated
        control. Note that this is the exact size of the control for any control
        other than the CompoundControl. In case of unknown Ids zero is returned.
    getControlOuterSize(s) yields the current ControlOuterSize of the indicated
        control. Note that this is the exact size of the control. In case of unknown
        Ids zero is returned.
```

```
*/

getControlSelectStates  :: ![Id] !WState -> [(Bool,SelectState)]
getControlSelectState   :: ! Id  !WState ->  (Bool,SelectState)
/*  getControlSelectState(s) yields the current SelectState of the indicated
    control. In case of unknown Ids Able is returned.
*/

getControlShowStates    :: ![Id] !WState -> [(Bool,Bool)]
getControlShowState     :: ! Id  !WState ->  (Bool,Bool)
/*  getControlShowState(s) yields True if the indicated control is visible, and
    False otherwise. The latter is also returned in case of unknown Ids.
*/

getControlTexts         :: ![Id] !WState -> [(Bool,Maybe String)]
getControlText          :: ! Id  !WState ->  (Bool,Maybe String)
/*  getControlText(s) yields (Just text) of the indicated (PopUp/Text/Edit/Button)
    Control. If the control is not such a control, then Nothing is yielded.
*/

getControlNrLines       :: ![Id] !WState -> [(Bool,Maybe NrLines)]
getControlNrLine        :: ! Id  !WState ->  (Bool,Maybe NrLines)
/*  getControlNrLine(s) yields (Just nrlines) of the indicated EditControl.
    If the control is not such a control, then Nothing is yielded.
*/

getControlLooks         :: ![Id] !WState -> [(Bool,Maybe (Bool,Look))]
getControlLook          :: ! Id  !WState ->  (Bool,Maybe (Bool,Look))
/*  getControlLook(s) yields the (render/look) of the indicated
    (Custom/CustomButton/Compound)Control. If the control is not such a control, or
    is a transparant CompoundControl, then Nothing is yielded.
*/

getControlMinimumSizes  :: ![Id] !WState -> [(Bool,Maybe Size)]
getControlMinimumSize   :: ! Id  !WState ->  (Bool,Maybe Size)
/*  getControlMinimumSize(s) yields (Just minimumsize) if the indicated control had
    a ControlMinimumSize attribute and Nothing otherwise.
*/

getControlResizes       :: ![Id] !WState -> [(Bool,Maybe ControlResizeFunction)]
getControlResize        :: ! Id  !WState ->  (Bool,Maybe ControlResizeFunction)
/*  getControlResize(s) yields (Just resizefunction) if the indicated control had a
    ControlResize attribute and Nothing otherwise.
*/

getRadioControlItems    :: ![Id] !WState -> [(Bool,Maybe [String])]
getRadioControlItem     :: ! Id  !WState ->  (Bool,Maybe [String])
/*  getRadioControlItem(s) yields the TextLines of the items of the indicated
    RadioControl. If the control is not such a control, then Nothing is yielded.
*/

getRadioControlSelections:: ![Id] !WState -> [(Bool,Maybe Index)]
getRadioControlSelection :: ! Id  !WState ->  (Bool,Maybe Index)
/*  getRadioControlSelection(s) yields the index of the selected radio item of the
    indicated RadioControl. If the control is not such a control, then Nothing is
    yielded.
*/

getCheckControlItems    :: ![Id] !WState -> [(Bool,Maybe [String])]
getCheckControlItem     :: ! Id  !WState ->  (Bool,Maybe [String])
/*  getCheckControlItem(s) yields the TextLines of the items of the indicated
    CheckControl. If the control is not such a control, then Nothing is yielded.
*/

getCheckControlSelections:: ![Id] !WState -> [(Bool,Maybe [Index])]
getCheckControlSelection :: ! Id  !WState ->  (Bool,Maybe [Index])
/*  getCheckControlSelection(s) yields the indices of the selected checkitems of the
```

```
    indicated CheckControl. If the control is not such a control, then Nothing is
    yielded.
*/

getPopUpControlItems    :: ![Id] !WState -> [(Bool,Maybe [String])]
getPopUpControlItem     :: ! Id  !WState ->  (Bool,Maybe [String])
/*  getPopUpControlItem(s) yields the TextLines of the items of the indicated
    PopUpControl. If the control is not such a control, then Nothing is yielded.
*/

getPopUpControlSelections:: ![Id] !WState -> [(Bool,Maybe Index)]
getPopUpControlSelection :: ! Id  !WState ->  (Bool,Maybe Index)
/*  getPopUpControlSelection(s) yields the Index of the indicated PopUpControl.
    If the control is not such a control, then Nothing is yielded.
*/

getSliderDirections     :: ![Id] !WState -> [(Bool,Maybe Direction)]
getSliderDirection      :: ! Id  !WState ->  (Bool,Maybe Direction)
/*  getSliderDirection(s) yields (Just Direction) of the indicated SliderControl.
    If the control is not such a control, then Nothing is yielded.
*/

getSliderStates         :: ![Id] !WState -> [(Bool,Maybe SliderState)]
getSliderState          :: ! Id  !WState ->  (Bool,Maybe SliderState)
/*  getSliderState(s) yields (Just SliderState) of the indicated SliderControl.
    If the control is not such a control, then Nothing is yielded.
*/

getControlViewFrames    :: ![Id] !WState -> [(Bool,Maybe ViewFrame)]
getControlViewFrame     :: ! Id  !WState ->  (Bool,Maybe ViewFrame)
/*  getControlViewFrame(s) yields (Just ViewFrame) of the indicated CompoundControl.
    If the control is not such a control, then Nothing is yielded.
*/

getControlViewDomains   :: ![Id] !WState -> [(Bool,Maybe ViewDomain)]
getControlViewDomain    :: ! Id  !WState ->  (Bool,Maybe ViewDomain)
/*  getControlViewDomain(s) yields (Just ViewDomain) of the indicated
    CompoundControl. If the control is not such a control, then Nothing is yielded.
*/

getControlScrollFunctions
                        :: ![Id] !WState
                        -> [(Bool,Maybe ((Direction,Maybe ScrollFunction)
                                        ,(Direction,Maybe ScrollFunction)
                                        ))]
getControlScrollFunction:: ! Id  !WState
                        ->  (Bool,Maybe ((Direction,Maybe ScrollFunction)
                                        ,(Direction,Maybe ScrollFunction)
                                        ))
/*  getControlScrollFunction(s) yields the ScrollFunctions of the indicated
    CompoundControl. If the control is not such a control, then Nothing is yielded.
*/

getControlItemSpaces    :: ![Id] !WState -> [(Bool,Maybe (Int,Int))]
getControlItemSpace     :: ! Id  !WState ->  (Bool,Maybe (Int,Int))
/*  getControlItemSpace(s) yields (Just (horizontal space,vertical space)) of the
    indicated (Compound/Layout)Control. If the control is not such a control, then
    Nothing is yielded.
*/

getControlMargins       :: ![Id] !WState -> [(Bool,Maybe ((Int,Int),(Int,Int)))]
getControlMargin        :: ! Id  !WState ->  (Bool,Maybe ((Int,Int),(Int,Int)))
/*  getControlMargins yields (Just (ControlHMargin,ControlVMargin)) of the
    indicated (Compound/Layout)Control. If the control is not such a control, then
    Nothing is yielded.
*/
```

A.5 StdControlAttribute

```
definition module StdControlAttribute


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdControlAttribute specifies which ControlAttributes are valid for each of the
//  standard controls.
//  Basic comparison operations and retrieval functions are also included.
//  ****************************************************************************


import StdControlDef


/*  The following functions specify the valid attributes for each standard control.
*/

isValidButtonControlAttribute :: !(ControlAttribute .st) -> Bool
/*  ButtonControl       (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       . |
    ControlFunction     y | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction y | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                  | ControlWidth        y |
*/

isValidCheckControlAttribute :: !(ControlAttribute .st) -> Bool
/*  CheckControl        (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       . |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                  | ControlWidth        . |
*/

isValidCompoundControlAttribute :: !(ControlAttribute .st) -> Bool
/*  CompoundControl     (y = valid, . = invalid)
    ControlActivate     y | ControlKeyboard     y | ControlPos          y |
    ControlDeactivate   y | ControlLook         y | ControlResize       y |
    ControlFunction     . | ControlMinimumSize  y | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      y | ControlMouse        y | ControlViewDomain   y |
    ControlHScroll      y | ControlOrigin       y | ControlViewSize     y |
    ControlId           y | ControlOuterSize    y | ControlVMargin      y |
    ControlItemSpace    y | ControlPen          y | ControlVScroll      y |
                                                  | ControlWidth        . |
*/

isValidCustomButtonControlAttribute :: !(ControlAttribute .st) -> Bool
/*  CustomButtonControl (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       y |
    ControlFunction     y | ControlMinimumSize  y | ControlSelectState  y |
    ControlHide         y | ControlModsFunction y | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          y | ControlVScroll      . |
                                                  | ControlWidth        . |
*/

isValidCustomControlAttribute :: !(ControlAttribute .st) -> Bool
/*  CustomControl       (y = valid, . = invalid)
    ControlActivate     y | ControlKeyboard     y | ControlPos          y |
    ControlDeactivate   y | ControlLook         . | ControlResize       y |
    ControlFunction     . | ControlMinimumSize  y | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        y | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          y | ControlVScroll      . |
                                                  | ControlWidth        . |
*/

isValidEditControlAttribute :: !(ControlAttribute .st) -> Bool
/*  EditControl         (y = valid, . = invalid)
    ControlActivate     y | ControlKeyboard     y | ControlPos          y |
    ControlDeactivate   y | ControlLook         . | ControlResize       y |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                  | ControlWidth        . |
*/

isValidLayoutControlAttribute :: !(ControlAttribute .st) -> Bool
/*  LayoutControl       (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       y |
    ControlFunction     . | ControlMinimumSize  y | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          . |
    ControlHMargin      y | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     y |
    ControlId           y | ControlOuterSize    y | ControlVMargin      y |
    ControlItemSpace    y | ControlPen          . | ControlVScroll      . |
                                                  | ControlWidth        . |
*/

isValidPopUpControlAttribute :: !(ControlAttribute .st) -> Bool
/*  PopUpControl        (y = valid, . = invalid)
    ControlActivate     y | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   y | ControlLook         . | ControlResize       . |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                  | ControlWidth        y |
*/

isValidRadioControlAttribute :: !(ControlAttribute .st) -> Bool
/*  RadioControl        (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       . |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                | ControlWidth        . |
*/

isValidSliderControlAttribute :: !(ControlAttribute .st) -> Bool
/*  SliderControl       (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       y |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  y |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                | ControlWidth        . |
*/

isValidTextControlAttribute :: !(ControlAttribute .st) -> Bool
/*  TextControl         (y = valid, . = invalid)
    ControlActivate     . | ControlKeyboard     . | ControlPos          y |
    ControlDeactivate   . | ControlLook         . | ControlResize       . |
    ControlFunction     . | ControlMinimumSize  . | ControlSelectState  . |
    ControlHide         y | ControlModsFunction . | ControlTip          y |
    ControlHMargin      . | ControlMouse        . | ControlViewDomain   . |
    ControlHScroll      . | ControlOrigin       . | ControlViewSize     . |
    ControlId           y | ControlOuterSize    . | ControlVMargin      . |
    ControlItemSpace    . | ControlPen          . | ControlVScroll      . |
                                                | ControlWidth        y |
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isControlActivate       :: !(ControlAttribute .st) -> Bool
isControlDeactivate     :: !(ControlAttribute .st) -> Bool
isControlFunction       :: !(ControlAttribute .st) -> Bool
isControlHide           :: !(ControlAttribute .st) -> Bool
isControlHMargin        :: !(ControlAttribute .st) -> Bool
isControlHScroll        :: !(ControlAttribute .st) -> Bool
isControlId             :: !(ControlAttribute .st) -> Bool
isControlItemSpace      :: !(ControlAttribute .st) -> Bool
isControlKeyboard       :: !(ControlAttribute .st) -> Bool
isControlLook           :: !(ControlAttribute .st) -> Bool
isControlMinimumSize    :: !(ControlAttribute .st) -> Bool
isControlModsFunction   :: !(ControlAttribute .st) -> Bool
isControlMouse          :: !(ControlAttribute .st) -> Bool
isControlOrigin         :: !(ControlAttribute .st) -> Bool
isControlOuterSize      :: !(ControlAttribute .st) -> Bool
isControlPen            :: !(ControlAttribute .st) -> Bool
isControlPos            :: !(ControlAttribute .st) -> Bool
isControlResize         :: !(ControlAttribute .st) -> Bool
isControlSelectState    :: !(ControlAttribute .st) -> Bool
isControlTip            :: !(ControlAttribute .st) -> Bool
isControlViewDomain     :: !(ControlAttribute .st) -> Bool
isControlViewSize       :: !(ControlAttribute .st) -> Bool
isControlVMargin        :: !(ControlAttribute .st) -> Bool
isControlVScroll        :: !(ControlAttribute .st) -> Bool
isControlWidth          :: !(ControlAttribute .st) -> Bool


/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getControlActivateFun   :: !(ControlAttribute .st) -> IdFun .st
getControlDeactivateFun :: !(ControlAttribute .st) -> IdFun .st
getControlFun           :: !(ControlAttribute .st) -> IdFun .st
```

```
getControlHMarginAt    :: !(ControlAttribute .st) -> (Int,Int)
getControlHScrollFun    :: !(ControlAttribute .st) -> ScrollFunction
getControlIdAtt         :: !(ControlAttribute .st) -> Id
getControlItemSpaceAtt  :: !(ControlAttribute .st) -> (Int,Int)
getControlKeyboardAtt   :: !(ControlAttribute .st) -> ( KeyboardStateFilter
                                                      , SelectState
                                                      , KeyboardFunction .st
                                                      )
getControlLookAtt       :: !(ControlAttribute .st) -> (Bool,Look)
getControlMinimumSizeAtt:: !(ControlAttribute .st) -> Size
getControlModsFun       :: !(ControlAttribute .st) -> ModifiersFunction .st
getControlMouseAtt      :: !(ControlAttribute .st) -> ( MouseStateFilter
                                                      , SelectState
                                                      , MouseFunction .st
                                                      )
getControlOriginAtt     :: !(ControlAttribute .st) -> Point2
getControlOuterSizeAtt  :: !(ControlAttribute .st) -> Size
getControlPenAtt        :: !(ControlAttribute .st) -> [PenAttribute]
getControlPosAtt        :: !(ControlAttribute .st) -> ItemPos
getControlResizeFun     :: !(ControlAttribute .st) -> ControlResizeFunction
getControlSelectStateAtt:: !(ControlAttribute .st) -> SelectState
getControlTipAtt        :: !(ControlAttribute .st) -> String
getControlViewDomainAtt :: !(ControlAttribute .st) -> ViewDomain
getControlViewSizeAtt   :: !(ControlAttribute .st) -> Size
getControlVMarginAtt    :: !(ControlAttribute .st) -> (Int,Int)
getControlVScrollFun    :: !(ControlAttribute .st) -> ScrollFunction
getControlWidthAtt      :: !(ControlAttribute .st) -> ControlWidth
```

A.6 StdControlClass

```
definition module StdControlClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdControlClass define the standard set of controls instances.
//  ********************************************************************************


import  StdControlDef
from    windowhandle    import ControlState
from    StdPSt          import PSt, IOSt


class Controls cdef where
    controlToHandles    :: !(cdef      .ls (PSt .l)) !(PSt .l)
                    -> (![ControlState .ls (PSt .l)], !PSt .l)
    getControlType      ::  (cdef      .ls .pst)
                    -> ControlType

instance Controls (AddLS  c)            | Controls c
instance Controls (NewLS  c)            | Controls c
instance Controls (ListLS c)            | Controls c
instance Controls NilLS
instance Controls ((:+:) c1 c2)         | Controls c1 & Controls c2

instance Controls ButtonControl
instance Controls CheckControl
instance Controls (CompoundControl c)   | Controls c
instance Controls CustomButtonControl
instance Controls CustomControl
instance Controls EditControl
instance Controls (LayoutControl   c)   | Controls c
instance Controls PopUpControl
instance Controls RadioControl
instance Controls SliderControl
instance Controls TextControl
```

A.7 StdControlDef

```
definition module StdControlDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdControl contains the types to define the standard set of controls.
//  ********************************************************************************


import  StdIOCommon, StdPictureDef


::  ButtonControl       ls pst
 =  ButtonControl       String                         [ControlAttribute *(ls,pst)]
::  CheckControl        ls pst
 =  CheckControl        [CheckControlItem *(ls,pst)] RowsOrColumns
                                                       [ControlAttribute *(ls,pst)]
::  CompoundControl      c ls pst
 =  CompoundControl     (c ls pst)                     [ControlAttribute *(ls,pst)]
::  CustomButtonControl ls pst
 =  CustomButtonControl Size Look                      [ControlAttribute *(ls,pst)]
::  CustomControl       ls pst
 =  CustomControl       Size Look                      [ControlAttribute *(ls,pst)]
::  EditControl         ls pst
 =  EditControl         String ControlWidth NrLines    [ControlAttribute *(ls,pst)]
::  LayoutControl        c ls pst
 =  LayoutControl       (c ls pst)                     [ControlAttribute *(ls,pst)]
::  PopUpControl        ls pst
 =  PopUpControl        [PopUpControlItem *(ls,pst)] Index
                                                       [ControlAttribute *(ls,pst)]
::  RadioControl        ls pst
 =  RadioControl        [RadioControlItem *(ls,pst)] RowsOrColumns Index
                                                       [ControlAttribute *(ls,pst)]
::  SliderControl       ls pst
 =  SliderControl       Direction ControlWidth SliderState (SliderAction *(ls,pst))
                                                       [ControlAttribute *(ls,pst)]
::  TextControl         ls pst
 =  TextControl         String                         [ControlAttribute *(ls,pst)]

::  CheckControlItem st :== (String, Maybe ControlWidth, MarkState, IdFun st)
::  PopUpControlItem st :== (String,                               IdFun st)
::  RadioControlItem st :== (String, Maybe ControlWidth,            IdFun st)
::  NrLines             :== Int
::  RowsOrColumns
    =   Rows            Int
    |   Columns         Int
::  ControlWidth                // The width of the control:
    =   PixelWidth      Int     // the exact number of pixels
    |   TextWidth       String  // the exact string width in dialog font
    |   ContentWidth    String  // width of the control as if string is its content

::  ControlAttribute st                             // Default:
 // General control attributes:
    =   ControlActivate     (IdFun             st)  // id
    |   ControlDeactivate   (IdFun             st)  // id
    |   ControlFunction     (IdFun             st)  // id
    |   ControlHide                                 // initially visible
    |   ControlId           Id                      // no id
    |   ControlKeyboard     KeyboardStateFilter SelectState (KeyboardFunction st)
                                                    // no keyboard input/overruled
    |   ControlMinimumSize  Size                    // zero
    |   ControlModsFunction (ModifiersFunction st)  // ControlFunction
    |   ControlMouse        MouseStateFilter    SelectState (MouseFunction st)
                                                    // no mouse input/overruled
    |   ControlPen          [PenAttribute]          // default pen attributes
    |   ControlPos          ItemPos                 // (RightTo previous,zero)
    |   ControlResize       ControlResizeFunction   // no resize
    |   ControlSelectState  SelectState             // control Able
    |   ControlTip          String                  // no tip
    |   ControlWidth        ControlWidth            // system derived
 // For CompoundControls only:
    |   ControlHMargin      Int Int                 // system dependent
    |   ControlHScroll      ScrollFunction          // no horizontal scrolling
    |   ControlItemSpace    Int Int                 // system dependent
    |   ControlLook         Bool Look               // control is transparant
    |   ControlOrigin       Point2                  // Left top of ViewDomain
    |   ControlOuterSize    Size                    // enclose elements
    |   ControlViewDomain   ViewDomain              // {zero,max range}
    |   ControlViewSize     Size                    // enclose elements
    |   ControlVMargin      Int Int                 // system dependent
    |   ControlVScroll      ScrollFunction          // no vertical   scrolling

::  ControlResizeFunction
    :== Size ->                                     // current control outer size
        Size ->                                     // old     parent  view  size
        Size ->                                     // new     parent  view  size
        Size                                        // new     control outer size
::  ControlType
    :== String
```

A.8 StdControlReceiver

```
definition module StdControlReceiver


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdControlReceiver defines Receiver(2) controls instances.
//  ********************************************************************************


import  StdReceiverDef, StdControlClass


instance Controls (Receiver  m  )
instance Controls (Receiver2 m r)
```

A.9 StdEventTCP

```
definition module StdEventTCP

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdEventTCP provides functions for using event driven TCP
//  ********************************************************************************

import  StdChannels, StdTCPDef
from    StdString       import String
from    StdReceiver     import Receivers, ReceiverType
from    StdPSt          import PSt, IOSt
from    StdIOCommon     import ErrorReport
from    tcp_bytestreams import TCP_SCharStream_
from    StdPStClass     import FileEnv, Files, TimeEnv, Date, Tick, Time

instance ChannelEnv (PSt .l), (IOSt .l)

instance Receivers      TCP_ListenerReceiver
instance Receivers      TCP_Receiver
instance Receivers      TCP_CharReceiver

openSendNotifier        ::  .ls !(SendNotifier *(*ch .a) .ls (PSt .l))
                            !(PSt .l)
                        ->  (!ErrorReport,!*(*ch .a),!PSt .l)
                        |   accSChannel ch & Send ch
/*  opens a send notifier, which informs the application, that sending on the
    channel is again possible due to flow conditions. Possible error reports are
    NoError and ErrorNotifierOpen
*/

closeSendNotifier       ::  !*(*ch .a) !(IOSt .l)
                        -> (!*(*ch .a), !IOSt .l)
                        |   accSChannel ch
/*  to close a send notifier. This function will be called implicitly if a send
    channel is closed, so there is no need to do it explicitly then.
*/

lookupIPAddress_async   ::  !String !(InetLookupFunction (PSt .l)) !(PSt .l)
                        ->  (PSt .l)
/*  lookupIPAddress_async asynchronously looks up an IP address. The String can be
    in dotted decimal form or alphanumerical. The InetLookupFunction will be called
    with the IP address, if this address was found, otherwise with Nothing.
*/

connectTCP_async        ::  !(!IPAddress,!Port) !(InetConnectFunction (PSt .l))
                            !(PSt .l)
                        ->  (PSt .l)
/*  connectTCP_async asynchronously tries to establish a new connection. The
    InetConnectFunction will be called with the new duplex channel if this attempt
    was succesful, otherwise with Nothing
*/

class accSChannel ch    ::  (TCP_SChannel -> (x, TCP_SChannel)) *(*ch .a)
                        ->  (x, *(*ch .a))
/*  This overloaded function supports the openSendNotifier function. It applies an
    access function on the underlying TCP_SChannel
*/

instance accSChannel TCP_SChannel_
instance accSChannel TCP_SCharStream_
```

A.10 StdFileSelect

```
definition module StdFileSelect


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdFileSelect defines the standard file selector dialogue.
//  ********************************************************************************


import StdMaybe, StdString

class FileSelectEnv env where
    selectInputFile ::                 !*env -> (!Maybe String,!*env)
    selectOutputFile:: !String !String !*env -> (!Maybe String,!*env)
    selectDirectory ::                 !*env -> (!Maybe String,!*env)
/*  selectInputFile
        opens a dialogue in which the user can browse the file system to select an
        existing file.
        If a file has been selected, the String result contains the complete
        pathname of the selected file.
        If the user has not selected a file, Nothing is returned.
    selectOutputFile
        opens a dialogue in which the user can browse the file system to save a
        file.
        The first argument is the prompt of the dialogue (default: "Save As:")
        The second argument is the suggested filename.
        If the indicated directory already contains a file with the indicated name,
        selectOutputFile opens a new dialogue to confirm overwriting of the existing
        file.
        If either this dialogue is not confirmed or browsing is cancelled then
        Nothing is returned, otherwise the String result is the complete pathname of
        the selected file.
    selectDirectory
        opens a dialogue in which the user can browse the file system to select a
        directory.
        If a directory has been selected, the String result contains the complete
        pathname of the selected directory.
        If the user has not selected a directory, Nothing is returned.
*/

instance FileSelectEnv World
```

A.11 StdId

```
definition module StdId


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdId specifies the generation functions for identification values.
//  ********************************************************************************


from    StdMaybe    import Maybe, Just, Nothing
from    id          import Id, RId, R2Id, RIdtoId, R2IdtoId, toString, ==
from    iostate     import PSt, IOSt

class Ids env where
    openId      ::      !*env -> (!Id,          !*env)
    openIds     :: !Int !*env -> (![Id],        !*env)

    openRId     ::      !*env -> (!RId  m,      !*env)
    openRIds    :: !Int !*env -> (![RId m],     !*env)

    openR2Id    ::      !*env -> (!R2Id  m r,   !*env)
    openR2Ids   :: !Int !*env -> (![R2Id m r],  !*env)
/*  There are three types of identification values:
    -   RId  m:     for uni-directional message passing (see StdReceiver)
    -   R2Id m r:   for bi-directional  message passing (see StdReceiver)
    -   Id:         for all other Object I/O library components
    Of each generation function there are two variants:
    -   to create exactly one identification value.
    -   to create a number of identification values.
            If the integer argument <=0, then an empty list of identification values
            is generated.
*/

instance Ids World,
             IOSt .l,
             PSt  .l

getParentId :: !Id !(IOSt .l) -> (!Maybe Id,!IOSt .l)
/*  getParentId returns the Id of the parent top-level GUI object
    of the GUI component identified by the argument Id.
    If the GUI component could not be found then Nothing is returned.
*/
```

A.12 StdIO

```
definition module StdIO

//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdIO contains all definition modules of the Object I/O library.
//  ****************************************************************************

import
    StdId,                  // The operations that generate identification values
    StdIOBasic,             // Function and type definitions used in the library
    StdIOCommon,            // Function and type definitions used in the library
    StdKey,                 // Function and type definitions on keyboard
    StdMaybe,               // The Maybe data type
    StdPSt,                 // Operations on PSt that are not device related
    StdPStClass,            // PSt/IOSt instances of common classes
    StdSystem,              // System dependent operations

    StdFileSelect,          // File selector dialogues

    StdPictureDef,          // Type definitions for picture handling
    StdPicture,             // Picture handling operations
    StdBitmap,              // Defines an instance for drawing bitmaps

    StdProcessDef,          // Type definitions for process handling
    StdProcessAttribute,    // ProcessAttribute access operations
    StdProcess,             // Process handling operations

    StdClipboard,           // Clipboard handling operations

    StdPrint,               // General printing functions
    StdPrintText,           // Specialised text printing functions

    StdControlDef,          // Type definitions for controls
    StdControlAttribute,    // ControlAttribute access operations
    StdControlClass,        // Standard controls class instances
    StdControlReceiver,     // Receiver controls class instances
    StdControl,             // Control handling operations

    StdMenuDef,             // Type definitions for menus
    StdMenuAttribute,       // MenuAttribute access operations
    StdMenuElementClass,    // Standard menus class instances
    StdMenuReceiver,        // Receiver menus class instances
    StdMenuElement,         // Menu element handling operations
    StdMenu,                // Menu handling operations

    StdReceiverDef,         // Type definitions for receivers
    StdReceiverAttribute,   // ReceiverAttribute access operations
    StdReceiver,            // Receiver handling operations

    StdTimerDef,            // Type definitions for timers
    StdTimerAttribute,      // TimerAttribute access operations
    StdTimerElementClass,   // Standard timer class instances
    StdTimerReceiver,       // Receiver timer class instances
    StdTimer,               // Timer handling operations
    StdTime,                // Time related operations

    StdWindowDef,           // Type definitions for windows
    StdWindowAttribute,     // WindowAttribute access operations
    StdWindow               // Window handling operations
```

A.13 StdIOBasic

```
definition module StdIOBasic


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdIOBasic defines basic types and access functions for the I/O library.
//  ********************************************************************************


import  StdOverloaded, StdString


/*  General type constructors for composing context-independent data structures.
*/
::  :^:     t1 t2           = (:^:) infixr 9 t1 t2


/*  General type constructors for composing context-dependent data structures.
*/
::  :~:     t1 t2       cs  = (:~:) infixr 9 (t1 cs) (t2 cs)
::  ListCS      t       cs  = ListCS [t cs]
::  NilCS               cs  = NilCS


/*  General type constructors for composing local and context-dependent
    data structures.
*/
::  :+:     t1 t2   ls  cs  = (:+:) infixr 9 (t1 ls cs) (t2 ls cs)
::  ListLS      t   ls  cs  = ListLS [t ls cs]
::  NilLS           ls  cs  = NilLS
::  NewLS       t   ls  cs  = E..new: {newLS::new, newDef:: t   new     cs}
::  AddLS       t   ls  cs  = E..add: {addLS::add, addDef:: t *(add,ls) cs}

noLS ::     (.a->.b)    (.c,.a) -> (.c,.b)  // Lift function    a  ->  b
                                            // to            (c,a)->(c,b)
noLS1:: (.x->.a->.b) .x (.c,.a) -> (.c,.b)  // Lift function x->  a  ->  b
                                            // to            x->(c,a)->(c,b)


::  Index           :== Int
::  Title           :== String


::  Vector2         =   {vx::!Int,vy::!Int}

instance            ==          Vector2     // @1-@2==zero
instance            +           Vector2     // {vx=@1.vx+@2.vx,vy=@1.vy+@2.vy}
instance            -           Vector2     // {vx=@1.vx-@2.vx,vy=@1.vy-@2.vy}
instance            zero        Vector2     // {vx=0,vy=0}
instance            ~           Vector2     // zero-@1
instance            toString    Vector2

class toVector      x :: !x ->  Vector2


::  Size            =   {w ::!Int,h ::!Int}

instance            ==          Size        // @1.w==@2.w && @1.h==@2.h
instance            zero        Size        // {w=0,h=0}
instance            toVector    Size        // {w,h}->{vx=w,vy=h}
instance            toString    Size


::  Point2
```

```
    =   {   x       :: !Int
        ,   y       :: !Int
        }
::  Rectangle
    =   {   corner1 :: !Point2
        ,   corner2 :: !Point2
        }

instance            ==          Point2      // @1-@2==zero
instance            +           Point2      // {x=@1.x+@2.x,y=@1.y+@2.y}
instance            -           Point2      // {x=@1.x-@2.x,y=@1.y-@2.y}
instance            zero        Point2      // {x=0,y=0}
instance            ~           Point2      // zero-@1
instance            toVector    Point2      // {x,y}->{vx=x,vy=y}
instance            toString    Point2

instance            ==          Rectangle   //    @1.corner1==@2.corner1
                                            // && @1.corner2==@2.corner2
instance            zero        Rectangle   // {corner1=zero,corner2=zero}
instance            toString    Rectangle

rectangleSize       :: !Rectangle -> Size   // {w=abs (@1.corner1-@1.corner2).x,
                                            //  h=abs (@1.corner1-@1.corner2).y}
movePoint   :: !Vector2 !Point2 -> .Point2  // {vx,vy} {x,y} -> {vx+x,vy+y}


::  IdFun st        :== st -> st

::  Void            =   Void
```

A.14 StdIOCommon

```
definition module StdIOCommon


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdIOCommon defines common types and access functions for the I/O library.
//  ********************************************************************************


import  StdOverloaded, StdString
import  StdBitmap, StdIOBasic, StdKey, StdMaybe
from    id          import  Id, RId, R2Id, RIdtoId, R2IdtoId, toString, ==


/*  The SelectState and MarkState types.                */

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

enabled     :: !SelectState -> Bool         // @1 == Able
marked      :: !MarkState   -> Bool         // @1 == Mark

instance ==       SelectState               // Constructor equality
instance ==       MarkState                 // Constructor equality
instance ~        SelectState               // Able <-> Unable
instance ~        MarkState                 // Mark <-> NoMark
instance toString SelectState
instance toString MarkState



/*  The KeyboardState type.                             */

::  KeyboardState
    =   CharKey     Char        KeyState    // ASCII character input
    |   SpecialKey  SpecialKey  KeyState Modifiers
                                            // Special key input
    |   KeyLost                             // Key input lost while key was down
::  KeyState
    =   KeyDown     IsRepeatKey             // Key is down
    |   KeyUp                               // Key goes up
::  IsRepeatKey                             // Flag on key down:
    :== Bool                                // True iff key is repeating
::  Key
    =   IsCharKey    Char
    |   IsSpecialKey SpecialKey
::  KeyboardStateFilter                     // Predicate on KeyboardState:
    :== KeyboardState -> Bool               // evaluate KeyFunction only if True

getKeyboardStateKeyState:: !KeyboardState -> KeyState   // KeyUp   if KeyLost
getKeyboardStateKey     :: !KeyboardState -> Maybe Key  // Nothing if KeyLost

instance ==       KeyboardState             // Equality on KeyboardState
instance ==       KeyState                  // Equality on KeyState
instance toString KeyboardState
instance toString KeyState


/*  The MouseState type.                                */

::  MouseState
    =   MouseMove   Point2 Modifiers        // Mouse is up     (position,modifiers)
    |   MouseDown   Point2 Modifiers Int    // Mouse goes down (and nr down)
    |   MouseDrag   Point2 Modifiers        // Mouse is down   (position,modifiers)
    |   MouseUp     Point2 Modifiers        // Mouse goes up   (position,modifiers)
    |   MouseLost                           // Mouse input lost while mouse was down
::  ButtonState
    =   ButtonStillUp                       // MouseMove
    |   ButtonDown                          // MouseDown _ _ 1
    |   ButtonDoubleDown                    //           _ _ 2
    |   ButtonTripleDown                    //           _ _ >2
    |   ButtonStillDown                     // MouseDrag
    |   ButtonUp                            // MouseUp/MouseLost
::  MouseStateFilter                        // Predicate on MouseState:
    :== MouseState -> Bool                  // evaluate MouseFunction only if True

getMouseStatePos        :: !MouseState  -> Point2       // zero        if MouseLost
getMouseStateModifiers  :: !MouseState  -> Modifiers    // NoModifiers if MouseLost
getMouseStateButtonState:: !MouseState  -> ButtonState  // ButtonUp    if MouseLost

instance ==       MouseState                // Equality on MouseState
instance ==       ButtonState               // Constructor equality
instance toString MouseState
instance toString ButtonState

/*  The SliderState type.                               */

::  SliderState
    =   {   sliderMin   :: !Int
        ,   sliderMax   :: !Int
        ,   sliderThumb :: !Int
        }

instance == SliderState                     // @1.sliderMin   == @2.sliderMin
                                            // @1.sliderMax   == @2.sliderMax
                                            // @1.sliderThumb == @2.sliderThumb
instance toString SliderState


/*  The UpdateState type.                               */

::  UpdateState
    =   {   oldFrame    :: !ViewFrame
        ,   newFrame    :: !ViewFrame
        ,   updArea     :: !UpdateArea
        }
::  ViewDomain          :== Rectangle
::  ViewFrame           :== Rectangle
::  UpdateArea          :== [ViewFrame]

instance toString UpdateState

RectangleToUpdateState  :: !Rectangle -> UpdateState
                                            // r -> {oldFrame=newFrame=r,updArea=[r]}

/*  viewDomainRange defines the minimum and maximum values for ViewDomains.
    viewFrameRange  defines the minimum and maximum values for ViewFrames.
*/
viewDomainRange         :== {   corner1 = {x = 0-(2^30),y = 0-(2^30)}
                            ,   corner2 = {x =    2^30 ,y =    2^30 }
                            }
viewFrameRange          :== {   corner1 = {x = 1-(2^31),y = 1-(2^31)}
                            ,   corner2 = {x = (2^31)-1,y = (2^31)-1}
                            }


/*  Modifiers indicates the meta keys that have been pressed (True) or not (False).
*/
::  Modifiers
    =   {   shiftDown   :: !Bool            // True iff shift   down
        ,   optionDown  :: !Bool            // True iff option  down
        ,   commandDown :: !Bool            // True iff command down
        ,   controlDown :: !Bool            // True iff control down
        ,   altDown     :: !Bool            // True iff alt     down
        }

//  Constants to check which of the Modifiers are down.

NoModifiers :== {shiftDown  = False
                ,optionDown = False
                ,commandDown= False
                ,controlDown= False
                ,altDown    = False
                }
ShiftOnly   :== {shiftDown  = True
                ,optionDown = False
                ,commandDown= False
                ,controlDown= False
                ,altDown    = False
                }
OptionOnly  :== {shiftDown  = False
                ,optionDown = True
                ,commandDown= False
                ,controlDown= False
                ,altDown    = True
                }
CommandOnly :== {shiftDown  = False
                ,optionDown = False
                ,commandDown= True
                ,controlDown= True
                ,altDown    = False
                }
ControlOnly :== {shiftDown  = False
                ,optionDown = False
                ,commandDown= True
                ,controlDown= True
                ,altDown    = False
                }
AltOnly     :== {shiftDown  = False
                ,optionDown = True
                ,commandDown= False
                ,controlDown= False
                ,altDown    = True
                }

instance ==         Modifiers
instance toString   Modifiers


/*  The layout language used for windows and controls.  */
::  ItemPos
    :== (   ItemLoc
        ,   ItemOffset
        )
::  ItemLoc
 // Absolute:
    =   Fix
 // Relative to corner:
    |   LeftTop
    |   RightTop
    |   LeftBottom
    |   RightBottom
 // Relative in next line:
    |   Left
    |   Center
    |   Right
 // Relative to other item:
    |   LeftOf  Id
    |   RightTo Id
    |   Above   Id
    |   Below   Id
 // Relative to previous item:
    |   LeftOfPrev
    |   RightToPrev
    |   AbovePrev
    |   BelowPrev
::  ItemOffset
    =   NoOffset                          // Shorthand for OffsetVector zero
    |   OffsetVector Vector2              // A constant offset vector
    |   OffsetFun    ParentIndex OffsetFun  // Offset depends on orientation
::  ParentIndex
    :== Int                               // The number of parents (1..)
::  OffsetFun
    :== (ViewDomain,Point2) -> Vector2    // Current view domain and origin

instance    zero     ItemOffset           // zero == NoOffset
instance    ==       ItemLoc              // Constructor and value equality
instance    toString ItemLoc              // Constructor and value as String


/*  The Direction type.                           */

::  Direction
    =   Horizontal
    |   Vertical

instance    ==       Direction            // Constructor equality
instance    toString Direction            // Constructor as String


/*  The CursorShape type.                         */

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

instance    ==       CursorShape          // Constructor equality
instance    toString CursorShape          // Constructor as String


/*  Document interface of interactive processes.  */

::  DocumentInterface
    =   NDI                               // No       Document Interface
    |   SDI                               // Single   Document Interface
    |   MDI                               // Multiple Document Interface

instance    ==       DocumentInterface    // Constructor equality
instance    toString DocumentInterface    // Constructor as String


/*  Process attributes.                                 */

::  ProcessAttribute st                                 // Default:
    =   ProcessActivate     (IdFun st)                  // No action on activate
    |   ProcessDeactivate   (IdFun st)                  // No action on deactivate
    |   ProcessClose        (IdFun st)                  // Process is closed
 // Attributes for (M/S)DI process only:
    |   ProcessOpenFiles    (ProcessOpenFilesFunction st)
                                                        // Request to open files
    |   ProcessWindowPos    ItemPos                     // Platform dependent
    |   ProcessWindowSize   Size                        // Platform dependent
    |   ProcessWindowResize (ProcessWindowResizeFunction st)
                                                        // Platform dependent
    |   ProcessToolbar      [ToolbarItem st]            // Process has no toolbar
 // Attributes for MDI processes only:
    |   ProcessNoWindowMenu                             // Process has WindowMenu

::  ProcessWindowResizeFunction st
    :== Size                                            // Old ProcessWindow size
     -> Size                                            // New ProcessWindow size
     -> st -> st
::  ProcessOpenFilesFunction st
    :== [String]                                        // The filenames to open
     -> st -> st

::  ToolbarItem st
    =   ToolbarItem Bitmap (Maybe String) (IdFun st)
    |   ToolbarSeparator


/*  Frequently used function types.                     */

::  ModifiersFunction   st  :== Modifiers       ->  st -> st
::  MouseFunction       st  :== MouseState      ->  st -> st
::  KeyboardFunction    st  :== KeyboardState   ->  st -> st
::  SliderAction        st  :== SliderMove      ->  st -> st
::  SliderMove
    =   SliderIncSmall
    |   SliderDecSmall
    |   SliderIncLarge
    |   SliderDecLarge
    |   SliderThumb Int

instance toString SliderMove


/*  Scrolling function.                                 */

::  ScrollFunction
    :== ViewFrame       ->                  // Current  view
        SliderState     ->                  // Current  state of scrollbar
        SliderMove      ->                  // Action of the user
        Int                                 // New thumb value

stdScrollFunction :: !Direction !Int -> ScrollFunction
/*  stdScrollFunction direction d implements standard scrolling behaviour:
    - direction indicates scrolling for Horizontal or Vertical scroll bar.
    - d         is the stepsize with which to scroll (taken absolute).
    stdScrollFunction lets the system scroll as follows:
    - Slider(Inc/Dec)Small: d
    - Slider(Inc/Dec)Large: viewFrame size modulo d
    - SliderThumb x:        x modulo d
*/


/*  Standard GUI object rendering function.             */

::  Look
    :== SelectState ->                      // Current SelectState of GUI object
        UpdateState ->                      // The area to be rendered
        *Picture    -> *Picture             // The rendering action

stdUnfillNewFrameLook:: SelectState !UpdateState !*Picture -> *Picture
stdUnfillUpdAreaLook :: SelectState !UpdateState !*Picture -> *Picture
/*  Two convenience functions for simple Look functions:
    stdUnfillNewFrameLook _ {newFrame} = unfill newFrame
    stdUnfillUpdAreaLook  _ {updArea}  = seq (map unfill updArea)
*/


/*  Common error report types.                          */

::  ErrorReport                     // Usual cause:
    =   NoError                     // Everything went allright
    |   ErrorViolateDI              // Violation against DocumentInterface
    |   ErrorIdsInUse               // Object contains Ids that are bound
    |   ErrorUnknownObject          // Object can not be found
    |   ErrorNotifierOpen           // It was tried to open a second send notifier
    |   OtherError !String          // Other kind of error

instance    ==          ErrorReport // Constructor equality
instance    toString    ErrorReport // Constructor as String

::  OkBool                          // True iff the operation was successful
    :== Bool
```

A.15 StdKey

```
definition module StdKey


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdKey defines the special keys for the Object I/O library.
//  ********************************************************************************


from    key     import  SpecialKey,
                            BackSpaceKey, BeginKey,
                            ClearKey,
                            DeleteKey, DownKey,
                            EndKey, EnterKey, EscapeKey,
                            F1Key,  F2Key,  F3Key,  F4Key,  F5Key,
                            F6Key,  F7Key,  F8Key,  F9Key,  F10Key,
                            F11Key, F12Key, F13Key, F14Key, F15Key,
                            HelpKey,
                            LeftKey,
                            PgDownKey, PgUpKey,
                            RightKey,
                            UpKey,
                        ==, toString
```

A.16 StdMaybe

```
definition module StdMaybe

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMaybe defines the Maybe type.
//  ********************************************************************************

from    StdFunc         import St
from    StdOverloaded   import ==
from    StdIOBasic      import IdFun

::  Maybe x
    =   Just x
    |   Nothing

isJust      :: !(Maybe .x) -> Bool      // case @1 of (Just _) -> True; _ -> False
isNothing   :: !(Maybe .x) -> Bool      // not o isJust
fromJust    :: !(Maybe .x) -> .x        // \(Just x) -> x

// for possibly unique elements:
u_isJust    :: !(Maybe .x) -> (!Bool, !Maybe .x)
u_isNothing :: !(Maybe .x) -> (!Bool, !Maybe .x)

accMaybe    :: .(St .x .a) !(Maybe .x) -> (!Maybe .a,!Maybe .x)
// accMaybe f (Just x) = (Just (fst (f x)),Just (snd (f x)))
// accMaybe f Nothing  = (Nothing,Nothing)

mapMaybe    :: .(.x -> .y) !(Maybe .x) -> Maybe .y
// mapMaybe f (Just x) = Just (f x)
// mapMaybe f Nothing  = Nothing

instance ==       (Maybe x) | == x
//  Nothing==Nothing
//  Just a ==Just b <= a==b
```

A.17 StdMenu

```
definition module StdMenu


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenu defines functions on menus.
//  ********************************************************************************


import  StdMenuElementClass
from    iostate import PSt, IOSt


//  Operations on unknown Ids are ignored.

class Menus mdef where
    openMenu    :: .ls !(mdef .ls (PSt .l)) !(PSt .l) -> (!ErrorReport,!PSt .l)
    getMenuType ::      (mdef .ls .pst)               -> MenuType
/*  Open the given menu definition for this interactive process.
    openMenu may not be permitted to open a menu depending on its DocumentInterface
        (see the comments at the shareProcesses instances in module StdProcess).
    In case a menu with the same Id is already open then nothing happens. In case
        the menu has the WindowMenuId Id then nothing happens. In case the menu does
        not have an Id, it will obtain an Id which is fresh with respect to the
        current set of menus. The Id can be reused after closing this menu. In case
        menu elements are opened with duplicate Ids, the menu will not be opened.
    In case the menu definition does not have a MenuIndex attribute (see StdMenuDef)
        it will be opened behind the last menu. In case the menu definition has a
        MenuIndex attribute it will be placed 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.
*/

instance Menus (Menu      m) | MenuElements      m
instance Menus (PopUpMenu m) | PopUpMenuElements m
/*  PopUpMenus can only be opened in a SDI or MDI process. If the parent process is
        a NDI process, then no PopUpMenu is opened and ErrorViolateDI is returned.
    The elements of a PopUpMenu are the same as for standard menus except for
        SubMenus. For elements the same restrictions hold as for standard menus.
    The PopUpMenu will be closed as soon as the user has dismissed it either by
        selecting an item or clicking outside the menu.
*/


closeMenu :: !Id !(IOSt .l) -> IOSt .l
/*  closeMenu closes the indicated Menu and all of its elements.
    The WindowMenu can not be closed by closeMenu (in case the Id argument equals
    WindowMenuId).
*/


openMenuElements    :: !Id !Index .ls (m .ls (PSt .l))      !(PSt .l)
                                          -> (!ErrorReport,!PSt .l)
                                          |  MenuElements m
openSubMenuElements :: !Id !Index .ls (m .ls (PSt .l))      !(PSt .l)
                                          -> (!ErrorReport,!PSt .l)
                                          |  MenuElements m
openRadioMenuItems  :: !Id !Index ![MenuRadioItem (PSt .l)] !(IOSt .l)
                                          -> (!ErrorReport,!IOSt .l)
/*  Add menu elements to the indicated Menu, SubMenu, or RadioMenu.
    openRadioMenuItems checks the first item in the list if the RadioMenu was empty.
    Menu elements are added after the item with the specified index. The index of a
        menu element starts from one for the first menu element in the indicated
        menu.
        If the index is negative or zero, then the new menu elements are added
        before the first menu element of the indicated menu.
        If the index exceeds the number of menu elements in the indicated menu, then
        the new menu elements are added behind the last menu element of the
        indicated menu.
    No menu elements are added if the indicated menu does not exist. In this case
        ErrorUnknownObject is returned.
    open(Sub)MenuElements have no effect in case menu elements with duplicate Ids
    are opened. In this case ErrorIdsInUse is returned.
*/


closeMenuElements :: !Id ![Id] !(IOSt .l) -> IOSt .l
/*  closeMenuElements
        closes menu elements of the Menu identified by the first Id argument by
        their Ids. The elements of (Sub/Radio)Menus will be removed first.
*/


closeMenuIndexElements      :: !Id ![Index] !(IOSt .l) -> IOSt .l
closeSubMenuIndexElements   :: !Id ![Index] !(IOSt .l) -> IOSt .l
closeRadioMenuIndexElements :: !Id ![Index] !(IOSt .l) -> IOSt .l
/*  Close menu elements of the indicated Menu, SubMenu, or RadioMenu by their Index
    position.
    Analogous to openMenuElements and openRadioMenuItems indices range from one to
        the number of menu elements in a menu. Invalid indices (less than one or
        larger than the number of menu elements of the menu) are ignored.
    If the currently checked element of a RadioMenu is closed, the first remaining
        element of that RadioMenu will be checked.
    Closing a (Sub/Radio)Menu closes the indicated (Sub/Radio)Menu and all of its
    elements.
*/


enableMenuSystem    :: !(IOSt .l) -> IOSt .l
disableMenuSystem   :: !(IOSt .l) -> IOSt .l
/*  Enable/disable the menu system of this interactive process. When the menu system
    is re-enabled the previously selectable menus and elements will become
    selectable again.
    Enable/disable operations on the menu(element)s of a disabled menu system take
    effect when the menu system is re-enabled.
    enableMenuSystem has no effect in case the interactive process has a (number of)
    modal dialogue(s).
*/


enableMenus         :: ![Id] !(IOSt .l) -> IOSt .l
disableMenus        :: ![Id] !(IOSt .l) -> IOSt .l
/*  Enable/disable individual menus.
    The WindowMenu can not be enabled/disabled.
    Disabling a menu overrules the SelectStates of its elements, which become
    unselectable.
    Enabling a disabled menu re-establishes the SelectStates of its elements.
    Enable/disable operations on the elements of a disabled menu take effect when
    the menu is re-enabled.
*/


getMenuSelectState  :: !Id !(IOSt .l) -> (!Maybe SelectState,!IOSt .l)
/*  getMenuSelectState yields the current SelectState of the indicated menu. In case
    the menu does not exist, Nothing is returned.
*/
```

```
getMenus            :: !(IOSt .l) -> (![(Id,MenuType)],!IOSt .l)
/*  getMenus yields the Ids and MenuTypes of the current set of menus of this
    interactive process.
*/


getMenuPos          :: !Id !(IOSt .l) -> (!Maybe Index,!IOSt .l)
/*  getMenuPos yields the index position of the indicated menu in the current list
    of menus.
    In case the menu does not exist, Nothing is returned.
*/


setMenuTitle        :: !Id !Title !(IOSt .l) -> IOSt .l
getMenuTitle        :: !Id        !(IOSt .l) -> (!Maybe Title,!IOSt .l)
/*  setMenuTitle sets the title of the indicated menu.
        In case the menu does not exist or refers to the WindowMenu, nothing
        happens.
    getMenuTitle retrieves the current title of the indicated menu.
        In case the menu does not exist, Nothing is returned.
*/
```

A.18 StdMenuAttribute

```
definition module StdMenuAttribute


//  **************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenuAttribute specifies which MenuAttributes are valid for each of the
//  standard menus and menu elements.
//  Basic comparison operations and retrieval functions are also included.
//  **************************************************************************


import StdMenuDef


/*  The following functions specify the valid attributes for each standard menu
    (element).
*/

isValidMenuAttribute :: !(MenuAttribute .st) -> Bool
/*  Menu            (y = valid, . = invalid)
    MenuFunction    . | MenuInit            y | MenuSelectState y |
    MenuId          y | MenuMarkState       . | MenuShortKey    . |
    MenuIndex       y | MenuModsFunction    . |
*/

isValidSubMenuAttribute :: !(MenuAttribute .st) -> Bool
/*  SubMenu         (y = valid, . = invalid)
    MenuFunction    . | MenuInit            . | MenuSelectState y |
    MenuId          y | MenuMarkState       . | MenuShortKey    . |
    MenuIndex       . | MenuModsFunction    . |
*/

isValidRadioMenuAttribute :: !(MenuAttribute .st) -> Bool
/*  RadioMenu       (y = valid, . = invalid)
    MenuFunction    . | MenuInit            . | MenuSelectState y |
    MenuId          y | MenuMarkState       . | MenuShortKey    . |
    MenuIndex       . | MenuModsFunction    . |
*/

isValidMenuItemAttribute :: !(MenuAttribute .st) -> Bool
/*  MenuItem        (y = valid, . = invalid)
    MenuFunction    y | MenuInit            . | MenuSelectState y |
    MenuId          y | MenuMarkState       y | MenuShortKey    y |
    MenuIndex       . | MenuModsFunction    y |
*/

isValidMenuSeparatorAttribute :: !(MenuAttribute .st) -> Bool
/*  MenuSeparator   (y = valid, . = invalid)
    MenuFunction    . | MenuInit            . | MenuSelectState . |
    MenuId          y | MenuMarkState       . | MenuShortKey    . |
    MenuIndex       . | MenuModsFunction    . |
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isMenuFunction          :: !(MenuAttribute .st) -> Bool
isMenuId                :: !(MenuAttribute .st) -> Bool
isMenuIndex             :: !(MenuAttribute .st) -> Bool
isMenuInit              :: !(MenuAttribute .st) -> Bool
isMenuMarkState         :: !(MenuAttribute .st) -> Bool
isMenuModsFunction      :: !(MenuAttribute .st) -> Bool
isMenuSelectState       :: !(MenuAttribute .st) -> Bool
```

```
isMenuShortKey          :: !(MenuAttribute .st) -> Bool


/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getMenuFun              :: !(MenuAttribute .st) -> IdFun .st
getMenuIdAtt            :: !(MenuAttribute .st) -> Id
getMenuIndexAtt         :: !(MenuAttribute .st) -> Index
getMenuInitFun          :: !(MenuAttribute .st) -> IdFun .st
getMenuMarkStateAtt     :: !(MenuAttribute .st) -> MarkState
getMenuModsFun          :: !(MenuAttribute .st) -> ModifiersFunction .st
getMenuSelectStateAtt   :: !(MenuAttribute .st) -> SelectState
getMenuShortKeyAtt      :: !(MenuAttribute .st) -> Char
```

A.19 StdMenuDef

```
definition module StdMenuDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenu contains the types to define the standard set of menus and their
//  elements.
//  ********************************************************************************


import  StdIOCommon, StdMaybe


/*  Menus:              */
::  Menu        m ls pst = Menu          Title         (m ls pst)
                                         [MenuAttribute *(ls,pst)]
::  PopUpMenu   m ls pst = PopUpMenu                   (m ls pst)

/*  Menu elements:      */
::  MenuItem      ls pst = MenuItem      Title
                                         [MenuAttribute *(ls,pst)]
::  MenuSeparator ls pst = MenuSeparator [MenuAttribute *(ls,pst)]
::  RadioMenu     ls pst = RadioMenu     [MenuRadioItem *(ls,pst)] Index
                                         [MenuAttribute *(ls,pst)]
::  SubMenu     m ls pst = SubMenu       Title         (m ls pst)
                                         [MenuAttribute *(ls,pst)]

::  MenuRadioItem st  :== (Title,Maybe Id,Maybe Char,IdFun st)

::  MenuAttribute           st                      // Default:
 // Attributes for Menus and MenuElements:
    =   MenuId              Id                      // no Id
    |   MenuSelectState     SelectState             // menu(item) Able
 // Attributes only for Menus:
    |   MenuIndex           Int                     // end of current menu list
    |   MenuInit            (IdFun st)              // no actions after opening menu
 // Attributes ignored by (Sub)Menus:
    |   MenuFunction        (IdFun              st) // \x->x
    |   MenuMarkState       MarkState               // NoMark
    |   MenuModsFunction    (ModifiersFunction  st) // MenuFunction
    |   MenuShortKey        Char                    // no ShortKey

::  MenuType        :== String
::  MenuElementType :== String
```

A.20 StdMenuElement

```
definition module StdMenuElement


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenuElement specifies all functions on menu elements.
//  ********************************************************************************


import  StdMenuDef
from    iostate import IOSt


/*  Functions that change the state of menu elements.
    Only those Id arguments that refer to menu elements within the same interactive
    process are used to change the corresponding menu elements.
*/

enableMenuElements      :: ![Id] !(IOSt .l) -> IOSt .l
disableMenuElements     :: ![Id] !(IOSt .l) -> IOSt .l
/*  (en/dis)ableMenuElements set the SelectState of the indicated menu elements.
    Disabling a (Sub/Radio)Menu overrules the SelectStates of its elements, which
        become unselectable.
    Enabling a disabled (Sub/Radio)Menu re-establishes the SelectStates of its
        elements.
    (En/Dis)able operations on the elements of a disabled (Sub/Radio)Menu take
        effect when the (Sub/Radio)Menu is re-enabled.
*/

setMenuElementTitles    :: ![(Id,Title)] !(IOSt .l) -> IOSt .l
/*  setMenuElementTitles sets the titles of the indicated menu elements.
*/

markMenuItems           :: ![Id] !(IOSt .l) -> IOSt .l
unmarkMenuItems         :: ![Id] !(IOSt .l) -> IOSt .l
/*  (un)markMenuItems sets the MarkState of the indicated MenuItems.
*/

selectRadioMenuItem     :: !Id !Id    !(IOSt .l) -> IOSt .l
selectRadioMenuIndexItem:: !Id !Index !(IOSt .l) -> IOSt .l
/*  selectRadioMenu(Index)Item
        selects the indicated MenuRadioItem of a RadioMenu, causing the mark of the
        previously marked MenuRadioItem to disappear.
    selectRadioMenuItem
        indicates the MenuRadioItem by the Id of its parent RadioMenu and its Id.
    selectRadioMenuIndexItem
        indicates the MenuRadioItem by the Id of its parent RadioMenu and its index
        position (counted from 1).
*/


/*  Access functions on MState. To read the state of a menu element, a MState is
    required which can be obtained by the getMenu function. The MState value
    represents the state of a menu at that particular moment.
*/

::  MState

getMenu         :: !Id !(IOSt .l) -> (!Maybe MState, !IOSt .l)
getParentMenu   :: !Id !(IOSt .l) -> (!Maybe MState, !IOSt .l)
/*  getMenu returns a read-only MState for the indicated menu.
        In case the indicated menu does not exist Nothing is returned.
    getParentMenu returns a read-only MState for the indicated menu element.
        In case the Id does not correspond with a menu element, then Nothing
        is returned.
*/

getMenuElementTypes         ::     !MState -> [(MenuElementType,Maybe Id)]
getCompoundMenuElementTypes :: !Id !MState -> [(MenuElementType,Maybe Id)]
/*  getMenuElementTypes
        yields the list of MenuElementTypes of all menu elements of this menu.
    getCompoundMenuElementTypes
        yields the list of MenuElementTypes of all menu elements of this
        (Sub/Radio)Menu.
    Both functions return (Just id) if the element has a MenuId attribute, and
    Nothing otherwise.
    Ids are not collected recursively through (Sub/Radio)Menus.
*/

/*  Functions that return the current state of menu elements.
    For each access there is one singular and one plural version. In case of the
    plural version the result list is of equal length as the argument Id list. Each
    result list element corresponds in order with the argument Id list.
    In both versions the first Boolean result is False in case of invalid Ids (if so
    dummy values are returned - see comment).
    Important: menu elements with no MenuId attribute, or illegal ids, can not be
    found in the MState!
*/
getSelectedRadioMenuItems   :: ![Id]  !MState -> [(!Index,!Maybe Id)]
getSelectedRadioMenuItem    :: ! Id   !MState ->  (!Index,!Maybe Id)
/*  getSelectedRadioMenuItem(s)
        returns the Index and Id, if any, of the currently selected MenuRadioItem of
        the indicated RadioMenu.
    If the RadioMenu does not exist or is empty, the Index is zero and the Id is
    Nothing.
*/

getMenuElementSelectStates  :: ![Id] !MState -> [(Bool,SelectState)]
getMenuElementSelectState   :: ! Id  !MState ->  (Bool,SelectState)
/*  getMenuElementSelectState(s) yields the SelectStates of the indicated elements.
    If the element does not exist Able is returned.
*/

getMenuElementMarkStates    :: ![Id] !MState -> [(Bool,MarkState)]
getMenuElementMarkState     :: ! Id  !MState ->  (Bool,MarkState)
/*  getMenuElementMarkState(s) yields the MarkState of the indicated elements.
    If the element does not exist NoMark is returned.
*/

getMenuElementTitles        :: ![Id] !MState -> [(Bool,Maybe String)]
getMenuElementTitle         :: ! Id  !MState ->  (Bool,Maybe String)
/*  getMenuElementTitle(s) yields (Just title) of the indicated (SubMenu/MenuItem),
    Nothing otherwise.
*/

getMenuElementShortKeys     :: ![Id] !MState -> [(Bool,Maybe Char)]
getMenuElementShortKey      :: ! Id  !MState ->  (Bool,Maybe Char)
/*  getMenuElementShortKey(s) yields (Just key) of the indicated MenuItem, Nothing
    otherwise.
*/
```

A.21 StdMenuElementClass

```
definition module StdMenuElementClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenuElementClass defines the standard set of menu element instances.
//  ********************************************************************************


import  StdMenuDef
from    StdPSt      import PSt, IOSt
from    menuhandle  import MenuElementState


/*  Menu elements for Menus:
*/
class MenuElements m where
    menuElementToHandles    :: !(m .ls (PSt .l)) !(PSt .l)
            -> (![MenuElementState .ls (PSt .l)], !PSt .l)
    getMenuElementType      ::  (m .ls .pst)
            -> MenuElementType

instance MenuElements (AddLS    m)  | MenuElements m
instance MenuElements (NewLS    m)  | MenuElements m
instance MenuElements (ListLS   m)  | MenuElements m
instance MenuElements NilLS
instance MenuElements ((:+:) m1 m2) | MenuElements m1
                                    & MenuElements m2
instance MenuElements (SubMenu  m)  | MenuElements m
instance MenuElements RadioMenu
instance MenuElements MenuItem
instance MenuElements MenuSeparator

/*  Menu elements for PopUpMenus:
*/
class PopUpMenuElements m where
    popUpMenuElementToHandles   :: !(m .ls (PSt .l)) !(PSt .l)
                -> (![MenuElementState .ls (PSt .l)], !PSt .l)
    getPopUpMenuElementType     ::  (m .ls .pst)
                -> MenuElementType

instance PopUpMenuElements (AddLS   m)      | PopUpMenuElements m
instance PopUpMenuElements (NewLS   m)      | PopUpMenuElements m
instance PopUpMenuElements (ListLS  m)      | PopUpMenuElements m
instance PopUpMenuElements NilLS
instance PopUpMenuElements ((:+:) m1 m2)    | PopUpMenuElements m1
                                            & PopUpMenuElements m2
instance PopUpMenuElements RadioMenu
instance PopUpMenuElements MenuItem
instance PopUpMenuElements MenuSeparator
```

A.22 StdMenuReceiver

```
definition module StdMenuReceiver


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdMenuReceiver defines Receiver(2) menu element instances.
//  ********************************************************************************


import  StdReceiverDef, StdMenuElementClass


//  Receiver components:
instance MenuElements (Receiver  m  )
instance MenuElements (Receiver2 m r)
```

A.23 StdPicture

```
definition module StdPicture


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPicture contains the drawing operations and access to Pictures.
//  ********************************************************************************


from    StdFunc     import  St
from    osfont      import  Font
from    ospicture   import  Picture
import  StdPictureDef


//  Pen attribute functions:

setPenAttributes        :: ![PenAttribute]      !*Picture -> *Picture
getPenAttributes        ::                      !*Picture
                            -> (![PenAttribute],!*Picture)

//  Pen position attributes:
setPenPos               :: !Point2              !*Picture -> *Picture
getPenPos               ::                      !*Picture -> (!Point2,!*Picture)

class movePenPos figure :: !figure              !*Picture -> *Picture
//  Move the pen position as much as when drawing the figure.
instance movePenPos Vector2
instance movePenPos Curve

//  Pen size attributes:
setPenSize              :: !Int                 !*Picture -> *Picture
getPenSize              ::                      !*Picture -> (!Int,!*Picture)

setDefaultPenSize       ::                      !*Picture -> *Picture
//  setDefaultPenSize = setPenSize 1

//  Pen colour attributes:
setPenColour            :: !Colour              !*Picture -> *Picture
getPenColour            ::                      !*Picture -> (!Colour,!*Picture)
setPenBack              :: !Colour              !*Picture -> *Picture
getPenBack              ::                      !*Picture -> (!Colour,!*Picture)

setDefaultPenColour     ::                      !*Picture -> *Picture
setDefaultPenBack       ::                      !*Picture -> *Picture

toRGBColour             :: !Colour -> RGBColour // Convert a colour to RGBColour

//  setDefaultPenColour = setPenColour Black
//  setDefaultPenBack   = setPenBack   White

//  Pen font attributes:
setPenFont              :: !Font                !*Picture -> *Picture
getPenFont              ::                      !*Picture -> (!Font,!*Picture)

setDefaultPenFont       ::                      !*Picture -> *Picture


/*  Font operations:
*/
openFont        :: !FontDef !*Picture -> (!(!Bool,!Font),!*Picture)
openDefaultFont ::          !*Picture -> (!Font,        !*Picture)
openDialogFont  ::          !*Picture -> (!Font,        !*Picture)
/*  openFont
```

```
        creates the font as specified by the name, stylistic variations, and size.
        The Boolean result is True only if the font is available and need not be
        scaled.
        In all other cases, an existing font is returned (depending on the system).
    openDefaultFont
        returns the font used by default by applications.
    openDialogFont
        returns the font used by default by the system.
*/

getFontNames    ::                      !*Picture -> (![FontName],   !*Picture)
getFontStyles   ::           !FontName  !*Picture -> (![FontStyle],  !*Picture)
getFontSizes    :: !Int !Int !FontName  !*Picture -> (![FontSize],   !*Picture)
/*  getFontNames
        returns the FontNames  of all available fonts.
    getFontStyles
        returns the FontStyles of all available styles of a particular FontName.
    getFontSizes
        returns all FontSizes in increasing order of a particular FontName that are
        available without scaling. The sizes inspected are inclusive between the two
        Integer arguments. (Negative values are set to zero.)
        In case the requested font is unavailable, the styles or sizes of the
        default font are returned.
*/

getFontDef :: !Font -> FontDef
/*  getFontDef returns the name, stylistic variations and size of the argument Font.
*/

getPenFontCharWidth     ::       ! Char     !*Picture -> (! Int,        !*Picture)
getPenFontCharWidths    ::       ![Char]    !*Picture -> (![Int],       !*Picture)
getPenFontStringWidth   ::       ! String   !*Picture -> (! Int,        !*Picture)
getPenFontStringWidths  ::       ![String]  !*Picture -> (![Int],       !*Picture)
getPenFontMetrics       ::                  !*Picture -> (!FontMetrics, !*Picture)

getFontCharWidth        :: !Font ! Char     !*Picture -> (!Int,         !*Picture)
getFontCharWidths       :: !Font ![Char]    !*Picture -> (![Int],       !*Picture)
getFontStringWidth      :: !Font ! String   !*Picture -> (!Int,         !*Picture)
getFontStringWidths     :: !Font ![String]  !*Picture -> (![Int],       !*Picture)
getFontMetrics          :: !Font            !*Picture -> (!FontMetrics, !*Picture)
/*  get(Pen)Font(Char/String)Width(s)
        return the width of the argument (Char/String)(s) given the Font argument
        or current PenFont attribute.
    get(Pen)FontMetrics
        returns the FontMetrics of the Font argument or current PenFont attribute.
*/


/*  Region functions.
    A Region is defined by a collection of shapes.
*/
::  Region

//  Basic access functions on Regions:

isEmptyRegion   :: !Region -> Bool
getRegionBound  :: !Region -> Rectangle
/*  isEmptyRegion
        holds if the argument region covers no pixels (it is empty).
    getRegionBound
        returns the smallest enclosing rectangle of the argument region.
        If the region is empty, zero is returned.
*/

//  Constructing a region:

class toRegion area :: !area -> Region
```

```
::  PolygonAt
    =   {   polygon_pos :: !Point2
        ,   polygon     :: !Polygon
        }

instance toRegion Rectangle
instance toRegion PolygonAt
instance toRegion [area]            | toRegion area
instance toRegion (:^: area1 area2) | toRegion area1 & toRegion area2


//  Drawing and restoring picture attributes:

appPicture      :: !.(IdFun *Picture) !*Picture -> *Picture
accPicture      :: !.(St *Picture .x) !*Picture -> (.x,!*Picture)
/*  (app/acc)Picture f pict
        apply f to pict. After drawing,  the picture attributes of the result
        picture are restored to those of pict.
*/


//  Drawing within in a clipping region:

appClipPicture :: !Region !.(IdFun *Picture) !*Picture -> *Picture
accClipPicture :: !Region !.(St *Picture .x) !*Picture -> (.x,!*Picture)


//  Drawing in 'exclusive or' mode:

appXorPicture   :: !.(IdFun *Picture) !*Picture -> *Picture
accXorPicture   :: !.(St *Picture .x) !*Picture -> (.x,!*Picture)
/*  (app/acc)XorPicture f pict
        apply f to pict in the appropriate platform xor mode.
*/


//  Drawing in 'hilite' mode:

class Hilites figure where
    hilite  ::         !figure !*Picture -> *Picture
    hiliteAt:: !Point2 !figure !*Picture -> *Picture
/*  hilite
        draws figures in the appropriate 'hilite' mode at the current  pen position.
    hiliteAt
        draws figures in the appropriate 'hilite' mode at the argument pen position.
    Both functions reset the 'hilite' mode after drawing.
*/

instance Hilites Box        // Hilite a box
instance Hilites Rectangle  // Hilite a rectangle (note: hiliteAt pos r = hilite r)


//  Drawing points:

drawPoint       ::                !*Picture -> *Picture
drawPointAt     :: !Point2        !*Picture -> *Picture
/*  drawPoint
        plots a point at the current  pen position p and moves to p+{vx=1,vy=0}
    drawPointAt
        plots a point at the argument pen position, but retains the pen position.
*/

//  Drawing lines:

drawLineTo      :: !Point2         !*Picture -> *Picture
drawLine        :: !Point2 !Point2 !*Picture -> *Picture
```

```
/*  drawLineTo
        draws a line from the current pen position to the argument point which
        becomes the new pen position.
    drawLine
        draws a line between the two argument points, but retains the pen position.
*/


/*  Drawing and filling operations.
    These functions are divided into the following classes:
    Drawables:
        draw     'line-oriented' figures at the current  pen position.
        drawAt   'line-oriented' figures at the argument pen position.
        undraw     f = appPicture (draw     f o setPenColour background)
        undrawAt x f = appPicture (drawAt x f o setPenColour background)
    Fillables:
        fill     'area-oriented' figures at the current  pen position.
        fillAt   'area-oriented' figures at the argument pen position.
        unfill     f = appPicture (fill     f o setPenColour background)
        unfillAt x f = appPicture (fillAt x f o setPenColour background)
*/
class Drawables figure where
    draw    ::         !figure !*Picture -> *Picture
    drawAt  :: !Point2 !figure !*Picture -> *Picture
    undraw  ::         !figure !*Picture -> *Picture
    undrawAt:: !Point2 !figure !*Picture -> *Picture

class Fillables figure where
    fill    ::         !figure !*Picture -> *Picture
    fillAt  :: !Point2 !figure !*Picture -> *Picture
    unfill  ::         !figure !*Picture -> *Picture
    unfillAt:: !Point2 !figure !*Picture -> *Picture


//  Text drawing operations:
//  Text is always drawn with the baseline at the y coordinate of the pen.

instance Drawables  Char
instance Drawables  {#Char}
/*  draw     text:
        draws the text starting at the current pen position.
        The new pen position is directly after the drawn text including spacing.
    drawAt p text:
        draws the text starting at p.
*/


//  Line2 drawing operations:
instance Drawables Line2
/*  draw     l:
        is equal to drawLine l.line_end1 l.line_end2.
    drawAt p l:
        draw l
    None of these functions change the pen position.
*/

//  Vector2 drawing operations:
instance Drawables Vector2
/*  draw     v:
        draws a line from the current pen position pen to pen+v.
    drawAt p v:
        draws a line from p to p+v.
*/


/*  Oval drawing operations:
    An Oval o is a transformed unit circle
    with    horizontal radius rx    o.oval_rx
            vertical   radius ry    o.oval_ry
    Let (x,y) be a point on the unit circle:
        then (x‘,y‘) = (x*rx,y*ry) is a point on o.
    Let (x,y) be a point on o:
        then (x‘,y‘) = (x/rx,y/ry) is a point on the unit circle.
*/
instance Drawables Oval
instance Fillables Oval
/*  draw     o:
        draws an oval with the current pen position being the center of the oval.
    drawAt p o:
        draws an oval with p being the center of the oval.
    fill     o:
        fills an oval with the current pen position being the center of the oval.
    fillAt p o:
        fills an oval with p being the center of the oval.
    None of these functions change the pen position.
*/


/*  Curve drawing operations:
    A Curve c is a slice of an oval o
    with    start angle a   c.curve_from
            end   angle b   c.curve_to
            direction   d   c.curve_clockwise
    The angles are taken in radians (counter-clockwise).
    If d holds then the drawing direction is clockwise, otherwise drawing occurs
    counter-clockwise.
*/
instance Drawables Curve
instance Fillables Curve
/*  draw     c:
        draws a curve with the starting angle a at the current pen position.
        The pen position ends at ending angle b.
    drawAt p c:
        draws a curve with the starting angle a at p.
    fill     c:
        fills the figure obtained by connecting the endpoints of the drawn curve
        (draw c) with the center of the curve oval.
        The pen position ends at ending angle b.
    fillAt p c:
        fills the figure obtained by connecting the endpoints of the drawn curve
        (drawAt p c) with the center of the curve oval.
*/


/*  Box drawing operations:
    A Box b is a horizontally oriented rectangle
    with    width  w        b.box_w
            height h        b.box_h
    In case w==0 (h==0),   the Box collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Box collapses to a point.
*/
instance Drawables  Box
instance Fillables  Box
/*  draw     b:
        draws a box with left-top corner at the current pen position p and
        right-bottom corner at p+(w,h).
    drawAt p b:
        draws a box with left-top corner at p and right-bottom corner at p+(w,h).
    fill     b:
        fills a box with left-top corner at the current pen position p and
        right-bottom corner at p+(w,h).
    fillAt p b:
        fills a box with left-top corner at p and right-bottom corner at p+(w,h).
    None of these functions change the pen position.
```

```
*/


/*  Rectangle drawing operations:
    A Rectangle r is always horizontally oriented
    with    width  w        abs (r.corner1.x-r.corner2.x)
            height h        abs (r.corner1.y-r.corner2.y)
    In case w==0 (h==0),   the Rectangle collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Rectangle collapses to a point.
*/
instance Drawables Rectangle
instance Fillables Rectangle
/*  draw     r:
        draws a rectangle with diagonal corners r.corner1 and r.corner2.
    drawAt p r:
        draw r
    fill     r:
        fills a rectangle with diagonal corners r.corner1 and r.corner2.
    fillAt p r:
        fill r
    None of these functions change the pen position.
*/


/*  Polygon drawing operations:
    A Polygon p is a figure
    with    shape   p.polygon_shape
    A polygon p at a point base is drawn as follows:
        drawPicture [setPenPos base:map draw shape]++[drawToPoint base]
*/
instance Drawables Polygon
instance Fillables Polygon
/*  None of these functions change the pen position.
*/


getResolution :: !*Picture -> (!(!Int,!Int),!*Picture)
/*  getResolution returns the horizontal and vertical resolution of a Picture in dpi
    (dots per inch).
    In case of a printer Picture:
        the return values are the printer resolution (if it is not emulating the
        screen resolution).
    In case of a screen Picture:
        the resolution is the number of pixels that fit in one "screen inch".
        A "screen inch" is the physical size of a 72 point font on the screen.
        Although some screens allow the user to alter the screen resolution,
        getResolution always returns the same value for the same screen.
        The reason is that if a user for instance increases the screen resolution,
        not only the size of a pixel decreases, but also the size of a "screen
        inch". So a 12 point font will not appear with 12 point size, but smaller
        (A point is a physical unit, defined as 1 point is approximately 1/72 inch.)
*/
```

A.24 StdPictureDef

```
definition module StdPictureDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPictureDef contains the predefined figures that can be drawn.
//  ********************************************************************************


import  StdIOBasic
from    osfont      import  Font


::  Line2                                   // A line connects two points
    =   {   line_end1       :: !Point2      // The first  point
        ,   line_end2       :: !Point2      // The second point
        }
::  Box                                     // A box is a rectangle
    =   {   box_w           :: !Int         // The width  of the box
        ,   box_h           :: !Int         // The height of the box
        }
::  Oval                                    // An oval is a stretched unit circle
    =   {   oval_rx         :: !Int         // The horizontal radius (stretch)
        ,   oval_ry         :: !Int         // The vertical   radius (stretch)
        }
::  Curve                                   // A curve is a slice of an oval
    =   {   curve_oval      :: !Oval        // The source oval
        ,   curve_from      :: !Real        // Starting angle (in radians)
        ,   curve_to        :: !Real        // Ending   angle (in radians)
        ,   curve_clockwise :: !Bool        // Direction: True iff clockwise
        }
::  Polygon                                 // A polygon is an outline shape
    =   {   polygon_shape   :: ![Vector2]   // The shape of the polygon
        }
::  FontDef
    =   {   fName           :: !FontName    // Name of the font
        ,   fStyles         :: ![FontStyle] // Stylistic variations
        ,   fSize           :: !FontSize    // Size in points
        }
::  FontMetrics
    =   {   fAscent         :: !Int         // Distance between top    and base line
        ,   fDescent        :: !Int         // Distance between bottom and base line
        ,   fLeading        :: !Int         // Distance between two text lines
        ,   fMaxWidth       :: !Int         // Max character width including spacing
        }
::  FontName    :== String
::  FontStyle   :== String
::  FontSize    :== Int
::  Colour
    =   RGB RGBColour
    |   Black       | White
    |   DarkGrey    | Grey      | LightGrey // 75%, 50%, and 25% Black
    |   Red         | Green     | Blue
    |   Cyan        | Magenta   | Yellow
::  RGBColour
    =   {   r   :: !Int                     // The contribution of red
        ,   g   :: !Int                     // The contribution of green
        ,   b   :: !Int                     // The contribution of blue
        }
::  PenAttribute                            // Default:
    =   PenSize     Int                     // 1
    |   PenPos      Point2                  // zero
    |   PenColour   Colour                  // Black
    |   PenBack     Colour                  // White
    |   PenFont     Font                    // DefaultFont

//  Colour constants:
BlackRGB                :== {r=MinRGB,g=MinRGB,b=MinRGB}
WhiteRGB                :== {r=MaxRGB,g=MaxRGB,b=MaxRGB}
MinRGB                  :== 0
MaxRGB                  :== 255

//  Font constants:
SerifFontDef            :== {fName="Times New Roman",fStyles=[],fSize=10}
SansSerifFontDef        :== {fName="Arial",          fStyles=[],fSize=10}
SmallFontDef            :== {fName="Small Fonts",    fStyles=[],fSize=7 }
NonProportionalFontDef  :== {fName="Courier New",    fStyles=[],fSize=10}
SymbolFontDef           :== {fName="Symbol",         fStyles=[],fSize=10}

//  Font style constants:
ItalicsStyle            :== "Italic"
BoldStyle               :== "Bold"
UnderlinedStyle         :== "Underline"

//  Standard lineheight of a font is the sum of its leading, ascent and descent:
fontLineHeight fMetrics :== fMetrics.fLeading + fMetrics.fAscent + fMetrics.fDescent

//  Useful when working with Ovals and Curves:
PI                      :== 3.1415926535898
```

A.25 StdPrint

Cancelled, StartedPrinting, PrintEnvironments

```
definition module StdPrint

//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPrint specifies general printing functions.
//  ****************************************************************************


from    StdIOCommon     import  UpdateState, ViewFrame, UpdateArea
from    StdIOBasic      import  IdFun, Size, Rectangle, Point2
from    StdOverloaded   import  ==
from    ospicture       import  Picture
from    osprint         import  PrintSetup, JobInfo, PrintInfo, Alternative,
                                Cancelled, StartedPrinting, PrintEnvironments
from    iostate         import  IOSt, PSt
from    StdFile         import  FileEnv, Files

::  PageDimensions
    =   {   page        :: !Size        // Size of the drawable area of the page
        ,   margins     :: !Rectangle   // This field contains information about the
                                        // size of the margins on a sheet in pixels.
                                        // Drawing can't occur within these margins.
                                        // The margin Rectangle is bigger than the
                                        // page size. Its values are:
                                        // corner1.x<=0 && corner1.y<=0 &&
                                        // corner2.x>=page.w && corner2.y>=page.h
        ,   resolution  :: !(!Int,!Int) // Horizontal and vertical printer
                                        // resolution in dpi
        }

defaultPrintSetup   :: !*env -> (!PrintSetup,!*env)
                    | FileEnv env
/*  defaultPrintSetup returns a default print setup.
*/

printSetupDialog    :: !PrintSetup !*env -> (!PrintSetup,!*env)
                    | PrintEnvironments env
/*  printSetupDialog lets the user choose a print setup via the print setup dialog.
*/

getPageDimensions   :: !PrintSetup !Bool -> PageDimensions
instance == PageDimensions

fwritePrintSetup    :: !PrintSetup !*File -> *File
/*  fwritePrintSetup writes PrintSetup to file (text or data).
*/

freadPrintSetup     :: !*File !*env -> (!Bool,!PrintSetup,!*File,!*env)
                    |  FileEnv  env
/*  freadPrintSetup reads PrintSetup from File (text or data).
    If the resulting Boolean is True: success, otherwise the default PrintSetup is
    returned.
*/

print   ::  !Bool !Bool
            .(PrintInfo !*Picture -> ([IdFun *Picture],!*Picture))
            !PrintSetup !*env
        -> (!PrintSetup,!*env)
        |  PrintEnvironments env
/*  print doDialog emulateScreen pages printSetup env
    sends output to the printer and returns the used print setup, which can differ
    from the input print setup.
    doDialog:
        if True a dialog will pop up that lets the user choose all printing options,
        otherwise printing will happen in the default way.
    emulateScreen:
        if True, the printing routine will emulate the resolution of the screen.
        That means that a pixel on paper has the same dimension as on screen.
        Otherwise, the used resolution will be the printer resolution, with the
        effect that coordinates get much "tighter".
    pages:
        this function should calculate a list of functions, each function
        representing one page to be printed. Each of these drawing functions is
        applied to an initial printer Picture.
    env:
        a PrintEnvironment is either the PSt or the Files system.
*/

printUpdateFunction
        ::  !Bool (UpdateState -> *Picture -> *Picture) [Rectangle]
            !PrintSetup !*env
        -> (!PrintSetup,!*env)
        |  PrintEnvironments env
/*  printUpdateFunction doDialog update area printSetup env
    sends the content of the update function of a given area to the printer:
    doDialog:
        identical to print.
    update:
        this function will be applied to an UpdateState of value
            {oldFrame=area,newFrame=area,updArea=[area]}.
    area:
        the area to be sent to the printer. If a rectangle of this area does not
        fit on one sheet, it will be distributed on several sheets.
    printSetup,env,result value:
        identical to print.
*/

printPagePerPage
        :: !Bool !Bool
           .x
           .(.x -> .(PrintInfo -> .(*Picture -> ((.Bool,Point2),(.state,*Picture)))))
           ((.state,*Picture) -> ((.Bool,Point2),(.state,*Picture)))
           !PrintSetup           !*env
        -> (Alternative .x .state,!*env)
        |  PrintEnvironments env
/*  printPagePerPage doDialog emulateScreen x prepare pages printSetup env
    sends output to the printer.
    This function can be used more efficiently than print. The major difference is
    that the pages function is a state transition function instead of a page list
    producing function. Each page transition function generates one page for the
    printer. An additional feature of printPagePerPage is that it is possible to
    set the origin of the printer Pictures.
    doDialog:
        identical to print.
    emulateScreen:
        identical to print.
    x:
        this value is passed to the prepare function.
    prepare:
        this function calculates the initial page print state.
        Iff there are no pages to print, the return Boolean must be True.
        The returned Point is the Origin of the first printer Picture.
    pages:
        this state transition function produces the printed pages.
        The state argument consists of the state information and an initial printer
        Picture which Origin has been set by the previous return Point value.
        If there are no more pages to print, the return Boolean must be True. In
        that case the result of printPagePerPage is (StartedPrinting state),
        with state the current state value. If printing should continue, the
        return Boolean is False.
        The returned Point is the Origin of the next printer Picture.
    printSetup, env:
        identical to print.

    If printing is cancelled via the print dialog, then (Cancelled x) will be
    returned, otherwise (StartedPrinting ...)
*/

instance PrintEnvironments World
/*  Other instances are the Files subworld and PSt.
*/
```

A.26 StdPrintText

```
definition module StdPrintText

//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPrintText specifies functions to print text.
//  ****************************************************************************

from    StdPictureDef   import FontDef, FontName, FontStyle, FontSize
import  StdPrint
from    StdString       import String

::  WrapMode    :== Int

NoWrap          :== 0
LeftJustify     :== 1
RightJustify    :== 2

class CharStreams cs where
    getChar     :: !*cs -> (!Bool,!Char,!*cs)
                // getChar returns the next character of the stream. The Boolean
                // result indicates whether this operation was successful.
    savePos     :: !*cs -> *cs
                // savePos saves actual position of charstream to enable the
                // restorePos function to restore it.
    restorePos  :: !*cs -> *cs
    eos         :: !*cs -> (!Bool,!*cs)
                // eos checks for end of stream.

instance CharStreams FileCharStream

::  *FileCharStream

fileToCharStream :: !*File -> *FileCharStream
charStreamToFile :: !*FileCharStream -> *File


printText1 :: !Bool !WrapMode !FontDef !Int
              !*charStream !PrintSetup  !*env
        -> (!(!*charStream,!PrintSetup),!*env)
        |  CharStreams charStream & PrintEnvironments env
/*  printText1 doDialog wrapMode font spacesPerTab
               charStream printSetup env
    prints a CharStream:
    doDialog:
        identical to print (StdPrint)
    wrapMode:
        controls word wrapping in case lines do not fit. NoWrap suppresses wrapping.
        LeftJustify and RightJustify wrap text to the left and right respectively.
    font:
        the text will be printed in this font.
    spacesPerTab:
        the number of spaces a tab symbol represents.
    charStream:
        the charStream to be printed.
    printSetup, env:
        identical to print (StdPrint)
*/

printText2 :: !String !String
              !Bool !WrapMode !FontDef !Int
              !*charStream !PrintSetup  !*env
        -> (!(!*charStream,!PrintSetup),!*env)
        |  CharStreams charStream & PrintEnvironments env
```

```
/*  printText2 titleStr pageStr
               doDialog wrapMode fontParams spacesPerTab
               charStream printSetup env
    prints a charStream with a header on each page.
    titleStr:
        this String will be printed on each page at the left corner of the header
    pageStr:
        this String and the actual page number are printed on the right corner of
        the header
    The other parameters are identical to printText1.
*/

printText3 :: !Bool !WrapMode !FontDef !Int
              .(PrintInfo *Picture -> (state, (Int,Int), *Picture))
               (state Int *Picture -> *Picture)
              !*charStream !PrintSetup  !*env
        -> (!(!*charStream,!PrintSetup),!*env)
        |  CharStreams charStream & PrintEnvironments env
/*  printText3 doDialog wrapMode font spacesPerTab
               textRange
               eachPageDraw
               charStream printSetup env
    prints a charStream with a header and trailer on each page.
    textRange:
        this function takes a PrintInfo record and the printer Picture on which the
        text will be printed. It returns a triple (state,range,picture):
        state:
            a value of arbitrary type that can be used to pass data to the page
            printing function pages.
        range:
            a pair (top,bottom), where top<bottom. The printed text will appear
            within these y-coordinates only, so a header and a trailer can be
            printed for each page.
    eachPageDraw:
        this function draws the header and/or trailer for the current page. Its
        arguments are the data produced by textRange, the actual page number, and
        an initial printer Picture. This function is applied by printText3 before
        each new page receives its text.
    The other parameters are identical to printText1.
*/

/*  If a file is openend with FReadData, then all possible newline conventions
    (unix,mac,dos) will be recognized. All these printing functions will replace
    nonprintable characters of the font with ASCII spaces. Exceptions are: newline,
    formfeed and tab. So the ASCII space has to be a printable character in the used
    font. A form feed character will cause a form feed, and it will also end a line.
*/
```

A.27 StdProcess

```
definition module StdProcess


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdProcess contains the process creation and manipulation functions.
//  ****************************************************************************


import  StdProcessDef
from    iostate import PSt, IOSt


/*  General process topology creation functions:
*/

class Processes pdef where
    startProcesses  :: !pdef !*World   -> *World
    openProcesses   :: !pdef !(PSt .l) -> PSt .l
/*  (start/open)Processes creates an interactive process topology specified by
        the pdef argument.
        All interactive processes can communicate with each other by means of the
        file system or by message passing.
    startProcesses terminates as soon as all interactive processes that are
        created by startProcesses and their child processes have terminated.
    openProcesses schedules the interactive processes specified by the pdef argument
        to be created.
*/

instance Processes [pdef]   | Processes pdef
instance Processes Process

startIO :: !DocumentInterface !.l !(ProcessInit (PSt .l))
                              ![ProcessAttribute (PSt .l)]
            !*World -> *World
/*  startIO creates one process group of one interactive process.
*/


//  Process access operations:

closeProcess        :: !(PSt .l) -> PSt .l
/*  closeProcess removes all abstract devices that are held in the interactive
    process.
    If the interactive process has processes that share its GUI then these will also
    be closed recursively. As a result evaluation of this interactive process
    including GUI sharing processes will terminate.
*/


hideProcess         :: !(PSt .l) -> PSt .l
showProcess         :: !(PSt .l) -> PSt .l
/*  If the interactive process is active, hideProcess hides the interactive process,
    and showProcess makes it visible. Note that hiding an interactive process does
    NOT disable the process but simply makes it invisible.
*/

getProcessWindowPos :: !(IOSt .l) -> (!Point2,!IOSt .l)
/*  getProcessWindowPos returns the current position of the ProcessWindow.
*/

getProcessWindowSize:: !(IOSt .l) -> (!Size,!IOSt .l)
/*  getProcessWindowSize returns the current size of the ProcessWindow.
*/
```

A.28 StdProcessAttribute

```
definition module StdProcessAttribute


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdProcessAttribute specifies which ProcessAttributes are valid for each of the
//  standard interactive processes.
//  Basic comparison operations and retrieval functions are also included.
//  ********************************************************************************


import StdProcessDef


/*  The following function specifies the valid attributes for each standard
    interactive process, specialised by its DocumentInterface.
*/

isProcessKindAttribute :: !DocumentInterface !(ProcessAttribute .st) -> Bool
/*  (The document interface is given for which the attribute is valid)
    ProcessActivate     NDI SDI MDI | ProcessToolbar        SDI MDI
    ProcessClose        NDI SDI MDI | ProcessWindowPos      SDI MDI
    ProcessDeactivate   NDI SDI MDI | ProcessWindowResize   SDI MDI
    ProcessNoWindowMenu         MDI | ProcessWindowSize     SDI MDI
    ProcessOpenFiles        SDI MDI
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isProcessActivate           :: !(ProcessAttribute .st) -> Bool
isProcessClose              :: !(ProcessAttribute .st) -> Bool
isProcessDeactivate         :: !(ProcessAttribute .st) -> Bool
isProcessNoWindowMenu       :: !(ProcessAttribute .st) -> Bool
isProcessOpenFiles          :: !(ProcessAttribute .st) -> Bool
isProcessToolbar            :: !(ProcessAttribute .st) -> Bool
isProcessWindowPos          :: !(ProcessAttribute .st) -> Bool
isProcessWindowResize       :: !(ProcessAttribute .st) -> Bool
isProcessWindowSize         :: !(ProcessAttribute .st) -> Bool


/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getProcessActivateFun       :: !(ProcessAttribute .st) -> IdFun .st
getProcessCloseFun          :: !(ProcessAttribute .st) -> IdFun .st
getProcessDeactivateFun     :: !(ProcessAttribute .st) -> IdFun .st
getProcessOpenFilesFun      :: !(ProcessAttribute .st)
                                               -> ProcessOpenFilesFunction .st
getProcessToolbarAtt        :: !(ProcessAttribute .st) -> [ToolbarItem .st]
getProcessWindowPosAtt      :: !(ProcessAttribute .st) -> ItemPos
getProcessWindowResizeFun   :: !(ProcessAttribute .st)
                                               -> ProcessWindowResizeFunction .st
getProcessWindowSizeAtt     :: !(ProcessAttribute .st) -> Size
```

A.29 StdProcessDef

```
definition module StdProcessDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdProcessDef contains the types to define interactive processes.
//  ********************************************************************************


import  StdIOCommon
from    iostate import PSt, IOSt


::  Process
    =   E. .l: Process
                    DocumentInterface           // The process DocumentInterface
                    l                           // The process private state
                    (ProcessInit      (PSt l))  // The process initialisation
                    [ProcessAttribute (PSt l)]  // The process attributes

/*  NDI processes can't open windows and menus.
    SDI processes can have at most one window open.
    MDI processes can open an arbitrary number of device instances.
*/

::  ProcessInit pst
    :== IdFun pst
```

A.30 StdPSt

```
definition module StdPSt


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPSt defines operations on PSt and IOSt that are not abstract device related.
//  ********************************************************************************


from    StdFunc     import St
from    StdIOCommon import IdFun, DocumentInterface, MDI, SDI, NDI
from    StdPicture  import Picture
from    iostate     import PSt, IOSt


/*  accScreenPicture provides access to an initial Picture as it would be created in
    a window or control.
*/
class accScreenPicture env :: !.(St *Picture .x) !*env -> (!.x,!*env)

instance accScreenPicture World
instance accScreenPicture (IOSt .l)


beep :: !(IOSt .l) -> IOSt .l
/*  beep emits the alert sound.
*/


//  Operations on the DocumentInterface of an interactive process:

getDocumentInterface :: !(IOSt .l) -> (!DocumentInterface, !IOSt .l)
/*  getDocumentInterface returns the DocumentInterface of the interactive process.
*/


//  Operations on the attributes of an interactive process:

setProcessActivate  :: !(IdFun (PSt .l)) !(IOSt .l) -> IOSt .l
setProcessDeactivate:: !(IdFun (PSt .l)) !(IOSt .l) -> IOSt .l
/*  These functions set the ProcessActivate and ProcessDeactivate attribute of the
    interactive process respectively.
*/


//  Coercing PSt component operations to PSt operations.

appListPIO  :: ![.IdFun (IOSt .l)]  !(PSt .l) ->        PSt .l
appListPLoc :: ![.IdFun .l]         !(PSt .l) ->        PSt .l

appPIO      :: !.(IdFun (IOSt .l))  !(PSt .l) ->        PSt .l
appPLoc     :: !.(IdFun .l)         !(PSt .l) ->        PSt .l

//  Accessing PSt component operations.

accListPIO  :: ![.St (IOSt .l) .x]  !(PSt .l) -> (![.x],!PSt .l)
accListPLoc :: ![.St .l       .x]  !(PSt .l) -> (![.x],!PSt .l)

accPIO      :: !.(St (IOSt .l) .x)  !(PSt .l) -> (! .x, !PSt .l)
accPLoc     :: !.(St .l       .x)  !(PSt .l) -> (! .x, !PSt .l)
```

A.31 StdPStClass

```
definition module StdPStClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdPStClass collects (PSt .l) and (IOSt .l) class instances.
//  ********************************************************************************


import  StdFile, StdFileSelect, StdSound, StdTime
from    iostate     import PSt, IOSt


/*  PSt is an environment instance of the following classes:
    - FileSystem    (see StdFile)
    - FileEnv       (see StdFile)
    - FileSelectEnv (see StdFileSelect)
    - TimeEnv       (see StdTime)
    - playSoundFile (see StdSound)

    IOSt is also an environment instance of the classes FileEnv, TimeEnv
*/
instance FileSystem     (PSt .l)
instance FileEnv        (PSt .l), (IOSt .l)
instance FileSelectEnv  (PSt .l)
instance TimeEnv        (PSt .l), (IOSt .l)
instance playSoundFile  (PSt .l)
```

A.32 StdReceiver

```
definition module StdReceiver

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdReceiver specifies all receiver operations.
//  ********************************************************************************

import  StdReceiverDef, StdMaybe
from    iostate import  PSt, IOSt
from    id      import  RId, R2Id, RIdtoId, R2IdtoId, ==


//  Open uni- and bi-directional receivers:

class Receivers rdef where
    openReceiver   :: .ls !*(*rdef .ls (PSt .l)) !(PSt .l) -> (!ErrorReport,!PSt .l)
    getReceiverType::      *(*rdef .ls .pst)               -> ReceiverType
/*  openReceiver
        opens the given receiver if no receiver currently exists with the given
        R(2)Id. The R(2)Id has to be used to send messages to this receiver.
    getReceiverType
        returns the type of the receiver (see also getReceivers).
*/

instance Receivers (Receiver  msg)
instance Receivers (Receiver2 msg resp)


closeReceiver           :: !Id !(IOSt .l) -> IOSt .l
/*  closeReceiver closes the indicated uni- or bi-directional receiver.
    Invalid Ids have no effect.
*/

getReceivers            :: !(IOSt .l) -> (![(Id,ReceiverType)], !IOSt .l)
/*  getReceivers returns the Ids and ReceiverTypes of all currently open uni- or
    bi-directional receivers of this interactive process.
*/

enableReceivers         :: ![Id] !(IOSt .l) ->                      IOSt .l
disableReceivers        :: ![Id] !(IOSt .l) ->                      IOSt .l
getReceiverSelectState  :: ! Id  !(IOSt .l) -> (!Maybe SelectState,!IOSt .l)
/*  (en/dis)ableReceivers
        (en/dis)able the indicated uni- or bi-directional receivers.
        Note that this implies that in case of synchronous message passing messages
        can fail (see the comments of syncSend and syncSend2 below). Invalid Ids
        have no effect.
    getReceiverSelectState
        yields the current SelectState of the indicated receiver. In case the
        receiver does not exist, Nothing is returned.
*/

//  Inter-process communication:

//  Message passing status report:
::  SendReport
    =   SendOk
    |   SendUnknownReceiver
    |   SendUnableReceiver
    |   SendDeadlock
    |   OtherSendReport !String

instance ==       SendReport
instance toString SendReport
```

```
asyncSend :: !(RId msg) msg !(PSt .l) -> (!SendReport, !PSt .l)
/*  asyncSend posts a message to the receiver indicated by the argument RId. In case
    the indicated receiver belongs to this process, the message is simply buffered.
    asyncSend is asynchronous: the message will at some point be received by the
    indicated receiver.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent.
                Note that even though the message has been sent, it cannot be
                guaranteed that the message will actually be handled by the
                indicated receiver because it might become closed, forever disabled,
                or flooded with synchronous messages.
    -   SendUnknownReceiver:
                The indicated receiver does not exist.
    -   SendUnableReceiver:
                Does not occur: the message is always buffered, regardless whether
                the indicated receiver is Able or Unable. Note that in case the
                receiver never becomes Able, the message will not be handled.
    -   SendDeadlock:
                Does not occur.
*/

syncSend :: !(RId msg) msg !(PSt .l) -> (!SendReport, !PSt .l)
/*  syncSend posts a message to the receiver indicated by the argument RId. In case
    the indicated receiver belongs to the current process, the corresponding
    ReceiverFunction is applied directly to the message argument and current process
    state.
    syncSend is synchronous: this interactive process blocks evaluation until the
    indicated receiver has received the message.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent and
                handled by the indicated receiver.
    -   SendUnknownReceiver:
                The indicated receiver does not exist.
    -   SendUnableReceiver:
                The receiver exists, but its ReceiverSelectState is Unable.
                Message passing is halted. The message is not sent.
    -   SendDeadlock:
                The receiver is involved in a synchronous, cyclic communication
                with the current process. Blocking the current process would result
                in a deadlock situation. Message passing is halted to circumvent the
                deadlock. The message is not sent.
*/

syncSend2 :: !(R2Id msg resp) msg  !(PSt .l)
    -> (!(!SendReport,!Maybe resp), !PSt .l)
/*  syncSend2 posts a message to the receiver indicated by the argument R2Id. In
    case the indicated receiver belongs to the current process, the corresponding
    Receiver2Function is applied directly to the message argument and current
    process state.
    syncSend2 is synchronous: this interactive process blocks until the indicated
    receiver has received the message.
    The SendReport can be one of the following alternatives:
    -   SendOk: No exceptional situation has occurred. The message has been sent and
                handled by the indicated receiver. The response of the receiver is
                returned as well as (Just response).
    -   SendUnknownReceiver:
                The indicated receiver does not exist.
    -   SendUnableReceiver:
                The receiver exists, but its ReceiverSelect is Unable.
                Message passing is halted. The message is not sent.
    -   SendDeadlock:
                The receiver is involved in a synchronous, cyclic communication
                with the current process. Blocking the current process would result
                in a deadlock situation. Message passing is halted to circumvent the
                deadlock. The message is not sent.
    In all other cases than SendOk, the optional response is Nothing.
*/
```

A.33 StdReceiverAttribute

```
definition module StdReceiverAttribute


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdReceiverAttribute specifies which ReceiverAttributes are valid for each of
//  the standard receivers.
//  Basic comparison operations and retrieval functions are also included.
//  ********************************************************************************


import StdReceiverDef


/*  The following functions specify the valid attributes for each standard receiver.
*/

isValidReceiverAttribute :: !(ReceiverAttribute .st) -> Bool
/*  Receiver            (y = valid, . = invalid)
    ReceiverInit    y | ReceiverSelectState y |
*/

isValidReceiver2Attribute :: !(ReceiverAttribute .st) -> Bool
/*  Receiver2           (y = valid, . = invalid)
    ReceiverInit    y | ReceiverSelectState y |
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isReceiverInit              :: !(ReceiverAttribute .st) -> Bool
isReceiverSelectState       :: !(ReceiverAttribute .st) -> Bool
isReceiverConnectedReceivers:: !(ReceiverAttribute .st) -> Bool // MW11++

/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getReceiverInitFun              :: !(ReceiverAttribute .st) -> IdFun .st
getReceiverSelectStateAtt       :: !(ReceiverAttribute .st) -> SelectState
getReceiverConnectedReceivers   :: !(ReceiverAttribute .st) -> [Id] // MW11++
```

A.34 StdReceiverDef

```
definition module StdReceiverDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdReceiverDef contains the types to define the standard set of receivers.
//  ********************************************************************************


import  StdIOCommon


::  Receiver  m   ls pst = Receiver  (RId  m)   (ReceiverFunction  m   *(ls,pst))
                                                [ReceiverAttribute     *(ls,pst)]
::  Receiver2 m r ls pst = Receiver2 (R2Id m r) (Receiver2Function m r *(ls,pst))
                                                [ReceiverAttribute     *(ls,pst)]

::  ReceiverFunction  m   st :== m -> st ->    st
::  Receiver2Function m r st :== m -> st -> (r,st)

::  ReceiverAttribute   st                  // Default:
    =   ReceiverInit        (IdFun st)      // no actions after opening receiver
    |   ReceiverSelectState SelectState     // receiver Able
    |   ReceiverConnectedReceivers [Id]     // []
::  ReceiverType
    :== String
```

A.35 StdSound

```
definition module StdSound


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdSound specifies sound playing functions.
//  ********************************************************************************

from    StdString   import String


class playSoundFile env :: !String !*env -> (!Bool,!*env)

/*  playSoundFile filename
        opens the sound file at filename and plays it synchronously.
        The Boolean result indicates whether the sound file could be succesfully
        played.
*/

instance playSoundFile World
```

A.36 StdStringChannels

```
definition module StdStringChannels

//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdStringChannels provides channel instances to send and receive Strings.
//  These channels use their own protocol above TCP.
//  ****************************************************************************

from    StdString       import String
import  StdTCPDef, StdChannels, StdEventTCP
from    StdReceiver     import Receivers, ReceiverType, RId
from    StdTCPChannels  import SelectSend, SelectReceive, getNrOfChannels

/*  If a string via a StringChannel is sent, then first the length of the string is
    sent, and then the string itself, e.g. sending the string "abc" will result in
    "3 abc\xD"
*/

//  ****************************************************************************
//  StringChannels to receive
//  ****************************************************************************

::  *StringRChannel_ a
::  *StringRChannel     :== StringRChannel_ String
::  *StringRChannels    =   StringRChannels [StringRChannel]
::  *StringChannelReceiver ls pst
 =  StringChannelReceiver
        (RId (ReceiveMsg String)) StringRChannel
        (ReceiverFunction (ReceiveMsg String) *(ls,pst))
        [ReceiverAttribute                    *(ls,pst)]

toStringRChannel        :: TCP_RChannel -> StringRChannel

instance Receivers      StringChannelReceiver
instance Receive        StringRChannel_
instance closeRChannel  StringRChannel_
instance MaxSize        StringRChannel_


//  ****************************************************************************
//  StringChannels to send
//  ****************************************************************************

::  *StringSChannel_ a
::  *StringSChannel     :== StringSChannel_ String
::  *StringSChannels    =   StringSChannels [StringSChannel]

toStringSChannel        ::  TCP_SChannel -> StringSChannel

instance Send StringSChannel_

//  For openSendNotifier, closeSendNotifier
instance accSChannel        StringSChannel_

//  For selectChannel
instance SelectSend         StringSChannels
instance SelectReceive      StringRChannels
instance getNrOfChannels    StringRChannels
```

A.37 StdSystem

```
definition module StdSystem


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdSystem defines platform dependent constants and functions.
//  ********************************************************************************


import  StdIOBasic


//  System dependencies concerning the file system.

dirseparator    :: Char     // Separator between folder- and filenames in a pathname
homepath        :: !String -> String
applicationpath :: !String -> String
/*  dirseparator
        is the separator symbol used between folder- and filenames in a file path.
    homepath
        prefixes the 'home' directory file path to the given file name.
    applicationpath
        prefixes the 'application' directory file path to the given file name.
    Use these directories to store preference/options/help files of an application.
*/

newlineChars    :: !String
/*      the newline characters in a textfile
*/

printSetupTypical   :: Bool

//  System dependencies concerning the time resolution

ticksPerSecond  :: Int
/*  ticksPerSecond returns the maximum timer resolution per second.
*/

//  System dependencies concerning the screen resolution.

mmperinch       :== 25.4

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

maxScrollWindowSize :: Size
maxFixedWindowSize  :: Size
/*  maxScrollWindowSize
        yields the range at which scrollbars are inactive.
    maxFixedWindowSize
        yields the range at which a window still fits on the screen.
*/
```

A.38 StdTCP

```
definition module StdTCP

import  StdChannels,
        StdTCPChannels,
        StdEventTCP,
        StdStringChannels,
        StdTCPDef
```

A.39 StdTCPChannels

```
definition module StdTCPChannels

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTCPChannels provides instances to use TCP.
//  ********************************************************************************

import  StdTCPDef, StdChannels
from    StdString        import String
from    StdIOCommon      import OkBool
from    StdIOBasic       import Void, :^:
from    tcp_bytestreams import TCP_SCharStream_, TCP_RCharStream_


//  ********************************************************************************
//  Listeners
//  ********************************************************************************

instance Receive            TCP_Listener_
instance closeRChannel      TCP_Listener_
/*  Receiving on a listener will accept a TCP_DuplexChannel. eom never becomes True
    for listeners.
*/

//  ********************************************************************************
//  TCP send channels
//  ********************************************************************************

instance Send               TCP_SChannel_

//  ********************************************************************************
//  TCP receive channels
//  ********************************************************************************

instance Receive            TCP_RChannel_
instance closeRChannel      TCP_RChannel_
instance MaxSize            TCP_RChannel_

//  ********************************************************************************
//  TCP char streams to receive
//  ********************************************************************************

::  *TCP_RCharStream    :== TCP_RCharStream_ Char
::  *TCP_RCharStreams   =   TCP_RCharStreams [TCP_RCharStream]

toRCharStream           :: !TCP_RChannel -> TCP_RCharStream

instance Receive            TCP_RCharStream_
instance closeRChannel      TCP_RCharStream_

//  ********************************************************************************
//  TCP char streams to send
//  ********************************************************************************

::  *TCP_SCharStream    :== TCP_SCharStream_ Char
::  *TCP_SCharStreams   =   TCP_SCharStreams [TCP_SCharStream]

toSCharStream           ::  !TCP_SChannel -> TCP_SCharStream

instance Send               TCP_SCharStream_

//  ********************************************************************************
//  establishing connections
//  ********************************************************************************
```

```
lookupIPAddress :: !String !*env
                -> (!Maybe IPAddress, !*env)
                |  ChannelEnv env
connectTCP_MT   :: !(Maybe !Timeout) !(!IPAddress,!Port) !*env
                -> (!TimeoutReport, !Maybe TCP_DuplexChannel, !*env)
                |  ChannelEnv env
openTCP_Listener:: !Port !*env
                -> (!OkBool, !Maybe TCP_Listener, !*env)
                |  ChannelEnv env
tcpPossible     :: !*env
                -> (!Bool, !*env)
                |  ChannelEnv env
/*  lookupIPAddress
        input String can be in dotted decimal form or alphanumerical. In the latter
        case the DNS is called.
    connectTCP
        tries to establish a TCP connection.
    openTCP_Listener
        to listen on a certain port.
    tcpPossible
        whether tcp can be started on this computer.
*/

//  ****************************************************************************
//  multiplexing
//  ****************************************************************************

selectChannel_MT:: !(Maybe !Timeout)         !*r_channels !*s_channels !*World
                -> (![(!Int, !SelectResult)],!*r_channels,!*s_channels,!*World)
                |  SelectReceive r_channels & SelectSend s_channels
/*  selectChannel_MT mbTimeout r_channels s_channels world
        determines the first channel on which "something happens".
        If the result is an empty list, then the timeout expired, otherwise each
        (who,what) element of the result identifies one channel in r_channels or
        s_channels. The what value determines whether available/eom/disconnected
        on the identified channel would have returned True.
        what==SR_Sendable indicates that it is possible to send non blocking on the
        identified channel. If r_channels contains r channels and if s_channels
        contains s channels, then the following holds:
            isMember what [SR_Available,SR_EOM]          => 0<=who<r
            isMember what [SR_Sendable ,SR_Disconnected] => 0<=who<s
*/

instance ==       SelectResult
instance toString SelectResult

/*  The following classes support the selectChannel_MT function:
*/
class SelectReceive channels where
    accRChannels    :: (PrimitiveRChannel -> (x, PrimitiveRChannel)) !*channels
                    -> (![x], !*channels)
    getRState       :: !Int !*channels !*World
                    -> (!Maybe !SelectResult, !*channels, !*World)
/*  accRChannels f channels
        applies a function on each channel in channels and returns a list which
        contains the result for each application.
    getRState
        applies available and eom on the channel which is identified by the Int
        parameter and returns SR_Available or SR_EOM or Nothing.
*/

class SelectSend channels where
    accSChannels    :: (TCP_SChannel -> (x, TCP_SChannel)) !*channels
                    -> (![x], !*channels)
    appDisconnected :: !Int !*channels !*World
                    -> (!Bool, !*channels, !*World)
```

```
/*  accSChannels
        applies a function on each channel in channels and returns a list which
        contains the result for each application.
    appDisconnected
        returns whether disconnected is True for the channel which is identified by
        the Int parameter.
*/

class getNrOfChannels channels :: !*channels -> (!Int, !*channels)
/*  getNrOfChannels channels
        returns the number of channels in channels.
*/

instance SelectReceive TCP_RChannels,TCP_Listeners,TCP_RCharStreams,Void
instance SelectReceive (:^: *x *y)      | SelectReceive, getNrOfChannels x
                                        & SelectReceive y

instance SelectSend TCP_SChannels,TCP_SCharStreams,Void
instance SelectSend (:^: *x *y)         | SelectSend, getNrOfChannels x
                                        & SelectSend y

instance getNrOfChannels TCP_RChannels,TCP_Listeners,TCP_RCharStreams,
                         TCP_SChannels,TCP_SCharStreams,Void
instance getNrOfChannels (:^: *x *y)    | getNrOfChannels x & getNrOfChannels y
```

A.40 StdTCPDef

```
definition module StdTCPDef

//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTCPDef provides basic definitions for using TCP.
//  ********************************************************************************

from    StdMaybe        import  Maybe
from    StdReceiverDef  import  Id, ReceiverFunction, ReceiverAttribute
from    StdOverloaded   import  toString, ==
from    StdChannels     import  DuplexChannel, ReceiveMsg, SendEvent
from    tcp             import  TCP_SChannel_,TCP_RChannel_,TCP_Listener_,IPAddress


::  *TCP_SChannel       :== TCP_SChannel_ ByteSeq
::  *TCP_RChannel       :== TCP_RChannel_ ByteSeq
::  *TCP_Listener       :== TCP_Listener_ (IPAddress, TCP_DuplexChannel)

::  Port                :== Int

::  *TCP_DuplexChannel  :== DuplexChannel *TCP_SChannel_ *TCP_RChannel_ ByteSeq

::  ByteSeq
//  A sequence of bytes

instance toString   ByteSeq
instance ==         ByteSeq
toByteSeq       ::  !x          -> ByteSeq  | toString x
byteSeqSize     ::  !ByteSeq    -> Int
//  byteSeqSize returns the size in bytes

instance toString IPAddress
//  returns ip address in dotted decimal form


//  ********************************************************************************
//  for event driven processing
//  ********************************************************************************

//  To receive byte sequences
::  *TCP_Receiver ls pst
 =   TCP_Receiver
        Id TCP_RChannel
        (ReceiverFunction (ReceiveMsg ByteSeq) *(ls,pst))
        [ReceiverAttribute                     *(ls,pst)]

::  SendNotifier sChannel ls pst
 =  SendNotifier
        sChannel
        (ReceiverFunction SendEvent *(ls,pst))
        [ReceiverAttribute          *(ls,pst)]

//  To accept new connections
::  *TCP_ListenerReceiver ls pst
 =   TCP_ListenerReceiver
        Id TCP_Listener
        ((ReceiveMsg (IPAddress,TCP_DuplexChannel)) -> *(*(ls,pst) -> *(ls,pst)))
        [ReceiverAttribute *(ls,pst)]

//  To receive characters
::  *TCP_CharReceiver ls pst
 =   TCP_CharReceiver
        Id TCP_RChannel (Maybe NrOfIterations)
        (ReceiverFunction (ReceiveMsg Char) *(ls,pst))
        [ReceiverAttribute                  *(ls,pst)]

/*  For efficency the receiver function of a TCP_CharReceiver will be called from
    a loop. Within this loop no other events can be handled. The NrOfIterations
    parameter limits the maximum number of iterations.
*/
::  NrOfIterations          :== Int
::  InetLookupFunction  st  :== (Maybe IPAddress)         -> *(st -> st)
::  InetConnectFunction st  :== (Maybe TCP_DuplexChannel) -> *(st -> st)

//  ****************************************************************************
//  for multiplexing
//  ****************************************************************************

:: *TCP_RChannels = TCP_RChannels [TCP_RChannel]
:: *TCP_SChannels = TCP_SChannels [TCP_SChannel]
:: *TCP_Listeners = TCP_Listeners [TCP_Listener]

::  *PrimitiveRChannel
 =  TCP_RCHANNEL TCP_RChannel
 |  TCP_LISTENER TCP_Listener

::  SelectResult
 =  SR_Available
 |  SR_EOM
 |  SR_Sendable
 |  SR_Disconnected
```

A.41 StdTime

```
definition module StdTime


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTime contains time related operations.
//  ****************************************************************************

from    StdOverloaded import <
from    ostick import Tick

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

wait                :: !Int .x -> .x

/*  wait n x suspends the evaluation of x modally for n ticks.
    If n<=0, then x is evaluated immediately.
*/

instance < Tick

intPlusTick         ::  !Int  !Tick -> Tick
tickDifference      ::  !Tick !Tick -> Int

class TimeEnv env where
    getBlinkInterval:: !*env -> (!Int,  !*env)
    getCurrentTime  :: !*env -> (!Time, !*env)
    getCurrentDate  :: !*env -> (!Date, !*env)
    getCurrentTick  :: !*env -> (!Tick, !*env)
/*  getBlinkInterval
        returns the time interval in ticks that should elapse between blinks of
        e.g. a cursor. This interval may be changed by the user while the
        interactive process is running!
    getCurrentTime
        returns the current Time.
    getCurrentDate
        returns the current Date.
    getCurrentTick
        returns the current Tick.
*/

instance TimeEnv World
```

A.42 StdTimer

```
definition module StdTimer


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTimer specifies all timer operations.
//  ********************************************************************************


import  StdTimerElementClass, StdMaybe
from    StdSystem   import ticksPerSecond
from    iostate     import PSt, IOSt


class Timers tdef where
    openTimer   :: .ls !(tdef .ls (PSt .l)) !(PSt .l)   -> (!ErrorReport,!PSt .l)
    getTimerType::      (tdef .ls .pst)                 -> TimerType
/*  Open a new timer.
    This function has no effect in case the interactive process already contains a
    timer with the same Id. In case TimerElements are opened with duplicate Ids, the
    timer will not be opened. Negative TimerIntervals are set to zero.
    In case the timer does not have an Id, it will obtain an Id which is fresh with
    respect to the current set of timers. The Id can be reused after closing this
    timer.
*/

instance Timers (Timer t)   | TimerElements t


closeTimer :: !Id !(IOSt .l) -> IOSt .l
/*  closeTimer closes the timer with the indicated Id.
*/


getTimers  ::     !(IOSt .l) -> (![(Id,TimerType)],!IOSt .l)
/*  getTimers returns the Ids and TimerTypes of all currently open timers.
*/


enableTimer         :: !Id !(IOSt .l) -> IOSt .l
disableTimer        :: !Id !(IOSt .l) -> IOSt .l
getTimerSelectState :: !Id !(IOSt .l) -> (!Maybe SelectState,!IOSt .l)
/*  (en/dis)ableTimer (en/dis)ables the indicated timer.
    getTimerSelectState yields the SelectState of the indicated timer. If the timer
    does not exist, then Nothing is yielded.
*/


setTimerInterval    :: !Id !TimerInterval   !(IOSt .l) -> IOSt .l
getTimerInterval    :: !Id                  !(IOSt .l)
                    -> (!Maybe TimerInterval,!IOSt .l)
/*  setTimerInterval
        sets the TimerInterval of the indicated timer.
        Negative TimerIntervals are set to zero.
    getTimerInterval
        yields the TimerInterval of the indicated timer.
        If the timer does not exist, then Nothing is yielded.
*/
```

A.43 StdTimerAttribute

```
definition module StdTimerAttribute


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTimerAttribute specifies which TimerAttributes are valid for each of the
//  standard timers.
//  Basic comparison operations and retrieval functions are also included.
//  ****************************************************************************


import StdTimerDef


/*  The following functions specify the valid attributes for each standard timer.
*/

isValidTimerAttribute :: !(TimerAttribute .st) -> Bool
/*  Timer           (y = valid, . = invalid)
    TimerFunction   y | TimerInit           y |
    TimerId         y | TimerSelectState    y |
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isTimerFunction         :: !(TimerAttribute .st) -> Bool
isTimerId               :: !(TimerAttribute .st) -> Bool
isTimerInit             :: !(TimerAttribute .st) -> Bool
isTimerSelectState      :: !(TimerAttribute .st) -> Bool


/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getTimerFun             :: !(TimerAttribute .st) -> TimerFunction .st
getTimerIdAtt           :: !(TimerAttribute .st) -> Id
getTimerInitFun         :: !(TimerAttribute .st) -> IdFun .st
getTimerSelectStateAtt  :: !(TimerAttribute .st) -> SelectState
```

A.44 StdTimerDef

```
definition module StdTimerDef


//  ******************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTimerDef contains the types to define the standard set of timers.
//  ******************************************************************************


import  StdIOCommon


::  Timer t ls pst  = Timer TimerInterval (t ls pst) [TimerAttribute *(ls,pst)]

::  TimerInterval
    :== Int

::  TimerAttribute  st                          // Default:
    =   TimerFunction       (TimerFunction st)  // \_ x->x
    |   TimerId             Id                  // no Id
    |   TimerInit           (IdFun st)          // no actions after opening timer
    |   TimerSelectState    SelectState         // timer Able

::  TimerFunction   st  :== NrOfIntervals -> st -> st
::  NrOfIntervals       :== Int

::  TimerType           :== String
::  TimerElementType    :== String
```

A.45 StdTimerElementClass

```
definition module StdTimerElementClass


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTimerElementClass define the standard set of timer element instances.
//  ********************************************************************************


import  StdIOCommon, StdTimerDef
from    iostate     import PSt, IOSt
from    timerhandle import TimerElementState


class TimerElements t where
    timerElementToHandles   :: !(t  .ls (PSt .l)) !(PSt .l)
            -> (![TimerElementState .ls (PSt .l)], !PSt .l)
    getTimerElementType     ::  (t  .ls .pst)
            -> TimerElementType

instance TimerElements (NewLS   t)   | TimerElements t
instance TimerElements (AddLS   t)   | TimerElements t
instance TimerElements (ListLS  t)   | TimerElements t
instance TimerElements NilLS
instance TimerElements ((:+:) t1 t2) | TimerElements t1
                                     & TimerElements t2
```

A.46 StdTimerReceiver

```
definition module StdTimerReceiver


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdTimerReceiver defines Receiver(2) timer element instances.
//  ********************************************************************************


import  StdReceiverDef, StdTimerElementClass


//  Receiver components for timers:
instance TimerElements  (Receiver  m  )
instance TimerElements  (Receiver2 m r)
```

A.47 StdWindow

```
definition module StdWindow


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdWindow defines functions on windows and dialogues.
//  ****************************************************************************


from    StdFunc         import St
import  StdControlClass, StdWindowDef
from    StdPSt          import PSt, IOSt


//  Functions applied to non-existent windows or unknown ids have no effect.
class Windows wdef where
    openWindow      :: .ls !(wdef .ls (PSt .l)) !(PSt .l)
                                -> (!ErrorReport,!PSt .l)
    getWindowType   ::      (wdef .ls .pst) -> WindowType

class Dialogs wdef where
    openDialog      :: .ls !(wdef .ls (PSt .l))    !(PSt .l)
                    ->  (  !ErrorReport,            !PSt .l)
    openModalDialog :: .ls !(wdef .ls (PSt .l))    !(PSt .l)
                    ->  (!(!ErrorReport,!Maybe .ls),!PSt .l)
    getDialogType   ::      (wdef .ls .pst) -> WindowType

/*  open(Window/Dialog) opens the given window(dialog).
    If the Window(Dialog) has no WindowIndex attribute (see StdWindowDef), then the
    new window is opened frontmost.
    If the Window(Dialog) has a WindowIndex attribute, then the new window is
    opened behind the window indicated by the integer index:
        Index value 1 indicates the top-most window.
        Index value M indicates the bottom-most modal window, if there are M modal
            windows.
        Index value N indicates the bottom-most window, if there are N windows.
        If index<M, then the new window is added behind the bottom-most modal window
            (at index M).
        If index>N, then the new window is added behind the bottom-most window
            (at index N).
    openModalDialog always opens a window at the front-most position.
    openWindow may not be permitted to open a window depending on its
        DocumentInterface (see the comments at the ShareProcesses instances in
        module StdProcess).
    In case the window does not have an Id, it will obtain a fresh Id. The Id can
    be reused after closing this window.
    In case a window with the same Id is already open the window will not be opened.
    In case controls are opened with duplicate Ids, the window will not be opened.
    openModalDialog terminates when:
        the modal dialog has been closed (by means of closeWindow), or the process
        has been terminated (by means of closeProcess). If the ErrorReport==NoError,
        then also the final local state of the modal dialog is returned, otherwise
        Nothing.
*/

instance Windows (Window c) | Controls c
instance Dialogs (Dialog c) | Controls c


closeWindow         :: !Id !(PSt .l) -> PSt .l
closeActiveWindow   ::     !(PSt .l) -> PSt .l
/*  If the indicated window is not an inactive modal dialog, then closeWindow closes
        the window/dialogue.
        In case the Id of the window was generated by open(Window/Dialog), it will
        become reusable for new windows/dialogues.
        No window is closed in case of an unknown Id.
    closeActiveWindow closes the currently active window/dialogue (see also
        getActiveWindow) if such a window could be found.
*/


setActiveWindow :: !Id      !( PSt .l) -> PSt .l
getActiveWindow ::          !(IOSt .l) -> (!Maybe Id,!IOSt .l)
/*  setActiveWindow makes the indicated window the active window.
        If there are modal dialogues, then the window will be placed behind the last
        modal dialog.
        setActiveWindow has no effect in case the window is unknown or is a modal
        dialog.
    getActiveWindow returns the Id of the window that currently has the input focus
        of the interactive process.
        Nothing is returned if there is no such window.
*/

setActiveControl:: !Id      !( PSt .l) -> PSt .l
getActiveControl::          !(IOSt .l) -> (!(!Bool,!Maybe Id),!IOSt .l)
/*  setActiveControl makes the indicated (PopUp/Edit/Custom/Compound)Control the
        active control. This succeeds only if its parent window is already active.
    getActiveControl returns the Id of the (PopUp/Edit/Custom/Compound)Control that
        currently has the input focus.
        The Boolean result is True only iff such a control could be found.
        Nothing is returned if the control has no Id attribute or if the Boolean
        result is False.
*/


stackWindow     :: !Id !Id  !(PSt .l) -> PSt .l
/*  stackWindow id1 id2 places the window with id1 behind the window with id2.
    If id1 or id2 is unknown, or id1 indicates a modal window, stackWindow does
    nothing.
    If id2 indicates a modal window, then the window with id1 is placed behind the
    last modal window.
*/

getWindowStack  :: !(IOSt .l) -> (![(Id,WindowType)],   !IOSt .l)
getWindowsStack :: !(IOSt .l) -> (![Id],                !IOSt .l)
getDialogsStack :: !(IOSt .l) -> (![Id],                !IOSt .l)
/*  getWindowStack returns the Ids and WindowTypes of all currently open windows,
    in the current stacking order starting with the active window.
    get(Windows/Dialogs)Stack is equal to getWindowStack, restricted to Windows
    instances and Dialogs instances respectively.
*/


getDefaultHMargin   :: !Bool    !(IOSt .l) -> ((Int,Int),       !IOSt .l)
getDefaultVMargin   :: !Bool    !(IOSt .l) -> ((Int,Int),       !IOSt .l)
getDefaultItemSpace :: !Bool    !(IOSt .l) -> ((Int,Int),       !IOSt .l)
getWindowHMargin    :: !Id      !(IOSt .l) -> (!Maybe (Int,Int),!IOSt .l)
getWindowVMargin    :: !Id      !(IOSt .l) -> (!Maybe (Int,Int),!IOSt .l)
getWindowItemSpace  :: !Id      !(IOSt .l) -> (!Maybe (Int,Int),!IOSt .l)
/*  getDefault((H/V)Margin)/ItemSpace) isWindow return the default values for the
        horizontal and vertical window (if isWindow)/dialogue (if (not isWindow))
        margins and item spaces.
    getWindow((H/V)Margin/ItemSpace) return the current horizontal and vertical
        margins and item spaces of the indicated window. These will have the default
        values in case they were not specified as an attribute.
        In case the window does not exist, Nothing is yielded.
*/


enableWindow            :: !Id  !(IOSt .l) -> IOSt .l
disableWindow           :: !Id  !(IOSt .l) -> IOSt .l
```

```
enableWindowMouse       :: !Id  !(IOSt .l) -> IOSt .l
disableWindowMouse      :: !Id  !(IOSt .l) -> IOSt .l
enableWindowKeyboard    :: !Id  !(IOSt .l) -> IOSt .l
disableWindowKeyboard   :: !Id  !(IOSt .l) -> IOSt .l
/*  (en/dis)ableWindow
        (en/dis)ables the indicated window.
    (en/dis)ableWindowMouse
        (en/dis)ables mouse handling of the indicated window.
    (en/dis)ableWindowKeyboard
        (en/dis)ables keyboard handling of the indicated window.
    Disabling a window overrules the SelectStates of its elements, which all become
    Unable.
    Reenabling the window reestablishes the SelectStates of its elements.
    The functions have no effect in case of invalid Ids or Dialogs instances.
    The latter four functions also have no effect in case the Window does not have
    the indicated attribute.
*/


getWindowSelectState        :: !Id  !(IOSt .l) ->(!Maybe SelectState,!IOSt .l)
getWindowMouseSelectState   :: !Id  !(IOSt .l) ->(!Maybe SelectState,!IOSt .l)
getWindowKeyboardSelectState:: !Id  !(IOSt .l) ->(!Maybe SelectState,!IOSt .l)
/*  getWindowSelectState
        yields the current SelectState of the indicated window.
    getWindow(Mouse/Keyboard)SelectState
        yields the current SelectState of the mouse/keyboard of the indicated
        window.
    The functions return Nothing in case of invalid Ids or Dialogs instances or if
    the Window does not have the indicated attribute.
*/


getWindowMouseStateFilter   :: !Id                           !(IOSt .l)
                            -> (!Maybe MouseStateFilter,    ! IOSt .l)
getWindowKeyboardStateFilter:: !Id                           !(IOSt .l)
                            -> (!Maybe KeyboardStateFilter, ! IOSt .l)
setWindowMouseStateFilter   :: !Id !MouseStateFilter         !(IOSt .l)
                                                            -> IOSt .l
setWindowKeyboardStateFilter:: !Id !KeyboardStateFilter      !(IOSt .l)
                                                            -> IOSt .l
/*  getWindow(Mouse/Keyboard)StateFilter yields the current
        (Mouse/Keyboard)StateFilter of the indicated window. Nothing is yielded in
        case the window does not exist or has no Window(Mouse/Keyboard) attribute.
    setWindow(Mouse/Keyboard)StateFilter replaces the current
        (Mouse/Keyboard)StateFilter of the indicated window. If the indicated window
        does not exist the function has no effect.
*/


appWindowPicture:: !Id !.(IdFun *Picture) !(IOSt .l) -> IOSt .l
accWindowPicture:: !Id !.(St *Picture .x) !(IOSt .l) -> (!Maybe .x,!IOSt .l)
/*  (app/acc)WindowPicture applies the given drawing function to the Picture of
    the indicated window (behind all controls).
    Both functions have no effect in case the window is unknown or is a Dialog.
    In that case, accWindowPicture also returns Nothing.
*/


updateWindow    :: !Id !(Maybe ViewFrame)   !(IOSt .l) -> IOSt .l
/*  updateWindow applies the WindowLook attribute function of the indicated window.
    The Look attribute function is applied to the following arguments:
    The current SelectState of the window, and
    the UpdateState argument
        {oldFrame=viewframe,newFrame=viewframe,updArea=[frame]}
    where viewframe is the current ViewFrame of the window;
    and frame depends on the optional ViewFrame argument:
        in case of (Just rectangle):
            the intersection of viewframe and rectangle.
        in case of Nothing:
            viewframe.
    updateWindow has no effect in case of unknown windows, or if the indicated
    window is a Dialog, or the optional viewframe argument is an empty rectangle.
*/

setWindowLook   :: !Id !Bool !(!Bool,!Look) !(IOSt .l) -> IOSt .l
getWindowLook   :: !Id                      !(IOSt .l)
                      -> (!Maybe (Bool,Look),!IOSt .l)
/*  setWindowLook sets the (render/look) of the indicated window.
        The window is redrawn only if the Boolean argument is True.
        setWindowLook has no effect in case the window does not exist, or is a
        Dialog.
    getWindowLook returns the (Just (render/look)) of the indicated window.
        In case the window does not exist, or is a Dialog, or has no WindowLook
        attribute, the result is Nothing.
*/


setWindowPos    :: !Id !ItemPos !(IOSt .l) -> IOSt .l
getWindowPos    :: !Id          !(IOSt .l) -> (!Maybe Vector2,!IOSt .l)
/*  setWindowPos places the window at 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.
        If the ItemPos argument is one of (LeftOf/RightTo/Above/Below)Prev, then the
        previous window is the window that is before the window in the current
        stacking order.
        If the window is frontmost, setWindowPos has no effect. setWindowPos also
        has no effect if the window would be moved outside the screen, or if the Id
        is unknown or refers to a modal Dialog.
    getWindowPos returns the current item offset position of the indicated window.
        The corresponding ItemPos is (LeftTop,OffsetVector offset). Nothing is
        returned in case the window does not exist.
*/


moveWindowViewFrame :: !Id Vector2 !(IOSt .l) -> IOSt .l
/*  moveWindowViewFrame moves the orientation of the view frame of the indicated
    window over the given vector, and updates the window if necessary. The view
    frame is not moved outside the ViewDomain of the window.
    In case of unknown Id, or of Dialogs, moveWindowViewFrame has no effect.
*/

getWindowViewFrame  :: !Id !(IOSt .l) -> (!ViewFrame,!IOSt .l)
/*  getWindowViewFrame returns the current view frame of the window in terms of the
    ViewDomain. Note that in case of a Dialog, getWindowViewFrame returns
    {zero,size}.
    In case of unknown windows, the ViewFrame result is zero.
*/

setWindowViewSize   :: !Id !Size    !(IOSt .l) -> IOSt .l
getWindowViewSize   :: !Id          !(IOSt .l) -> (!Size,!IOSt .l)
/*  setWindowViewSize
        sets the size of the view frame of the indicated window as given, and
        updates the window if necessary. The size is fit between the minimum size
        and the screen dimensions.
        In case of unknown Ids, or of Dialogs, setWindowViewSize has no effect.
    getWindowViewSize yields the current size of the view frame of the indicated
        window. If the window does not exist, zero is returned.
*/

setWindowOuterSize  :: !Id !Size    !(IOSt .l) -> IOSt .l
getWindowOuterSize  :: !Id          !(IOSt .l) -> (!Size,!IOSt .l)
/*  setWindowOuterSize
        sets the size of the outer frame of the indicated window as given, and
        updates the window if necessary. The size is fit between the minimum size
        and the screen dimensions.
        In case of unknown Ids, or of Dialogs, setWindowOuterSize has no effect.
    getWindowOuterSize yields the current size of the outer frame of the indicated
        window. If the window does not exist, zero is returned.
*/

setWindowViewDomain :: !Id ViewDomain   !(IOSt .l) -> IOSt .l
getWindowViewDomain :: !Id              !(IOSt .l)
                   -> (!Maybe ViewDomain,!IOSt .l)
/*  setWindowViewDomain
        sets the view domain of the indicated window as given. The window view frame
        is moved such that a maximum portion of the view domain is visible. The
        window is not resized.
        In case of unknown Ids, or of Dialogs, setWindowViewDomain has no effect.
    getWindowViewDomain
        returns the current ViewDomain of the indicated window.
        Nothing is returned in case the window does not exist or is a Dialog.
*/

setWindowScrollFunction :: !Id Direction ScrollFunction !(IOSt .l) -> IOSt .l
getWindowScrollFunction :: !Id Direction                !(IOSt .l)
                               -> (!Maybe ScrollFunction,!IOSt .l)
/*  setWindowScrollFunction
        changes the current scroll function of the indicated Window and direction
        only if the indicated window already had a scroll bar in that direction.
        In all other cases setWindowScrollFunction has no effect.
    getWindowScrollFunction
        returns the current scroll function in the argument direction if the
        indicated Window had one.
        In all other cases Nothing is returned.
*/


setWindowTitle  :: !Id Title       !(IOSt .l) -> IOSt .l
setWindowOk     :: !Id Id          !(IOSt .l) -> IOSt .l
setWindowCancel :: !Id Id          !(IOSt .l) -> IOSt .l
setWindowCursor :: !Id CursorShape !(IOSt .l) -> IOSt .l
getWindowTitle  :: !Id             !(IOSt .l) -> (!Maybe Title,      !IOSt .l)
getWindowOk     :: !Id             !(IOSt .l) -> (!Maybe Id,         !IOSt .l)
getWindowCancel :: !Id             !(IOSt .l) -> (!Maybe Id,         !IOSt .l)
getWindowCursor :: !Id             !(IOSt .l) -> (!Maybe CursorShape,!IOSt .l)
/*  setWindow(Title/Ok/Cancel/Cursor) set the indicated window attributes.
        In case of unknown Ids, these functions have no effect.
    getWindow(Title/Ok/Cancel/Cursor) get the indicated window attributes.
        In case of unknown Ids, the result is Nothing.
*/
```

A.48 StdWindowAttribute

```
definition module StdWindowAttribute


//  ****************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdWindowAttribute specifies which WindowAttributes are valid for Windows
//  and Dialogs.
//  Basic comparison operations and retrieval functions are also included.
//  ****************************************************************************


import StdWindowDef


/*  The following functions specify the valid attributes for each standard window.
*/

isValidWindowAttribute  :: !(WindowAttribute .st) -> Bool
/*  Window              (y = valid, . = invalid)
    WindowActivate      y | WindowInit          y | WindowPen           y |
    WindowCancel        . | WindowInitActive    y | WindowPos           y |
    WindowClose         y | WindowItemSpace     y | WindowSelectState   y |
    WindowCursor        y | WindowKeyboard      y | WindowViewDomain    y |
    WindowDeactivate    y | WindowLook          y | WindowViewSize      y |
    WindowHMargin       y | WindowMouse         y | WindowVMargin       y |
    WindowHScroll       y | WindowOk            . | WindowVScroll       y |
    WindowId            y | WindowOrigin        y |
    WindowIndex         y | WindowOuterSize     y |
*/

isValidDialogAttribute  :: !(WindowAttribute .st) -> Bool
/*  Dialog              (y = valid, . = invalid)
    WindowActivate      y | WindowInit          y | WindowPen           . |
    WindowCancel        y | WindowInitActive    y | WindowPos           y |
    WindowClose         y | WindowItemSpace     y | WindowSelectState   . |
    WindowCursor        . | WindowKeyboard      . | WindowViewDomain    . |
    WindowDeactivate    y | WindowLook          . | WindowViewSize      y |
    WindowHMargin       y | WindowMouse         . | WindowVMargin       y |
    WindowHScroll       . | WindowOk            y | WindowVScroll       . |
    WindowId            y | WindowOrigin        . |
    WindowIndex         y | WindowOuterSize     y |
*/


/*  The following functions return True only iff the attribute equals the
    indicated name.
*/
isWindowActivate            :: !(WindowAttribute .st) -> Bool
isWindowCancel              :: !(WindowAttribute .st) -> Bool
isWindowClose               :: !(WindowAttribute .st) -> Bool
isWindowCursor              :: !(WindowAttribute .st) -> Bool
isWindowDeactivate          :: !(WindowAttribute .st) -> Bool
isWindowHMargin             :: !(WindowAttribute .st) -> Bool
isWindowHScroll             :: !(WindowAttribute .st) -> Bool
isWindowId                  :: !(WindowAttribute .st) -> Bool
isWindowIndex               :: !(WindowAttribute .st) -> Bool
isWindowInit                :: !(WindowAttribute .st) -> Bool
isWindowInitActive          :: !(WindowAttribute .st) -> Bool
isWindowItemSpace           :: !(WindowAttribute .st) -> Bool
isWindowKeyboard            :: !(WindowAttribute .st) -> Bool
isWindowLook                :: !(WindowAttribute .st) -> Bool
isWindowMouse               :: !(WindowAttribute .st) -> Bool
isWindowOk                  :: !(WindowAttribute .st) -> Bool
isWindowOrigin              :: !(WindowAttribute .st) -> Bool
```

```
isWindowOuterSize           :: !(WindowAttribute .st) -> Bool
isWindowPen                 :: !(WindowAttribute .st) -> Bool
isWindowPos                 :: !(WindowAttribute .st) -> Bool
isWindowSelectState         :: !(WindowAttribute .st) -> Bool
isWindowViewDomain          :: !(WindowAttribute .st) -> Bool
isWindowViewSize            :: !(WindowAttribute .st) -> Bool
isWindowVMargin             :: !(WindowAttribute .st) -> Bool
isWindowVScroll             :: !(WindowAttribute .st) -> Bool


/*  The following functions return the attribute value if appropriate.
    THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
    attribute.
*/
getWindowActivateFun        :: !(WindowAttribute .st) -> IdFun .st
getWindowCancelAtt          :: !(WindowAttribute .st) -> Id
getWindowCloseFun           :: !(WindowAttribute .st) -> IdFun .st
getWindowCursorAtt          :: !(WindowAttribute .st) -> CursorShape
getWindowDeactivateFun      :: !(WindowAttribute .st) -> IdFun .st
getWindowHMarginAtt         :: !(WindowAttribute .st) -> (Int,Int)
getWindowHScrollFun         :: !(WindowAttribute .st) -> ScrollFunction
getWindowIdAtt              :: !(WindowAttribute .st) -> Id
getWindowIndexAtt           :: !(WindowAttribute .st) -> Int
getWindowInitFun            :: !(WindowAttribute .st) -> IdFun .st
getWindowInitActiveAtt      :: !(WindowAttribute .st) -> Id
getWindowItemSpaceAtt       :: !(WindowAttribute .st) -> (Int,Int)
getWindowKeyboardAtt        :: !(WindowAttribute .st) -> ( KeyboardStateFilter
                                                         , SelectState
                                                         , KeyboardFunction .st
                                                         )
getWindowLookAtt            :: !(WindowAttribute .st) -> (Bool,Look)
getWindowMouseAtt           :: !(WindowAttribute .st) -> ( MouseStateFilter
                                                         , SelectState
                                                         , MouseFunction .st
                                                         )
getWindowOkAtt              :: !(WindowAttribute .st) -> Id
getWindowOriginAtt          :: !(WindowAttribute .st) -> Point2
getWindowOuterSizeAtt       :: !(WindowAttribute .st) -> Size
getWindowPenAtt             :: !(WindowAttribute .st) -> [PenAttribute]
getWindowPosAtt             :: !(WindowAttribute .st) -> ItemPos
getWindowSelectStateAtt     :: !(WindowAttribute .st) -> SelectState
getWindowViewDomainAtt      :: !(WindowAttribute .st) -> ViewDomain
getWindowViewSizeAtt        :: !(WindowAttribute .st) -> Size
getWindowVMarginAtt         :: !(WindowAttribute .st) -> (Int,Int)
getWindowVScrollFun         :: !(WindowAttribute .st) -> ScrollFunction
```

A.49 StdWindowDef

```
definition module StdWindowDef


//  ********************************************************************************
//  Clean Standard Object I/O library, version 1.2
//
//  StdWindowDef contains the types to define the standard set of windows and
//  dialogues.
//  ********************************************************************************


import  StdIOCommon, StdPictureDef


::  Dialog c ls pst = Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]
::  Window c ls pst = Window Title (c ls pst) [WindowAttribute *(ls,pst)]

::  WindowAttribute st                         // Default:
 // Attributes for Windows and Dialogs:
    =   WindowActivate      (IdFun st)          // id
    |   WindowClose         (IdFun st)          // user can't close window
    |   WindowDeactivate    (IdFun st)          // id
    |   WindowHMargin       Int Int             // system dependent
    |   WindowId            Id                  // system defined id
    |   WindowIndex         Int                 // open front-most
    |   WindowInit          (IdFun st)          // no actions after opening window
    |   WindowInitActive    Id                  // system dependent
    |   WindowItemSpace     Int Int             // system dependent
    |   WindowOuterSize     Size                // screen size
    |   WindowPos           ItemPos             // system dependent
    |   WindowViewSize      Size                // screen size
    |   WindowVMargin       Int Int             // system dependent
 // Attributes for Dialog only:
    |   WindowCancel        Id                  // no cancel  (Custom)ButtonControl
    |   WindowOk            Id                  // no default (Custom)ButtonControl
 // Attributes for Windows only:
    |   WindowCursor        CursorShape         // no change of cursor
    |   WindowHScroll       ScrollFunction      // no horizontal scrolling
    |   WindowKeyboard      KeyboardStateFilter SelectState (KeyboardFunction st)
                                                // no keyboard input
    |   WindowLook          Bool Look           // show system dependent background
    |   WindowMouse         MouseStateFilter    SelectState (MouseFunction    st)
                                                // no mouse input
    |   WindowOrigin        Point2              // left top of picture domain
    |   WindowPen           [PenAttribute]      // default pen attributes
    |   WindowSelectState   SelectState         // Able
    |   WindowViewDomain    ViewDomain          // {zero,max range}
    |   WindowVScroll       ScrollFunction      // no vertical   scrolling

::  WindowType
    :== String
```

Index