Corona™ SDK
Tools Guide
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Ansca Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Ansca's copyright notice.

The Ansca logo is a trademark of Ansca Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Ansca retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications using Ansca software. Every effort has been made to ensure that the information in this document is accurate. Ansca is not responsible for typographical errors.

Even though Ansca has reviewed this document, ANSCA MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL ANSCA BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Ansca dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.
# Contents

**Introduction**  
4  
  Limitations  
  See Also  
4  

**Overview**  
5  
  Opening a Sample Project  
5  
  Command-line  
5  
  Open Dialog  
6  

**Corona Simulator and Debugger**  
7  
  Corona Terminal: Everyday development  
7  
  Debugging  
8  
  Corona Simulator: Demonstrations  
8  

**Text Editors**  
9  

**Sample Debugging Session**  
10  
  Basics  
10  
  Sample Session  
10  

**Revision History**  
14
Introduction

Ansca Corona fundamentally change how engineers approach mobile software development.

Today, mobile development is a horror story. Even a team of seasoned professional engineers will end up wasting a lot of time, enduring a lot of pain/hassle, and spending a lot of money attempting to successfully navigate through the many mobile technology terrains.

Our technology dramatically reduces these barriers to entry. We facilitate rapid development of visually rich applications. With our technology, the creative and business potential of mobile will finally be unleashed!

We provide you with a consistent platform to build graphically-rich applications. As a developer, you will program in Lua, a simple and intuitive scripting language, and leverage Corona's innovative and robust API's.

This document will discuss how to use Corona's API's to maximize your productivity.

Limitations

Currently, the tools only work on Mac OS X 10.5.6 or later.

See Also

See Getting Started Guide for a quick overview on how to get started.

See Applications Programming Guide for more information on how to create an application using Corona
Overview

In this chapter, we’ll explain the contents of the SDK and then show how to use the simulator to launch a sample application.

The following files in the SDK are worth noting:

- **Documentation/** is a directory containing this guide and the Applications Programming Guide.
- **debugger** is the command line debugger.
- **Corona Simulator** launches your application for testing. It simulates the mobile device on your local computer.
- **Corona Terminal** is useful for everyday development. It launches the Corona Simulator and opens a Terminal window to display error messages and print/trace statements. This enables you to do rapid iterations on your code.
- **simulator** does the same thing as Corona Terminal, but is more convenient to use on the command line.
- **SampleCode/** is a directory containing various sample applications to help you get started.

We’ll assume that the SDK directory is `/Applications/Corona/`.

Let’s launch the **fishies** application in the simulator. The actual application code and assets (image files) are located at `/Applications/Corona/SampleCode/fishies/`. We launch it by invoking the simulator with the path to that directory.

Opening a Sample Project

In order to launch an application, the simulator needs to know the *folder* where the application resides.

You can launch the simulator in two ways: via the command-line or via an Open dialog.

Command-line

Let’s open the Fishies sample project from the command line

Open a Terminal window and `cd` into the SDK directory (`/Applications/Corona`). Then launch the simulator, passing the path of the Fishies sample project. Your command-line session should look something like:
A copyright notice similar to the above will appear in the Terminal and the simulator window at left should appear. Try clicking on the fish and see what happens. The source for this application is available in the file `SampleCode/fishies/main.lua`.

Open Dialog

Open the SDK folder and double click on **Corona Terminal**. You should see an Open dialog displayed (below). Navigate to the SampleCode folder and select the Fishies sample folder.

The **Hardware** menu lets you rotate the simulator orientation and suspend/resume the application.
Corona Simulator and Debugger

Any time you create an application, the file must be named `main.lua` in order to be recognized by the simulator.

There are several ways to launch the simulator, depending on your scenario:

- Double click **Corona Terminal**. This is useful for everyday development, since it will launch the Corona Simulator and automatically bring up a Terminal window where all error messages and print/trace results will be displayed. For convenience, an alias called `simulator` can be invoked from the command-line to do the same thing.

- Double click **debugger**. This will launch an interactive session in a Terminal window and launch the Corona Simulator.

- Double click **Corona Simulator**. You typically only use this when you are demonstrating your application, since error messages are hidden.

Corona Terminal: Everyday development

To load a sample application, double click on **Corona Terminal**. You will see errors or print statements in a Terminal window. In the example below, we have loaded the **Button** sample and then clicked on the image for Button 2.

The sample code prints the click location to the terminal:
Debugging

The debugger is a useful tool for finding and isolating problems with your code. As with most debuggers, it will allow you to step through events. Access the debugger by pointing your terminal to it, or drag it into the terminal. Typing “help” will give you a list of commands available in the debugger.

Corona Simulator: Demonstrations

Launching Corona Simulator the terminal to execute code. This is useful in situations where you don’t want error messages to display (e.g. you are demonstrating your application to a client).

To run Corona Simulator, simply double click on its icon and navigate to the project folder (the folder containing your project’s main.lua file).
Text Editors

On MacOS, there are a variety of text editors available for editing Lua code that have syntax coloring support:

- Eclipse via the [Lua Eclipse](http://lua-users.org/wiki/LuaEditorSupport) plugin
- Emacs via the major mode [lua-mode](http://lua-users.org/wiki/LuaEditorSupport)
- BBEdit is a popular text editor for most major languages
- [Smultron](http://lua-users.org/wiki/LuaEditorSupport)
- [SubEthaEdit](http://lua-users.org/wiki/LuaEditorSupport)
- [TextWrangler](http://lua-users.org/wiki/LuaEditorSupport)
- [Vim](http://lua-users.org/wiki/LuaEditorSupport)
- XCode via the [specification files](http://lua-users.org/wiki/LuaEditorSupport) for Lua

A comprehensive list is available at [http://lua-users.org/wiki/LuaEditorSupport](http://lua-users.org/wiki/LuaEditorSupport)
Sample Debugging Session

Debugging is an essential part of software development. ANSCA offers a command-line debugger called debugger that offers all the standard debugger features:

- breakpoints
- stepping through code line-by-line
- stepping into functions
- listing the local variables
- inspecting the values of local variables
- inspecting the internal contents of Lua tables
- obtaining function stack traces
- switching into different stack frames
- printing and evaluating Lua expressions
- etc.

Basics

A debugger runs your application in the simulator. It allows you to specify what events trigger the debugger to pause execution of the application.

A most common trigger is a breakpoint. This allows you to pause execution anytime a line of code is about to execute. This provides you with the opportunity to look at the state of the application at that particular point in time, including the values of variables or the function stack trace.

Sample Session

The best way to understand how to use a debugger is to walk through a sample session.

To start a debug session, open a Terminal session, go to the directory containing debugger, and invoke debugger at the prompt:

```
[~]% cd /Volumes/ANSCA
[/Volumes/ANSCA]% ./debugger
ANSCA Remote Debugger
Run the program you wish to debug
```
At this point, an “Open” dialog will appear just as you saw when launching the simulator.

Navigate to a directory containing the main.lua file. As with the simulator, all assets that the main.lua file references must be in the same directory as the main.lua file itself.

Once you click okay, your application will launch under the simulator. The simulator will pause execution waiting for further commands from the debugger. Back in the terminal, the debugger will tell you that your program is paused.

The debugger always pauses at the first line of code to give you the opportunity to set breakpoints in your sample application. You can obtain a list of commands available in the debugger:

```
> help
backtrace(bt)       -- show backtrace
frame(f) <num>      -- switch to frame <num>
locals(l)           -- show local variables
dump(d)             -- prints value of variable
setb(b) <file> <line> -- sets a breakpoint
delb <file> <line>  -- removes a breakpoint
delallb             -- removes all breakpoints
setw <exp>          -- adds a new watch expression
delw <index>        -- removes the watch expression at index
delallw             -- removes all watch expressions
run                  -- run until next breakpoint
continue(c)          -- same as 'run'
step(si)             -- run until next line, stepping into function calls
over(so)             -- run until next line, stepping over function calls
listb                -- lists breakpoints
listw                -- lists watch expressions
eval <exp>           -- evaluates expression on the current context and returns its value
print(p) <exp>       -- same as 'eval'
exec <stmt>          -- executes statement on the current context
basedir [<path>]     -- sets the base path of the remote application, or shows the current one
exit                  -- exits debugger
```

Note in the help that certain commands have abbreviations for convenience. These are listed in parentheses. When possible, the abbreviations match those in gdb, a popular open source debugger.

For example, backtraces (i.e. function stack traces) can be obtained using an alias `bt` instead of typing out `backtrace`:

```
> bt
[* 1] (nil) at /Volumes/rtt/apps/SampleCode/Fishies/main.lua:46
```
Here, we are at the outermost scope because we haven’t called any functions. Therefore, the stack trace is one function deep. Note that line 46 is the first line of actual code in main.lua so that’s where the debugger pauses when it launches your application.

Let’s set some breakpoints, step through some code, and display values of local variables. To see what’s happening, here’s a snippet of the main.lua code we’ll be stepping through:

```lua
local buttonListener = function( event )
    local group = event.target
    -- tap only triggers change from original to different color
    local topObject = group[1]
    ...
end
```

This code is the listener function that gets called when we tap on the fish. We’ll set a breakpoint at line 98 inside the listener (or callback) method that gets invoked when a tap event gets fired. Then tell the debugger to continue execution.

The listener where we set the breakpoint does not get called unless a tap event is fired. So once we click on a fish, the event fires and the debugger will pause execution at the breakpoint we set inside the listener. From there, we’ll be able to step over code one line at a time, and list all the local variables:

```lua
paused at file main.lua line 98
> over
paused at file main.lua line 100
> locals
group = table: 0x1cda88c0
event = table: 0x160f90
```

Both `group` and `event` are tables, so we can print the value of a specific property of the table or the entire contents of a table via `dump`:

```lua
> dump event.x
  event.x(number) = 193
> dump event.y
  event.y(number) = 52
> dump event
  event(table: 0x160f90) =
    { y = 52     
      x = 193
      name = "touch"
      phase = "ended"
      target = table: 0x1cda88c0
    }
```
We can also do a stack trace like we did before:

```lua
> backtrace
[ 2] (nil) at ?:0
[ 3] (nil) at ?:0
```

Function stack trace information is only available in your code. In this particular stack trace, no information is available for stack frames 2 and 3 because these are internal to ANSCA.

Finally, we let the application continue running. The debugger remains active until the simulator quits.

```lua
> continue
Program finished
```
Revision History

This table describes the changes to *Tools Guide*:

<table>
<thead>
<tr>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-23-01</td>
<td>Initial draft</td>
</tr>
<tr>
<td>2009-26-04</td>
<td>Update</td>
</tr>
<tr>
<td>2009-16-06</td>
<td>Update</td>
</tr>
</tbody>
</table>