# API documentation for libui

## Contents

# 1 Overview

User interface library.

# 2 ui.eh — Screen based user interface.

```
use "ui.eh"
```

## 2.1 Description

Graphical application defines one or more full screen windows, in terminology of Alchemy OS called Screens. You may choose from different types of screens:

- the set of predefined screens in stdscreens.eh;
- *canvas* from canvas.eh on which you can draw freely;
- *form* from form.eh to build high level dialogs.

Initially application has no active screen (is in console mode). Switching to graphical mode is done by calling ui_set_screen(screen). If argument is `null` then screen is removed from the display and application switches back to the console mode.

### 2.1.1 Screen menus

Every screen has a menu. This menu always contains item "Switch to..." which is used to switch between graphical applications. Custom menu items may be attached to a screen or detached from it anytime. Menu items are values of type Menu.

Menu item has type, label and priority. Priority determines how menus are organized. The less priority number the higher menu item will be in the menu. Menu type is a hint about intent of menu item. Depending on type platform may assign menu items to different buttons or add icons to them.

### 2.1.2 Event handling

When user interacts with the screen, application generates an UIEvent value that contains information about what happened and on which screen. These events are then read by functions ui_read_event and ui_wait_event.

`ui_wait_event()` will wait until something happens on the screen, and then return that event. This function is appropriate when you just passively waiting for the event.

`ui_read_event()` returns immediately. If there are no events, it returns `null`. It is appropriate for cases when you constantly need to do something (redraw screen for example).

### 2.1.3 Simple example

```
use "ui"
use "stdscreens"

def main(args: [String]) {
  // creating new screen
  var screen = new MsgBox("This is an example of graphical program")
  screen.title = "Example"
  // attaching menu to the screen
  var mclose = new Menu("Close", 1)
  screen.add_menu(mclose)
  // showing screen to the user
  ui_set_screen(screen)
  // waiting for Close menu
  var e = ui_wait_event()
  while (e.value != mclose) {
    e = ui_wait_event()
  }
}
```

## 2.2 Constants

```
const MT_SCREEN = 1;
```

Default menu type, with no specific intent.

```
const MT_BACK = 2;
```

Menu item that returns the user to the previous screen.

```
const MT_CANCEL = 3;
```

Menu item that is a standard negative answer to a dialog.

```
const MT_OK = 4;
```

Menu item that is a standard positive answer to a dialog.

```
const MT_HELP = 5;
```

Menu item that shows help information.

```
const MT_STOP = 6;
```

Menu item that stops currently running operation.

```
const MT_EXIT = 7;
```

Menu item for exiting from application.

```
const EV_SHOW = -1;
```

Event of this kind is generated when screen gains focus. The field `UIEvent.value` is `null` for this kind of event.

```
const EV_HIDE = -2;
```

Event of this kind is generated when screen losts focus. The field `UIEvent.value` is `null` for this kind of event.

```
const EV_MENU = 1;
```

Event of this kind is generated when user chooses screen menu item. In this case the field `UIEvent.value` will contain chosen Menu item.

```
const EV_ITEM = 2;
```

Event of this kind is generated when user activates interactive item. In this case the field `UIEvent.value` will contain activated Item.

```
const EV_KEY = 3;
```

Event of this kind is generated by canvas on key press. The field `UIEvent.value` will contain Int code of pressed key.

```
const EV_KEY_HOLD = 4;
```

Event of this kind is generated by canvas repeatedly if key is held down. The field `UIEvent.value` will contain Int code of pressed key.

```
const EV_KEY_RELEASE = 5;
```

Event of this kind is generated by canvas on key release. The field `UIEvent.value` will contain Int code of pressed key.

```
const EV_PTR_PRESS = 6;
```

Event of this kind is generated by canvas when the pointer is pressed (screen touched). In this case `UIEvent.value` will contain Point value with coordinates of pointer position.

```
const EV_PTR_RELEASE = 7;
```

Event of this kind is generated by canvas when the pointer is released. In this case `UIEvent.value` will contain Point value with coordinates of pointer position.

```
const EV_PTR_DRAG = 8;
```

Event of this kind is generated by canvas when the pointer is dragged. In this case `UIEvent.value` will contain Point value with coordinates of pointer position.

```
const EV_ITEMSTATE = 9;
```

Event of this kind is generated by form when the state of an item in it changes. In this case `UIEvent.value` will contain Item which changed state.

## 2.3 Types

```
type Screen < Any;
```

An application window which can be shown on the device display.

```
type Menu < Any;
```

Menu item that can be attached to a screen.

```
type UIEvent = {
  kind: Int,
  source: Screen,
  value: Any
}
```

An event from the graphical user interface. Fields:

- *kind* - kind of event, one of predefined `EV_*` constants;
- *source* - screen that generated this event;
- *value* - depends on the event kind. See description of each `EV_*` constant to find out particular value.

```
type Point = {
  x: Int,
  y: Int
}
```

2-dimensional point on the canvas. Used as return value in pointer events.

## 2.4 Functions

```
def ui_set_app_title(title: String);
```

Sets default title for all screens of application.

```
def ui_set_app_icon(icon: Image);
```

Sets application icon, which will appear in "Switch to..." dialog.

```
def ui_vibrate(millis: Int): Bool;
```

Requests device to vibrate for specified number of milliseconds. This function returns immediately, vibration happens in background. To stop vibrator, call this function with `0`. Note, that device may limit or override duration of vibration.

Returns `true` if vibration is supported, `false` otherwise.

```
def ui_flash(millis: Int): Bool;
```

Requests device to flash backlight for specified number of milliseconds. The exact effect is device dependent, examples are are cycling the backlight on and off or from dim to bright repeatedly. This function returns immediately, flashing happens in background. To stop flashing effect, call this function with `0`. Note, that device may limit or override duration of flashing.

Returns `true` if flashing is supported, `false` otherwise.

```
def Screen.get_height(): Int;
```

Returns height of the screen available to application.

```
def Screen.get_width(): Int;
```

Returns width of the screen available to application.

```
def Screen.get_title(): String;
```

Returns title of the screen.

```
def Screen.set_title(title: String);
```

Sets new title to the screen.

```
def Screen.is_shown(): Bool;
```

Returns `true` if this screen is shown on the phone display.

```
def ui_get_screen(): Screen;
```

Returns current screen associated with the application. If application is in console mode, this method returns `null`.

```
def ui_set_screen(scr: Screen);
```

Shows given screen on the display.

```
def Menu.new(text: String, priority: Int, mtype: Int = MT_SCREEN): Menu;
```

Creates new menu item that can be attached to a screen. Menu label is defined by *text* argument. Priority determines how menus are arranged in a list - lower number means higher priority. Menu type is one of `MT_*` constants defined in this header.

```
def Menu.get_text(): String;
```

Returns text label of this menu item.

```
def Menu.get_priority(): Int;
```

Returns priority of given menu item.

```
def Screen.add_menu(menu: Menu);
```

Attaches given menu item to the screen.

```
def Screen.remove_menu(menu: Menu);
```

Detaches given menu item from the screen.

```
def ui_read_event(): UIEvent;
```

Reads next event from the event queue of the application. If there are no pending events this function returns `null`.

```
def ui_wait_event(): UIEvent;
```

Reads next event from the event queue of the application. If there are no pending events this function waits until an event is available.

# 3 stdscreens.eh — Useful builtin screens.

```
use "stdscreens.eh"
```

## 3.1 Description

This header defines set of screens which may be used to build user interface.

### 3.1.1 Message box

`MsgBox` is a screen that displays static text and can optionally contain icon. This screen is convenient for making dialog windows.

## 3.2 Edit box

`EditBox` is a screen that allows user to enter and edit text.

## 3.3 List box

Screen that presents a list of strings from which user can pick one. Each string can optionally be accompanied by icon.

## 3.4 Types

```
type MsgBox < Screen;
```

Screen which displays static text.

```
type EditBox < Screen;
```

Screen that allows user to enter and edit text.

```
type ListBox < Screen;
```

List of strings from which user can pick one.

## 3.5 Functions

```
def MsgBox.new(msg: String, icon: Image = null): MsgBox;
```

Creates new message box with given message and icon. If *icon* is `null` then message box contains no icon.

```
def MsgBox.get_text(): String;
```

Returns text contained in this message box.

```
def MsgBox.set_text(text: String);
```

Sets new text to this message box.

```
def MsgBox.get_image(): Image;
```

Returns image contained in this message box.

```
def MsgBox.set_image(img: Image);
```

Sets new image to this message box.

```
def EditBox.new(mode: Int = EDIT_ANY): EditBox;
```

Creates new editbox. Argument *mode* must be one of constants defined in ui_edit.eh.

```
def EditBox.get_text(): String;
```

Returns text currently contained in this editbox.

```
def EditBox.set_text(text: String);
```

Sets new text to this editbox.

```
def EditBox.get_maxsize(): Int;
```

Returns maximum length of text this editbox can store.

```
def EditBox.set_maxsize(size: Int);
```

Sets new maximum length of text this editbox can store. Note, that actual maximum length may be overriden by platform.

```
def EditBox.get_size(): Int;
```

Returns number of characters this editbox currently stores.

```
def EditBox.get_caret(): Int;
```

Returns current input position. On most devices this function simply returns cursor position. On some devices, however, it blocks and asks the user to set position.

```
def ListBox.new(strings: [String], images: [Image], select: Menu): ListBox;
```

Creates new listbox. Array *strings* is used as the initial contents of the list. If *images* is not `null` then it must have the same length as *strings*. The contents of *images* array is used as icons for list items. Some elements of *images* array may be `null`, the corresponding list item has no icon in this case. Menu given as *select* argument is added to the screen and returned in an event when user selects an item from the list.

```
def ListBox.get_index(): Int;
```

Returns index of the selected item in the listbox.

```
def ListBox.set_index(index: Int);
```

Sets selection to the item with the specified index in the listbox.

```
def ListBox.add(str: String, img: Image = null);
```

Adds new item to the end of the list. Argument *img* may be `null`.

```
def ListBox.insert(at: Int, str: String, img: Image = null);
```

Inserts new item at the specified position of the list. Argument *img* may be `null`.

```
def ListBox.set(at: Int, str: String, img: Image = null);
```

Replaces item at the specified position of the list with given one.

```
def ListBox.delete(at: Int);
```

Removes item at the specified position of the list.

```
def ListBox.get_string(at: Int): String;
```

Returns string part of the item at the specified position of the list.

```
def ListBox.get_image(at: Int): Image;
```

Returns image part of the item at the specified position of the list.

```
def ListBox.clear();
```

Removes all items from this list.

```
def ListBox.len(): Int;
```

Returns current number of items in this list.

# 4 canvas.eh — Screen on which you can do arbitrary drawing.

```
use "canvas.eh"
```

## 4.1 Description

This header provides a `Screen` called "canvas". Canvas provides facilities of low-level drawing on the display and allows to read key presses and touch events. Canvas is double-buffered, all drawings are performed on off-screen buffer that is obtained with Canvas.graphics. After drawing is finished, Canvas.refresh should be called to present changes on the display.

### 4.1.1 Key events

If you press the key while the active screen is canvas, the UIEvent will be generated. Its `kind` field will be EV_KEY and its `value` field will contain the code of the pressed key. Since returned type of the `UIEvent.value` is not known at the time of compilation, you have to cast it manually

```
// read the next event
var e = ui_wait_event()
if (e.kind == EV_KEY) {
  // obtain key code
  var key = e.value.cast(Int)
  // if it is a character, print it in the terminal
  if (key > 0) {
    write(key)
  }
}
```

Similarly, when key is released, EV_KEY_RELEASE event is generated. Also, if key is held down, device may generate EV_KEY_HOLD event repeatedly. The last event type is optional, to test if device supports hold events, use Canvas.has_hold_event.

This header defines key codes for standard phone keypad (keys 0..9, * and #). If the phone has other alphanumeric keys, the code (most probably) will be the corresponding Unicode character code. If phone has non-alphanumeric keys (for example, joystick), those keys will (most probably) return negative key codes. However, actual key codes may be different on distinct hardware. If you want your application to be portable, you should use only the standard key codes provided by this header. Or use the action codes.

### 4.1.2 Action codes

Since keypads differ from device to device, platform provides the following portable *action codes*: UP, DOWN, LEFT, RIGHT, FIRE, ACT_A, ACT_B, ACT_C and ACT_D. Each key code is mapped to at most one action code.

However, multiple keys may be mapped to the same action code. For example, if the phone has joystick, both moving the joystick up and pressing '2' key will generate UP action. To get action code for the pressed key use Canvas.action_code(key).

### 4.1.3 Pointer events

If device has touch screen, the canvas may generate pointer events. To test whether the device supports touch events, use Canvas.has_ptr_events and Canvas.has_ptrdrag_event. For pointer events `value` field of the event will be of type Point. Since compiler cannot predict type of the value, you have to cast it to the needed type manually to extract values $x$ and $y$.

```
var e = ui_wait_event()
if (e.kind == EV_PTR_PRESS) {
  var p = e.value.cast(Point)
  do_something_at(p.x, p.y)
}
```

## 4.2 Constants

```
const KEY_0 = '0';
```

Key code for key 0.

```
const KEY_1 = '1';
```

Key code for key 1.

```
const KEY_2 = '2';
```

Key code for key 2.

```
const KEY_3 = '3';
```

Key code for key 3.

```
const KEY_4 = '4';
```

Key code for key 4.

```
const KEY_5 = '5';
```

Key code for key 5.

```
const KEY_6 = '6';
```

Key code for key 6.

```
const KEY_7 = '7';
```

Key code for key 7.

```
const KEY_8 = '8';
```

Key code for key 8.

```
const KEY_9 = '9';
```

Key code for key 9.

```
const KEY_STAR = '*';
```

Key code for key *.

```
const KEY_HASH = '#';
```

Key code for key #.

```
const UP = 1;
```

Constant for the UP action.

```
const DOWN = 6;
```

Constant for the DOWN action.

```
const LEFT = 2;
```

Constant for the LEFT action.

```
const RIGHT = 5;
```

Constant for the RIGHT action.

```
const FIRE = 8;
```

Constant for the FIRE action.

```
const ACT_A = 9;
```

Constant for the general purpose "A" action.

```
const ACT_B = 10;
```

Constant for the general purpose "B" action.

```
const ACT_C = 11;
```

Constant for the general purpose "C" action.

```
const ACT_D = 12;
```

Constant for the general purpose "D" action.

## 4.3 Types

```
type Canvas < Screen;
```

Screen for low-level drawing.

## 4.4 Functions

```
def Canvas.new(full: Bool = false): Canvas;
```

Creates new canvas screen. If *fullscreen* is `true` then created canvas is in fullscreen mode.

```
def Canvas.graphics(): Graphics;
```

Returns the graphical buffer for this canvas on which you can draw.

```
def Canvas.read_key(): Int;
```

Returns key code of the last pressed key. If no key was pressed, returns `0`. This is convenient function, if you want to receive only key presses from canvas. If you want to receive other kinds of events, use event framework.

```
def Canvas.refresh();
```

Refreshes displayed content of the canvas.

```
def Canvas.action_code(key: Int): Int;
```

Returns the action code for the specified key code. If given key has no associated action, then `0` is returned.

```
def Canvas.has_ptr_events(): Bool;
```

Checks if the platform supports pointer press and release events. If returns `true`, the canvas will generate EV_PTR_PRESS and EV_PTR_RELEASE events.

```
def Canvas.has_ptrdrag_event(): Bool;
```

Checks if the platform supports pointer dragging events. If returns `true`, the canvas will generate EV_PTR_DRAG event.

```
def Canvas.has_hold_event(): Bool;
```

Checks if the platform supports key hold event. If returns `true`, the canvas will generate EV_KEY_HOLD event.

# 5 graphics.eh — Graphical context for low-level drawing.

```
use "graphics.eh"
```

## 5.1 Description

Type `Graphics` represents graphical context which can be rendered both on the display and to the offscreen images. Drawing primitives are provided for text, images, lines, rectangles, rounded rectangles, and arcs. Rectangles and arcs may also be filled with a solid color.

All drawing operations are performed with the current color of graphics. New color is set with set_color function in form of `0x00RRGGBB`. You can specify color by its components using expression `(red<<16)|(green<<8)|blue` where `red`, `green` and `blue` are numbers in range `0..255`.

Line drawing operations are performed with the current stroke style of graphics. New stroke style is set by set_stroke and may be SOLID or DOTTED. Stroke style does not affect `fill_*` functions, images or text.

Strings are rendered with the current font of graphics. New font is set by set_font. Font constants and functions are defined in font.eh.

## 5.2 Constants

```
const SOLID = 0;
```

Constant for the solid stroke style.

```
const DOTTED = 1;
```

Constant for the dotted stroke style.

```
const TR_NONE = 0;
```

No transformation applied.

```
const TR_ROT90 = 5;
```

Image is rotated clockwise by 90 degrees.

```
const TR_ROT180 = 3;
```

Image is rotated clockwise by 180 degrees.

```
const TR_ROT270 = 6;
```

Image is rotated clockwise by 270 degrees.

```
const TR_HMIRROR = 2;
```

Image is mirrored horizontally.

```
const TR_HMIRROR_ROT90 = 7;
```

Image is mirrored horizontally and then rotated clockwise by 90 degrees.

```
const TR_VMIRROR = 1;
```

Image is mirrored vertically.

```
const TR_VMIRROR_ROT90 = 4;
```

Image is mirrored vertically and then rotated clockwise by 90 degrees.

## 5.3 Types

```
type Graphics < Any;
```

Graphical context to draw on.

## 5.4 Functions

```
def Graphics.get_color(): Int;
```

Returns current color to render with. Color is returned as `0x00RRGGBB` value.

```
def Graphics.set_color(rgb: Int);
```

Sets color used to draw new primitives. Color is specified in form of `0x00RRGGBB`.

```
def Graphics.get_stroke(): Int;
```

Returns the stroke style used for drawing operations.

```
def Graphics.set_stroke(stroke: Int);
```

Sets the stroke style used for drawing lines, arcs, rectangles, and rounded rectangles. This does not affect fill, text, and image operations. The value of *stroke* must be one of SOLID, DOTTED.

```
def Graphics.get_font(): Int;
```

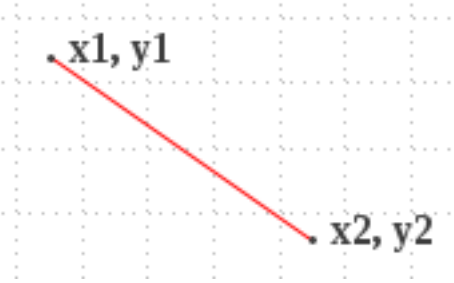
Returns current font to render strings with.

```
def Graphics.set_font(font: Int);
```

Sets new font to render new strings with.

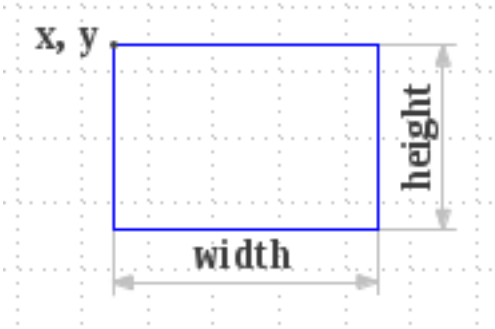
```
def Graphics.draw_line(x1: Int, y1: Int, x2: Int, y2: Int);
```

Draws a line between the coordinates (*x1*,*y1*) and (*x2*,*y2*) using the current color and stroke style.



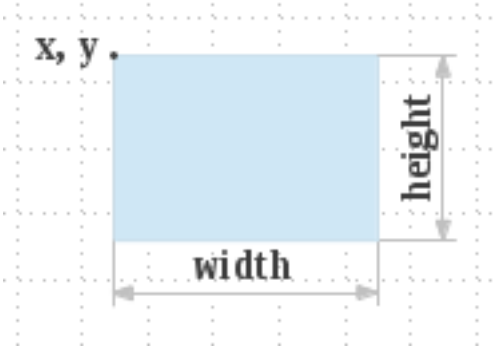

```
def Graphics.draw_rect(x: Int, y: Int, w: Int, h: Int);
```

Draws the outline of the specified rectangle using the current color and stroke style.



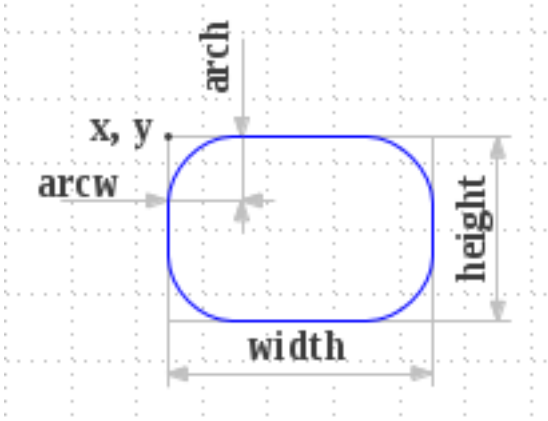

```
def Graphics.fill_rect(x: Int, y: Int, w: Int, h: Int);
```

Fills the specified rectangle with the current color.



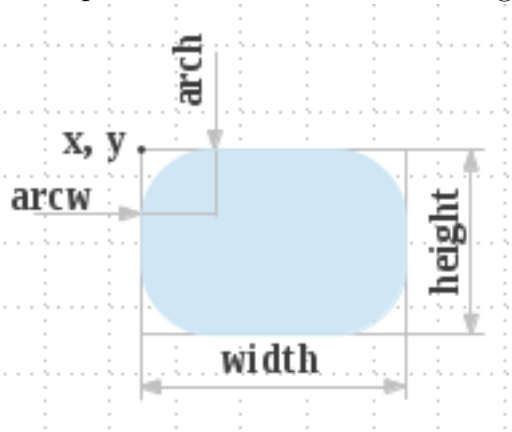
```
def Graphics.draw_roundrect(x: Int, y: Int, w: Int, h: Int,
        arcw: Int, arch: Int);
```

Draws the outline of the specified rounded corner rectangle using the current color and stroke style.



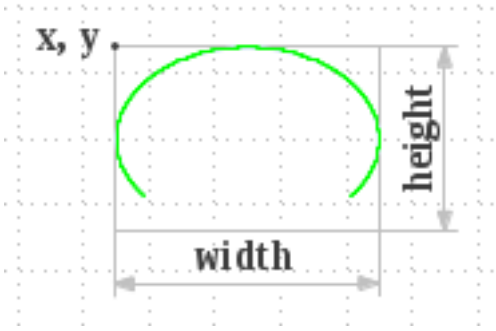

```
def Graphics.fill_roundrect(x: Int, y: Int, w: Int, h: Int,
        arcw: Int, arch: Int);
```

Fills the specified rounded corner rectangle with the current color.



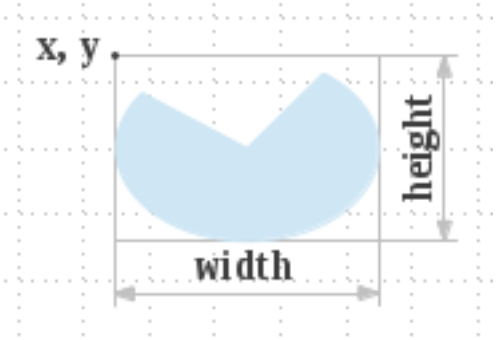

```
def Graphics.draw_arc(x: Int, y: Int, w: Int, h: Int,
        startangle: Int, arcangle: Int);
```

Draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style. The resulting arc begins at *startangle* and extends for *arcangle* degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.




```
def Graphics.fill_arc(x: Int, y: Int, w: Int, h: Int,
        startangle: Int, arcangle: Int);
```

Fills a circular or elliptical arc covering the specified rectangle. The resulting arc begins at *startangle* and extends for *arcangle* degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation. The filled region consists of the "pie wedge" region bounded by the arc segment as if drawn by `draw_arc()`, the radius extending from the center to this arc at *startangle* degrees, and radius extending from the center to this arc at *startangle*+*arcangle* degrees.




```
def Graphics.fill_triangle(x1: Int, y1: Int, x2: Int, y2: Int,
        x3: Int, y3: Int);
```

Fills the specified triangle will the current color. The lines connecting each pair of points are included in the filled triangle.

```
def Graphics.draw_string(str: String, x: Int, y: Int);
```

Draws the specified string using the current font and color. The width and height of the rendered string may be obtained using font_height and str_width functions with current drawing font.

```
def Graphics.draw_image(im: Image, x: Int, y: Int);
```

Draws specified image to the given location.

```
def Graphics.draw_rgb(rgb: [Int], ofs: Int, scanlen: Int,
        x: Int, y: Int, w: Int, h: Int, alpha: Bool);
```

Renders an ARGB pixel data to the given location. Pixels are stored as 24-bit color with 8-bit alpha channel in form `0xAARRGGBB`. The first pixel is stored at the specified *offset*. The *scanlen* specifies the relative offset within the array between the corresponding pixels of consecutive rows. Any value for *scanlen* is acceptable (even negative values) provided that all resulting references are within the bounds of the array. The ARGB data is rasterized horizontally from left to right within each row. The ARGB values are rendered in the region specified by *x*, *y*, *width* and *height*.

If *alpha* is `false`, then transparency is not processed and all values are assumed to be fully opaque.

```
def Graphics.copy_area(xsrc: Int, ysrc: Int, w: Int, h: Int,
        xdst: Int, ydst: Int);
```

Copies the contents of a rectangular area (*xsrc*, *ysrc*, *width*, *height*) in given graphics to a destination area, whose top left angle is located at (*xdest*, *ydest*).

```
def Graphics.draw_region(im: Image, xsrc: Int, ysrc: Int,
        w: Int, h: Int, trans: Int, xdst: Int, ydst: Int);
```

Draws a region of the specified source image to the given location, possibly transforming (rotating and reflecting) the image data using the chosen transformation. The transformation is one of `TR_*` constants.

# 6 form.eh — A screen that contains a list of components.

```
use "form.eh"
```

## 6.1 Description

A form is a screen that contains a list of interactive components called Items. Items are arranged vertically - one item per line.


Every item can have a text label assigned to it. Label appears before or above an item and visually differs from contents of the item. Item label may be received or set by functions Item.get_label and Item.set_label respectively. You may specify empty string as label to create item without label.

When the state of the item is changed by the user, form generates EV_ITEMSTATE event.

### 6.1.1 Text item

Text item is non-interactive item that just displays plain text. Text may contain line breaks (`'\n'` characters) in which case it spans several rows. Text also will be wrapped if it doesn't fit in a single line.

Constructor: new TextItem(label, text)

Properties:

- *text* - text displayed by this item;
- *font* - font to display the text with. On how to define a font, see font.eh.

### 6.1.2 Hyperlink item

Text item which contains hyperlink. This item inherits all properties from text item, but it can be activated, e.g. by setting focus on it and pressing softkey or clicking it on touch screen. When item is activated EV_ITEM event is generated.

Constructor: new HyperlinkItem(label, text)

### 6.1.3 Image item

Image item is used to display an image.

Constructor: new ImageItem(label, image)

Properties:

- *image* - displayed image;
- *alttext* - a string to be shown in place of the image if the image exceeds the capacity of the display.

### 6.1.4 Hyperimage item

Hyperimage is a hyperlink image. This item inherits all properties from image item, but it can be activated, e.g. by setting focus on it and pressing softkey or clicking it on touch screen. When item is activated EV_ITEM event is generated.

Constructor: new HyperimageItem(label, image)

### 6.1.5 Edit item

Edit item is an editable text field in which user can input arbitrary text. Edit item has a maximum size which limits number of characters that may be entered in it. With specifying input mode actual input may be restricted to accept only numeric input/e-mail addresses, etc... and/or hide input characters (e.g. when entering a password). Constants to use as `mode` argument may be found in ui_edit.eh.

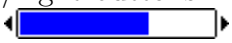
Constructor: new EditItem(label, text, mode, size)

Properties:

- *text* - text currently contained within this item;
- *maxsize* - maximum number of characters this item can store;
- *size* (read only) - number of characters currently stored in this item;
- *caret* (read only) - current cursor position.

### 6.1.6 Gauge item

Gauge is a graphical item usually represented by horizontal bar or bar graph. It contains integer value between 0 and *maxvalue* which user can change using left/right buttons.



Constructor: new GaugeItem(label, max, init)
Properties:

- *value* - current value of the gauge;
- *maxvalue* - maximum value of the gauge.

### 6.1.7 Date item

Item for presenting and choosing date and time. Date is represented by Long number of milliseconds since "the epoch". Functions to work with dates may be found in time.eh.

Constructor: new DateItem(label, mode)
Properties:

- *date* - date currently stored in this item.

### 6.1.8 Check item

An item which have two states - checked and unchecked. Usually represented as square box which is either empty or contains a tick mark or X.

Constructor: new CheckItem(label, text, checked)
Properties:

- *checked* - whether this item checked or not;
- *text* - text that follows check box. Do not be confused with *label* which precedes an item. In the screenshot above all three checkboxes have text assigned to them, but only the first is labeled.

### 6.1.9 Radio item

This item represents a list of strings only one of which can be selected at a time. Selection usually visualized via "radio buttons" preceding strings.

Constructor: new RadioItem(label, strings)
Properties:

- *index* - index of selected element, starting with 0.

### 6.1.10 Popup item

The compact version of radio item. Selected string is shown, all others are hidden. When user chooses this item, popup menu appears, allowing to choose one of strings.

Constructor: new PopupItem(label, strings)

Properties:

- *index* - index of selected element, starting with 0.

## 6.2 Constants

```
const DATE_ONLY = 1;
```

Input mode for date item that allows to input only date.

```
const TIME_ONLY = 2;
```

Input mode for date item that allows to input only time.

```
const DATE_TIME = 3;
```

Input mode for date item that allows to input both date and time.

## 6.3 Types

```
type Form < Screen;
```

A screen that contains a list of interactive components.

```
type Item < Any;
```

A component of the form.

```
type TextItem < Item;
```

An item that displays plain text.

```
type HyperlinkItem < TextItem;
```

An interactive text item, activating which generates an event.

```
type ImageItem < Item;
```

An item that displays image.

```
type HyperimageItem < ImageItem;
```

An interactive image item, activating which generates an event.

```
type EditItem < Item;
```

An item in which user can input arbitrary text.

```
type GaugeItem < Item;
```

A graphical gauge display.

```
type DateItem < Item;
```

An item that allows to choose date and time.

```
type CheckItem < Item;
```

An item that can be checked and unchecked.

```
type RadioItem < Item;
```

A list of strings only one of which can be selected at a time.

```
type PopupItem < RadioItem;
```

A list of strings that uses popup menu.

## 6.4 Functions

```
def Form.new(): Form;
```

Creates new empty form.

```
def Item.get_label(): String;
```

Returns label assigned to this item.

```
def Item.set_label(label: String);
```

Sets new label to this item. If argument is `null` then item has no label.

```
def Form.add(item: Item);
```

Adds new item to the end of this form.

```
def Form.get(at: Int): Item;
```

Returns item in the specified row of this form.

```
def Form.set(at: Int, item: Item);
```

Sets new item in the specified row of this form replacing previous item.

```
def Form.insert(at: Int, item: Item);
```

Inserts new item in the specified row of this form moving all subsequent items lower.

```
def Form.remove(at: Int);
```

Removes item in the specified row of this form.

```
def Form.size(): Int;
```

Returns current number of items in this form.

```
def Form.clear();
```

Removes all items from this form.

```
def TextItem.new(label: String, text: String): TextItem;
```

Creates new item that shows given text.

```
def TextItem.get_text(): String;
```

Returns text contained in this text item.

```
def TextItem.set_text(text: String);
```

Sets new text to this text item.

```
def TextItem.get_font(): Int;
```

Returns font used in this text item.

```
def TextItem.set_font(font: Int);
```

Sets font for this text item to display text with.

```
def HyperlinkItem.new(label: String, text: String);
```

Creates new hyperlink item with specified label and text.

```
def ImageItem.new(label: String, img: Image): ImageItem;
```

Creates new image item.

```
def ImageItem.get_image(): Image;
```

Returns image contained in this item.

```
def ImageItem.set_image(img: Image);
```

Sets new image to this item.

```
def ImageItem.get_alttext(): String;
```

Gets the text string to be used if the image exceeds the device's capacity to display it.

```
def ImageItem.set_alttext(text: String);
```

Sets the alternate text of the ImageItem. If `null` no alternate text is provided.

```
def HyperimageItem.new(label: String, img: Image): ImageItem;
```

Creates new hyperimage item with specified label and image.

```
def EditItem.new(label: String, text: String = "",
        mode: Int = EDIT_ANY, maxsize: Int = 50): EditItem;
```

Creates new editable text item. Argument *mode* must be one of `EDIT_*` constants from ui_edit.eh. Argument *maxsize* specifies maximum length of string that user can input in this item. Note that actual maximum size may be even less than this argument due to platform limitations. To get actual maximum size use get_maxsize.

```
def EditItem.get_text(): String;
```

Returns text currently stored in the editable item.

```
def EditItem.set_text(text: String);
```

Sets new text to this edit item.

```
def EditItem.get_maxsize(): Int;
```

Returns maximum length of text this item can store.

```
def EditItem.set_maxsize(size: Int);
```

Sets new maximum length of text this item can store.

```
def EditItem.get_size(): Int;
```

Returns number of characters this item currently stores.

```
def EditItem.get_caret(): Int;
```

Returns current input position. On most devices this function simply returns cursor position. On some devices, however, it blocks and asks the user to set position.

```
def GaugeItem.new(label: String, max: Int, init: Int): GaugeItem;
```

Creates new gauge item. Gauge item represents Int value between zero and *max* value and initially set to *init*. User can decrease or increase this value by pressing left and right buttons respectively.

```
def GaugeItem.get_value(): Int;
```

Returns current value of the gauge item.

```
def GaugeItem.set_value(val: Int);
```

Sets new value to the gauge item.

```
def GaugeItem.get_maxvalue(): Int;
```

Returns maximum value of the gauge item.

```
def GaugeItem.set_maxvalue(val: Int);
```

Sets new maximum value to the gauge item.

```
def DateItem.new(label: String, mode: Int = DATE_ONLY): DateItem;
```

Creates new item that allows to input date or/and time. Argument *mode* must be one of constants DATE_ONLY, TIME_ONLY or DATE_TIME. Date item initially have no date set.

```
def DateItem.get_date(): Long;
```

Returns date currently set in this date item. If date is not set then `null` is returned.

```
def DateItem.set_date(date: Long);
```

Sets new date to this date item.

```
def CheckItem.new(label: String, text: String, checked: Bool): CheckItem;
```

Creates new item that has one of two states - checked or unchecked.

```
def CheckItem.get_checked(): Bool;
```

Tests whether this check item is checked.

```
def CheckItem.set_checked(checked: Bool);
```

Checks or unchecks this check item.

```
def CheckItem.get_text(): String;
```

Returns text of this check item.

```
def CheckItem.set_text(text: String);
```

Sets new text to this check item.

```
def RadioItem.new(label: String, strings: [String]): RadioItem;
```

Creates new item where user can pick one from the list of choices. Array argument must contain only String values, these values are used as choices in this item. You may use zero length array (`[]`), if you want to fill this item with strings later.

```
def RadioItem.get_index(): Int;
```

Returns index of selected string in this item.

```
def RadioItem.set_index(index: Int);
```

Selects new choice in this item.

```
def RadioItem.add(str: String);
```

Appends new string to the end of list.

```
def RadioItem.insert(at: Int, str: String);
```

Inserts new string in the specified position of this item.

```
def RadioItem.set(at: Int, str: String);
```

Replaces string in the specified position of this item.

```
def RadioItem.delete(at: Int);
```

Removes string in the specified position of this item.

```
def RadioItem.get(at: Int): String;
```

Returns string in the specified position of this item.

```
def RadioItem.clear();
```

Removes all strings from this item.

```
def RadioItem.len(): Int;
```

Returns current number of strings in this item.

```
def PopupItem.new(label: String, strings: [String]): PopupItem;
```

Creates new popup item with specified list of strings.

# 7 image.eh — Images.

```
use "image.eh"
```

## 7.1 Description

The Image type holds graphical image data. Images can be painted on the screen or placed in visual elements of interface. Using constructor Image.new you can create mutable image on which you can then draw. Binary image data processed by functions image_from_file, image_from_data or image_from_stream must be in one of image formats supported by the phone. Note that the only format that is guaranteed to be supported is PNG.

## 7.2 Types

```
type Image < Any;
```

Graphical image data.

## 7.3 Functions

```
def Image.new(w: Int, h: Int): Image;
```

Creates new mutable image with given *width* and *height*. Initially every pixel of the image is white. You can draw on this image by obtaining its `Graphics` with Image.graphics.

```
def Image.graphics(): Graphics;
```

Creates a Graphics that renders to this image. This image must be mutable, i.e. created by Image.new constructor.

```
def image_from_argb(argb: [Int], w: Int, h: Int, alpha: Bool): Image;
```

Creates an immutable image from a sequence of ARGB values, specified as `0xAARRGGBB`. The ARGB data within the *argb* array is arranged horizontally from left to right within each row, row by row from top to bottom. If *alpha* is `true`, the high-order byte specifies opacity; that is, `0x00RRGGBB` specifies a fully transparent pixel and `0xFFRRGGBB` specifies a fully opaque pixel. Intermediate alpha values specify semitransparency. If *alpha* is `false`, the alpha values are ignored and all pixels are treated as fully opaque.

```
def image_from_file(file: String): Image;
```

Reads image from file.

```
def image_from_stream(in: IStream): Image;
```

Decodes image data from input stream. Stream is left open after reading.

```
def image_from_data(data: [Byte]): Image;
```

Creates an immutable image which is decoded from the data stored in the specified byte array.

```
def image_from_image(im: Image, x: Int, y: Int, w: Int, h: Int): Image;
```

Creates an immutable image using pixel data from the specified region of a source image. This function can also be used if you want to create immutable image from mutable one.

```
def Image.get_argb(argb: [Int], ofs: Int, scanlen: Int, x: Int,
        y: Int, w: Int, h: Int);
```

Obtains ARGB pixel data from the specified region of this image and stores it in the array *argb* as integer values. The *scanlen* specifies the relative offset within the array between the corresponding pixels of consecutive rows. In order to prevent rows of stored pixels from overlapping, the absolute value of *scanlen* must be greater than or equal to *width*. Negative values of *scanlen* are allowed.

# 8 font.eh — Font handling.

```
use "font.eh"
```

## 8.1 Description

Font in Alchemy UI is specified as OR-combined mask of constants defined in this header. For example, to set large italicized font on TextItem you should use

```
item.set_font(SIZE_LARGE | STYLE_ITALIC)
```

If requested font does not exist then system will provide closest match.

## 8.2 Constants

```
const FACE_SYSTEM = 0;
```

Default font face for the system.

```
const FACE_MONO = 32;
```

Monospace font face.

```
const FACE_PROP = 64;
```

Proportional font face.

```
const STYLE_PLAIN = 0;
```

Plain font style. Can be combined with other style constants.

```
const STYLE_BOLD = 1;
```

Bold font style. Can be combined with other style constants.

```
const STYLE_ITALIC = 2;
```

Italicized font style. Can be combined with other style constants.

```
const STYLE_ULINE = 4;
```

Underlined font style. Can be combined with other style constants.

```
const SIZE_SMALL = 8;
```

The "small" system-dependent font size.

```
const SIZE_MED = 0;
```

The "medium" system-dependent font size.

```
const SIZE_LARGE = 16;
```

The "large" system-dependent font size.

## 8.3 Functions

```
def str_width(font: Int, str: String): Int;
```

Returns the width given string will occupy when rendered with specified font.

```
def font_height(font: Int): Int;
```

Returns the standard height of a line of text in specified font.

```
def font_baseline(font: Int): Int;
```

Gets the distance in pixels from the top of the text to the text's baseline in specified font.

# 9 ui_edit.eh — Constants for text editing UI components.

```
use "ui_edit.eh"
```

## 9.1 Description

This header defines constants that may be used to specify mode of text editing components - EditItem and EditBox. Depending on input mode component may restrict range of available characters or switch to special input mode. For example, EDIT_NUMBER mode accepts only numeric input. In EDIT_PASSWORD mode input is hidden, usually by displaying all characters as `"*"` though platform may use other ways of obscuring.

There is no need to include this header explicitly with `use` directive - it is used included automatically by both form.eh and stdscreens.eh.

## 9.2 Constants

```
const EDIT_ANY = 0;
```

The user is allowed to enter any text. The input is not restricted and may contain line breaks.

```
const EDIT_EMAIL = 1;
```

The user is allowed to enter an e-mail address. The input is restricted to characters allowed in e-mail addresses.

```
const EDIT_NUMBER = 2;
```

The user is allowed to enter only an integer value. The input is restricted to digits and minus sign. Unless the text of component is empty, it will be successfully parsable using String.toint.

```
const EDIT_PHONE = 3;
```

The user is allowed to enter a phone number. The exact set of characters allowed is specific to the device and to the device's network and may include non-numeric characters, such as a "+" prefix character.

```
const EDIT_URL = 4;
```

The user is allowed to enter a URL. The input is restricted to characters allowed in URL addresses.

```
const EDIT_DECIMAL = 5;
```

The user is allowed to enter numeric values with optional decimal fractions such as "-123", "0.123", or ".5". The input is restricted so that only decimal numbers are allowed. Unless the text of component is empty, it will be successfully parsable using String.todouble.

```
const EDIT_PASSWORD = 0x10000;
```

Indicates that text entered in a editbox is confidential data that should be obscured. This mode is useful for entering confidential information such as passwords. This mode can be OR-combined with another `EDIT_*` mode, for example `(EDIT_NUMBER|EDIT_PASSWORD)` is allowed.