How to create sample PNaCl application

Published 2014-10-27 | (Compatible with SDK 4.5,5.0,5.1 and 2013,2014 models)

> Tutorial on creating sample Portable Native Client applications, which can be run on Smart TV.

Contents

This tutorial provides information on how to create a simple application that uses a PNaCl module. The application can be tested with Samsung Smart TV Emulator or directly on Samsung Smart TV with Native Client. You can also test it in your Google Chrome browser.

Prerequisites

To create application using a PNaCl module, you'll need NaCl SDK and a text editor for creating HTML, JavaScript, CSS, C/C++ and configuration files. To test the application you will need Samsung Smart TV Emulator running in VirtualBox.

Getting the SDK Emulator

Download the latest version of SDK and Emulator from **http://developer.samsung.com/tv/develop/tools/tizen-studio**. Follow the installation instructions for NaCl SDK and set up the working environment accordingly.

Introduction to PNaCl

PNaCl (Portable Native Client) implemented on Samsung Smart TV enables running native code that is a part of a widget.

Native Client

Native Client is an open-source technology developed by Google. It was ported into Samsung Smart TV, Samsung Smart TV Emulator. This technology enables executing native code within a Smart TV widget. Currently supported languages are C

and C++.

Native Client module is embedded in the HTML page. The communication, between the JavaScript/HTML layer of the application and the NaCl module, is provided by the Pepper API, as shown in Figure 1 Basic concept of Native Client application.

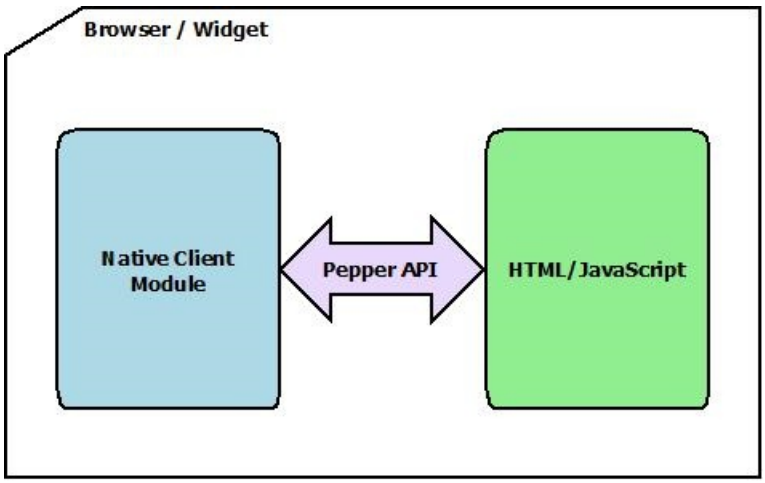

Figure 1 Basic concept of Native Client application.

Portability

The native code from the developed module is compiled independently for each target platform. The output of such compilation is a set of files with extension .nexe. When loading a module, through manifest file, the appropriate version is chosen, based on system architecture. This means that the module compiled into several files can be executed on each system specified in the manifest file without additional compilation.

In practice, this means that the same module can be tested on Google Chrome browser and on Samsung Smart TV Emulator, and executed on Samsung Smart TV.

More information

Learn more about Native Client for Samsung Smart TV here Getting started with NaCl.

For more information about the Native Client visit **https://developers.google.com/native-client/overview**

PNaCl Application

PNaCl application consists of three core elements:

The **widget / web page** components, such as JavaScript, CSS files, HTML pages.

PNaCl modules - files containing native (currently C or C++) code. Later, they are compiled into a set of files with extension .nexe. There is one .nexe file per every architecture type.

Manifest - This is a file containing information on where to find the PNaCl module file. This file has the extension .nmf.

Developing your first PNaCl application

In these few steps you will learn how to create your first PNaCl application. There are two possibilities: you can create source code for the native module in either C or C++ language. This tutorial presents simple examples of a native client module written in both languages.

All source code for the discussed application is available here:

stub files **d19_SampleAppStubs.zip**,

for a sample C implementation of a NaCl module **d19_SampleAppC.zip**,

for a sample C++ implementation of a NaCl module **d19_SampleAppCPP.zip**,

for a sample game of Tic-Tac-Toe **d19_TicTacToe.zip**.

Before running the applications, you have to build the applications first. When using Windows system just double-click on file make.bat provided with the application. For Linux machines execute make in the application directory. Find out more about building provided examples in the Compiling the Sample Applications section of this document.

Creating a PNaCl project

Every widget, that can be executed using Samsung Emulator, contains specific files:
the config.xml file,
the widget.info file,
JavaScript and CSS files,
index.html and other HTML files.

Creating the config.xml file

Create a new file and name it config.xml. This file will describe widget configuration. Here (Writing the config.xml file) you can find a tutorial on creating config.xml file.

Here is simple config.xml file that the sample application will require:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget>
    <ThumbIcon itemtype="string"></ThumbIcon>
    <BigThumbIcon itemtype="string"></BigThumbIcon>
    <ListIcon itemtype="string"></ListIcon>
    <BigListIcon itemtype="string"></BigListIcon>

    <category itemtype="string"></category>

    <type itemtype="string">user</type>
    <cpname itemtype="string"></cpname>

    <ver itemtype="string"></ver>
    <mgver itemtype="string"></mgver>

    <fullwidget itemtype="boolean">y</fullwidget>
    <srcctl itemtype="boolean">y</srcctl>
    <dcont itemtype="string">y</dcont>
    <mouse itemtype="boolean">y</mouse>

    <widgetname itemtype="string">Sample App</widgetname>
    <description itemtype="string"></description>

    <width itemtype="number">1920</width>
    <height itemtype="number">1080</height>

    <author itemtype="group">
        <name itemtype="string"></name>
        <email itemtype="string"></email>
        <link itemtype="string"></link>
        <organization itemtype="string"></organization>
    </author>
</widget>
```

To enable mouse or keyboard usage in widget, make sure that the config.xml file contains an adequate line:

```
<mouse itemtype="boolean">y</mouse>
<keyboard itemtype="boolean">y</keyboard>
```

Creating the widget.info file

Create a new file and name it widget.info. This file is used for the widget body opacity adjustment. Insert the following lines into this file:

Use Alpha Blending? = Yes
Screen Resolution = 1920x1280
More information about the widget.info can be found in Implementation details.

Creating JavaScript files

Embedding the NaCl module

To use Native Client technology in widgets you have to directly create an embed element for your PNaCl module inside the HTML page's body. For clarity, and learning purposes please also add listeners for the element, as shown in the section below.

```
1  <embed  name="nacl_module"
2        id="sample_app"
3        width=200 height=200
4        src="sample_app.nmf"
5        type="application/x-nacl" />
```

In the <embed> element:

name	the DOM name attribute for the Native Client module ("nacl_module" is often used as a convention)
id	specifies the DOM ID for the Native Client module
width, **height**	specify the size in pixels of the rectangle on the web page that is managed by the Native Client module (if the module does not have a visible area, these values can be 0)
src	refers to the Native Client manifest file that is used to determine which version of a module to load, based on the architecture of the user's computer
type	specifies the MIME type of the embedded content; for Native Client modules the type must be "application/x-nacl"

```
1  var listenerDiv = document.getElementById('listener');
2  listenerDiv.addEventListener('load', moduleDidLoad, true);
3  listenerDiv.addEventListener('message', handleMessage, true);
```

In line 2, the element listenerDiv subscribes for messages of type load sent from the NaCl module. This message is sent when the module is loaded properly. If event of type load is received, the function moduleDidLoad() is executed. In line 3, element listenerDiv subscribes for messages of type message sent from the NaCl module. This message is sent, when function PostMessage() is called in the NaCl module. If event of type message is called function handleMessage() is executed.

Creating the sample_app.js file

For the actual message handling we will create the sample_app.js file:

```
1  function handleMessage(message) {
2     console.log(message.data);
3     document.getElementById('outputString').innerHTML = message.data;
4  }
```

Line 2	message is logged into the JavaScript console
Line 3	message is written into HTML object named outputString

Creating index.html file

Create simple HTML file with PNaCl module embedded using a div marker.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Sample Application.</title>
5    <script type="text/javascript" src="sample_app.js"></script>
6    <script type='text/javascript' language='javascript' src='$MANAGER_WIDGET/Common/API/Widget.js'></script>
7    <script type="text/javascript">
8      function pageDidLoad() {
9        var widgetAPI = new Common.API.Widget();
10       widgetAPI.sendReadyEvent();
11       document.body.style.background = "#FFFFFF";
12     }
13   </script>
14 </head>
15 <body onload="pageDidLoad();">
16   <h1>Sample Application.</h1>
17   <h2>Status:<code id="statusField">NO-STATUS</code></h2>
18   <div id="listener">
19     <embed name="nacl_module"
20       id="sample_app"
21       width=200 height=200
22       src="sample_app.nmf"
23       type="application/x-nacl" />
24   </div>
25   <h1>OUTPUT</h1>
26   <pre>
27     <p><b id='outputString'></b></p>
28   </pre>
29 </body>
30 </html>
```

Line 5	Optional	Importing JavaScript file sample_app.js which contains functions handling messages sent from PNaCl module.
Lines 7 - 13	**Obligatory**	To run the Smart TV application. For details refer to Opening and Closing Applications
Line 18	**Obligatory**	In the following example, the div element subscribes for events sent from the NaCl module.
Line 19	**Obligatory**	The NaCl module is embedded in the HTML page.
Line 27	Optional	This element stores messages from PNaCl module, and presents them in the HTML view.

Creating a PNaCl Module

Once the set of stub files is created, we can start developing actual core of the application - the PNaCl module.

There are two possibilities, when it comes to creating the PNaCl module. For those who prefer the classic approach, we present chapter Creating the PNaCl Module using C language. For others, who prefer to use object oriented programming, we offer the section Creating PNaCl Module using C++ language.

Creating the PNaCl Module using C language

Create a new file in your project folder and name it sample_app.c.

Every PNaCl module has to implement three basic methods. These methods are PPP_InitalizeModule(), PPP_ShutdownModule() and PPP_GetInterface().

At the beginning, place the following code in your sample_app.c file.

```
#include <stddef.h>
#include "ppapi/c/pp_errors.h"
#include "ppapi/c/ppp.h"

PP_EXPORT int32_t PPP_InitializeModule(PP_Module module_id,
                                       PPB_GetInterface get_browser_interface)
{
  return PP_OK;
}

PP_EXPORT void PPP_ShutdownModule()
{
}

PP_EXPORT const void* PPP_GetInterface(const char* interface_name)
{
  return NULL;
}
```

The PPP_InitializeModule() is the application entry point, and is called when the module is loaded.

This is the application stub - right now, the bodies of these functions are empty.

The next step is to implement PPP_Instance interface struct, and later, pass it as a result when the PPP_GetInterface() is called with a suitable parameter.

Now, we need the PPP_Instance interface struct in sample_app.c. First, we need five functions with signatures corresponding to the functions from PPP_Instance interface struct. These are:

DidCreate(),
DidDestroy(),
DidChangeView(),
DidChangeFocus(),
HandleDocumentLoad().

In SampleApp module these functions will have names with the Instance_ prefix to stress that these methods belong to the PPP_Instance interface struct. Insert the following lines to the already existing sample_app.c right after include directives.

```
#include "ppapi/c/ppp_instance.h"
#include "ppapi/c/pp_bool.h"

static PP_Bool Instance_DidCreate(PP_Instance instance,
                        uint32_t argc,
                        const char* argn[],
                        const char* argv[])
{
   return PP_TRUE;
}

static void Instance_DidDestroy(PP_Instance instance)
{
}

static void Instance_DidChangeView(PP_Instance pp_instance,
                         PP_Resource view)
{
}

static void Instance_DidChangeFocus(PP_Instance pp_instance, PP_Bool has_focus)
{
}

static PP_Bool Instance_HandleDocumentLoad(PP_Instance pp_instance,
                                PP_Resource pp_url_loader)
{
   return PP_FALSE;
}
```

All the necessary functions have been created now. We need to create interface struct pointing to created functions. In order for these functions to be called, create this struct and pass it as a method result when PPP_GetInterface() is called with the interface name as a parameter:

Your PPP_GetInterface() should look like the following:

```
#include <string.h>

PP_EXPORT const void* PPP_GetInterface(const char* interface_name)
{
   if (strcmp(interface_name, PPP_INSTANCE_INTERFACE) == 0)
   {
      static PPP_Instance instance_interface = {
         &Instance_DidCreate,
         &Instance_DidDestroy,
         &Instance_DidChangeView,
         &Instance_DidChangeFocus,
         &Instance_HandleDocumentLoad
      };
      return &instance_interface;
   }
   return NULL;
}
```

> Important
>
> Every module has to implement PPP_Instance interface. Otherwise, it will not be loaded by

> the Native Client plug-in.
>
> Important
>
> Make sure all PPP_Instance interface functions are declared **before** creating the actual instance of PPP_Instance structure.

The module is now ready for compilation, but does not do anything yet. To be completely sure that the module is loaded and working properly, we will use the PPB_Messaging interface that enables sending events from the PNaCl module to the JavaScript layer of the application. The PPB_Messaging interface provides a function PostMessage(), which takes parameters of type PP_Instance and PP_Var. The PP_Var parameter is carrying the message body. To create objects of type PP_Var, remember to include files containing the PPB_Var interface.

> Note
>
> The interface names start with prefixes PPP_ and PPB_. The PPP_* interfaces contain functions that are implemented by the plug-in (“P”) that can be called by the browser. The PPB_* interfaces contain functions that are implemented by the browser (“B”) and can be called from within the Native Client module.
>
> Another group with the PP_ prefix contains constants and structs that are common for both groups mentioned before.

To obtain the pointer to the PPB_Messaging and PPB_Var implementation, use the following code and place it in your sample_app.c file.

Add all the necessary includes:

```
#include "ppapi/c/ppb_var.h"
#include "ppapi/c/ppb_messaging.h"
```

Declare the global variables to store the interface pointers:

```
const PPB_Messaging* g_varMessagingInterface;
const PPB_Var* g_varInterface;
```

Initialization of the created pointers will be performed in the PPP_InitializeModule() function body:

```
PP_EXPORT int32_t PPP_InitializeModule(PP_Module module_id,
                                       PPB_GetInterface get_browser_interface)
{
  /* Initializing global pointers */
  g_varMessagingInterface = (const PPB_Messaging*) get_browser_interface(PPB_MESSAGING_INTERFACE);
  g_varInterface = (const PPB_Var*) get_browser_interface(PPB_VAR_INTERFACE);

  return PP_OK;
}
```

Now all we want is to send a message saying “Hello world!” to JavaScript as soon as the module has been loaded. The function Instance_DidCreate() is called once, when the module is loaded. Update its body as follows:

```
#include "ppapi/c/pp_var.h"

static PP_Bool Instance_DidCreate(PP_Instance instance,
                        uint32_t argc,
                        const char* argn[],
                        const char* argv[])
{
   /* Create PP_Var containing the message body */
   struct PP_Var varString = g_varInterface->VarFromUtf8("Hello world!", strlen("Hello world!"));

   /* Post message to the JavaScript layer. */
   g_varMessagingInterface->PostMessage(instance, varString);
   return PP_TRUE;
}
```

Creating PNaCl Module using C++ language

Create a new file in your project folder and name it sample_app.cpp.

First, create the Module and Instance classes that are required for every Native Client module.

Creating the Instance: class:

```
#include "ppapi/cpp/instance.h"

class SampleAppInstance: public pp::Instance {
public:
   explicit SampleAppInstance(PP_Instance instance)
      : pp::Instance(instance) {
   }

   virtual ~SampleAppInstance() {}
};
```

Creating the Module class:

```
#include "ppapi/cpp/module.h"

class SampleAppModule: public pp::Module {
public:
   SampleAppModule(): pp::Module(){
   }
   virtual ~SampleAppModule(){}

   virtual pp::Instance* CreateInstance(PP_Instance instance) {
      return new SampleAppInstance(instance);
   }
};
```

Native Client modules do not contain the main() method. Instead, there is the CreateModule() function which triggers the creation of the module. The binding between the JavaScript part of the application and the Native Client module is done by the Init(), which is called when the module is embedded in a web page.

Don’t forget the CreateModule() function:

```
namespace pp
{
Module* CreateModule() {
   return new SampleAppModule();
}
}
```

Fill the Init() function in Instance class to send a message to the JavaScript layer of the application. The message will be sent when the page is loaded.

Include the header containing data type passed between module and the page:

```
#include "ppapi/cpp/var.h"
```

Inside the SampleAppInstance class add Init() function implementation:

```
bool Init(int argc, const char* argn[], const char* argv[]) {
  std::string msg = "Hello world!";
  PostMessage(pp::Var(msg));
  return true;
}
```

Creating a more complex application

In the attached examples, there is a more complex application, which involves graphics and event handling. This is a simple game of Tic Tac Toe, but it shows more possibilities in creating NaCl modules. Get the application's code here **d19_TicTacToe.zip**

The application's elements that deserve special recognition:

Widget:

The widget uses of a button, placed in the HTML page, to send a message to NaCl module informing that the user wants to start a new game.

Clicking the "NEW GAME" button calls a function that uses PostMessage() that enables sending messages to the NaCl module.

```
document.getElementById('nacl_module').postMessage('newGame');
```

The message is handled in HandleMessage() method, where depending on the message content different actions can be performed.

NaCl module:

Drawing elements using graphics 2D.

Graphics object is initialized when DidChangeView() method is called and the NaCl module's size has changed

```
graphics2DContext_ = pp::Graphics2D(this, view.GetRect().size(), isAlwaysOpaque);
BindGraphics(graphics2DContext_);
```

The graphics elements will be repainted when Flush() method is called, it obviously has to be called every time the game state has changed and some new objects (X or O) should appear on the screen.

```
graphics2DContext_.ReplaceContents(imageData);
graphics2DContext_.Flush(callbackFactory_.NewCallback(&SampleAppInstance::didFlush));
```

User interaction using mouse events.

The game is based on user mouse clicks. The user points and clicks on the board and basing on the mouse cursor position the right symbol (X or O) should appear.

The mouse input events can be received by the NaCl module after prior subscribing to the specific class of events, during the module initialization.

```
RequestInputEvents(PP_INPUTEVENT_CLASS_MOUSE);
```

The events are handled within HandleInputEvent() method. The application can perform specific action basing on type or other event's parameters.

Game logic.

The game logic is very simple, the playing board is only 3x3 and the user places O or X symbol on the board. The player wins when he manages to put three of the same symbols in a column, row or across the board. Despite the simplicity of this example, it shows that the module can perform some more complicated calculations, and check if the state of the game indicates whether the play has ended.

Compiling the Sample Applications

The NaCl module build procedure depends on GNU Make based build file. The sample applications are provided with makefiles that build executables for each of three achitecture types:

x86_64 - target environment is 64-bit architecture machine,
x86_32 - target environment is 32-bit architecture machine,
arm - target environment is arm based architecture machine.

Build procedure for Windows OS

1. Right-click on My Computer and choose Properties.
2. Chose Advanced system settings for Windows 7 and Advanced tab for Windows XP.
3. Click Environment Variables.
4. Add New... variable named NACL_SDK_ROOT with value pointing to the path with pepper bundle: <your_nacl_sdk_path>\nacl_sdk\pepper_<version>. The pepper version has to be supported by Samsung Smart TV and Emulator. For detailed information about the pepper version check: **http://developer.samsung.com/tv/develop/tools/tizen-studio**
5. From command line navigate to the directory of the sample application to compile, for example:
   cd <path_to_your_sample_app>/SampleAppC
6. From command line execute **make** command:
   make

Build procedure for UNIX-based OS

1. Export NACL_SDK_ROOT variable with the path to NaCl SDK's pepper API. The pepper version has to be supported by Samsung Smart TV and Emulator. For detailed information about the pepper version check: **http://developer.samsung.com/tv/develop/tools/tizen-studio/a>**
   export NACL_SDK_ROOT=<your_nacl_sdk_path>/nacl_sdk/pepper_<version>
2. Navigate to directory with your project:
   cd <path_to_your_sample_app>/SampleAppC
3. Compile your project using **make** command:
   make

Compilation result

If no compilation errors occurred, the following files should appear in project folder:

sample_app.nmf,
sample_app_arm.d,
sample_app_arm.nexe,
sample_app_arm.o,
sample_app_x86_32.d,
sample_app_x86_32.nexe,
sample_app_x86_32.o,
sample_app_x86_64.d,
sample_app_x86_64.nexe,
sample_app_x86_64.o.

Having manifest file(sample_app.nmf) and .nexe files for different architectures, you can run your application.

Running PNaCl Application

Test your application with Google Chrome

In order to test your application with Google Chrome browser, you need to:

1. Build the PNaCl module

   For detailed information please refer to Compiling the Sample Applications
2. Move files to examples folder

Copy following files into the examples folder in your pepper bundle:

index.html (and other HTML files, if there are any),
JavaScript files - common.js, sample_app.js (and other, if there are any),
sample_app.nmf,
.nexe files.

If any of those files is in a subfolder of a main project folder, it should get into the folder with the same name as original one.

3. Run server

Run server for NaCl applications by running httpd file following instructions from NaCl Getting Started guide: **https://developers.google.com/native-client/devguide/tutorial#server**.

4. Run Google Chrome

Run Google Chrome with –no-sandbox option and configure it for NaCl plug-in according to NaCl Getting Started guide: **https://developers.google.com/native-client/devguide/tutorial#verify**.

5. Run your application

Assuming you are using the local server and the project directory is named SampleApp and it is placed in examples folder in appropriate pepper bundle, you can load the application web page into Chrome by visiting the following URL: http://localhost:5103/SampleApp/.

> Important
>
> Make sure you placed your application in the same examples folder, from which you run httpd server.

Test your application with Samsung Emulator

In order to test your application with Emulator provided with Samsung Smart TV SDK, you need to:

1. Build the PNaCl module.

For detailed information please refer to Compiling the Sample Applications

2. Move files to Apps folder.

Copy following files into the Apps folder:

widget.info,
config.xml,
index.html (and other HTML files, if there are any),
JavaScript files - common.js, sample_app.js (and other, if there are any),
sample_app.nmf,
.nexe files.

If any of those files is in a subfolder of a main project folder, it should get into the folder with the same name as original one.

3. Run Emulator.

Launch Samsung Emulator.

4. Run SampleApp.

Select Menu Open App -> SampleApp -> OK.

The Emulator should present result similar to the Figure 2 Simple PNaCl widget.

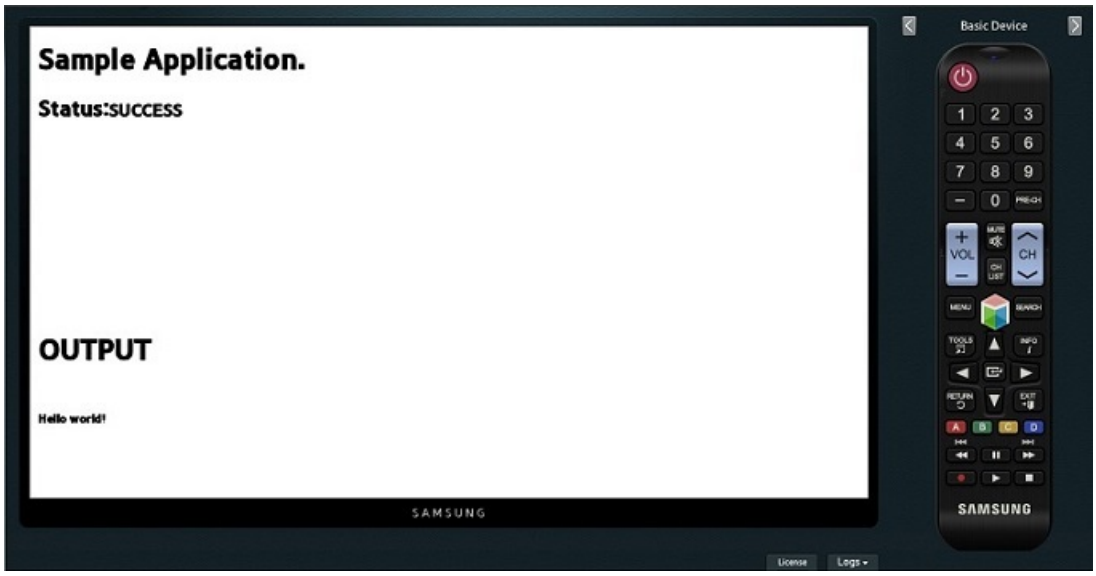

Figure 2 Simple PNaCl widget.

Problems that may occur

1. Widget is loading, but then “Missing Plug-in” appears where the PNaCl module should be.

   The example of “Missing Plug-in” error is shown in Figure 3 “Missing Plug-in” error.

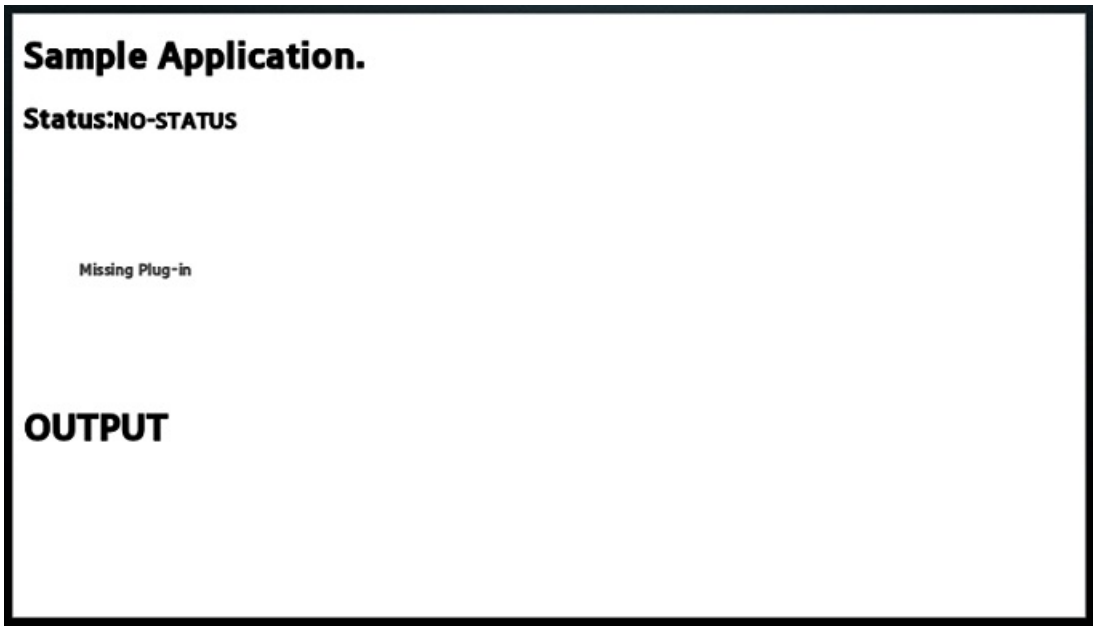

   Figure 3 “Missing Plug-in” error.

   To avoid “Missing Plug-in” error, please make sure that you have the latest Emulator version that supports Native Client applications.

2. An error appears: Error Detail: NaCl module load failed: could not load nexe url.

   To avoid this problem, make sure the following conditions are satisfied:

   The compilation of the PNaCl module was successful, ended without errors;

   The path specified in the manifest file is valid and points to the file generated as the output of PNaCl module compilation.

3. An error appears: Error Detail: NaCl module load failed: could not load manifest url.

   To avoid this problem, make sure the object embedding the PNaCl module in the widget’s HTML page has the attribute src set to the value matching the manifest filename (set to relative path, if the file wasn’t placed directly under the project’s root).