The Clock Language

Reference Manual

T.C. Nicholas Graham
York University
graham@cs.yorku.ca

Electronic Technical Report CS-ETR-95-01

Abstract: This document serves as preliminary documentation for the Clock language. Clock is a purely declarative language supporting the development of highly interactive graphical user interfaces. For now, this document concentrates on the syntax of the language and the predefined functions the language provides.

Acknowledgements

Clock was designed and implemented by T.C. Nicholas Graham, Tore Urnes, Catherine A. Morton and Roy Nejabi. Other contributers to the Clock project were Herbert Damker and Gekun Song.

We gratefully acknowledge the financial support of the ITRC, NSERC, and the Royal Norwegian Research Council.

Context Free Syntax of Clock

```
componentDef   ::=   { definition }

definition     ::=   equation
                |    typeDefinition
                |    dataDefinition

equation       ::=   id { simplePattern }+ = expn .

typeDefinition ::=   type typeId = typeSpec .

dataDefinition ::=   data constructorId =
                         constructorId { typeSpec }
                       { | constructorId { typeSpec }} .

expn           ::=   simpleExpn
                |    simpleExpn simpleExpn
                |    debug simpleExpn


simpleExpn     ::=   id
                |    literal
                |    constructorId
                |    fn { simplePattern }+ -> expn end fn
                |    if expn then expn
                     { elsif expn then expn }
                     else expn
                     end if
                |    case expn of
                       { pattern -> expn }+
                     end case
                |    let { pattern = expn }+ in expn end let
                |    [ expn { , expn } ]
                |    unaryOp simpleExpn
                |    simpleExpn binaryOp simpleExpn
                |    ( expn )
```

simplePattern	`::=`	`_`
	|	*id*
	|	*literal*
	|	*constructorId*
	|	**`[`** *pattern* { **`,`** *pattern* } **`]`**
	|	**`(`** *pattern* **`)`**
pattern	`::=`	*simplePattern*
	|	*constructorId* { *simplePattern* }
	|	*simpleExpn* **`:`** *simpleExpn*
unaryOp	`::=`	**`-`**
binaryOp	`::=`	**`+ | - | * | div | mod`**
	|	**`= | ~= | < | > | <= | >=`**
	|	**`and | or | not`**
	|	`@ | #` **`| ++`**
typeSpec	`::=`	**`Num | String | Bool`**
	|	*typeId*
	|	**`[`** *typeSpec* **`]`**
	|	**`(`** *typeSpec* { **`,`** *typeSpec* } **`)`**

Lexical Rules

numLiteral	`::=`	`[0-9]*`
stringLiteral	`::=`	**`"`** `[any char]*` **`"`**
boolLiteral	`::=`	**`True | False`**
id	`::=`	`[a-z]* [a-zA-Z0-9_]* [']*`
constructorId	`::=`	`[A-Z]* [a-zA-Z0-9_]* [']*`
typeId	`::=`	`[A-Z]* [a-zA-Z0-9_]* [']*`

Predefined Functions

List manipulation functions:

```
take :: Num -> [a] -> [a]
```

Takes the first n elements of the given list; ie, `take 3 [1,2,3,4] = [1,2,3]`.

```
: :: a -> [a] -> [a]
```

The 'cons' function places a given element onto the front of a list.

```
append :: [a] -> a -> [a]
```

Appends the given value to the end of the given list.

```
++ :: [a] -> [a] -> [a]
```

Concatenates two lists.

```
assign :: [a] -> Num -> a -> [a]
```

Gives a list where the element at the given position is replaced with the new given element. Ie, `assign [10,11,12,13] 2 1001 = [10,1001,12,13]`.

```
@ :: [a] -> Num -> a
```

Returns the value at the given position of the list. Ie, `[10,9,8] @ 2 = 9`. List indeces are based at 1. List indeces out of bounds result in an error.

```
# :: [a] -> (Num,Num) -> [a]
```

Returns all elements in the given list between the two given numbers. Ie, `[1,2,3,4,5,6] # (2,5) = [2,3,4,5]`. If the right index exceeds the left index by 1, the null list is returned: ie, `[1,2,3] # (1,0) = []`. If either index is otherwise out of bounds, an error results.

```
length :: [a] -> Num
```

Returns the number of elements in the given list.

Dictionary functions:

A dictionary is a list of key/value pairs. The dictionary functions make it easy to retrieve values from a dictionary given a key.

```
        enterDict :: a -> b -> [(a,b)] -> [(a,b)]
```

Given a key *a*, a value b, and a dictionary, returns a new dictionary where b has been entered under key a. If the key a already exists in the given dictionary, this operation overwrites the old value.

```
        lookup :: a -> [(a,b)] -> b
```

Given a key *a*, finds the first value b with that key in the dictionary.

```
        lookupPos a -> [(a,b)] -> Num
```

Given a key a, finds the position of the first occurence of that key in the dictionary.

```
        lookupAll :: a -> [(a,b)] -> [b]
```

Given a key a, returns the list of all values b with that key in the dictionary.

```
        index a -> [a] -> Num
```

In a simple list, finds the position of the first occurence of a value in that list. Ie, `index 3 [1,3,10,4] = 2`.

Combinators:

```
        map :: (a->b) -> [a] -> [b]
```

Given a function *f* and a list of elements *as*, gives the result of applying *f* to each element of *as*.

```
        fold :: (a->a->a) -> a -> [a] -> a
```

Given a function f, initial value a0, and list of elements [a1,a2,...,an], returns the result of applying f a0 (f a1 (f a2 (..... (f an-1 an))).

Mathematical functions:

```
min :: Num -> Num -> Num
max :: Num -> Num -> Num
```

Return the min/max of the given pair of numbers.

Tuple functions:

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

Return the first/second items of a tuple.

String manipulation functions:

```
ord :: String -> Num
```

Returns the ordinal number of the first character of the given string; eg, in ASCII, `ord ("A") = 65`. The given string must not be null.

```
chr :: Num -> String
```

Gives as a string the character value of the given integer. Eg, `ord (65) = "A"`.

```
numstr :: Num -> String
```

Converts an integer into a string. Eg, `numstr 123 = "123"`.

```
strnum :: String -> Num
```

Converts a string into an integer. Eg, `strnum "123" = 123`. The string must contain a valid integer; currently this is not checked, so if the string is not valid, a segmentation violation style of crash will result.

Date/time manipulation functions:

Date and Time are predefined abstract data types representing a date/time. These functions implement computations based on dates. Note that in Clock, there is currently no way of accessing the current date or time.

```
textToDate :: String -> String -> String -> Date
```

Creates a date type from the given day, month and year. The month must be the full English name of the month: "January", "February", etc. For example, `textToDate "10" "January" "1995"` gives the date for Jan 10, 1995.

```
dateToString :: Date -> String
```

Gives a string representation of a date.

```
dayOfWeek :: Date -> String
```

Returns what the weekday is of the current date.

```
daysInMonth :: Date -> Num
```

Returns how many days are in the current month.

```
daysInYear :: Date -> Num
```

Returns how many days are in the given year. Does a reasonable job with leap years, but not perfect.

```
tomorrow :: Date -> Date
```

Gives the date following the given date.

```
yesterday :: Date -> Date
```

Gives the date preceding the given date.

```
stringToTime :: String -> Time
```

Converts a string to a time. Time strings can be in two formats: 24 hour, such as `stringToTime "13:45"` or 12 hour, such as `stringToTime "1:45 PM"`. This function is very sensitive to format, and will crash if there is any deviation.

```
timeToString :: Time -> Num -> String
```

Returns a string version of the given time. The time may be in 12 or 24 hour format, as specified by the numeric parameter.

```
earlier :: (Date,Time) -> (Date,Time) -> Boolean
```

Determines whether the first of two given dates+times occurs earlier than the second.

Environment query functions:

```
getenv :: String -> String
```

Given the string name of an environment variable, returns the variable's value. Eg, `getenv “PRINTER”` returns the name of the current printer.

```
homedir :: String
```

Returns the directory in which this program is located. This function is useful if the program makes use of auxiliary files located in the program directory (e.g., image files.)

Views in Clock

Clock provides a simple language from which graphical displays can be constructed. While the language is simple, the abstraction powers of functional programming can allow complex displays to be easily constructed. Currently, the view language does not support all forms of graphical primitives or layout that would be desirable; the most glaring problem right now is the lack of any circle or elipse primitives.

This presentation begins by outlining the primitive constructs from which views are built, and then describes the predefined functions that simplify view manipulation.

Primitive Constructs

Views are elements of the data type `DisplayView`. This data type has the following definition:

```
data DisplayView =                     | Font FontName DisplayView
    Views [DisplayView]                | FontColour Colour DisplayView
  | Line Coord Coord                   | Inverted DisplayView
  | Arrow Coord Coord                  | LineWidth Num DisplayView
  | At Coord Coord DisplayView         | LineStyle Num DisplayView
  | Box DisplayView                    | LineColour Colour DisplayView
  | Text String                        | SaveProps DisplayView
  | NumText Num                        | RestoreProps DisplayView
  | CharText Char
  | BooleanText Bool                   | DarkReliefShade Colour DisplayView
  | InstanceOf SubViewName SubViewId   | LightReliefShade Colour DisplayView
                                       | ReliefWidth Num DisplayView
  | BorderStyle Num DisplayView        | Relief String DisplayView
  | BorderColour Colour DisplayView    | PolyLine [Coord]
  | BorderWidth Num DisplayView        | Pile [(Num, DisplayView)]
  | FillPattern Num DisplayView        | Image String
  | FillColour Colour DisplayView      | Crop Coord Coord DisplayView.
```

Views in Clock are therefore values that are built from data constructors. For example, the view `Box (Text “Hello world”)` represents the text “Hello world” drawn with a box around it.

Text, NumText, CharText, BooleanText

These constructors are used to display text. `Text “Hello”` displays the text “Hello”. `NumText 123` displays the text “123”. `CharText 65` displays the text “A”. `BooleanText True` displays the text “True”. The text is drawn in the current font and current font colour. For example, `Font largeItalicFont (FontColour red (Text “Hello world”))` draws the text “Hello world” in large, red, italic text.

Fonts are strings in the standard X font description format. The Unix command `xlsfonts` will give you the complete list of X fonts available on your server. The file `$CLOCKSYS/Source/clocklib/viewUtils` gives a list of predefined font names.

Colours have the definition:

```
type Colour = (Num,Num,Num).
```

That is, colours are triples consisting of an integer value representing the red, green and blue intensities of the desired colour. The file `$CLOCKSYS/Source/clocklib/colourUtils` gives a list of several hundred predefined colour names.

Box

A box surrounds whatever its parameter view is. E.g., `Box (Text “Hello”)` draws a box correctly sized to fit the text “Hello”. By default, boxes are drawn in black, with a width of one pixel. The constructors `BorderStyle`, `BorderColour`, `BorderWidth`, `FillPattern` and `FillColour` permit the attributes of boxes to be adjusted. For example,

```
BorderWidth 3 (
  FillColour yellow (
    BorderColour green (
      Box (Text “Hello”)
    )
  )
)
```

displays the text “Hello” surrounded by a 3 pixel wide green border, on a yellow background. Border styles may be: `solidBorder` (default), `dashedBorder`, or `doubleDashedBorder`. Fill patterns may be: `solidFill` (default), `hashedFill`, `screenDoorFill`, or `tiledFill`.

At

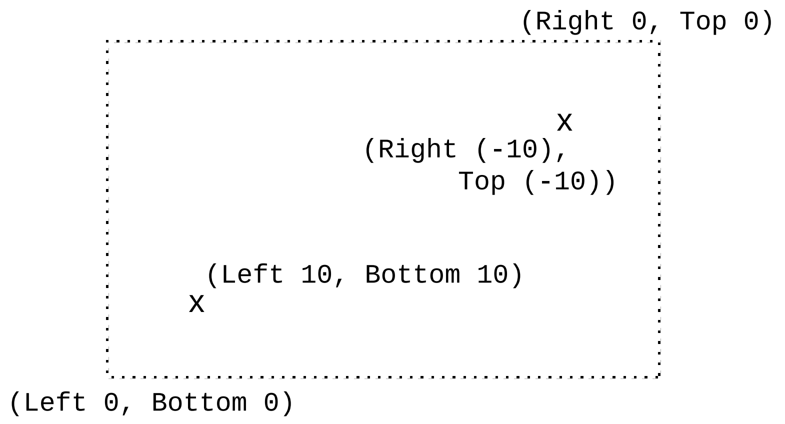
To place Clock primitives on the display, coordinates must be used. Coordinates refer to positions within the current *canvas*. The contents of each box construct is considered to be a separate canvas with its own coordinate space, thus giving Clock a hierarchical graphics system. The *Coord* data type defines the form of coordinates:

```
type Coord = (Ordinate,Ordinate).
type Offset = Num.
type OrdinateLabel = Num.

data Ordinate =
    XBaseOffset OrdinateLabel Offset | YBaseOffset OrdinateLabel Offset
```

```
  | Left Offset  | Bottom Offset | Right Offset | Top Offset
  | XSomewhere | YSomewhere.
```

The simplest form of coordinate is the absolute coordinate. The form `(Left 0, Bottom 0)` is used to specify the position 0 units to the left of the lower-left corner of the coordinate space. Similarly, the form `(Right 0, Top 0)` specifies the upper-right corner of the coordinate space. This form of absolute coordinate allows primitives to be located without knowing the position or size of the coordinate space itself:



Primitives can be positioned with the `At` constructor. For example:

```
At (Left 10, Bottom 10) (Right (-10), Top (-10)) (Box noView)
```

would display a box whose size would be adjusted to always be 10 pixels from the border of the canvas.

As a convenience, the functions:

```
x xpos = Left x.
y ypos = Bottom y.
origin = (x 0, y 0).
```

are predefined. These functions allow simpler forms for coordinates that are expressed in terms of the lower-left corner of the canvas.

Sometimes, the programmer does not know the position of both coordinates for a primitive. For example, when positioning text, it is possible to write:

```
At (x 20, y 13) (XSomewhere, YSomewhere) (Text “Hello”)
```

This states simply that the lower-left corner of the text is to be positioned at (20, 13) within the current coordinate space, and the upper-right corner’s position is not specified.

As a convenience, the function:

```
stretching = (XSomewhere, YSomewhere).
```

is defined. With this form, text can be simply positioned as:

```
At (x 20, y 13) stretching (Text “Hello”)
```

Views

The views constructor allows multiple primitives to be placed in the same canvas. For example,

```
Views [
   At (x 10, y 10) stretching (Text “Hello”),
   At (x 100, y 10) stretching (Text “there”)
]
```

places two text primitives in the same canvas.

In specifying the positions of primitives within a `Views` construct, primitive coordinates can also be expressed relative to the positions of other primitives. For example, to place two texts on the display separated by 10 pixels, one writes:

```
Views [
   At (x 10, y 10) (XBaseOffset 1 0, YSomewhere) (Text “Hello”),
   At (XBaseOffset 1 10, y 10) stretching (Text “there”)
]
```

The `XBaseOffset` constructor takes two parameters: the first is a numeric name for a position on the display. In this case, the name “1” is used to refer to the X-position of the right extent of the text “Hello”, wherever that may be. The second parameter is an offset (positive or negative) from that position. Therefore, the text “there” is positioned at `XBaseOffset 1 10`, which is 10 pixels to the right of the rightmost position of the text “Hello”.

Line, Arrow, PolyLine

These constructors allow lines and arrows to be drawn. For example,

```
Line origin (x 10, y 10)
```

is a line stretching from the origin to the position (10, 10) in the current canvas. The constructors `LineWidth`, `LineStyle` and `LineColour` allow the attributes of lines to be set. Currently, arrows don’t have arrow heads.

PolyLine allows polygons to be built from a list of coordinates. The current fill colour is applied to the region contained by the polyline.

Arrow may have unpredictable effects in the current version of Clock.

SaveProps, RestoreProps

`SaveProps` saves the current state of all properties (line colour, font, fill colour, etc). `RestoreProps` restores the last saved set of properties. For example, to define a function to draw a green box around a given view, we would write:

```
greenBox v = SaveProps (BorderColour green (RestoreProps v)).
```

Ie, we save the properties prior to changing the border colour, and then restore them before evaluating `v`.

Image

The `Image` constructor allows the inclusion of JPEG images in Clock programs. For example, `Image "foo.jpg"` reads the image contained in the file "foo.jpg". Images are first class Clock views; for example, `Box (Image "foo.jpg")` draws a box surrounding the image.

Clip

Clip clips the given view to fit within the given coordinates. For example,

```
Clip (x 10, y 10) (x 30, y 30) (Image "foo.jpg")
```

displays the portion of the image contained in the region (10,10) -> (30,30).

Relief

Relief allows the current view to be given the 3D style of relief commonly used in modern toolkits. The form `Relief "raised" v` draws v slightly raised over the surrounding view; `Relief "sunken" v` draws v slightly sunken. Relief properties include the `ReliefWidth`, the number of pixels wide the relief shading will be, and `DarkReliefShade` and `LightReliefShade`, which are used to specify the colours to be used in relief shading.

Predefined View Functions

The last section introduced the low level primitives from which all Clock views are constructed. In fact, Clock programmers do much of their view construction using predefined functions that abstract from the detailed level of the primitives. These definitions are all contained in the files `$CLOCKSYS/Source/clocklib/viewTypes` and `$CLOCKSYS/Source/clocklib/viewUtils`. Reading through these files is a very useful way of finding out how to write sophisticated view functions in Clock.

Coordinates

```
x :: Num -> Ordinate
y :: Num -> Ordinate
```

Allow the specification of coordinates relative to the lower-left corner of the current canvas. Eg, `(x 10, y 20)` is a coordinate.

```
xOrigin :: Ordinate
yOrigin :: Ordinate
```

Return the lower left X- and Y-positions of the the current canvas.

```
origin :: Coordinate
```

Returns the lower-left coordinate of the current canvas.

```
mostHigh :: Ordinate
mostRight :: Ordinate
topRight :: Coordinate
```

Return the positions of the top-right extent of the current canvas.

```
stretching :: Coordinate
```

Returns a coordinate at an unspecified location.

```
noView :: DisplayView
```

Returns no view at all. Useful in, for example `Box noView`, a box containing nothing.

Layout

```
    beside :: [DisplayView] -> DisplayView
    above :: [DisplayView] -> DisplayView
```

Given a list of display views, returns the views laid out horizontally/vertically respectively. For example, `beside [Text “a”, Text “b”, Text “c”]` would display the text “abc”.

```
    size :: (Num,Num) DisplayView -> DisplayView
```

Makes the given view into the specified size. This is useful, for example, in creating a box of a specific size without having to specify its position: `Size (10, 10) (Box noView)`.

```
   group :: DisplayView -> DisplayView
```

Introduces a new canvas around the given display view. This allows a complex view to be included into a new view which may have a conflicting coordinate space.

Shadows

```
    shadow :: Num -> DisplayView -> DisplayView
```

Draws a drop-shadow around the lower-left of the given view. The current fill colour is used as the colour of the shadow.

```
     whiteShadow :: DisplayView -> DisplayView
     blackShadow :: DisplayView -> DisplayView
     greyShadow :: DisplayView -> DisplayView
     greenShadow :: DisplayView -> DisplayView
```

Draws shadows of the named colour.

```
    upperShadow :: Num -> DisplayView -> DisplayView
    whiteUpperShadow :: DisplayView -> DisplayView
```

Same idea as above, except the shadows are drawn to the upper-right.

Relief Helper Functions

```
    motifShading :: Bool -> Num -> DisplayView -> DisplayView
```

Adds a Motif-style border to the given display view. The numeric parameter specifies the width of the shading. The boolean parameter if True specifies sunken relief, if False specifies raised relief.

```
    groovyBox :: Num -> DisplayView -> DisplayView
```

Adds a “groovy” box of two times the specified width around the given display view.

Padding and Spacing

```
    pad :: Num -> DisplayView -> DisplayView
```

Adds a blank border of the specified number of pixels around the given display view.

```
    paddedText :: Num ->  String -> DisplayView
```

Returns a display view consisting of the string with a blank border of the specified number of pixels surrounding it.

```
    space :: Num -> DisplayView
    hSpace :: Num -> DisplayView
    vSpace :: Num -> DisplayView
```

Returns a space of the given size in pixels. This function is useful for spacing out items listed in a beside or above function. *space* returns a square space; *hSpace* and *vSpace* have height and width of 1 pixel respectively.

```
    spaceApart :: [DisplayView] -> [DisplayView]
```

Inserts a two-pixel wide space between each element of the given list of views.

```
    largeSpaceApart :: [DisplayView] -> [DisplayView]
```

Inserts a ten-pixel wide space between each element of the given list of views.

Grabbing

By default, input is directed to the component whose view is under the tracking symbol. Ie, clicking input is directed to whatever is clicked. Sometimes, it is desirable to explicitly grab inputs when some condition is met. This grabbing is specified in the view language. For example, the typical implementation for a button is:

```
view =
    if isDepressed then
        GrabbingMouseButton (Box (Text myId))
    else
        Box (Text myId)
    end if.
```

Here, whenever the button is depressed, all next mouse button inputs will be sent to the button until it is released. This means that if the user clicks on a button and then moves the mouse before releasing, the button "up" input will still be sent to the button.

The grabbing directives are:

```
GrabbingMouseButton :: DisplayView -> DisplayView
```

Grabs subsequent mouse button inputs.

```
GrabbingMouseMotion :: DisplayView -> DisplayView
```

Grabs subsequent mouse motion and relative motion inputs.

```
GrabbingMouse :: DisplayView -> DisplayView
```

Grabs subsequent mouse button, motion and relative motion inputs. This directive has the same effect as `GrabbingMouseButton` and `GrabbingMouseMotion` combined.

```
GrabbingKeyboard :: DisplayView -> DisplayView
```

Grabs subsequent keyboard inputs.

The grabbing directives are stack-based: if a component grabs a resource currently belonging to another component, it takes the resource. When the second component releases the resource, its ownership reverts to the first.

User Inputs

Clock supports a number of predefined user inputs to allow access to events generated by the mouse and keyboard input devices. To access these inputs, an update function must be placed in the event handler taking the input.

Mouse Button

By default, mouse button input is directed to the nearest enclosing event handler over which the mouse is clicked.

```
mouseButton :: String -> UpdateEvent
```

Registers that a mouse click has been performed over the view of the event handler taking the `mouseButton` input. The string parameter can be “`Down`”, indicating the mouse button has been depressed, or “`Up`”, indicating the mouse button has been released. On multi-button mice, all buttons generate the `mouseButton` input; there is no way of distinguishing which button was depressed.

Mouse Motion

By default, mouse motion events are directed to the event handler whose view most tightly encloses the postion at which the mouse motion occurs. The granualarity of mouse motion events is dependent on the windowing system; in most systems, moving the mouse rapidly causes motion events to be dropped, possible also leading to `enter` events being missed.

```
enter :: UpdateEvent
```

Indicates the mouse has entered over the view of the event handler taking this update.

```
leave :: UpdateEvent
```

Indicates that mouse was over the view of the event handler taking this update, but now is not.

```
motion (Num,Num) :: UpdateEvent
```

Indicates that mouse is now located at the given coordinate within the view of the event handler taking the update. The coordinate system of this event handler is used, so the coordinate `(0,0)` indicates the lower-left corner of the view.

```
relMotion (Num,Num) :: UpdateEvent
```

Indicates that the mouse has moved by the given number of pixels in the X and Y direction since the last `relMotion` update was given.

Keyboard

Keyboard input is directed to the last event handler to grab keyboard input. If nobody has grabbed the keyboard, inputs are discarded.

```
key :: Num -> UpdateEvent
```

Indicates that a key has been depressed. The parameter represents the ordinal of the key clicked.

```
arrowKey :: String -> UpdateEvent
```

Indicates that one of the arrow keys has been depressed. The string parameter indicates which arrow key, and can have values of “`Left`”, “`Right`”, “`Up`” or “`Down`”.

```
editKey :: String -> UpdateEvent
```

Indicates that a special key has been depressed. The string parameter indicates which key was depressed, and can have values of “`Tab`”, “`Backspace`”, “`Delete`”, “`Escape`” or “`Return`”.

```
functionKey :: String -> UpdateEvent
```

Indicates that a function key has been depressed. The string parameter indicates which function key, and can have values of “`1`” through “`35`”.

Predefined Updates

The following updates are predefined in the language. There are currently no predefined requests.

All or Nothing

```
    all :: [UpdateEvent] -> UpdateEvent
```

In order to permit update functions to return more than one update, the `all` update groups a list of updates into a single update. All updates are performed, in no specified order.

```
    noUpdate :: UpdateEvent
```

The update noUpdate indicates that no update is to be performed. This is equivalent to `all []`.

Known Problems

Clock is still an experimental system, and has a number of known bugs and shortcomings. These will hopefully be resolved over time.

Layout

There is a known bug in the layout routine that sometimes causes views to be allocated more space than they should. The cases under which this occurs are hard to describe precisely. So far, only I have actually come up with an example that triggers this error.

Type Checking

There is no type checker currently, so most type errors will result in “segmentation violation” or “bus error” types of crashes. Note that in functional languages, type errors include type mismatches on operations, and providing too many or too few parameters to functions or constructors. Tracking down type errors is not actually all that hard – use the trace option in cw to locate the component and function in which the crash occurs, and use the ‘debug’ function to check that you are indeed getting the values you expect at various points.

Request Caching

The incremental view update mechanism uses a complex algorithm based on caching request values and triggering view recomputation when these values change. There are no known bugs in this caching/triggering code, but I wouldn’t be surprised if there are still some lurking there. If you find that views are not being updated when they should, this may be the problem; please let me know of any such cases.

2.5 D

Currently, 2.5 D layout is supported through an awkward and inefficient mechanism. This is due for a total overhaul. In the current system, if you try placing objects in a layered manner, you may get unpredictable results.

Output

There is no way in Clock of drawing circles, elipses, or splines.

Reals

Clock currently provides no support for real numbers. The `Num` type is currently integer.

External Interface

There is currently no way of accessing code written in other languages or the environment in general.

Loading Libraries

Currently, when you load a library in ClockWorks, you get access to the class definitions in the library, but not to the actual code of the library components. If you run programs using library components, you will get error messages of the form:

```
Clint (Fatal Error):  Cannot open '/cs/u/graham/EClock/Programs/temp/Depressed'
```

To create links to the code of the library components, you must:

- cd to the project directory
- perform the command: `cplib` *libraryName*

For example, if the project is called 'MyProject', and the library name is 'Buttons', then type:

- `cd $CLOCKSYS/Programs/MyProject`
- `cplib Buttons`

It is often convenient to include `$CLOCKSYS/Programs` in your cdpath.