Sparkpad

User Manual

(for Developers)

Updated: Nov 7, 2010

Published by

Sparkpad LLC

46090 Lake Center Plaza, #206

Sterling, Virginia 20165

Phone: (888) 907-7275

Fax: (888) 391-7275

Printed in the United States of America

Table of Contents

1. Introduction

Sparkpad is a hardware and software platform solution developed by Sparkpad LLC for companies or individuals seeking an embedded device that is easily customizable. The Sparkpad platform offers various hardware options, including:

- TFT touch screen displays (7”, 8”, 10.1” & 10.4”)
- Non-touch screen (remote operated) displays (15” and above)
- “Video Box” (remote operated) for use with any monitor
- Networking support with both WiFi and Ethernet connections
- A complete Software Development Kit (SDK) and sample applications for creating customized functionality and user interfaces (UI)
- Operating System layer access (Linux)
- A full-featured Emulation and Development Environment for Windows

The iGala Wireless Digital Picture Frame (www.igala.com) is an example of a working commercial application developed on the Sparkpad platform.

This programming guide is intended for software engineers who plan to build customized applications on the Sparkpad platform. It provides information on how to use Sparkpad to develop application extensions and includes sample application code.

1.1. Conventions

The following font conventions are used:

`Code`: This font is used for any device-generated data such as program codes, web page source code and/or log messages generated.

Var: Similar to `code`, but text in this font involves variable identifiers and must be replaced by other meaningful data. There will be a description on how to replace those identifiers when they are mentioned in the document. For example, the command-line arguments:

{code} or {***var***}: Device-generated data enclosed by '{' '}' indicates the data is optional.

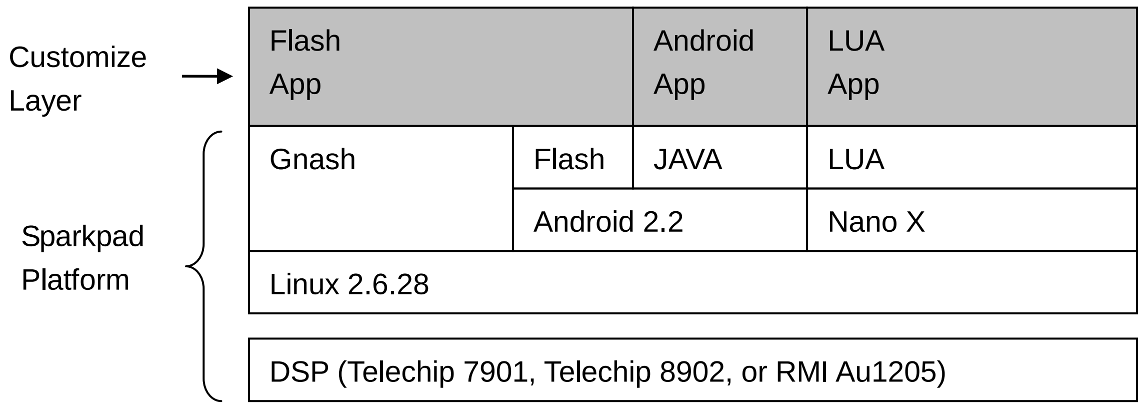
1.2. System architecture

Sparkpad hardware is available in different hardware configurations, including screen size. Some devices are even available without a screen. For each device there may be differences in the core DSP, memory size, board configuration, and interface ports. The following table lists Sparkpad's currently available (or soon-to-be available) hardware configurations:

Screen Size	Screen Resolution	Screen Ratio	Input Type	DSP	Ethernet	WiFi
7"	800x480	16:9	Touch Screen	Telechip 7901	USB Host Port	Marvell (integrated)
8"	800x600	4:3	Touch Screen	RMI Au1250	Optional USB Host	Internal USB module
10.1"	800x480 1024x600	16:9	Touch Screen	Telechip 8902	Included	Marvell (integrated)
10.4"	800x600	4:3	Touch	Telechip	Included	Marvell

	1024x768		Screen	8902		(integrated)
15”	1024x768	4:3	Remote Control	Telechip 7901	Included	Marvell (integrated)
42”	1080x1024	16:9	Touch Screen	Telechip 8902	Included	Marvell (integrated)
Video Box	HDMI & Component Video Output	N/A (No screen)	Remote Control	Telechip 8902	Included	Marvell (integrated)
Video Box	HDMI & Component Video Output	N/A (No screen)	Remote Control	Broadcom	Included	Broadcom (integrated)

All hardware options run a modified version of the Linux OS. On top of Linux, Sparkpad provides three development options for applications: LUA, Flash (coming soon) and Android (coming soon), as illustrated in the following stack chart:



Currently, this document only assists with the development LUA applications (with or without NanoX). Please note that for non-touch screen models, such as the 15” and “box” devices, the LUA layer is built directly on top of Linux without NanoX.

2. LUA application development

This section provides information on how to build applications on the Sparkpad platform using the LUA programming language. Sparkpad provides a LUA-based emulator for your Windows PC, so most development and testing of your application software can be completed before loading it onto the final end hardware.

2.1. Overview

Sparkpad hardware features different DSPs that run Linux. The display driver is configured to support digital LCD displays in their native resolutions. All models include at least 1GB of flash memory and 128MB RAM (most include 256MB).

The flash memory is partitioned into two sections:

- Application partition mounted under `/nand1` folder
- Content partition mounted under `/nand2` folder

Note: Developers should monitor memory consumption during development. The Linux system kernel uses about 15 - 20MB of RAM. The LUA layer handles its own memory management and consumes about 40MB per pool allocation.

Once the system boots up, the application partition will be mounted. The content partition is mounted in the background. Before using this partition, the application should c that the content partition has been properly mounted by checking for the directory `/nand1/load+found`. The system then checks for the existence of shell scripts with the name “`igala_api.sh`” under `/nand1/config`, and runs it automatically.

The development environment is structured as follows:

Operation System	Linux version 2.6.28

GCC	Version 4.1.2
UI	Microwindow nano-X 0.91
Programming Language	LUA

The following drivers are supported:

- LCD display
- Touch-Screen
- Wi-Fi 802.11b/g
- Ethernet
- USB host
- SD Card

2.2. Running applications on a PC

Applications developed for the Sparkpad platform using LUA can be run on any Windows PC (without the need of a Sparkpad hardware device) using the provided emulator. Developers can copy the application directory /nand1/bin to Windows and initiate the application from a Windows command terminal (cmd) using command:

```
lua.exe app.lua
```

In this case, the name of the LUA application file is app.lua, which can be replaced with any other LUA application files.

For standard Sparkpad applications, there is often a background process of fetching network data from online servers with the name of nettask.lua. <u>To completely emulate the applications, the developer needs to initiate both the app.lua and nettask.lua processes from two separate windows command terminals.</u>

2.3. Installing applications on a Sparkpad Device

Once application development is completed in the PC emulation environment, developers can use a USB flash drive or SD memory card to load their application onto the actual device.

To install applications to a device, copy the following `update.sh` file and the zipped bin directory `bin.zip` to the root directory of either a USB flash drive or SD memory card and insert it into the respective port on the device. Lastly, power on the device. The system will execute the `update.sh` file automatically.

Sample of `update.sh`

```
if [ -f "/mnt/SD/bin.zip" ]; then
  unzip -o /mnt/SD/bin.zip -d /nand1/config
  mv -f /mnt/SD/bin.zip /mnt/SD/updated_bin.zip
  echo 3 > /proc/sys/vm/drop_caches

  cd /nand1/

  if [ -d "bin" ]; then
      rm -f -r /nand1/bin
      mv -f  /nand1/config/bin /nand1
  fi
fi
if [ -f "/mnt/OHCI/bin.zip" ]; then
  unzip -o /mnt/OHCI/bin.zip -d /nand1/config
  mv -f /mnt/OHCI/bin.zip /mnt/OHCI/updated_bin.zip
  echo 3 > /proc/sys/vm/drop_caches

  cd /nand1/
  if [ -d "bin" ]; then
     rm -f -r /nand1/bin
     mv -f /nand1/config/bin /nand1
  fi
fi

mkdir -p /nand2/bones/flash/abies
cp -af /nand1/bin/mnt/* /nand2/bones
cd /nand2/bones/flash/abies
mkdir -p photo/net/picasa
cd /nand1/bin
cp igala_api.sh /nand1/config
```

When the install process is complete, the device will restart.

Note that `bin.zip` is renamed automatically upon completion of the install process. This ensures that the device will not be updated again if the developer has not yet removed the USB flash drive or SD card.

Developers can rename or rearrange the network download content in the sample application, as needed, located under /nand2/bones/flash/abies/photo/net/picasa.

2.4. Application invocation

Within the `update.sh` script, the file `igala_api.sh` is copied to the `/nand1/config` folder. When the system boots up, this file will run automatically.

Sample of igala_api.sh

```
if [ -f "/mnt/SD/bin.zip" ] && [ -f "/mnt/SD/update.sh" ]; then
  cd /nand1/bin
  ./lua update_info.lua
  aplay -B 4096 frameworks/frameImages/alert.wav
  cd /mnt/SD
  chmod +x update.sh
  ./update.sh
  mv update.sh update_pre.sh
  reboot
fi
sleep 3
if [ -f "/mnt/OHCI/bin.zip" ] && [ -f "/mnt/OHCI/update.sh" ]; then
  cd /nand1/bin
  ./lua update_info.lua
  aplay -B 4096 frameworks/frameImages/alert.wav
  cd /mnt/OHCI
  chmod +x update.sh
  ./update.sh
  mv update.sh update_pre.sh
  reboot
fi
####remote update####
if [ -f "/nand1/bin/update.sh" ]
then
  rm -f /nand1/bin/update.sh
fi
if [ -f "/nand1/bin.zip" ]
then
  unzip -o /nand1/bin.zip -d /nand1
  rm -f /nand1/bin.zip
  echo 3 > /proc/sys/vm/drop_caches
  if [ -f "/nand1/bin/update.sh" ]
```

```
    then
        cd /nand1/bin
        chmod +x update.sh
        ./update.sh
    fi
    rm -f /nand1/bin/update.sh
    echo 3 > /proc/sys/vm/drop_caches
fi

cd /nand1/bin
chmod +x ln.sh
chmod +x lua_8
chmod +x lua
chmod +x alert_play.sh
./ln.sh
./lua_8 nettask.lua&
sleep 2
./lua app.lua&
./lua_monitor&
Telnetd&
```

The first two sections of the `if-fi` code are to trigger the update process from USB/SD card, if present. The script starts the following three Sparkpad processes and the telnet daemon:

app.lua – Rendering of all UI

nettask.lua – Performs all network download activities

lua_monitor.lua – Monitors the health of application and nettask processes

2.5. Access Linux shell

The OS shell on the frame can be accessed only after the WiFi or Ethernet connection has been configured and is running. From a Telnet client on a computer connected via the same network, run:

```
telnet {IP address of the device, such as 192.168.0.102}
```

The shell access account is root/Udo. If successful, it will enter the Linux shell on the frame. You can run any shell script command to debug your application.

2.6. Network setup

Developers can manually set up the network using LUA code, or from a shell command line, using the following commands:

Ethernet:

- Ifconfig eth0 {IP address like 192.168.0.102}

WiFi:

- insmod /lib/modules/sd8686.ko
- iwconfig eth1 essid “{SSID of the wifi network}”
- ifconfig eth1 {IP address like 192.168.0.102}

Please note: For both WiFi and Ethernet, the previously established network must be disconnected before a new network attempt is attempted.

2.7. Touch screen, remote controls & more

To be provided. Please contact support@sparkpad.com meanwhile with questions.

2.8. LUA programming

This section describes how to develop a Sparkpad application using the LUA scripting language.

The Sparkpad SDK includes a LUA executable for both Windows PC and the hardware device. Developers can execute their LUA applications with commands such as "`lua.exe app.lua`" in Windows PC environment and get the exact same application behavior as they would on an actual Sparkpad hardware device. (One important exception: the emulator borrows the network connection from the PC and thus the network configuration process does not reflect the true experience on an actual device.)

Sparkpad strongly recommends that developers who are new to the platform begin by attempting to modify top-layer LUA application code, as provided with the SDK. All interactions with devices, such as the Wi-Fi module and the USB port, are handled in the provided application. The encrypted LUA files come with the Core Application code are compiled LUA codes

2.8.1. File listing

Directory Calendar (Touch screen only)

Directory frameworks

Directory mnt

Directory sdk

app_bone.lua

Main entry of Sparkpad LUA Bones Application (Touch screen only)

do_calendar.lua

do_gmail.lua

do_picasa.lua

fpDisplay.lua

fpFrequency.lua

fpGmail.lua

fpPicasa.lua

fpPicasaAlbum.lua

fpPowerSave.lua

fpTransition.lua

fpWifi.lua

fpWifiKeyboard.lua

fpWifiType.lua

fsCalendar.lua

fsMainMenu.lua

fsMedia.lua

fsSetup.lua

gmail.lua

gmail_setup.lua

media_manager.lua

nettask.lua

picasa.lua

playlist.lua

setting.lua

system_manager.lua

videoplay.lua

wlan.lua

wlan_check.lua

2.8.2. How to create a screen with buttons

The following example shows how to create a screen with buttons.

```
function create(scene, back_page)
  local page = glyph.glyph(scene,0,0,1024,768)  -- 15 inch screen
resolution here
  local page_title = component.text(page, 'Title here',
gui.WHITE,nil,font_factory.load('frameworks/font/arial.ttf',
50),bit._or(glyph.Alignment.hcenter, glyph.Alignment.vcenter))
  page_title:hcenter()
  page_title:offset(0, 20)

  local function create_buttons(parent, ...)
      local push_arg = {
            parent = parent,
            color = gui.WHITE,--WHITE,
            image = 
'frameworks/frameImages/blue_button_extra_big.png',
            caption = 'button',
            alignment = bit._or(glyph.Alignment.hleft,
glyph.Alignment.vcenter),
          font = font_factory.load('frameworks/font/arial.ttf',
21),
            no_auto_pop=true
         }
      local ret ={}
      local y = 200
      for _, f in ipairs(arg) do
         push_arg.caption = f
         ret[#ret + 1] = 
component.push_button(component.push_button_arg(push_arg))
         ret[#ret]:move(200, y)
         y = y + 60
      end
      return ret
  end


  local button = {}
  button = create_buttons(page, "button text here")

  local zone_list = {
      {-6, "  (UTC-06:00) Central Time (US & Canada)"},
      {-5, "  (UTC-05:00) Eastern Time (US & Canada)"},
      {-4, "  (UTC-04:00) Atlantic Time (Canada)"},
      {-3, "  (UTC-03:00) Buenos Aires"},
      {-2, "  (UTC-02:00) Mid-Atlantic"},
      {-1, "  (UTC-01:00) Cape Verde Is."},
      {-12, "  (UTC-12:00) International Date Line West"},
      {-11, "  (UTC-11:00) Midway Island, Samoa"},
      {-10, "  (UTC-10:00) Hawaii"},
      {-9, "  (UTC-09:00) Alaska"},
      {-8, "  (UTC-08:00) Pacific Time (US & Canada)"},
      {-7, "  (UTC-07:00) Mountain Time (US & Canada)"},
      {0, "  (UTC) GMT : Dublin, Edinburgh, Lisbon, London"},
      {1, "  (UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm,
Vienna"},
      {2, "  (UTC+02:00) Athens, Bucharest, Istanbul"},
      {3, "  (UTC+03:00) Moscow, St. Petersburg, Volgograd   "},
      {4, "  (UTC+04:00) Caucasus Standard Time"},
      {5, "  (UTC+05:00) Islamabad, Karachi"},
      {6, "  (UTC+06:00) Almaty, Novosibirsk"},
      {7, "  (UTC+07:00) Bangkok, Hanoi, Jakarta"},
      {8, "  (UTC+08:00) Kuala Lumpur, Singapore"},
      {9, "  (UTC+09:00) Osaka, Sapporo, Tokyo"},
      {10, "  (UTC+10:00) Canberra, Melbourne, Sydney"},
      {11, "  (UTC+11:00) Magadan, Solomon Is., New Caledonia"},
      {12, "  (UTC+12:00) Auckland, Wellington"},
  }
  local txt = component.text(page,'Current:',
gui.WHITE,nil,font_factory.load('frameworks/font/arial.ttf',
25),bit._or(glyph.Alignment.hleft, glyph.Alignment.vcenter))
  txt:move(200,100)
  local txt_next_caption = component.text(page,' ',
gui.WHITE,nil,nil,bit._or(glyph.Alignment.hleft,
glyph.Alignment.vcenter))
```

```
 txt_next_caption:move(300, 100)
 txt_next_caption:set_caption(setting.abies_config.zonetime[1
])
 local argu = {
    parent = page,
    caption = ' ',
    image = 'frameworks/frameImages/blue_triangle_left.png',
    color = gui.WHITE
 }
 local prebtn =
component.push_button(component.push_button_arg(argu))
 argu.image = 'frameworks/frameImages/blue_triangle_right.png'
 local nextbtn =
component.push_button(component.push_button_arg(argu))
 argu.image = 'frameworks/frameImages/green_button_small.png'
 argu.caption = 'Done'
 argu.font = font_factory.load('frameworks/font/arial.ttf', 30)
 local done =
component.push_button(component.push_button_arg(argu))
 prebtn:move(200, 600)
 layout.in_row_center(prebtn,{done, nextbtn}, 100)

 function button_caption(current_zone, pre_zone)
    local zone = (current_zone - pre_zone) * 3600
    local real_hour
    local os_time_year, os_time_month, os_time_date,
os_time_hour, os_time_minute, os_time_noon =
time_manager.get_time()
    if os_time_noon == "PM" and tonumber(os_time_hour) ~= 12 then
       real_hour = os_time_hour + 12
```

```
     else
        real_hour = os_time_hour
     end
     if setting.abies_config.daylight_config["dst"] == 1 then
        real_hour = real_hour - 1
     end
     local changetime = os.time{year = os_time_year, month =
os_time_month, day = os_time_date, hour = real_hour, min =
os_time_minute,isdst =
setting.abies_config.daylight_config["dst"]}
     local real_time = changetime + zone
     local d = os.date("*t",real_time)
     if constant.WIN32 then
        os.execute("date " .. d.year .."-".. d.month .."-"..
d.day )
        os.execute("time " .. d.hour ..":".. d.min ..":".. d.sec )
     else
       os.execute("date -s \"" .. d.year .. "-" .. d.month .. "-" ..
d.day .. " " .. d.hour .. ":" .. d.min .. ":" .. d.sec .. "\"")
     end
  end


  local flag = 0
  local index = 0
  local function click_zone(i)
     return function()
        windows:add( function()
           setting.load(setting.abies_config)
           local zone_pre = setting.abies_config.zonetime[2]
           local list_index  = ((i + index) % 25 + 24) % 25 + 1

  txt_next_caption:set_caption(zone_list[list_index][2])
           setting.abies_config.zonetime[1] =
zone_list[list_index][2]
           setting.abies_config.zonetime[2] =
zone_list[list_index][1]
           now = zone_list[list_index][1]
           local dlg = component.dialog(page, nil, '\n\nTime Zone
has been  successfully changed.',
bit._or(glyph.Alignment.vcenter, glyph.Alignment.hcenter),
'Done', nil,0, -20)
           if dlg:run() == 'Done' then
              setting.save(setting.abies_config)
              button_caption(now, zone_pre)
           end
        end)
     end
  end

  function on_click()
     windows:add(
        function()
           for i = 1 ,#button do
              local list_index = ((i + index) % 25 + 24) % 25 +
1
              button[i]:set_caption(zone_list[list_index][2])
           end
        end
     )
```

```
end
prebtn.on_click = function()
            if(flag > 0) then
               flag = flag - 1
            end
            if flag < 0 then
               flag = 0
            end
            index = flag * 6
            on_click()
         end

nextbtn.on_click = function()
            flag = flag + 1
            if flag > 25 then
               flag = 0
            end
            index = flag * 6
            on_click()
         end

for i = 1, #button do
   button[i].on_click = click_zone(i)
end

function done:on_click()
   page:hide()
   back_page:show()
end
```

```
   return page
  end
```

In the above code, a screen was created with a page title, text, a number of time zone buttons and a group of action buttons at the bottom of the page, including “previous page”, “next page” and “done” buttons. Developers can move the location of the buttons using screen coordinates. Button actions are defined in the `on_click` functions.

2.8.3. How to parse an XML file

With Sparkpad, developers can easily send a network request to an online server with input parameters, and then parse the returned XML into local content. In the following example, the developer first needs to construct the requesting URL with input parameters into a string of “url”.

```
   local http_content = http_request(url,
  setting.abies_config.proxy_config)
   if(http_content) then
      local result, result_set = pcall(collect, http_content)
      if not result then return nil, 115, "http request " .. url ..
  " get invalid content" end
      if (result_set[2].label == "xml label here") then
         local set = result_set[2][1]
         ob.weekday = set[1][3].xarg.abbrv
         ob.station = set[3][1]
         return ob, set[1], nil
      else
         return nil, 116, "http request " .. url .. " parse error"
      end
   else
```

```
      return nil, 404, "http request " .. url .. " get no content"
   end
```

In this example, the developer can retrieve XML attributes as well as the arguments from the incoming XML as long as the location and the sequence of the elements are known.

2.8.4. How to enable automatic remote firmware update

By following the sample in `autoupdate.lua`, developers can enable their application to check for firmware updates periodically.

2.8.4.1. Check Version

Here is the sample of how to check the current firmware version against the version available on the remote server. This code can be invoked at a different frequencies, depending on your needs:

```
local http_content, code, status
local update_version
if sysutil.fileexists('../' .. 'update_version.dat') then
  print('get version from cache file')
  update_version = sysutil.readlog('../' ..
'update_version.dat')
else
  print('get version from website')
  http_content, code, _, status = http.request(BASE_URL ..
VERSION_FILE)
  if tonumber(code) ~= 200 then
      print('http request version failure,', status)
      return
  end
  update_version = http_content
  sysutil.writelog('../' .. 'update_version.dat',
update_version)
end
local current_version = sysutil.readlog('version.txt')
print('update version:', update_version)
print('current version:', current_version)
if (current_version ~= update_version) then
  print('version changed, need upgrade')
else
  print('version is the latest.')
  sysutil.writelog('../' .. 'auto_update.dat',
tostring(os.time()))
  os.remove('../' .. 'update_version.dat')
  return
end
```

2.8.4.2. Get the new firmware package

After deciding a firmware update is needed, the following sample can be used to fetch the new upgrade package and initiate a self-installation:

```
local update_md5  -- for integrity check
if sysutil.fileexists('../' .. 'update_md5.dat') then
  print('get md5 from cache file')
  update_md5 = sysutil.readlog('../' .. 'update_md5.dat')
else
  print('get md5 from website')
  http_content, code, _, status = http.request(BASE_URL ..
MD5_FILE)
```

```
  if tonumber(code) ~= 200 then
     print('http request md5 failure')
     return
  end
  update_md5 = http_content
  sysutil.writelog('../' .. 'update_md5.dat', update_md5)
end

if sysutil.fileexists('../' .. 'file_to_get.zip') then
  print('zipfile exists, check md5')
  sysutil.clean_memory()
  local md5value = md5util.md5_file('../' .. file_to_get.zip')
  sysutil.clean_memory()
  if md5value ~= update_md5 then
     os.remove('../' .. ' file_to_get.zip')
  else
     print('zip file md5 check success')
     sysutil.writelog('../' .. 'auto_update.dat',
tostring(os.time()))
     os.remove('../' .. 'update_md5.dat')
     return
  end
end
print('getting zip file ...')
b, msg = get(BASE_URL .. ZIP_FILE, '../' .. ' file_to_get.zip' ..
'!')
if b then
  sysutil.clean_memory()
  local md5value = md5util.md5_file('../' .. ' file_to_get.zip' ..
'!')
```

```
    sysutil.clean_memory()
    if md5value ~= update_md5 then
       print('md5 check failure')
       os.remove('../' .. ' file_to_get.zip' .. '!')
       os.remove('../' .. 'update_md5.dat')
       return
    else
       print('md5 check success')
       os.remove('../' .. 'update_md5.dat')
       os.rename('../' .. ' file_to_get.zip' .. '!', '../' .. '
  file_to_get.zip')
       sysutil.writelog('../' .. 'auto_update.dat',
  tostring(os.time()))
```

2.8.5. Wi-Fi connection: wlan.lua

After calling wlan_on( ) , the network connection may fail. That could be cause by the delay of the wireless access point association and the allocation of DHCP. The call of wlan_stats ( ) may return the status of “connecting” during this time.

To get to the true status of a WiFi connection in real time, the code should use wlan_check( ) instead of wlan_on ( ) and wlan_status( ). However, the wlan_check( ) call may take up to 1-2 minutes to return a response, depending on how long it takes to connect to the access point.

The code in wlan.lua file shows the use of wlan_on( ), wlan_status( ) and wlan_check( ). Remember that a WiFi connection may not be successful even after the call of wlan_on( ).

Note that WiFi connection code can be completed before the GUI (graphical user interface) starts.

2.8.6. Memory management

Memory management is handled automatically in LUA. There is no need to free or allocate memory in the code. It uses a "Mark and Sweep" approach for garbage collection. Some more complicated cases involve the use of "Weak Reference", of which you will find several instances in the Sparkpad source code. Other than memory consumption by regular data structures, the closure routine in LUA could also contain references to some items, so please pay special attention to large data items such as images within the closure routines.

You can always trigger the garbage collection manually using collect_garbage(). Within the provided system, it is being done once for every 10 message event handling in the message loop, so that memory is freed faster.

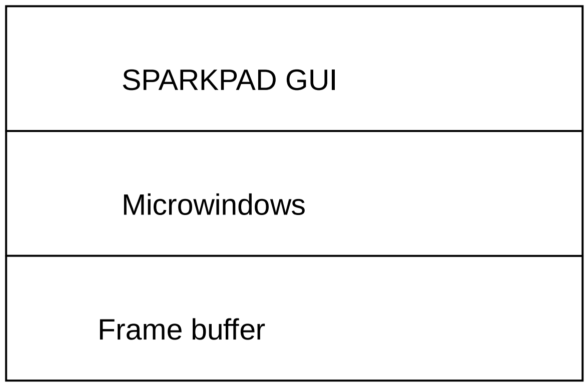
The consumption of the memory will NOT be reflected by checking the output of the system command "top". That is because memory management is done in a layer above the OS. Two big chunks of memory, each with a size of 16MB, are allocated when the process is started, and the memory manager handles them internally throughout the lifetime of the process. As a result, any attempt to use more than 16MB, or three chunks of 16MB, will cause internal errors.

Sparkpad's enhanced memory management system causes less defragmentation compared to the buddy algorithm used by Linux. However, there are also cases where LUA runs out of usable memory space, even though there is actually space left within the two 16MB chunks. This is trade off allows the embedded system to work more smoothly, and run longer, using less internal memory.

2.8.7. GUI

This section provides a general description of GUI components. Sparkpad GUI components provide functionality that is somewhat parallel to the Microwindows system, including window manipulation and some basic graphics functions.

The overall architecture looks like the following:



At the bottom is the frame buffer and touch panel drivers, which provide elementary touch panel input and screen output functions. Above that is a full Microwindows system and a Sparkpad GUI components layer. Sparkpad utilizes Microwindow's message loop as the underlying input facility and replaces its output with Sparkpad's own functions. Therefore, all Microwindow functions could be used in a Sparkpad environment. However Sparkpad usually doesn’t interact with the Microwindows system. The first decision for a GUI developer to make is which library to use as the GUI framework.

To help with this decision, developers should understand these core differences between Sparkpad and Microwindows:

Sparkpad only provides limited graphic functions. Only JPEG and PNG image and text output are supported. There are no other graphic primitives like cycle, ellipse, line, rectangle or polygon in Sparkpad. Conversely, Microwindows provides all those primitive graphic functions.

Sparkpad doesn't support window clipping, which means the graphic output in one Sparkpad component could easily overlap with other graphic outputs from other Sparkpad windows. This feature (or limitation) impacts application design in the following way:

It's easier to perform alpha blending in Sparkpad than in Microwindows.

There will be a performance impact if the UI is organized in a deep, hierarchical way. Because the output in the parent window won't be clipped off by the child window, even if it is not visible on the screen.

Sparkpad provides better image manipulation functions, and usually those functions outperform Microwindows on large image files.

2.8.7.1. LUA object model

(Please ignore this section if you choose Microwindows as your GUI framework.)

Most, though not all, of the Microwindows functions and constants are exported to LUA without any changes. Please review Microwindows documentation to get complete information on how to develop GUI programs in Microwindows.

If you are familiar with LUA, then you already know that LUA is not an object-oriented programming (OOP) language. It does provide enough facility to implement an object-oriented GUI library, however. This sections describes the approach used by the Sparkpad platform to establish an OOP framework in LUA.

The first concept that developers need to understand regarding the OOP framework of LUA is the concept of “closure” of LUA. Closure is a powerful language facility included with LUA that is a function with its own local states. You can treat closure as a function combined with its stack frame at the moment that the closure is created.

Here is basic example of closure:

```
function create_closure(x)
  return function()
          x = x + 1
          return x
      end
end

function use_closure()
  local closure = create_closure(5)
  print(closure())     --this will output 6 on the console
  print(closure())     --print 7 on the console
end
```

Function '`create_closure`' will return a closure that encapsulates a local variable x (which is ‘5’ at the moment of its creation) on its stack frame. And every time the closure is called, the variable will increment by 1 and be returned from that closure, so it will print ‘6’ on the console, and ‘7’ for the second time. As a functional language, the positive thing about LUA is that closure is a first class entity in the LUA language. It can be returned from a function, or be passed as an argument to a function or stored in a variable. The variable in a closure could be anything that is legal in the LUA language. In the example above, the variable is a number, but it could also be a table or another function/closure.

The second concept developers need to understand is the ':' syntax of the LUA language. The ':' in LUA is similar to '->' in C++, and 'object:methos()' in LUA means the same thing as 'object->method()' in C++. The details behind this syntax are described in LUA's full documentation.

With closure and ':' syntax, we have enough to implement an object-oriented framework in LUA. An object within the Sparkpad environment is simply a collection of closures that share some local variables in common and support inheritance and override.

Here is how it works:

```
function constructor(x)
  local y = 0
  local object = {}  --create an empty table
  function object:method1()
     x = x + 1
  end
  object.method2= function(obj)
     y = y + 1
  end
  function object:dump()
     print(x,y)
  end
  return object
end
local obj = constructor(5)
obj:method1() -- x = x + 1
obj:method2() -- y = y + 1
obj:dump()     -- print 6, 1
```

The following code illustrates how to inherit objects and the override method:

```
function sub_object(x)
  local super = constructor(x)
  local super_dump = super.dump
  function super:dump()
     print('super')
     super_dump(self)
  end
end
local s = sub_object(16)
s:method1()
s:method2()
s:dump()     -- print out 17, 1
```

The function `constructor` initializes an object with its argument x and a local variable y and three methods (method1, method2 and dump). Those three methods are closures and are stored in the returned object, which is a LUA table. The ':' syntax is used to define and call those methods. In this object, x is an input argument of `constructor` and y is a local variable. These two variables are shared by 'method1', 'method2' and 'dump'.

The function `sub_object` creates an object that is inherited from the object created by `constructor` and overrides its 'dump' method. So method1 and method2 could be called on `sub_object` without any change in its semantics. Method `dump` is overridden, which prints 'super' followed by its super method's output.

In the above example, there are two different syntaxes to define methods of an object. They are semantically equal.

```
1) local object = {}
   object.method = function(obj)
       ...
   end
2) local object={}
   function object:method()
       ...
   end
```

Both methods could be invoked by obj:method(). For additional context, please refer to LUA's full documentation for details.

So far, the essence of OOP/OOD can be conveniently represented in LUA. The major difference between Sparkpad's object model and other popular object models, such as C++/Java, is that Sparkpad has no classes. The terms ‘class’ and ‘object’ are used interchangeably in the following section. These patterns are used in Sparkpad’s library universally.

2.8.7.2. Runtime behavior of the Sparkpad GUI program

The cornerstone of Sparkpad GUI library is `glyph.lua` and `microwindow.lua`, which implement the fundamental GUI elements and interface with Microwindows' message loop, respectively.

Every Sparkpad GUI program begins with a call like “windows = Microwindow.create()”. Please note the variable name of "windows" is mandatory. This statement will initialize the Microwindows system, set up the root window and message filter, and create other data structures in Sparkpad’s GUI library.

The return object of Microwindow.create() is an object that interacts with Microwindows’ message queue. The most important one: `windows:process_message(timeout, event, allow, forbidden)`. This function drives the main message loop (all of its arguments are optional), so the simplest usage is windows:process_message(), which means: wait until the next event arrives and then dispatch it to the root Sparkpad scene that attached to windows previously. The root Sparkpad scene should be registered to windows by windows:attach(scene) before calling windows:process_message. Then the root scene will determine which component should be responsible for handling the message. For example, if a touch panel message arrives, the windows object will deliver the message to the current scene object and the scene will check the coordinates in the message. It will then dispatch accordingly.

The scene is defined in `glyph.lua` and it is the root window of the Sparkpad GUI program. Developers can create as many scenes as they want if there is sufficient memory, however only one scene can be attached to windows as the current scene. Scene is derived from glyph which is the root class of any GUI element, therefore any method that is applicable to a glyph is also applicable to a scene. Additionally, a scene is different from a glyph in the following aspects:

A scene carries with an off-screen buffer of size 800*600 pixels.

A scene carries with an optional background image, and the image could be changed by calling scene:set_background(image)

Scene can't be sub window of another window.

Scene always covers full screen of 800*600 pixels.

Now, here’s an example of a very basic Sparkpad GUI program:

```
require "Microwindow"
require "glyph"
windows = Microwindow.create()
local scene = glyph.scene()
scene:set_background('background.jpg')
local quit
function scene:on_pendown()
  quit = true
end
windows:attach(scene)
while(not quit) do
  windows:process_message()
end
```

This program creates an empty scene with a background image, and quits when the user taps the screen.

There are several concepts in the above code worth noting:

There is no 'local' keyword before variable 'windows', as 'windows' is a reserved global name in Sparkpad (not in LUA).

All methods of windows objects are called by the ':' syntax, while the windows object is created by '.' syntax. This is because Microwindow.create() is a function of module Microwindow, not a method of an object.

There is no problem with overriding the message handler in the main program. In fact, any method can be overridden anywhere.

The code above demonstrates the basic elements of a Sparkpad GUI program, but of course developers must use more components in most scenes. A Sparkpad component is anything that inherits from glyph directly or indirectly. A glyph is simply a window in the Sparkpad environment. It defines the fundamental behavior of GUI elements, such as show, hide, move, draw image or text, add a child glyph, etc.

There are several predefined basic components in `component.lua`, including button, label, single line editor, panel, thumbnail view, sandglass and dialog. If a developer's desired component is not in the list, or the given component doesn't fulfill all requirements, then the developer must implement a solution by overriding specific methods in glyph. In these cases, the code in `component.lua` is a good reference for implementing custom components.

2.8.7.3. Implement your own components

There are four methods in glyph that are intended to be overridden by developers:

`on_pendown`

`on_penup`

`on_penmove`

`on_draw`

The first three methods deal with the input from a touch panel. The last method is in charge of output on the screen. The following example brings these together in a simple component that displays a picture and invokes a user-defined method when the user selects the picture:

```
function picture(p, img, on_pendown)
  local g = glyph.glyph(p)
  img = display.to_image(img) --load image if it is a file name
  function g:on_draw(gc)
      print('on_draw')
      gc:blit(0,0, img, 0, 0, img:width(), img:height())
  end
  g.on_pendown = on_pendown or g.on_pendown
  return g
end
```

The function `picture` is the constructor of our new component, which takes three arguments. The first argument the parent glyph, the second argument is an image object or the file name of an image, and the third argument is a user-defined function. Here is how to use this new component:

```
require "Microwindow"
require "glyph"
require 'display'

local function picture(p, img, on_pendown)
  local g = glyph.glyph(p)
  img = display.to_image(img) --load image if it is a file name
  function g:on_draw(gc)
```

```
      print('on_draw')
      gc:blit(0,0, img, 0, 0, img:width(), img:height())
   end
   g.on_pendown = on_pendown or g.on_pendown
   return g
  end

  windows = Microwindow.create()
  local scene = glyph.scene(800,600)
  local quit
  local pic = picture(scene, 'frameImages/bg1.jpg',
                function()
                   quit = true
                   print ('quit')
                end)
  pic:move(0,0,scene:width(), scene:height())
  windows:attach(scene)
  while(not quit) do
   windows:process_message()
  end
```

In the above example, aside from the minimal Sparkpad GUI program and the definition of our new component, there are two lines before entering the message loop:

```
  local pic = picture(scene, 'frameImages/bg1.jpg',
                function()
                   quit = true
                   print ('quit')
                end)
  pic:move(0,0,scene:width(), scene:height())
```

The first line creates a new component by calling function `picture` and the second line displays the component as big as the scene. All components in the Sparkpad environment are created and used in this way – they mostly differ in how they respond to user inputs and how they perform the screen output. Please review glyph.txt for more details.

2.8.8. Audio Volume Control: music_manager.lua

The audio control is defined under file `music_manager.lua`. Here is sample code for controlling the volume:

```
local volume = 220
   os.execute("echo " .. volume .. " >
/proc/asound/wm8750L/wm8750_vol”)
```

Please note that while the legal value of variable `volume` is between 0 and 255, the value should always remain under 230.

2.8.9. Component: component.lua

This section describes the components in `component.lua`. Not all functions are listed here. This section only lists those that (1) have a clear semantic/interface, (2) are not specific to a certain application or (3) demonstrate important concepts.

`Component.lua` includes the implementation of most UI components. Usually the constructor of those components has many arguments, and many of them are optional and could be given a nil value when called with the constructor. However, if an argument has a value then all the arguments before it should be specified. To simplify the invocation of the constructor, a little trick is commonly used in `component.lua`.

For instance, function `panel (p, pixmap, caption, color, align, txt_margin,` `w, h, isSingleLine, n)` has 10 arguments, and it would be cumbersome to write down each of them every time we called the constructor. So, we use a helper:

```
function panel_arg(p)
  return p.parent, p.pixmap, p.caption, p.color, p.alignment,
p.text_margin, p.width, p.height, p.is_single_line, p.name
end
Tis function takes a table and flatten all its member fields. Then
we could have a much cleaner interface like:
local arg={
  pixmap='1.jpg',
  caption='hello world'
}
local p = panel(panel_arg(arg))
```

In the above example, only two arguments are given, and the rest are omitted.

2.8.9.1. Panel

```
function panel(p, pixmap, caption, color, align, txt_margin, w,
h, isSingleLine, n)
```

This function creates a panel. A panel is a glyph that can display a label and an image.

p:parent

pixmap:the initial value of the background image.

caption:the initial value of the text label

color:the color of the text

align: the alignment of the text

txt_margin: the margin around the text label

w,h: the width, height of the component

isSingleLine: a boolean value to indicate if the text should be displayed in one line

n:name of the component

2.8.9.2. Image Functions

```
function s:get_picture()
```

```
function s:set_picture(pic)
```

Get/change the background image, where 'pic' could be the image file name or an image object.

```
function s:set_color(clr)
```

```
function s:get_color()
```

Get/change the color of text.

```
function s:set_align(al)
```

```
function s:get_align()
```

Get/set the alignment of text, where the alignment should be the bitwise 'or' of glyph.Alignment.

```
function s:set_text_margin(ml)
```

```
function s:get_text_margin()
```

Get/set the margin around the text.

```
function s:set_caption(cp)
function s:get_caption()
```

Get/set the text caption.

```
function scrollbar(p, finished, unfinished, markup, markdown)
```

Create a horizontal scrollbar.

p:parent

finished: the image of a 100% scrollbar

unfinished: the image of a 0% scrollbar

markup: the image of a released handle

markdown: the image of a pressed handle

```
function bar:set_percent(pct, has_event)
function bar:get_percent()
```

Get/set the finished percentage of the scrollbar, where argument, has_event determines if an on_change event should be fired.

```
function sand_glass(x, y)
```

Create a sand_glass animation at x,y on the screen. The default value of x,y is the center of the screen.

```
function picture(p, img, margin)
```

Create a glyph to display an image, where the created glyph has the same size of the image plus the margin around it.

p:parent

img: the file name of the image or the image object

margin: the margin around the image

```
function text(p, caption, color, margin, fnt, align)
```

Create a glyph to display text.

p:parent

caption: the text to display

color: the color of the text

margin: the margin around the text

fnt: the font of the text, currently only one font face is supported, so the only option is a font with a different size

align: the alignment of the text

```
function g:set_caption(c)
function g:get_caption(c)
```

Get/set the caption of the glyph.

2.8.9.3. Glyph

```
function deligate(obj1, obj2, ...)
```

Typically a UI component is built from simpler primitive components. For instance, a dialog box is composed of a button, a label and their parent, the dialog itself. When the user selects the ‘OK’ button, the dialog box will be closed. In the Sparkpad environment, every event is delivered to the leaf component.

```
|--------------------------------------------------|
|                 Are you sure?                   |
|                                                 |
|                   |-----------|                 |
|                   |   OK   |                    |
|                   |-----------|                 |
|--------------------------------------------------|
```

Therefore, to handle the on_pendown event of selecting the ‘OK’ button, the developer would need to write the following:

```
function create_dialog()
   local dialog = glyph()
   local ok_button = create_button(dialog)
   function dialog:get_ok_button()
      return ok_button
   end
   return dialog

end
```

And in the application code, the developer would write:

```
dialog:get_ok_button().on_pendown = function(self, x,y)

  ...

End
```

This function will hook a handler in the `ok_button`. The function `dialog:get_ok_button` is mandatory, since we can't access `ok_button` outside the closure of `create_dialog`. Another solution is to call dialog's `on_pendown` in the handler of `ok_button`:

```
function create_dialog()

   local dialog = glyph()

   local ok_button = create_button(dialog)

   function ok_button:on_pendown(x,y)

      return dialog:on_pendown(x,y)

   end

   return dialog

end
```

…then in the application code, the developer would include:

```
dialog:on_pendown = function(self, x,y)

  ...

end
```

Both solutions are tedious, and would be even more cumbersome if a deeper hierarchical component tree were built. In practice, when moving from basic components to more complex ones, it's a common requirement to expose the event handler of the sub-component to its parent. Function `deligate` can help in these situations. Using `deligate`, the code becomes:

```
function create_dialog()

   local dialog = glyph()

   local ok_button = create_button(dialog)

   glyph.deligate(ok_button,
dialog,'on_pendown','on_penup','on_penmove')

   return dialog

end
```

… and then in the application code, the developer would write:

```
dialog:on_pendown = function(self, x,y)

  ...

End
```

Please note, in the above example, not only `on_pendown`, but also `on_penup` and `on_penmove` are all exposed to the dialog in a single function call.

`function deligate` has two fixed arguments, the first is the source of the event and the second is the receiver of event. Then the function expects a serial of function/event name. (In the Sparkpad environment, there is no difference between event and function).

```
function inherit(obj, method, overriden, mode)
```

In object oriented programming it's common to override a method of base class and calling base class's implementation in the function body.

```
function base_class()
    local base = {}
    function base:method()
        ...
    end
    return base
end
```

…then in the sub-class:

```
function sub_class()
    local sub_class = base_class()
    local method = sub_class.method
    function sub_class:method()
        method()
        ...
    end
    return sub_class
end
```

In this example, to call the implementation of base class, we have to store the base class' implementation in a local variable before we use it.

Function `inherit` could make this job a little easier. With inherit, the code looks like:

```
function sub_class()
   local sub_class = base_class()
   glyph.inherit(sub_class, 'method',function()  ... end, 'so')
   return sub_class
end
```

This function takes four arguments:

the first is the object whose method needs to be overridden;

the second is the name of the method;

the third is the new implementation of the method;

the fourth is a string that tells `inherit` how to deal with base class's implementation.

If the mode argument is 'so' then the super-class' implementation will be called before calling the overridden version. If this argument is 'os' then the overridden version will be called before super-class’ version. Otherwise, the super-class' version is totally omitted.

```
function gc(img, delta_x, delta_y,  w, h, uplist)
```

This function is the constructor of GC object, which is the first argument of `glyph:on_draw`. So it's not necessary to call this function directly. A GC object encapsulates the coordinates transformation and several methods to perform graphical operations.

```
function gc:blit(x1, y1, src, imgx, imgy, imgw, imgh)
```

This function will bitblit an image to the GC object.

x1,y1 is the left top corner in local coordinates

src, is the image to draw

imgx,imgy,imgw,imgh is the rectangle in image's coordinates

```
function gc:blend(x1, y1, src, imgx, imgy, imgw, imgh, alpha)
```

This function is similar to blit, except it performs alpha blending instead of a raw bitblit.

x1, y1, src, imgx, imgy, imgw, imgh are the same as blit (see above)

alpha, 0 < alpha < 255, specifies the value of alpha channel.

```
function gc:fill(color, alpha, x1,y1,w1,h1)
```

Draws a rectangle on GC.

color: the color of the rectangle

alpha: the alpha value of the rectangle

x1, y1, w1, h1: the coordinates of the rectangle (default is the extent of the target glyph)

```
function gc:text_out(txt, color, align, x1, y1, w1, h1)
```

Outputs text on the GC object.

txt: the output string

color: the color of the text, default is gui.BLACK

align: specifies the alignment of the text within the rectangle of (x1,y1,w1,h1); default is the bitwise 'or' of Alignment.hcenter and Alignment.vcenter

x1, y1, w1, h1: the coordinates of the rectangle (default is the extent of the target glyph) `function glyph(p,x, y, w, h, n)`

This function creates a blank glyph

p: parent should be a glyph or nil for root glyph

x, y, w,h: the initial value of the glyph's extent

n: the name of the glyph

`function g:set_enable(e)`

`function g:get_enable()`

Get or set the enabled property of a glyph to enable or disable the glyph. (A disabled glyph cannot a receive a mouse event.)

`function g:get_root()`

Retrieve the root glyph of the given glyph.

`function g:set_user_data(u)`

`function g:get_user_data()`

Get or set user data of a glyph. The user data of glyph is arbitrary data provided by the user, and the platform doesn't imply any structure to this data or interpretation of it.

`function g:get_visible()`

Check if the glyph is visible.

```
function g:get_children()
```

Return a table of all children of the glyph.

```
function g:top()
function g:left()
function g:width()
function g:height()
```

These four functions return the extent of the glyph.

```
function g:size()
```

Returns the width and height as a tuple.

```
function g:pos()
```

Returns left and top as a tuple.

```
function g:get_parent()
```

Return the parent of the glyph, or nil in case of root glyph.

```
function g:name()
```

Return the name of a glyph. The name is given by the sixth argument of the glyph's constructor, and is prefixed by all of its parent names.

```
function g:snap(x1, y1, w1, h1)
```

Returns a screen shot of the rectangle of (x1,y1,w1,h1), where the rectangle is in the local coordinates of the glyph.

```
function g:to_global(x1, y1)
```

Returns the global coordinates of (x1, y1), (x1,y1) should be in local coordinates of the glyph.

```
function g:to_local(x1,y1)
```

Returns the local coordinates of (x1,y1), where (x1,y1) is in the global coordinates.

```
function g:invalidate(x1,y1,w1,h1)
```

Force the glyph to redraw by calling `on_draw` method of the glyph as well as its children.

x1,y1,w1,h1, specifies the invalidation rectangle

```
function g:glyph_at(x1, y1)
```

Returns the leaf glyph under the point of (x1,y1).

```
function g:gc()
```

Returns the GC object of the glyph.

```
function g:remove()
```

Remove a child glyph from its parent.

`function g:insert(c1)`

Insert c1 into g's children table.

`function g:hide()`

Hide the glyph, change visible to false.

`function g:show()`

Show the glyph, change visible to true.

`function g:move(x1, y1, w1, h1)`

Change glyph's left ,top, width and height.

`function g:set_show_extent(s)`

Since glyph does not provide a default `on_draw` function nor does it clip the graphical output, it can be difficult to ascertain the position of a glyph at design time. This function overlaps a transparent rectangle on the glyph to indicate the glyph on the screen. The size of the rectangle is the extent of the glyph.

`function scene(w, h, background,name)`

Scene is the root window in the GUI library.

w,h: specifies the size of the root scene

background: the background image

name: the name of the scene

```
function s:set_background(bg)
```

Change the background of the scene.

```
function s:flush_update()
```

Scene will accumulate all graphical output in its off screen buffer -- this function will flush the off screen buffer to the real screen.

```
function content_size(g)
```

Return a tuple of (x,y,w,h), which is the smallest rectangle that could contain all the children of the glyph.

2.8.9.4. Buttons

`checkbox`, `radio_button` and `push_button` are three similar components in Sparkpad. Although their appearance is different, each has similar behaviors. (Basically they are all buttons.)

In the Sparkpad environment, a button can be any glyph that has two states: UP or DOWN. The function `buttonize` could convert a glyph to a button if the glyph implements a method called `change_state`. Please see function `button` for a complete example of how to implement a button.

The key method is `change_state`, in which the glyph changes its appearance according to the current state that is returned by method `get_up`.

```
function button(...)
  ...
```

```
   function s:change_state()
      if s:get_up() then
         s:set_picture(imgup)
      else
         s:set_picture(imgdown)
      end
   end
   ...
 End
```

`buttonize` is not a constructor, it just hooks `on_pendown` and `on_penup` to deal with the mouse events, and encapsulates the internal state of a button.

```
function buttonize(p,toggle, group)
```

p: is the glyph that will be converted to a button; note it is not the parent of the button

toggle: determines if the button is a toggle button.

group: this argument could be nil or a function returned by function 'group'. Function group () returns a function that represents a group object.

The buttons that share one group object will be mutually exclusive on its DOWN state, which means that at any time only one button in the group could be in DOWN state, this is the Sparkpad way to implement `radio_button`.

```
function checkbox(p, group, margin, imgup, imgdown)
```

Create a checkbox without caption.

p:the parent of the checkbox

group: the resule of group()

margin: a table with field top,left,right and bottom

imgup: the image to represents UP state

imgdown: the image to represents DOWN state

```
function radio_button(parent, caption, caption_left, group, color)
```

Create a radio button.

parent: the parent of the radio_button

caption: the caption of the button

caption_left: a boolean value to indicate if the text should put at the left side of the radio icon.

group: the result of group()

color: the color of text, default is gui.WHITE

```
function push_button(p, caption, color, fnt, image, toggle, group,
alignment, margin, xoffset, yoffset, no_auto_pop)
```

Create a standard button.

p:the parent of the button

caption: the caption of the button

color: the color of the caption

fnt: the font of the caption. In the current implementation, Sparkpad only supports one font face, so the only option here is the same font in a different size

image: the image of the button, push_button use only one image to represents UP and

DOWN states, and it offsets the image to right-bottom direction if the button is in DOWN state.

toggle: indicates if the button is a toggle button

group: the result of group()

alignment: the alignment of the caption

margin: the margin around the button image

xoffset, yoffset: how far should the button be moved if it is in DOWN state

no_auto_pop: obsolete, just ignore

```
function push_button:get_caption()
function push_button:set_caption(c)
```

Get/set the caption of `push_button`.

```
function push_button:set_font(f)
```

Change the font of `push_button`.

```
function push_button:get_image()
function push_button:set_image(im)
```

Get/set the image of `push_button`.

2.8.9.5. Focus Control

Besides the group property of `radio_button`, focus control is another feature in the Sparkpad environment that needs mutually exclusive control among a group of components. Therefore we can abstract the common aspect of these two features and the result is:

```
local function _group(outfunc, infunc)
```

This function encapsulates a reference to a glyph which will be called as “position” in the following remark. If the position is occupied by a glyph, then we say the glyph is in the state, otherwise we say the glyph is out of state. As there is only one position in a group, the semantic of 'mutually exclusive' is maintained naturally by this structure.

`outfunc`: is the name of the method that will be called when a glyph becomes out of state.

`infunc`: is the name of the method that will be called when a glyph becomes in the state.

This function returns an anonymous function (gly,query). When called, if the second argument is nil or false, then the current 'in the state' glyph will be replaced by the first argument and will return the previous 'in the state' glyph. If the second argument is not nil or false, then the first argument is ignored, and it will return the current 'in the state' glyph.

```
function focus_group()
```

Create a group that maintains the focus of components.

```
function group()
```

Create a group that maintains the UP/DOWN states of buttons.

```
function editor(p, cp, length, mask, insert_mode, focus_gp)
```

Create a single line editor.

p:parent of editor

cp:the initial text in the editor

length: the max length of the editor

mask: a function with prototype of mask(editor, caret, text, new_input). This function is used to validate/translate the input text

insert_mode: indicates if the editor is in insert mode

focus_group: the result of focus_group()

```
function editor:on_input(good)
```

An event that is fired when a new input character is accepted.

```
function editor:set_mask(m)
```

Change the mask function.

```
function editor:insert(c)
```

Insert a character.

```
function editor:delete()
```

Delete the character after the caret.

```
function editor:backspace()
```

Delete the character before the caret.

```
function editor:set_caret(ct)
function editor:get_caret()
```

Get/set the caret.

```
function editor:get_max_length()
```

Retrieve the max length of the editor.

```
function editor:set_insert_mode(i)
function editor:get_insert_mode()
```

Get/set insert mode of the editor.

```
function editor:set_color(f)
```

Change the color of the text.

```
function editor:set_focus_group(f)
function editor:get_focus_group()
```

Get/set the focus group.

```
function editor:on_focus(has_focus)
function editor:focus_in()
function editor:focus_out()
```

These three functions are required by focus group control.

`function dialog(p, icon, captions, align, ok, cancel, x, y)`

This function creates a dialog box that blocks other components on the screen.

p: the parent of the dialog

icon: an optional icon that display on the dialog

captions: a multi-line text message

align: the alignment of the message

ok, cancel: the caption of ok and cancel button

x,y: the offset along x and y axis relative to the center of its parent, the default value is 0,0

`function dialog:modal(loop)`

This function begins a new message loop to iterate the messages in Microwindows' message queue. It doesn't return until one of the following conditions is met:

user clicks ok button: it returns the label of the ok button, which is the fifth argument of the constructor

user clicks cancel button: it returns the label of the cancel button, which is the sixth argument of the constructor

The result of function loop is not nil nor false, it returns the result of loop().

`function msgbox(p, msg, x, y)`

Displays a simple message box on the screen.

p: parent of the message box

msg: message of the message box

x,y: the offset along x and y axis relative to the center of its parent, the default value is 0,0

2.8.9.6. Image Module

This section lists out all functions defined in the image module.

```
image.image(width, height)
```

Returns the width * height image.

```
image.size(filename)
```

Return the size of the given image filename.

filename: the name of image file

Return value: a tuple of (width, height), this function doesn't load the image

```
image.load(filename, width, height)
```

Load image from file.

filename: the name of image file

width: optional, the max width of the result image, default is the width of the image

height: optional, the max height of the result image, default is the width of the image

Return value: the image that loaded from filename, which is scaled up/down to width * height.

`image.load_fit(filename, width, height)`

Load image from file.

filename: the name of image file

width: the max width of the result image

height: the max height of the result image

Return value: the image that loaded from filename, which is scaled up/down to fit into the size of width * height. This function preserves the ratio of width/height of the original image.

`image.snap(x,y,width,height)`

Snap a rectangular area of the screen.

x: optional, the left coordinate of the rectangle, default 0

y: optional, the top coordinate of the rectangle, default 0

width: optional, the width of the rectangle, default 800

height: optional, the height of the rectangle, default 600

`image.rgba(r,g,b,a)`

Return a pixel of color (r,g,b,a).

r:the value of red channel

g:the value of green channel

b:the value of blue channel

a:the value of alpha channel

2.8.9.7. Instance method of image object

image object is created by image.image, image.load, image.load_fit or image.snap. As the following functions are the instance method of an image object, please note the first 'image' is not a module name but an image object.

```
image:draw(x,y,x1,y1,width,height)
```

Draw a rectangle area of the image on the screen.

x: the destination left coordinate on the screen

y: the destination top coordinate on the screen

x1: the source left coordinate in the image

y1: the source top coordinate in the image

width: the width of the source rectangle

height: the height of the source rectangle

```
image:rotate(angle, x, y)
```

Rotate the image in counter-clockwise direction.

angle: degree of the rotation

x,y: the center of the rotation

return value: a new image, which is the result of the operation, the original image won't change

```
image:scale(width, height)
```

Scale the image to width * height.

width: desired width

height: desired height

Return value: a new image, which is the result of the operation, the original image won't change.

`image:width()`

Return the width of the image.

`image:height()`

Return the height of the image.

`image:fill(color, x, y, width, height, alpha)`

Draw a rectangle on the image.

color: optional, the color of the rectangle, default is black

x: optional, the left coordinate of the rectangle, default is 0

y: optional, the top coordinate of the rectangle, default is 0

width: optional, the width of the rectangle, default is the width of the image

height:optional, the height of the rectangle, default is the height of the image

alpha:optional, the value of the alpha channel, default is 255

`image:clone()`

Return a new image which is identical to the original one.

`image:save(filename)`

Save the image to file in JPEG format.

filename: the name of the file

`image:name()`

Return the name of the image. If the image is created by image.load/image.load_fit then the name is the file name of the image file. Otherwise the name is empty, and it could be altered by `image:set_name`.

`image:set_name(name)`

Change the name of image.

name:the new name

`image:blit(x,y,source, x1,y1,w1,h1)`

Copy a rectangular area of pixel from source to the image.

x,y: the left,top coordinate in the destination image

source: the source image

x1,y1: the left, top coordinate in the source image

w1,h1: the width and height of the source rectangle

`image:blend(x,y,source ,x1,y1,w1,h1,alpha)`

Perform alpha blend of a rectangular area of source with the image.

x,y: the left, top coordinate in the destination image

source: the source image

x1,y1: the left, top coordinate in the source image

w1, h1: the width and height of the source rectangle

alpha: the value of alpha channel

2.9. Notes to application developers

Before beginning application development on the Sparkpad platform, please note the following miscellaneous items:

The memory usage of a scene is about 800 x 600 x 3 bytes, which equals about 1.4MB for one screen buffer. There are three (3) buffers for the Telechip 8901 board and one (1) buffer for the ADI Blackfin board.

For the Blackfin board, not all USB drive brands are supported.

All LUA and shell scripts (*.sh) are in Linux format. Please do NOT edit and save those as text files in a Windows environment, as text files contain different EOF markings and such files will not be picked up by the system after being copied to the device.

3. Additional Hardware Information

3.1.1.1. Working temperature range:

	Celsius	Fahrenheit
Minimum Temperature	0	32
Maximum Temperature	40	104

3.1.1.2. USB Port

All USB ports support USB 2.0 (or higher) only. Do not connect “Powered USB” cable or devices.

3.1.1.3. FCC Rules, Part 15

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

-Reorient or relocate the receiving antenna.

-Increase the separation between the equipment and receiver.

-Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.

-Consult the dealer or an experienced radio / TV technician for help.

This device complies with Part 15 of the FCC rules.

Operation is subject to the following two conditions:
- This device may not cause harmful interference.
- This device must accept any interference received, including interference that may cause undesired operation.

Changes or modifications not expressly approved by the party responsible for compliance could void the user’s authority to operate the equipment.

Responsible Party:
Sparkpad LLC
46090 Lake Center Plaza #206
Sterling VA 20165
USA
888-907-7275
support@sparkpad.com