SB1
PROGRAMMER'S GUIDE

SB1 PROGRAMMERS GUIDE

72E-170991-02

Rev. A

August 2014

No part of this publication may be reproduced or used in any form, or by any electrical or mechanical means, without permission in writing from Motorola. This includes electronic or mechanical means, such as photocopying, recording, or information storage and retrieval systems. The material in this manual is subject to change without notice.

The software is provided strictly on an “as is” basis. All software, including firmware, furnished to the user is on a licensed basis. Motorola grants to the user a non-transferable and non-exclusive license to use each software and firmware program delivered hereunder (licensed program). Except as noted below, such license may not be assigned, sublicensed, or otherwise transferred by the user without prior written consent of Motorola. No right to copy a licensed program in whole or in part is granted, except as permitted under copyright law. The user shall not modify, merge, or incorporate any form or portion of a licensed program with other program material, create a derivative work from a licensed program, or use a licensed program in a network without written permission from Motorola. The user agrees to maintain Motorolas copyright notice on the licensed programs delivered hereunder, and to include the same on any authorized copies it makes, in whole or in part. The user agrees not to decompile, disassemble, decode, or reverse engineer any licensed program delivered to the user or any portion thereof.

Motorola reserves the right to make changes to any software or product to improve reliability, function, or design.

Motorola does not assume any product liability arising out of, or in connection with, the application or use of any product, circuit, or application described herein.

No license is granted, either expressly or by implication, estoppel, or otherwise under any Motorola, Inc., intellectual property rights. An implied license only exists for equipment, circuits, and subsystems contained in Motorola products.

Revision History

Changes to the original guide are listed below:

Change	Date	Description
-01 Rev. A	1/10/2013	Initial release.
-02 Rev A	8/15/2014	Add description for Wait/Quit behavior. Add description for hour glass. Add details for additional configurations introduced in config.js. Add keyboard APIs and samples. Rev B.5 changes.

TABLE OF CONTENTS

Revision History iii
Introduction v
  Documentation Set v
Chapter Descriptions v
Notational Conventions vi
Related Documents and Software vi

Chapter 1: Introduction
SB1 System Applications 1-1
Namespaces 1-2
  SB1 Client Applications 1-2
    Backend 1-3
    Frontend 1-3

Chapter 2: Application Integration
Integration 2-1
  System Integration 2-1
  Visual Integration 2-2
Installation 2-2
  config.js File 2-3
  apps.json File 2-3
    Developer Content 2-4
  Application Timeout 2-5

Chapter 3: Application Shared Library
Application Services 3-1
  Running an Application 3-2
    Anonymous: asl.run(appUrl, data, callback); 3-2
    Installed: asl.run(appName, data, callback); 3-3
    asl.exit(data); 3-3
    asl.minimize(); 3-4

Events ... 3-4
asl.events.fire(eventName, data) ... 3-4
asl.events.subscribe(eventName, callback) ... 3-4
Native Shell Events ... 3-4
Notification Services ... 3-6
Parameters ... 3-6
Keyboard Services ... 3-9
Explicit Keyboard Execution ... 3-9
Parameters ... 3-10
Keyboard Events ... 3-11
Shared UI Services ... 3-13
Back Button ... 3-13
Parameters ... 3-13
Title Label ... 3-14
Options Menu Button ... 3-14
Parameters ... 3-14
Full Screen Mode ... 3-15
Resource Services ... 3-16
NPAPI ... 3-16
Messaging Services ... 3-18
Authentication Services ... 3-19
User Profiles and Authentication ... 3-19
Shell Authentication ... 3-19
Parameters ... 3-20
Shell Authentication Properties ... 3-20
Application Authentication ... 3-21
Badge Mode ... 3-21
Setting User Profile from an Application ... 3-21
Window Services ... 3-22
Alert Window ... 3-22
Confirm Window ... 3-22
Hourglass ... 3-22
Wait/Quit Feature ... 3-22
Startup Apps ... 3-23
Ghosting ... 3-23
QWERTY Keyboard ... 3-23
Process Application ... 3-24

Chapter 4: Configuration
ScanTo Application ... 4-1
Maximum Number of Running Applications ... 4-2
Application Source File Location ... 4-2
Push Notifications Port Number ... 4-3
Push Notifications Pass Key ... 4-3
Home Screen Shortcut Buttons ... 4-3
Touchzone Disable Timout ... 4-3
PIN Lock Require ... 4-3
Default Lock Screen Page ... 4-4
Lock Screen Timeout ... 4-4
Rotate Badge Screen Timeout ... 4-4

Disable Profile Button 4-4
Default User Information 4-5
Login Screen Require 4-5
Login Screen URL 4-5
Logoff URL 4-5
Login Timeout 4-6
Admin Settings PIN Require 4-6
Admin PIN value 4-6
Number of Levels for the Beeper Settings Screen 4-6
Beeper Duration 4-6
Beeper Frequency 4-7
Date Format 4-7
Time Format 4-7
Speaker Volume 4-7
Default Volume Level of the Beeper 4-7
Default Volume Level of the Speaker 4-7
Memory Threshold 4-8
Cradle Insert URL 4-8
Low Battery URL 4-8
Beeper for System Notifications 4-8
Beeper for Server Notifications 4-8
Beeper for Application Notifications 4-8
Number of Waits 4-9
Wait for Response Timeout 4-9
Retry URL 4-9
Reset History URL 4-9
Clear Local Storage on Reboot 4-9
Scanner Configuration for the scanTo Application 4-10
Push Notifications Number of Parameters 4-10
Startup Application 4-10
Cradle Insert Activities 4-11
Keyboard Configurations. 4-12

Chapter 5: Localization
String Resource Files 5-2
Keyboard Resource Files 5-2
Client Application Localization 5-2

Chapter 6: Additional Notes

Chapter 7: App Development and Deployment Guidelines
Application Naming 7-1
Content File Naming 7-2
Content File Location in SB1 7-2
Content File Location in Development Workstation 7-3
Applications Versions 7-4
SB1 Baseline Files and Baseline State 7-7
Creating an SB1 Baseline 7-8

Building an SB1 Baseline Package 7-8
Deploying an SB1 Baseline Package 7-9
Removing an SB1 Baseline Package 7-10
Developing and Testing an SB1 Application 7-10
Building an SB1 Application Package 7-11
Deploying an SB1 Application Package 7-12
Removing an SB1 Application Package 7-13

Chapter 8: Using Fonts on the SB1

Introduction 8-1
RhoElements Font System 8-1
Utilizing Current Fonts 8-1
Utilizing New Fonts 8-2
Installing A New Font To a Local File System 8-2
Font-Face/Webfonts 8-2
Transferring Fonts 8-3
Font Licenses 8-3

Chapter 9: Running off-line Web Applications

Introduction 9-1
How to Deploy a Web Application 9-1
Enabling the Cache Manifest in HTML 9-1
Cache Manifest 9-2
Structure 9-2
Header 9-2
Cache 9-2
Network 9-2
Fallback 9-2
Server 9-3
Tips 9-3
Revising the Cache Manifest 9-3
Off-line Mode 9-4
Discerning Connection Status 9-4
Network Module 9-4
WEBSQL 9-5
Example of Using WEBSQL in RhoElements 9-5
More Information 9-5
SB1 smart badge - Best Practices & User Training 9-6
Scanning Recommendations 9-6
Overall Usability 9-6
SB1 Accessories 9-6
Network Connectivity and Coverage 9-7
Configuration and Application Development 9-7

ABOUT THIS GUIDE

Introduction

This guide provides information developing applications for use on the SB1.

✓ ***NOTE*** Screens and windows pictured in this guide are samples and may differ from actual screens.

Documentation Set

The documentation set for the SB1 is divided into guides that provide information for specific user needs.

- *SB1 Regulatory Guide* - provides all regulatory and safety information.
- *SB1 User Guide* - describes how to use the SB1.
- *SB1 Integrator Guide* - describes how to set up the SB1 and accessories.
- *SB1 Programmers Guide* - describes how to develop applications for the SB1.

Chapter Descriptions

Topics covered in this guide are as follows:

- *Chapter 1, Introduction* - provides an introduction to the Programmers Guide.
- *Chapter 2, Application Integration* - Lists the steps required for integrating and installing applications on the SB1.
- *Chapter 3, Application Shared Library* - describes the modules of the Application Shared Library.
- *Chapter 4, Configuration* - Lists the options available in the SB1 configuration file.
- *Chapter 5, Localization* - describes localization support for the SB1.
- *Chapter 6, Additional Notes* - provides additional recommendations and best practices for using the SB1.

- *Chapter 7, App Development and Deployment Guidelines* - provides guidelines for application development and deployment.
- *Chapter 8, Using Fonts on the SB1* - provides information for using fonts.
- *Chapter 9, Running off-line Web Applications* - describes how to run application off-line.

Notational Conventions

The following conventions are used in this document:

- The term "SB1" refers to the Motorola SB1 smart badge.
- *Italics* are used to highlight the following:
  - Chapters and sections in this and related documents
  - Folder names
- **Bold** text is used to highlight the following:
  - Key names on a keypad
  - Button names on a screen.
  - Dialog box, window and screen names
  - Drop-down list and list box names
  - Check box and radio button names
  - Icons on a screen.
- **`Courier Bold`** text is used to highlight filenames.
- ***Bold Italic*** text is used to highlight MSP package names.
- Bullets (•) indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential.
- Sequential lists (e.g., those that describe step-by-step procedures) appear as numbered lists.

Related Documents and Software

The following items provide more information about the SB1.

- *SB1 Regulatory Guide*, p/n 72-162415-xx
- *SB1 User Guide*, p/n 72E-164711-xx
- *SB1 Integrator Guide*, p/n 72E-164712-xx
- *Administrating MSP 4.0*, p/n 72E-128775-04
- *Understanding Mobility Services Platform 4.0*, p/n 72E-128712-05
- Mobility Services Platform 4.0 Software Installation Guide, p/n 72E-100159-13
- Mobility Services Release Notes, p/n 72E-100160-17
- Using Mobility Services Platform 4.0, p/n 72E-128802-05

For the latest version of this guide and all guides, go to: http://www.motorolasolutions.com/support

CHAPTER 1 INTRODUCTION

This guide provides the starting point for developing applications for the SB1 Shell environment. It provides basic information about the system environment, what types of applications can run in this environment, system libraries and how to access device capabilities.

The SB1 Shell is a multi-instance, multi-tasking application which enables RhoElements “hybrid” (HTML/JavaScript) applications to run concurrently, interact with each other and share device and system resources. The Shell runs within RhoElements and uses the Webkit engine. This means that most rules and specific attributes that can be found in Webkit are available in the SB1 Shell.

SB1 System Applications

The SB1 Shell includes several system applications:

- **Home** - the starting point of the SB1 Shell. This application provides access to all the other applications and functions.
- **Applications** - lists and manages client applications.
- **Settings** - manages SB1 capabilities and Shell settings.
- **Lock** - security and badge functionality.
- **Notifications** - an inbox for viewing notifications sent by applications and the system.
- **Profile** - provides access to user settings for the SB1.

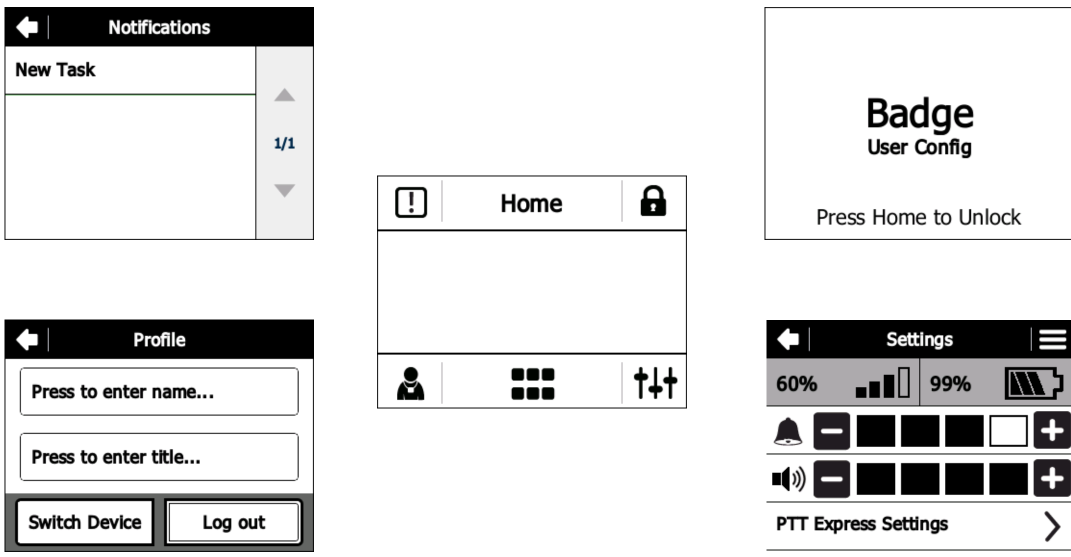


Figure 1-1 *System Applications*

Namespaces

This section provides information on the JavaScript objects namespaces, e.g. each layer in the Structure section is encapsulated by a JavaScript object (namespace):

- Application Shared Library - asl
- Shell UI Services - sui
- Shell System Library - sys

All of the modules described are provided with their namespaces.

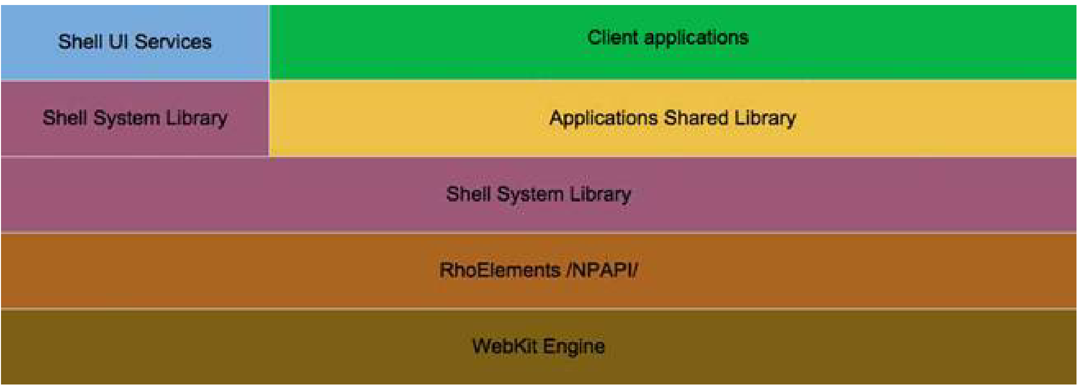


Figure 1-2 *Structure Diagram*

SB1 Client Applications

The SB1 Shell is designed to run custom built web based client applications. They are required to meet the following technical specifications:

Backend

In general, web applications can implement some specific logic based on the workflow required. The implementation can be made in various technologies and programming languages (C#.NET, RoR, PHP, etc.). The SB1 Shell does not interact with the application backend. Applications written in any client-server environment has to be hosted on a server. The SB1 Shell should be configured with the URL address of the application. With this information, the new application will be added to the SB1 Shell, enabling it to be run.

Frontend

- HTML - The SB1 Shell runs in a web browser that has support for HTML 5. SB1 applications can be written using the enhancements provided by the latest official specification of W3C. This allows application designers to take advantage of the latest improvements as markup elements, web databases, browser session storage and browser cache.
- CSS - The SB1 Shell supports the CSS 3 official specification with webkit extensions.
- JavaScript - The SB1 Shell supports the JavaScript ECMAScript 3 with some extensions from ECMAScript 5.

CHAPTER 2 APPLICATION INTEGRATION

Each client application that meets the technical specifications in *Chapter 1, Introduction* can be integrated and installed on the SB1. Client application developers should perform two steps:

- Integration
- Installation.

Integration

System Integration

NOTE If the client application page is to be served from a remote location, it must refer to a copy of the **`asl.js`** library from the same location. This is due to Web Security Standards which will not allow a remote page to load JavaScript from a local address.

Every HTML page within an SB1 application must include a special library called **asl.js**. This library provides the following services in the SB1 multi-instance environment:

- application services
- notification services
- keyboard services
- shared UI services
- resource services
- messaging services
- authentication services
- window services.

The following code shows an example of how to include asl.js within an HTML page:

```
<head>
   ...
   <script type="text/javascript"
   src="http://127.0.0.1:83/Application/www/sapp/src/asl.js"></script>
   ...
</head>
<body>
   ...
```

NOTE The **`asl.js`** file must be the first JavaScript file included in the client application document. **asl** is always the first library that access the document model and document events. This is required because the application system library needs to setup a connection with the SB1 Shell to initialize the device profile.

Client application logic implementation should never override the **asl** namespace. Application developers should never use statements in JavaScript that override the application shared library object - **asl**.

Visual Integration

The SB1 Shell includes a style sheet file that overrides the default styling of the web document. It is recommended that client applications running on the SB1 use these styles in order to achieve consistent application look and user experience. The style sheet can be included in HTML documents by adding the link tag that is shown in the following example:

```
<head>
   …
   <script type="text/javascript"
   src="http://127.0.0.1:83/Application/www/sapp/src/asl.js"></script>
   <link rel="stylesheet" type="text/css" href="path/asl.css"></link>
   …
</head>
<body>
   ...
```

Installation

Once the application is prepared to communicate with the SB1, it has to be installed. Depending on the configuration scheme, this process can be different.

In general, the SB1 has two primary configuration files:

- **`config.js`**
- **`apps.json`**

config.js File

The **config.js** file, in the *config* folder, is the general configuration file for SB1. One of the values it contains is called config.apps.src which holds the address to the application list that will be installed. By default, the value is:

```
config.apps.src = ''/UserDrive/config/apps.json;
```

NOTE The **apps.jason** file can be located at a remote location. For example:

config.apps.src = 'http://www.example.com/config/apps.json;

Once a remote location is set for this file, the system downloads it to a device location, configured in *config.apps* and then fetches the listed applications from this file.

See *Chapter 4, Configuration* for more information.

NOTE The **apps.json** file can be made variable based on the logged-in user. If Single Sign-On authentication is configured, the authentication server can send a different **apps.json** file depending on the user. The system loads the user applications once the user logs in successfully. See *Authentication Services on page 3-22* for more information.

apps.json File

The **apps.json** file, in the *config* folder*,* contains the list of the applications that are installed once the SB1 Shell is configured. It is formatted in JSON, a JavaScript native format. To add an application to this list, create a list element and insert it before or after any other list element in the **apps.json** file. Editing of the **apps.json** file is manual. Take caution in editing. Be sure to include the proper syntax (i.e., quotes, commas and slashes). Modifications to the **apps.json** file will not take effect unless the SB1 is rebooted.

Example

```
},
{
   "name": "App X",
   "url": "http://www.example.com/index.html",
   "icon": "http://www.example.com/assets/icon.png"
},
{
```

where:

- name = name of the application. This parameter is mandatory and must be unique for an **apps.json** file, otherwise the application will not be recognized by the system.
- url = address where the application can be found. This parameter is mandatory.
- icon = a path to the icon to be shown in the **Applications** application on the SB1. This parameter should be an absolute path. For example: http://example.com/images/icon.png. Relative paths will not work: /images/icon.png.

- Applications need a permission to overwrite custom badge. canOverwriteBadge = true can be configured to overwrite badge.

Once configured, the application is accessible from the SB1 **Application** screen.

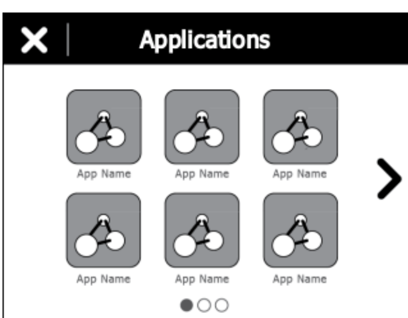


Figure 2-1 *Applications Screen*

NOTE The position of the application icon in the **Applications** screen depends on the position of the configuration element in the **`apps.json`** file.

Developer Content

All developer content can only be placed in the *\UserDrive* folder of the SB1. Basic configuration of an SB1 should have a folder structure similar to:

```
\UserDrive\config\
   apps.json
   config.js
   resources-en-US.js
   kbd-en-US.js
```

See the SB1 toolbox for sample(s) of a configured */UserDrive* folder.

- apps.json - list of applications can be configured.
- config.js - Necessary configuration values can be changed. And also apps.json path to be changed.
- resources-en-US - Users can copy required language resources to the *\UserDrive\config* folder
- kbd-en-US - Users can copy required language keyboard to the *\UserDrive\config* folder.

By default out of box device configured with sample demo applications in apps.json. Customers can configure applications in apps.json and make necessary configuration changes required for their business in config.js. Both apps.json and config.js can be copied to *\UserDrive\config* folder.

NOTE When connecting the SB1 to a host computer, the root of the *UserDrive* folder is shown. It may not be obvious of your location in the file system. So when placing content on the SB1 when connect to a host computer, developers should create a folder *\config* and place the appropriate configuration content in the folder.

```
\config (correct)
\UserDrive\config (incorrect)
\apps
   app1\app1.html
   app2\app2.html
   \src
      asl.js
```

After configuring the *UserDrive* content, the developer should:

1. Remove the USB cable.
2. Reboot to see the changes take effect.

Application Timeout

When the SB1 Shell opens an application, it creates a timeout function that waits for the application to respond. If that timeout function does not execute, the SB1 Shell considers the application unavailable and prompts the user with a notification if they want to quit the application or to wait until it loads.

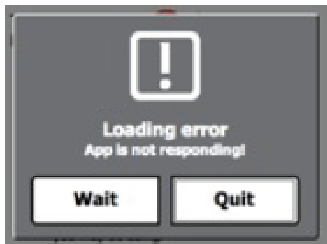


Figure 2-2 *Application Timeout Notification*

NOTE The only way for an application to respond to the SB1 Shell is through the **asl** library. The **asl** library should be included in the application code as it enables the application and stops the timeout.

CHAPTER 3 APPLICATION SHARED LIBRARY

The Application Shared Library contains eight modules for integrating client applications with the SB1 Shell:

- Application services
- Notification services
- Keyboard services
- Shared UI services
- Resource services
- Messaging services
- Authentication services
- Window services.

NOTE If the client application page is to be served from a remote location, it must refer to a copy of the **`asl.js`** library from the device. (http://127.0.0.1:83//Application/www/sapp/src/asl.js).

Application Services

The **asl** module provides several ways to manage and control applications from a client application (starting, exiting and switching applications). The library provides an event management mechanism with different options for application developers to communicate with other client or system applications. These are implemented through the SB1 Shell. Once the client application tries to use the application services, the **asl** module communicates with the SB1 Shell and the Shell executes the requested command.

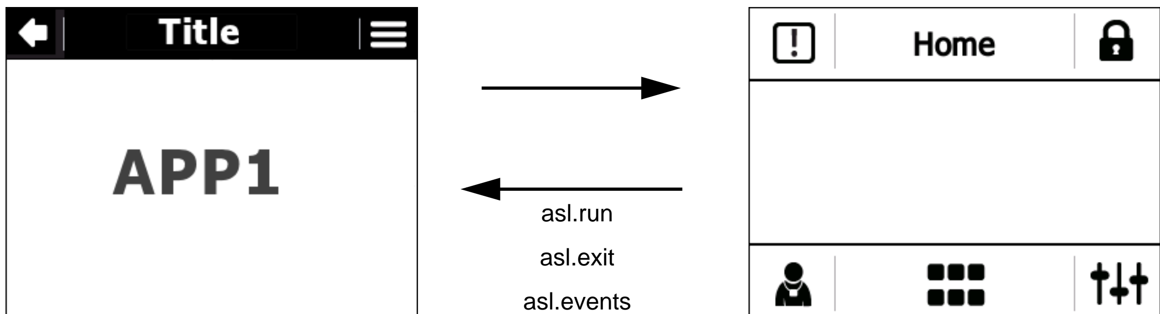


Figure 3-1 *asl Module Communication*

Running an Application

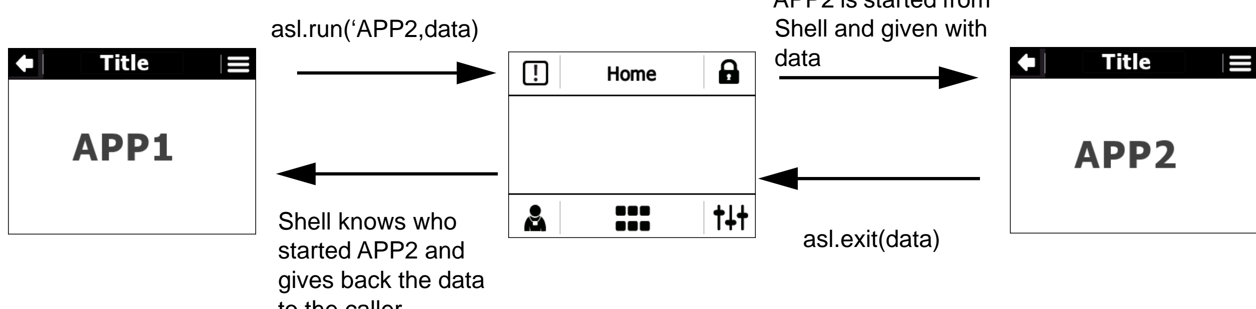


Figure 3-2 *Running an Application Flow*

Starting or switching to an application is done through the `asl.run` function call. Client applications do not need to know if the application that they call has been started or is running. Client applications can run two types of applications; anonymous and installed.

Anonymous: asl.run(appUrl, data, callback);

Anonymous applications are applications that are not defined in the **`apps.json`** file. They are always started in a new application instance. They are run by providing the URL of the app to the appUrl parameter.

Parameters

- appUrl (string)

  URL or application name. If the appUrl parameter is a URL, the SB1 Shell starts a new application instance with the given URL. It appears at the end of the Application list. If the parameter is an application name, the SB1 Shell looks for the application in the **`apps.json`** file. If it does not find the application, the SB1 Shell reports an error. If the SB1 Shell finds the application in the **`apps.json`** file, it starts the application and indicates that the requested application is opened in the **Application** screen.

- data (object)

  JSON formatted object of data that will be sent to the application defined in appUrl.

Example

```
asl.run('http://127.0.0.1:83/UserDrive/apps/product/index.html',{ProductId :
"101"}, function() {console.log('callback')});
```

The above data received in a app in loaded function.

Example

```
asl.events.subscribe(asl.events.types.loaded, function(data)
{console.log(data);
});
```

- callback (function)

  Function that is executed once the requested application exits. This callback is executed in the context of the invoking application.

Example

```
asl.run('http://www.motorola-solutions.com/citf/app1.html',{'taskId': '1024'});
```

Installed: asl.run(appName, data, callback);

Installed applications have their unique identifier in the SB1 system which can be used for either starting the application or switching to the already running application. For that purpose the caller has to know the `appName` (the unique identifier of the application).

Parameters

- appName (string)

  If the parameter is an application name, the SB1 Shell looks for the application in the **`apps.json`** file. If it does not find the application, the SB1 Shell reports an error. If the SB1 Shell finds the application in the **`apps.json`** file, it starts the application and indicates that the requested application is opened in the **Application** screen.

- data (object)

  JSON formatted object of data that will be sent to the application defined in appUrl.

Example

```
asl.run('http://127.0.0.1:83/UserDrive/apps/product/index.html',{ProductId :
"101"}, function() {console.log('callback')});
```

- callback (function)

  Function that is executed once the requested application exits. This callback is executed in the context of the invoking application.

Example

```
asl.run('TeamMate');
```

The data parameter can be used if the callee wants to provide specific data to the client application. The data parameter has to be json formatted object (or null).

Example

```
asl.run('myApplication', {'size': '1024'});
```

NOTE Installed applications run only one instance. Once the run method of an installed application is called a second time and the application is already running, its instance loads the default application url. The anonymous application can be run in multiply instances. Every time someone calls `asl.run` of a url it creates a new instance and loads the application from that url in this instance.

asl.exit(data);

Exiting an application can be called through `asl.exit` function. If the client application needs to provide a status code or some data to its callee this can be done by the data parameter of the exit function. The data parameter has to be json formatted object (or null).

Parameters

- data (object)

  json object that is passed to the invoking application once the current application exits.

Example

```
asl.exit({'message': 'Data was successfully saved.', 'status':
'1'});
```

If an application calls `asl.exit` and was started by another application by calling `asl.run`, the control of the screen is returned to this caller through the callback parameter of the `asl.run` function.This process is controlled by the Shell System Library. Callback function from the parent **`asl.run`** is executed with the parameters returned from **`asl.exit`** (data).

Example

```
asl.run('http://www.motorola-solutions.com/citf/app1.html',
{'taskId': '1024'}, function(){
   alert(sessionStorage.getItem("SuccessURL"));
});
```

asl.minimize();

A client application may request to hide its own window. This can be done by calling the minimize function. The minimize function does not close the application, it leaves it in background mode.

NOTE When a callback is defined, make sure that all variables and functions in the callback are in the global scope or are kept in session or local storage. When the callback function is defined, it is kept from the asl library during page preloads. Even if the application navigates to a different url, once the callback is called, the asl restores the function and executes it. It is a best practice to keep all the callback variables in a persistence store as session and local storage.

Events

Client applications are provided with a set of events to which they can subscribe or invoke when they want to notify the SB1 Shell. There are two functions in **asl** for managing events:

- asl.events.fire (eventName, data)
- asl.events.subscribe (eventName, callback)

asl.events.fire(eventName, data)

Parameters

- eventName (string)
  The name of the event that is fired.
- data (object)
  Data that is sent to the subscribers of eventName.

Example

```
asl.events.fire('onerror', 'Problem with connection to the server.');
```

asl.events.subscribe(eventName, callback)

Parameters

- eventName (string)
  The name of the event that is subscribed.
- callback (function)
  The function that is executed, if eventName occurs.

Example

```
asl.events.subscribe('onfocus', function(){changeBackground();
});
```

Native Shell Events

The following list contains all the events that are supported by the SB1 Shell for subscribing:

- **onFocus** - Called when the application receives focus.
- **onFocusOut** - Called when the application loses the focus. For example, Home button is pressed, lock/badge screen is turned on or a notification appears.
- **onBackPressed** - Called when the user presses the back button.
- **onOptionSelected** - Called when the user selects an option from the options menu.
- **onCradleInsert** - Called when the SB1 is placed in a cradle.
- **onCradleRemove** - Called when the SB1 is removed from a cradle.
- **onLowBattery** - Called when the battery level reaches a limit specified in the configuration file.
- **onCriticalBattery** - Called when the battery level reaches a limit specified in the configuration file.
- **onSignalLost** - Called when the RhoElements signal event is fired and the signal percentage is zero.
- **onSignalRestored** - Called when the RhoElements signal event is fired and the signal percentage is different than zero.
- **onKill** - Called when the application is forced to exit.
- **onExit** - Fired by the application when it is ready to exit.

- **onLoaded** - Called when the application has confirmed that all the scripts and files are loaded and it is able to communicate with SB1 Shell.

NOTE Subscribing for an event is valid only for the current page of the application. Navigating to a new page within the application removes the event subscriptions. Multiple subscriptions to the same event are executed in the same order as defined.

Notification Services

Notifications are messages sent to the SB1 user with specific properties (text, icon and actions). They are used to inform the user when some system, network or application event occurs. The **asl** module provides a function that can be used to put a notification to the notification queue based in the SB1 Shell.

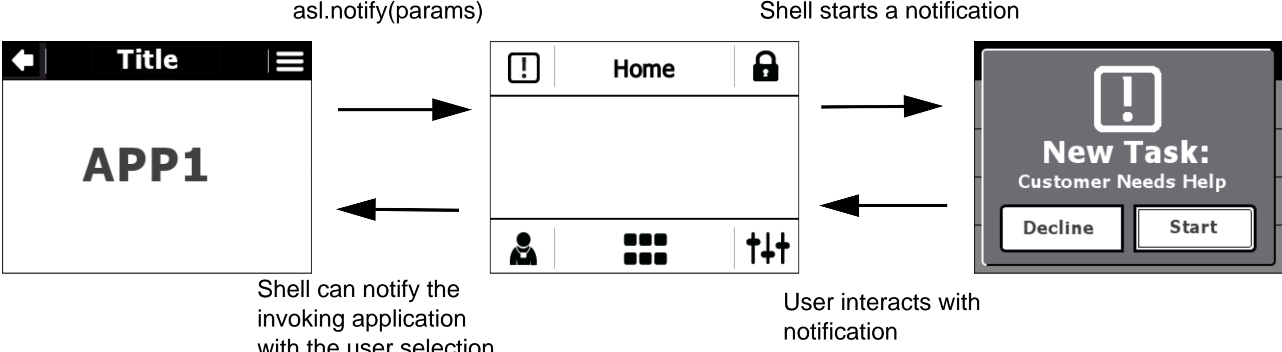


Figure 3-3 *Notifications*

A notification can be invoked with the following method:

```
asl.notify(type, priority, title, message, buttons, actions, timeout,
iconPath);
```

Parameters

- type (string)

  Type of the notification: application, server or system.

- priority (string)

  Priority of the notification: low, normal or urgent. Note that notification priority is not currently used by the SB1 Shell but has to be provided when a notification is created.

- title (string)

  Title of the notification that is shown in the notification popup or notification list.

NOTE If the title is blank, the default title **New Message** displays on a notification and in the notification list.

- message (string)

  The text of the notification that is shown in the detailed notification popup or view.

- buttons (array)

  Array of strings that contains the text for the notification buttons. Button length is a maximum of two items; If it is not provided (null value is provided) the notification shows the default buttons.

- actions (array)

  Array of functions that contains the actions for the notification buttons. The action at position 0 in the array corresponds to the item in the button list. The action at position 1 in the array corresponds to the item at position 1 in the button list.

- timeout (int)

NOTE If a notification times out, it will appear in the Notification tray.

- Number of seconds that the notification popup stays on the screen. 0 or null is for no timeout (the notification stays forever until a user action).
- iconPath (string)

  Absolute path to an image that can be used as a notification icon.

Example

```
asl.notify('application', 'normal', 'Message', 'Application A needs to close.',
['OK'], [function(){asl.exit()}], 10);
```

NOTE If you provide one button, it is automatically set as default button. If you provide two buttons, the second one is set as the default button.

To use one of the buttons to dismiss the notification, set its callback as null.

When a callback is defined, make sure that all variables and functions in the callback are in the global scope or are kept in session or local storage. When the callback function is defined, **asl** keeps it during page pre-loads. Even if the application navigates to a different url, once the callback is called, the **asl** restores the function and executes it. It is best to keep all callback variables in persistence store session and local storage.

Example

```
asl.notify('application', 'normal', 'Restock',
   'Produce needs to restock all brands of tomatoes, onions and fresh herbs.',
   ['Accept', 'Dismiss'],
   [function(){
asl.run('http://www.motorola-solutions.com/citf/app1.html',
{'taskId': '1024'}, true);
   }, null], 10, 'http://www.example.com/image/icon.png');
```

The above example produces a notification that displays a message to the user and two options; accept the task or dismiss it.

If the user accepts the task, the `asl.run` callback is called from within the client application that created the notification (note that the anonymous function in the callbacks parameter should be defined in the client application window). This `asl.run` call starts a Custom In-Task Flow (CITF) application located at the given URL with some additional data parameters (taskId). If the client application that created the notification has changed its location to a different URL by the time the user interacts with the notification, the **asl** library stores the callback function so it can be accessed by the notification.

The following example shows how to call a notification that uses a URL callback instead of function callback.

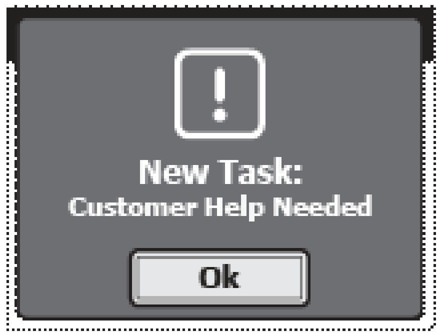
Example

```
asl.notify(
   'Produce needs to restock all brands of tomatoes, onions and fresh herbs.',
   ['Accept', 'Dismiss'],
   ['http://www.motorola-solutions.com/example/
callback_url.aspx?params=123']);
```

The difference in this example is that when the user selects **Accept**, the client application that created the notification is forwarded to the URL defined as the callback.

There are two types of notifications depending on how the notification is defined:

- blocked - where the user is forced to choose a specific action.
- normal - where the user is given options to choose.



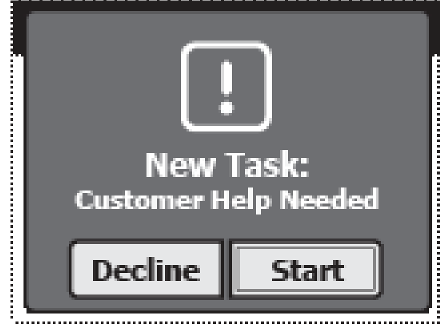


Figure 3-4 *Notification Options*

If the user receives more notifications, the notification stack displays.

Keyboard Services

Keyboard services provide automatic on-screen keyboards when the user touches a text field on the screen. The SB1 Shell provides various modes of keyboards, alpha, alphamore, symbol, currency and numeric, etc.

Depending upon the type of the html input field, the **asl** module activates the keyboard that is required. Numeric keyboard for number, date and time fields. Alpha keyboard for all other input fields.

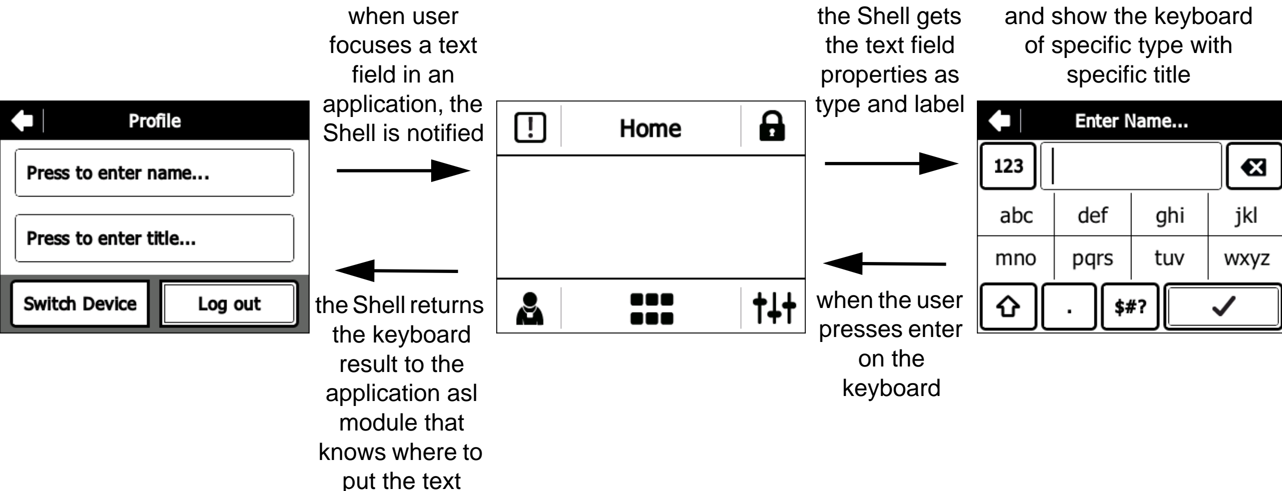


Figure 3-5 *Keyboards*

The appearance of the keyboard is controlled by the **asl** module. The **asl** module tracks the load event of the client applications document, discovers all the text fields in the document and uses the focus event of each text input field to activate the keyboard on text field focus. The same tracking and discovery procedure is executed for text inputs that are dynamically added to the document, after the load event, using the DOM Mutation events.

Explicit Keyboard Execution

Applications can invoke the keyboard manually by calling the `asl.showKeyboard()` or `asl.keyboard`() method. The text inputted into the keyboard will then be returned to the application via a callback.

```
asl.showKeyboard(params, callback);
```

Example

```
asl.showKeyboard({
   inputId: "test",
   mode: "number",
   title: "Enter Value",
   value: "predefined value"
}, function(re, val){ alert(val); });
```

```
asl.keyboard(type, callback, text);
```

Example

```
asl.keyboard('number', function(keyboard_input){
   //do something with the keyboard numeric input
   application_function(keyboard_input);
}, "Some text");
```

In the above example, the keyboard starts with default text written in the SIP text field, **Some text**.

Parameters

- type (string)

  The SB1 Shell on screen keyboard supports five types of keyboards: text, numeric, date, pin, currency and time. Any other type that is provided is translated from the shell to text type. The keyboard type can be set by setting the type of the input field or by setting the first parameter of the asl.keyboard method.

```
<input id="myInput" type="number" />
```

- title (string)

  Set the keyboard title by providing a title attribute of the text input field in HTML.

```
<input id="myInput" type="number" title="My Field" />
```

- maxLength (int)

  Set the fields maximum length by setting the maxlength attribute of the text input field within the HTML code. The keyboard considers this property and limits the input.

```
<input id="myInput" type="number" maxlength="4" />
```

- data-autoReturn (bool)

NOTE data-autoReturn (bool) is not available for scanned data.

- Setting this attribute will automatically close the keyboard and post the text back into the input field when the maxlength value has been hit. Note that data-autoReturn does not work without a maxlength set.

NOTE If a bar code is scanned with exact maximum length size, auto-return does not work. User needs to click on tick mark button.

```
<input id="myInput" type="number" maxlength="4" data-autoReturn="true" />
```

Keyboard Events

When the keyboard posts the text to the text field, the **asl** library fires the text input elements onchange event. By listening to this event, client applications can be notified when the value of the text field has changed.

```
<input id="myInput" type="number" onchange="alert('Event onchange is
defined.')" />
```

Shell version 2.8.3 and above provides a new features to the SIP which is starting the SIP in specific mode from the asl. Third party applications can have a text field of text type but start the SIP in numeric mode. In previous versions the Shell was invoking the SIP by default e.g. alpha mode. This feature is only applicable for text, bar code and password fields. To enable SIP mode, third party applications have to use data-mode attribute for text inputs.

The following provides the specifications of this feature:

```
<input type="currency" />
```

This will be ignored as the only available SIP type for currency field is numeric SIP.

Figure 3-6 *Currency Keyboard Sample*

```
<input type="text" data-mode="number" />
```

SIP will be started in numeric mode.

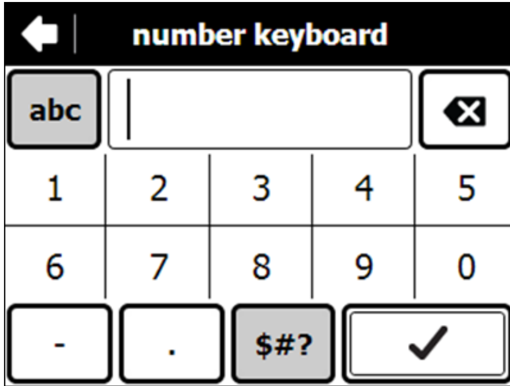


Figure 3-7 *Number Keyboard Sample*

```
<input type="password" />
```

SIP will be started in alphaMore mode e.g. if there is a second alpha chars mode it will be started by default.

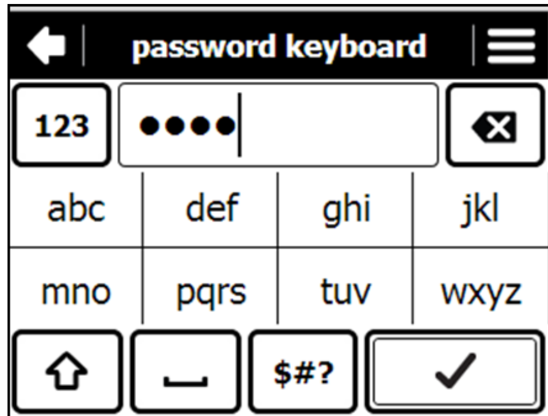


Figure 3-8 *Password Keyboard Sample*

```
<input type="pin" />
```

SIP will be started in PIN mode.

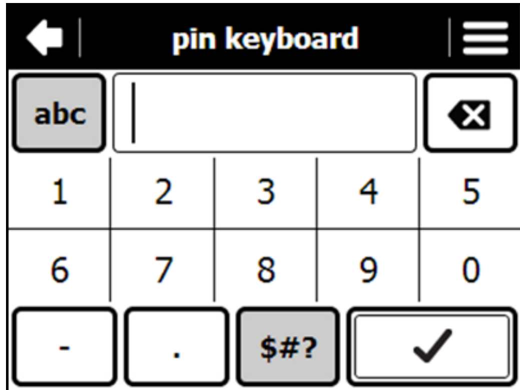


Figure 3-9 *PIN keyboard Sample*

```
<input type="text" data-mode="symbol" />
```

SIP will be started in symbol mode.

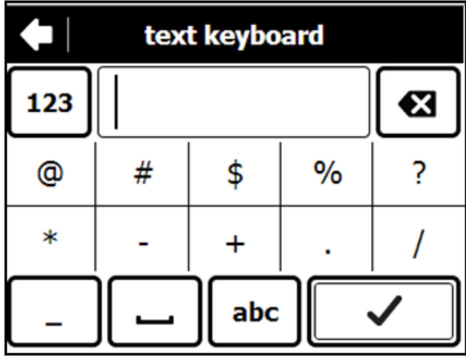


Figure 3-10 *Text Keyboard Sample*

Shared UI Services

Each client application can display and use a SB1 Shell specific title bar that provides a consistent user navigation across applications. The title bar contains three elements; a back button, title information and an options menu. The **asl** module provides an API to control the title bar elements.

These functions to be added in web pages as first statements.

- asl.title('Page Title');
- asl.back();
- asl.options();

Back Button

Applications can define a back action by calling the `asl.back` function with a callback function as a parameter. Defining the back action will show the back button on the title bar. Once the back button is pressed, the Shell executes the defined callback function. If the application wants to disable the back button it should call `asl.back` without parameters (or null for callback). Users can use whichever page they do not need back arrow. Options and back arrow will be persisted.

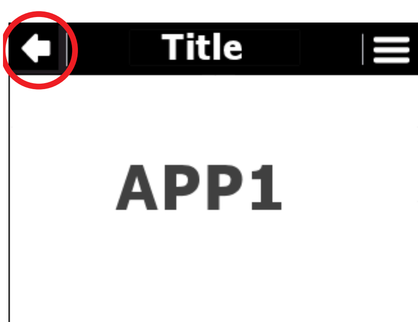


Figure 3-11 *Back Button*

```
asl.back(callback);
```

Example

```
asl.back(function(){
//do something
});
```

It is also possible to change the back button image by calling `asl.back` with an additional parameter:

```
asl.back(callback, imageUrl);
```

Parameters

- callback (function)

  The function that will be executed once the back button is pressed.

- imageUrl (string)

  An absolute path to an image to be used as the back button.

Example

```
asl.back(function(){
//do something
}, 'http://www.example.com/assets/back.gif');
```

Title Label

Applications can define title text by calling the `asl.title` function with a string parameter. This title text will be displayed in the title bar. If an application requires an empty title then it should call the `asl.title` function with an empty string or null.

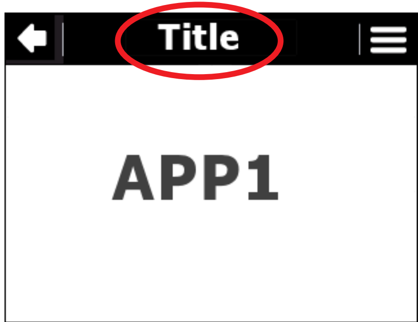


Figure 3-12 *Title Label*

For the English language, a maximum of 20 lowercase characters or a maximum of 11 uppercase characters can fit in title. For other languages the developer must verify and accommodate the text.

```
asl.title(titleText);
```

Example

```
asl.title('Title');
```

Options Menu Button

Applications can define an options menu by calling the `asl.options` function with a collection parameter containing the defined options. If an application requires no options menu it should call the `asl.options` function with empty or null parameter.

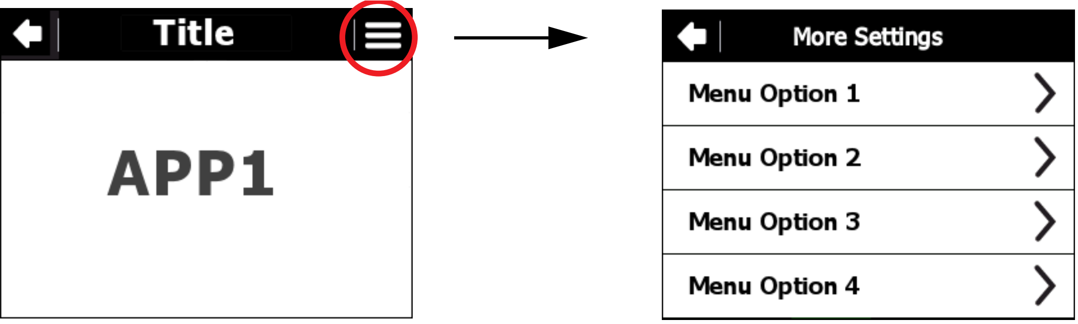


Figure 3-13 *Options Menu*

For the English language, a maximum of 25 lowercase characters or a maximum of 14 uppercase characters can fit in list item. For other languages the developer must verify and accommodate the text.

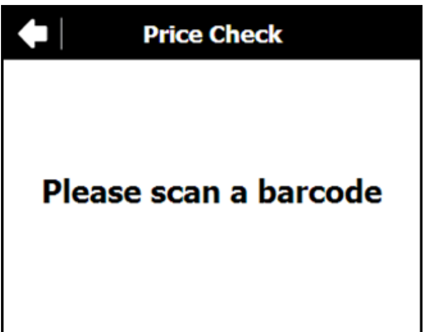


Figure 3-14 *Without Options*

Figure 3-15 *Without Options and Back Button*

Figure 3-16 *Custom Back Icon*

```
asl.options(optionsList);
```

Parameters

- optionsList (array)

  Array of objects containing title and callback property. Title will be used to set the text for the option menu. Callback is a function that will be executed once the option is selected.

  Option list should contain:

  - Title: Name of the menu item.
  - Callback: callback function for menu item.

Example

```
asl.options([
   {
   'title' : 'Menu Option 1',
   'callback' : function(){/*do something*/}
   },
   {
   'title' : 'Menu Option 2',
   'callback' : asl.exit
   },
   {
   'title' : 'Menu Option 3',
   'callback' : asl.exit
   },
   {
   'title' : 'Menu Option 4',
   'callback' : asl.exit
   }
]);
```

✓ ***NOTE*** All title bar elements are persistent between page loads. Once the back, title or options are defined, they will persist for each page within the same application. If you want to change or disable any of the back, title or options services, the application will need to call the `asl.back`, `asl.title` or `asl.options` function with either null or the new callback function.

Full Screen Mode

Applications can run in full screen mode by not defining, title, back button, and option menu.

Figure 3-17 *Full Screen Mode*

Resource Services

Access to device resources such as scanner, audio and battery is provided to client applications through the **asl** module. The asl module communicates with the System Library module to prevents conflicts which could arise when different client applications try to use same resources at the same time. The RhoElements NPAPI objects are overwritten by he asl library which will forward the necessary actions to the original RhoElements NPAPI objects depending on the application request. Note: The asl based overridden NPAPI objects are not defined under the **asl** namespace for backward compatibility with RhoElements Help documentation.

NPAPI

RhoElements uses an NPAPI interface to give applications access to the device capabilities via JavaScript. Client applications that need to access these device capabilities are able to do so through the asl library.

The following is a list of APIs available in **asl**:

- fileTransfer

  Note: Relative file path does not supported in SB1 device.Full File paths to be provided.

  Ex: file://\\UserDrive\\apps\\app7\\test_http_device.txt'
- audioCapture
- scanner
- wlan
- battery - Battery module is not overridden by the asl library and works entirely in RhoElements mode. asl library provides additional events for the status of the battery which are:
  - oncradleinsert
  - oncradleremove
  - onlowbattery
  - oncriticalbattery.
- push - Push port, push passkey and number of parameters for push notifications are available for configuration in the Shell in `config.js` file. List of push module settings:
  - detected - can be overriden by a client application. If a push event is available for the current client application, the callback or the url defined in detected parameter will be invoked
  - start - not available for use in client applications
  - stop - not available for use in client applications
  - port - not available for use in client applications
  - passKey - not available for use in client applications
  - response - not available for use in client applications
  - path - not available for use in client applications
  - unattended - not available for use in client applications

  port, passKey properties can be configured in `config.js`. Json formatted push event is not supported by the Shell;
- signal - asl library provides additional events for notifying client applications for the signal status:
  - onsignallost
  - onsignalrestored
- keyCapture - Only the keyEvent property is enabled in asl.

- generic - Available generic methods and properties:
  - InvokeMETAFunction
  - Log
  - LaunchProcess
  - LaunchProcessNonBlocking
  - CloseProcess
  - GetProcessExitCode
  - WaitProcess
  - SetRegistrySetting
  - GetRegistrySetting
  - PlayWave
  - ReadConfigSetting
  - WriteConfigSetting
  - ReadUserSetting
  - WriteUserSetting
  - OEMInfo
  - UUID

NOTE The push module works in a custom mode within the asl library. Since there is only one push server for the SB1 Shell, it is defined and started at Shell startup and is shared between all applications. It is predefined with a custom number of parameters which can be set in the configuration file (see *Push Notifications Number of Parameters on page 4-10*). Since the push service is shared between all client applications, to be able to forward the data to the correct application, the push request will need to contain an extra parameter (the name of the application). This parameter must be the first parameter in the push request. If the external source that calls the push applies more parameter than configured in config.js, they will not be passed to the client application.

NOTE All of the above APIs work in the same way as in a raw RhoElements environment. Client application developers will not have to do anything additional in order to use and work with the NPAPI object. The way NPAPI are supposed to be used is the same as how the RhoElements help documentation defines - using META tags, the generic object or by directly accessing the required module.

Scanner Limitations:

- scanner.autoTab = 'enabled' not supported due to technical limitations. Customers can use asl.showkeyboard() to call after successful receipt of scanned barcode data.
- scanner. auto Enter= 'enabled' not supported due to technical limitations. Customers can use submit the form on successful receipt of scanned barcode data.

Refer to the RhoElements Help Documentation for more information on the above APIs. Go to: http://docs.rhomobile.com/en/2.2.0/rhoelements/apicompatibility.

Messaging Services

Messaging services provide the ability for each application to communicate with the SB1 Shell and other applications. Messaging services are based on the HTML 5 postMessage method. The Shells Messaging services are not designed to be used by applications using the postMessage API. Instead, the **asl** module supports a common language between applications and the SB1 Shell based on messaging services which are responsible for passing between instances commands, events, notifications and NPAPI calls.

When an application is started, the Shell always sends a message (hi) to the application. With this message the SB1 Shell ensures that the client application is able to communicate with it and passes important parameters (like id and shell address) to the application. The SB1 Shell expects that the client application will respond to this message with an accepted status. After this handshake, both SB1 Shell and the client application establish a connection and are able to communicate with each other.

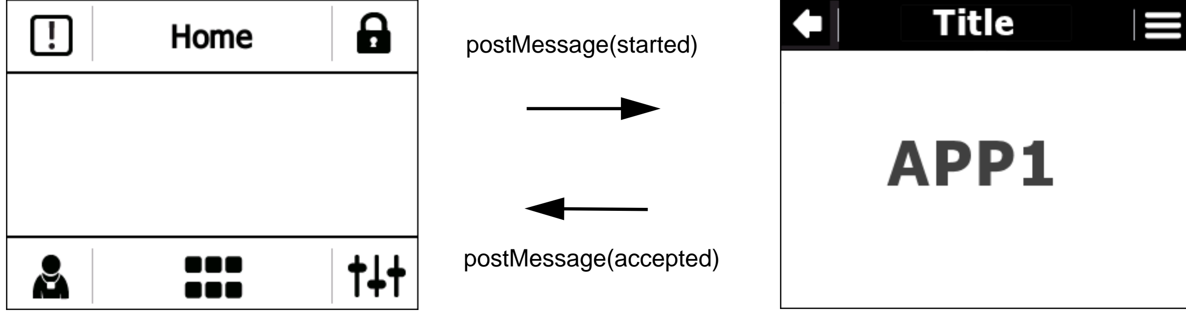


Figure 3-18 *Message Handshake*

NOTE The handshake process occurs on every page load in the client application.

Client application developers do not need to do anything explicit to make this handshake happen. They only need to include the asl library and it processes this communication.

Authentication Services

The SB1 Shell supports user profiles and two types of authentication; Shell authentication and Application authentication. By default, the SB1 does not use authentication.

User Profiles and Authentication

By default, the SB1 Shell has a system user, which is defined in the configuration file, and has the following properties:

- **name** - The name of the user to be displayed on the badge screen.
- **title** - The title of the user to be displayed on the badge screen.
- **pin** - The pin code of the user. A numeric value that is requested from the user if the lock screen is enabled.
- **dept** - Department of the user.
- **photo** - Photo of the user.
- **locale** - The current locale of the Shell.
- **tegroup** - PTT Express group (currently not implemented).

If the SB1 Shell authentication is enabled in **`config.js`**, then users may have additional properties coming from the authentication server that can be customized for a customer. Once the login is successful, the client application responsible for the login can change the user profile by calling the asl.auth API.

Shell Authentication

The SB1 Shell supports central authentication and has to be configured to use a client application as an authentication application and to enable the login in **`config.js`** file.

```
config.login.require = true;
config.login.page = "http://127.0.0.1:83/UserDrive/apps/auth/login.html";
```

The above configuration enables the authentication.

NOTE After you change the configuration in */UserDrive/config* folder you may need to go to the SB1 Shell profile screen and force log out in order to reset the browser to the modified configuration settings.

After the authentication is enabled on the device, start up users will see the authentication screen provided by the authentication application page (in the example case this will be the content of login.html file). This application should support forms authentication as this is the only supported method.

Users are able to populate their username, passwords or any custom fields using the keyboard.

Once the authentication application authenticates the user, it has to notify the SB1 Shell. This is done through the **asl** API called asl.auth.

```
asl.auth(status, data);
```

Parameters

- status (bool)

  True or false depending if the user has been authenticated successfully or not.

- data (object)

  The user object (described in shell authentication properties below) populated with the data of the authenticated user.

Example

```
<script type="text/javascript"
src="http://127.0.0.1:83/Application/www/sapp/src/asl.js"></script>
...
<script type="text/javascript">
   asl.events.subscribe(asl.events.types.loaded, function(){
      asl.auth(true, {
         apps: 'http://127.0.0.1:83/UserDrive/apps/auth/doe.json',
         name: 'John Doe',
         pin: '0001',
         idleTimeout: 1800
      });
   });
</script>
...
</html>
```

With the above call, the authentication application supplies the SB1 Shell with the specific user data.

This API call should be executed in the first page after the authentication client application authenticates the user.

NOTE Shell authentication supports a login timeout. If the auth message is not received within that timeout, the login is considered unsuccessful.

NOTE When using Shell authentication, application developers should provide a signout URL in order to allow the Shell to sign out the user in different circumstances like pressing sign-out button, cradle in or critical battery events or idleTimeout. This can be done by setting the config.login.signOutUrl parameter in `config.js` file.

Shell Authentication Properties

When using Shell authentication, the following properties can be set:

- name
- title
- pin

- apps
- idleTimeout
- badge.

Application Authentication

The SB1 Shell supports Application authentication. A client application that requires authentication needs to implement forms authentication and to integrate the **asl** library to its pages. Once the client application covers these conditions it does not need to inform the SB1 Shell for login status or any user data.

Badge Mode

The SB1 Shell has the ability to allow client applications provide the badge page that appears once the SB1 enters into badge mode. This can be done by using an **asl** method in the client applications page:

```
asl.badge(url);
```

Parameters

- url (string)
  The url of the badge page that displays once the SB1 gets locked or rotates.

Example

```
asl.badge('http://www.example.com/data/user1/badge.html');
```

NOTE For security purposes, smart badge should be enabled for a client application once it is installed in the **`apps.json`** file. This can be done by adding an attribute to the client application record called canOverwriteBadge which accepts true or false.

```
{
   "name": "App F",
   "url": "http://127.0.0.1:83/apps/app8/index.html",
   "icon": "default_app.png",
   "canOverwriteBadge": true
},
```

Setting User Profile from an Application

The user profile can be updated by any application that by using asl.profile method. The application needs to be configured as canOverwriteBadge (see above) in order to be able to overwrite the profile. After the profile is updated, the SB1 Shell restarts RhoElements to reload the new profile.

```
asl.profile({name: "James", photo:
"http://127.0.0.1:83/UserDrive/apps/app8/img.png",
apps:"http://127.0.0.1:83/UserDrive/apps/app8/fr.json"}
```

The above example updates the profile with a new name "James", a new photo of James and a new application list.

Window Services

The asl library includes a predefined version of alert and confirm pop-up native windows. It overrides the default alert and confirm functions and provides other custom designed pop-ups that acts in a similar way.

Alert Window

```
alert(message);
```

Example

```
alert('This application will close');
```

Confirm Window

```
confirm(message, ok_callback);
```

Example

```
confirm('This application wants to close', function(){
   save_current_data();
});
```

The difference between the native alert/confirm windows and the asl alert/confirm windows is that the asl representation does not stop the thread once the popup appears. For this reason the confirm function requires a callback for the OK action, because the function is executed only when the user confirms the action.

NOTE The native alert/confirm pop-up windows are still available through _alert/_confirm. It is highly recommend to use the native pop-ups only for debugging purposes.

Hourglass

In previous versions of the Shell there were circumstances in which a number of blank white screens displayed during page transitions and application loading. To avoid these white screens, an hourglass is displayed during application loading and page transitions. This feature is partiality implemented in this release which will cover only local and remote applications except in setting the screens.

Wait/Quit Feature

In previous versions of the Shell there were no indications to the user to show whether a page is loading or if there are any network issues when loading a page. Instead, a blank white screen would appear providing no information on what was happening on the device. In the current release of the Shell, a Wait/Quit notification is displayed if a page is not loaded within 10 seconds (default) after the navigation request occurs. A configuration option in the **config.js** file can be used to configure the default value of this delay. The option is:

```
config.waitForResponseTimeout = 10*1000;
```

Whenever there is no network or low network areas, the HTTP request of the client application may not reach the remote server and a Wait/Quit notification will be displayed. By default, the Shell is configured to show two Wait/Quit notifications and then a Retry/Quit notification. Once the user clicks on the Retry button, if there is no network connection, a Network Not Connected page displays. Once the SB1 connects to a network, the device's IP address displays and the client application automatically loads the previous GET request page.

To configure the number of consecutive Wait/Quit dialogs:

```
config.numberofWaits = 2;
```

If the Shell has been configured to require the user to logon, any "Wait/Quit" dialogs for the logon application will not display the "Quit" option. This stops the user from quitting the logon application and accessing the device without authentication.

The Network Not Connected page can be customized by changing the following **config.js** parameter to a user defined file:

```
config.retryURL = "/UserDrive/www/sapp/retrymanager.html";
```

This should point to an existing local file.

Startup Applications

In this and previous versions of the SB1 shell, developers can configure the shell to launch a startup application when the SB1 boots.

Upon booting the SB1, there can be a delay before connecting to a Wireless network. This can interfere with the behavior of remotely hosted startup applications as the application will be requested before the Wireless connection has been established. If the SB1 cannot connect to the startup application server after 10 seconds, a "Wait/Quit" dialog notification will display to the user. Pressing "Wait" will dismiss the notification and wait another 10 seconds for the page to load. If the page has still not loaded, the "Wait/Quit" dialog will appear again. If "Wait" is again selected and the page is not loaded after another 10 seconds, another "Retry/Quit" dialog displays. Pressing "Retry" will display a "Network not connected" page which will show realtime device IP information and will resend the HTTP request when reconnected.

If the user logs out from the profile screen, RhoElements will restart but will maintain the connection to the Wi-Fi network. Therefore, it does not need to re-negotiate its connection when starting, and the startup application should load without any network issue.

Ghosting

Sometimes ghosting of a previous screen might be observed during the usage of device. As per the e-ink display on the SB1 device, the screen may only update partially depending on how much of the screen needs updating and how many updates since the last full screen update. This may not be noticeable by regular users of SB1 device.

Cradle Insert and Removal Enhancements

A configurable delay has been introduced between the "oncradle" event and when the shell closes all the running applications (if configured to do so). This delay allows applications to correctly log out or shut down. The default value for this setting is 1 second.

Developers can configure applications to be startup applications. A new configuration option will open these a applications as soon as the device is removed from the cradle.

Ex: config.startUpApp = "PriceCheck";

```
config.cradle.openStartupAppOnCradleRemoval = false;
```

QWERTY Keyboard

Client applications can disable the default keyboard and use client specific keyboards. Ex: QWERTY keyboard.

To enable this feature, configure **apps.json** with disableKeyboard set totrue.

Example

Process Application

```
{
"name": "Customer app name",
"url": "http://127.0.0.1:83/UserDrive/apps/qwerty/Index.html",
"login": false,
"disableKeyboard":true
}
For CITF applications, keyboard can be disabled using asl.run() API.
asl.run(url, data ,callback, disableKeyboard)
```

The client applications can start an application as a process application to manage any special business use cases. For example; If a client application using a server sent events for the application instead of push notifications, the application need to open a session and maintain the session. The client application can open a process application for this purpose and manage the SSE connection and any other business process can be done. Asl.runProcess() method can be used for the process application. This method is similar kind of asl.run() method.

asl.runProcess(url, data, callback, disableKeyboard)

Parameters:

- url - URL of the application
- data - json data can be passed to the application
- callback - callback function will be called when application is closed.
- disableKeyboard - to enable/disable keyboard on process application.

CHAPTER 4 CONFIGURATION

This chapter details all of the configuration settings which have been provided to allow customization of the SB1 Shell. These configuration settings are contained in the **`config.js`** file.

NOTE The **`config.js`** is a JavaScript file which needs to be configured carefully. JavaScript is a case sensitive language and requires good syntax. Care must be take to produce a correct file.

ScanTo Application

```
config.scanToApp = {
   url: "http://localhost:83/UserDrive/apps/app5/scanto.html?data=%s"
};
```

ScanTo is a feature of the shell that enables bar codes to be scanned from the home page. The purpose of this feature is to get enable access to a bar code lookup application simply by pressing the home button and scanning a bar code. When a bar code is scanned from the home page, an application specified in this configuration setting is launched with the bar code data passed to the application as part of the query string of the URL.

This configuration parameter defines the URL to send the scanner decode information. This is equivalent to setting the RhoElements scanner module's “decodeEvent” event parameter. The syntax for both can be found in the RhoElements documentation. Refer to the RhoElements Help Documentation for more information. See http://docs.rhomobile.com/v/2.2/rhoelements/scanner.

Example

```
<!DOCTYPE html>
<html>
   <head>
      <script type="text/javascript"
      src="http://127.0.0.1:83/Application/www/sapp/src/asl.js"></script>
      <script>
         asl.title('ScanTo Example');
         asl.back(quit);

         function quit() {
            asl.exit();
         }
         function getDataFromQueryString() {
            var data = window.location.search.substring(6);
            //Do a lookup (i.e. AJAX) using the data variable
            document.getElementById('dataSpan').textContent = data;
         }
         window.addEventListener('load', getDataFromQueryString, false);
         </script>
      </head>
   <body>
      <div>Scanned Data = <span id="dataSpan"></span></div>
   </body>
</html>
```

Maximum Number of Running Applications

```
config.apps.maxRunning = 5;
```

Defines the number of applications that can be running at the same time. It is advised to run as few applications as possible to get the best performance from the device. The advised value is 5 or less. If the user requests 6th Iframe, the system displays a notification popup to indicate to the user that the system is closing the least used application.

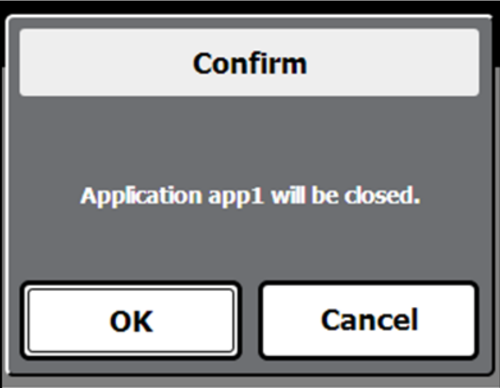


Figure 4-1 *Close Application Notification*

Application Source File Location

```
config.apps.src = '/UserDrive/config/apps.json';
```

Defines the location of the default application file. The list of applications that are installed on the SB1. This parameter can point to either a local or remote file. If the configured file does not exists , the shell loads the default **apps.json** file instead.

Push Notifications Port Number

```
config.push.port = '8080';
```

Port number for the push module.

Push Notifications Pass Key

```
config.push.passKey = null;
```

Defines the passKey value for the RhoElements push notification module.

Home Screen Shortcut Buttons

```
config.homeButton1 = {};
config.homeButton2 = {};
```

Configuration for a home page application shortcut. The provided icons are seen on the home screen, the app property must be set to an installed application name.

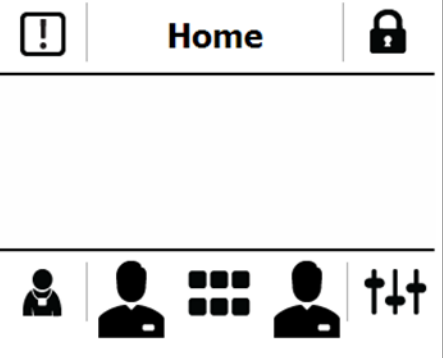


Figure 4-2 *Home Screen*

Example

```
config.homeButton1
{ imagePath: '/UserDrive/apps/myapp/myicon.png', app: 'General' };
```

PIN Lock Require

```
config.lockscreen.requirePIN = false;
```

Enables or disables the PIN for the SB1 Shell lock screen. The default PIN can be set through the user configuration object.

Default Lock Screen Page

```
config.lockscreen.page = "";
```

A URL to a custom lock screen page. The lock screen is used once the SB1 is locked.

Example

```
config.lockscreen.page = "http://www.example.com/badge.html";
```

Figure 4-3 *Default Badge Screen*

Figure 4-4 *Custom Badge Screen*

Lock Screen Timeout

```
config.lockscreen.timeOut = 60*1000;
```

Defines the time period, in milliseconds, of no user activity required to automatically lock the screen. Set to 0 to disable screen locking.

Rotate Badge Screen Timeout

```
config.lockscreen.timeOutOnRotate = 3*1000;
```

Defines the time period, in milliseconds, to wait after the SB1 is rotated into the up-side down orientation before the badge screen is shown automatically. Set to 0 to disable rotation.

Disable Profile Button

```
config.profile.disable = false;
```

Enables or disables the profile button in home screen.

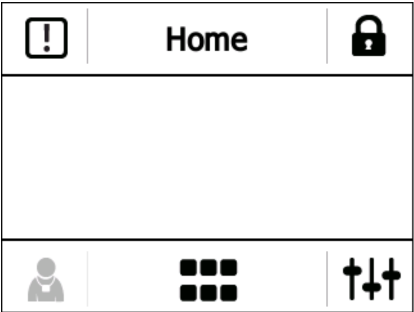


Figure 4-5 *Home Screen with Profile Button Disabled*

Default User Information

```
config.user.data = {
   name: "Badge",
   title: "User Config",
   pin: "0000",
   dept: '',
   photo: '',
   locale: 'en-US',
   local: '',
   tegroup: ''
};
```

Defines the default user information that is used in the profile and lock screens.

Example

```
name: "Nick Price",  (Name of the User)
title: "Sales Associate", (Title of the User)
pin: "0000",(Badge PIN)
dept: 'Sales',(Department of User)
photo: 'http://127.0.0.1:83/UserDrive/myPhoto.png',(Photo of the User)
locale: 'en-US',(Prefered Language of the User)
local: '', (This property for future purpose)
tegroup: ''  (This property for future purpose)
```

Login Screen Require

```
config.login.require = false;
```

Enables or disables the login screen. Works together with config.login.page (if this setting is not provided, login is not enabled).

Login Screen URL

```
config.login.page = "";
```

Defines the page address for the login screen. This page appears if the user is not authenticated against the SB1 Shell. Login should be enabled by config.login.require.

Example

```
http://127.0.0.1:83/UserDrive/apps/auth/login.html
```

Logoff URL

```
config.login.signOutUrl = "";
```

Defines the URL to be used for the signing off action when the SB1 Shell authentication is enabled. It is called once a user tries to sign out, switch devices or any other activity that requires logging off.

Example

```
config.login.signOutUrl = "http://example.com/logoff.html";
```

Login Timeout

```
config.login.wait = 30*1000;
```

Defines the time, in milliseconds, that the SB1 Shell should wait for the Single Sign On (SSO) application to respond with success or fail.

Admin Settings PIN Require

```
config.adminPINrequire = false;
```

Enables or disables the PIN for the Administrative Settings section.

Admin PIN value

```
config.adminPIN = '668765';
```

Defines the value of the Admin settings PIN. Default value: 668765.

Number of Levels for the Beeper Settings Screen

```
config.maxBeeperVolumeLevel = 4;
```

Defines the number of levels for the beeper settings control.

Beeper Duration

```
config.beeperDuration = 2000;
```

Defines the duration of the device beeper on SB1 Shell notifications (RhoElements documentation should be used for reference for the available values).

Beeper Frequency

```
config.beeperFrequency = 2900;
```

Defines the frequency of the beeper on SB1 Shell notifications (RhoElements documentation should be used for reference for the available values).

Date Format

```
config.dateFormat = 'DD/MM/YY';
```

Defines the date format used for the settings set date and time functionality. Options: DD/MM/YY, or DD/MM/YYYY, MM/DD/YYYY or MM/DD/YY.

Time Format

```
config.timeFormat = 'HH:MM';
```

Defines the time format string used for the settings set date and time functionality. Options: HH:MM:SS, HH:MM or MM:SS.

Default Volume Level of the Speaker

```
config.defaultVolume = 4;
```

Defines the level of the speaker volume value in config.volumeValues array. Options: 0 -4. Default = 4.

Cradle Insert URL

```
config.showCradleInsertURL = true;
```

Enables or disables the appearance of a system page when the SB1 is placed into a cradle.

Low Battery URL

```
config.showLowBatteryURL = true;
```

Enables or disables the battery low system page.

Beeper for System Notifications

```
config.beepForSystem = true;
```

Enables or disables beeper sound for system notifications. Can also be changed through settings page.

Beeper for Server Notifications

```
config.beepForServer = true;
```

Enables or disables beeper sound for server notifications. Can also be changed through settings page.

Beeper for Application Notifications

```
config.beepForApplication = true;
```

Enables or disables the beeper sound for application notifications. Can also be changed through settings page.

Number of Waits

```
config.numberofwaits=2;
```

Number of wait/quit notifications to be displayed to user incase of low network, no network or slow server responses.

Wait for Response Timeout

```
config.waitforResponseTimeout = 10*1000
```

Timeout for Wait/Kill notification on application start or application page transition.

Clear Local Storage on Reboot

```
config.clearLocalStorageOnReboot = true;
```

This configuration clears the local storage and session storage during device start up. It is up to the application user to configure this setting. Setting this to false will increase localStorage and sessionStorage resulting in performance issues.Only local storage data for local applications is cleared.

It is the application user responsibility to clear the localStorage and Session Storage at appropriate time if this is set to false.

Scanner Configuration for the scanTo Application

```
config.scanner = {
   duration: '5000',
   code128: 'enabled',
   ean13: 'enabled'
}
```

Holds the default settings for the scanner, when used in ScanTo function. The possible values that can be held in this configuration setting object are any scanner or decoder parameter detailed in the RhoElements scanner documentation pages:

- http://docs.rhomobile.com/rhoelements/toc_decoders
- http://docs.rhomobile.com/rhoelements/scanner

The format of each scanner setting within this object is:

[parameter]: '[value]',

Where [parameter] is the name of the parameter detailed in the RhoElements documentation, and [value] is the appropriate value, also detailed in the documentation. For example, to enable the code39 decoder, the setting would be:

code39: 'enabled',

Note the format for this setting is the regular JSON Object notation. Therefore, ensure the last scanner setting does not end with a comma.

Push Notifications Number of Parameters

```
config.pushCallbackParameters = 5;
```

Defines the maximum number of the parameters that the push notifications callback can receive.

Startup Application

```
config.startUpApp = "";
```

Holds a reference to an application to start on SB1 Shell launch. Can be a URL or installed application name.

Example

```
config.startUpApp = "General";
```

or

Example

```
config.startUpApp = "http://localhost:83/UserDrive/apps/sample.html";
```

Cradle Insert Activities

```
config.cradle.clearNotifications = true;
```

Enables or disables clearing notifications when the SB1 is placed in the cradle.

```
config.cradle.closeApplications = true;
```

Enables or disables closing applications when the SB1 is placed in the cradle.

If **`config.cradle.closeApplications`** is set to true, the Shell will close all running client applications, will minimize all running system applications like settings, PTT, Wireless, Profile, App Launcher, etc., will bring the home screen as a current Shell application and will show up the Cradle Insert Page if it is configured to be shown. The expectation is that Home button will not do anything as there is no background application. If Cradle Insert page is disabled, Shell should be in full operation mode.

If **`config.cradle.closeApplications`** is set to false, the Shell will leave all running client applications but will minimize them e.g. if they're background apps, they will be removed as background apps, Shell will minimize all running system applications like settings, PTT, Wireless, Profile, App Launcher, etc., will bring the home screen as a current Shell application and will show up the Cradle Insert Page if it is configured to be shown. The expectation is that Home button will not do anything as there is no background application just after Cradling. If Cradle Insert page is disabled, Shell should be in full operation mode.

If Cradle Insert Page is disabled, the device has been cradled and some operations were executed while it is cradled, on cradle remove, Shell will minimize all running client or system apps and will bring up the Home screen without a background running application.

```
config.cradle.cleanUserData = true;
```

Enables or disables cleaning user data when the SB1 is placed in the cradle. If the user configures the data from the profile screen or any login application, the data will be deleted

```
config.cradle.waitBeforeClose = 1 * 1000;
```

If user configured **config.cradle.closeApplications** = true, the Shell will send a cradle insert event to the client applications to logout their applications or follow closing procedures of client application. The Shell wait for one second to close the client applications.

```
config.cradle. openStartupAppOnCradleRemoval = true;
```

Enables or disables opening of a client application upon device removal from cradle. If user configured startup application and this configuration is enabled, the application will be opened as soon as the device is removed from the cradle.

Keyboard Configurations.

```
config.keyboardEnabled= true;
```

Enables or disables the default shell keyboard in order to allow client applications to use their own keyboards. This is a device level configuration to disable shell keyboard and allows users to disable the keyboard for all applications including the shell. Users cannot access the keyboard even in settings pages. It is recommended not to change this setting.

In addition to above configuration users can control the enable disable of shell keyboard to only desired client application.

```
{
"name": "MWM",
"url": "http://192.168.5.174/MWM/TSD",
"icon":"http://127.0.0.1:83/UserDrive/mwm/Taskworker_SB1.png",
"login": false,
"disableKeyboard" : true
}
```

Others

By default, the SB1 Shell is configured to display a maximum of eight notifications in the Notifications list page.

```
/**
 * @description holds the maximum number of notification 0 - infinite
 * @type {Number}
 */
config.maxNotifications = 8;
```

By default, the SB1 Shell disables the network login feature as a part of optimize the device performance.

```
/**
 * @description enable/disable loading of wireless authentication feature in the
iframe when shell loads
 * @type {Boolean}
 */
config.loadfusion = false;
```

Enable Shell in debug mode by making true for the config.debugInfo variable in config.js

```
config.debugInfo = false;
```

User can configure default beeper volume. The value can be from 0 to 4.

```
/**
* @description  default beeper volume value - can be from 0 to 4
* @type {Number}
*/
config.defaultBeeperVolume = 4;
```

CHAPTER 5 LOCALIZATION

The SB1 Shell supports one language for the current instance of the SB1 Shell. Only client applications (including an authentication application) can change the language of the SB1 Shell by changing the locale using asl.profile or asl.auth. Once the locale is changed the SB1 Shell restarts RhoElements and loads the new resource files.

In order to change the default locale of the SB1 Shell, change the **config.js** file:

```
config.user.data = {
   name: "Badge",
   title: "User Config",
   pin: "0000",
   dept: '',
   photo: '',
   locale: 'en-US',
   tegroup: ''
};
```

```
asl.profile(data);
```

Parameters

- data (object)

  An object containing a set of user properties from the user profile object that will be configured by the profiling callee.

Client applications can also change dynamically the locale of the SB1 Shell by using asl.profile call:

```
asl.profile({locale:'en-US', name: "George"});
```

String Resource Files

Programmers can provide a resource file in the */UserDrive/config* folder. To use this resource file it has to be named in a way that the SB1 Shell understands it. For example, for a locale en-US, the resource file should be named **`resources-en-US.js`**. Once the file is placed in the */UserDrive/config* folder and the above configuration is made, the SB1 Shell loads the new settings.

Example

```
resources.ConfirmCancel = "Cancel";
resources.ConfirmOK = "OK";
resources.AlertOK = "OK";
...
resources.NotificationsEmpty = "Empty";
resources.NotificationsTitle = "Notifications";
```

Keyboard Resource Files

Similar to the resource files, programmers can provision a custom keyboard file in the */UserDrive/config* folder. If the locale is set with the required string, the keyboard **`kbd-en-US.js`** file is loaded according to the locale.

Example

```
config.kbd.keyboards['en-US'] = {
   name: "English",
   delay: 1000,
   alpha: [
      {
         keys: [
            {
               label: "abc",
               actions: ["a", "b", "c"]
            },
...
```

Client Application Localization

Client applications may also use localization but they must implement their own localization strategies. What the SB1 Shell provides when it starts an application is to give them information about the locale of the SB1 Shell. This is done by using the asl.prv.locale property of the **asl**. This property is ready to use after **asl** onLoad event of the client application.

CHAPTER 6 ADDITIONAL NOTES

The following notes are recommendations and best practices for using the SB1 Shell.

- Please refrain from using the JQueryMobile library when developing SB1 applications as it can use a significant amount of memory at run time, which could lead to low memory issues on the SB1

NOTE Developers should reference the internal shell ASL.

- Integration of the **asl** library in all pages of the client application prevents unexpected behavior of the SB1 Shell.
- All the application pages must refer to **`asl.js`** from the SB1 location as a first line in head tag.

Example

```
<head>
<script type="text/javascript"
src="http://127.0.0.1:83/Application/www/sapp/src/asl.js"></script>
</head>
```

- It is recommended to call the asl.back, asl.title and asl.options functions on every page as they are persistent across page loads and may cause unexpected behavior of a client application. For example, if you set the title on one page and do not set the title on the next page, the next page uses the previously set title.
- Always clear application data (sessionStorage, localStorage, cookies, databases, caches when application onExit/onKill or onCradleInsert events). This prevents unexpected behavior of an application.
- It is recommended that all JavaScript calls in a page should be done in asl.events.types.loaded event subscription. This ensures application developers that the page is loaded and the communication between SB1 Shell and the client application has already been setup. . It is recommended that use asl.events.types.loaded event subscription instead of body onload or DOMContentLoaded.
- Careful consideration of the scope of application variables. Some of the callback functions that application developers can define might be executed in a completely different scope than the one that is expected. This prevents applications and callbacks from crashing.
- Logging of application specific events or errors in RhoElements log is important for future debugging of the applications.

- For production purposes only the use of the custom alert and confirm boxes is recommended. Sometimes stopping the thread from the native alert/confirm boxes may have unpredictable consequences.
- Some of the NPAPI modules have limitations within the multi-instance environment as the modules were not originally designed for multi-application use. One of which is the "push" module where most of its parameters are only modifiable through the **`config.js`** file. Refer to the documentation for more information.
- Resource management within client applications like js, css or image files - cleanup resources from pages that are not used in the current page. This increases the speed of the application load.
- Use validation tools to validate your HTML, JavaScript or CSS. Examples can be seen on:
  - http://validator.w3.org
  - http://www.jslint.com
- Compress and consolidate source files to decrease the size of the code and the number of requests to the server.

CHAPTER 7 APP DEVELOPMENT AND DEPLOYMENT GUIDELINES

All applications developed for the SB1 are developed for the SB1 Shell. The SB1 Shell environment imposes a few mandatory requirements which must be followed to achieve an acceptable result. This chapter presents some additional guidelines that provide the following significant benefits:

- Allows SB1 applications to be developed in parallel by multiple developers while providing a degree of confidence that such applications coexists in the same SB1.
- Allows a single common development and testing cycle to be used by all developers.
- Allows SB1 applications that are developed and tested to be easily packaged using the MSP Package Builder templates for the SB1 so that these packages can be deployed individually or in groups to SB1s using MSP Staging.

Application Naming

It is important that each application be given a unique application name. This ensures that different SB1 applications can be clearly distinguished from each other and that there is no confusion if multiple SB1 applications are deployed to the same SB1. The unique application name is used for a variety of purposes, including file naming, folder naming, etc. To ensure that the unique application name can be used universally for all such purposes, the unique application name should follow the rules for workstation and SB1 file names (minus the file extension).

Valid unique application names:

- MyApp
- myRhoApp
- MyCompany.MyApp.SB1

Invalid unique application names:

- MyApp\Test
- myRhoApp*Working?
- MyCompany>MyApp:SB1

The critical need is that the unique application name must ensure that two distinct applications never have identical names. An additional important consideration might also be to ensure that the purpose, owner, etc. of an SB1 application can be readily determined from its unique application name.

Content File Naming

Since SB1 applications vary in terms of complexity and content, it likely is not practical to impose highly rigid guidelines on how the content files that make up an application are named and organized. For example, it may not make sense to dictate how an SB1 application that consists of multiple linked .HTML files should name subsidiary .HTML files. Similarly, if an SB1 application has other subsidiary files, such as image files, JavaScript files, style sheets, etc., it may not make sense to dictate whether or not these files should be placed into subfolders.

Without some sort of guidelines, it could be quite difficult to ensure that independently developed applications can coexist peacefully. At a minimum, it is recommended that at least the main .HTML file for an SB1 application be named based on the unique application name. For example, if the unique application name for an SB1 application is "myapp", then using a main .HTML file name of "myapp.HTML" would be recommended. Any subsidiary files that are referenced from that main .HTML file could be named in whatever manner best suits the needs of the developer.

Content File Location in SB1

It is important that the content files that make up an SB1 application be located under a unique folder in the SB1. While this may not always be required, it is nonetheless highly recommended. The developer of one SB1 application cannot be certain that the developers of other applications will follow any specific conventions regarding the naming of content files. Consequently, if two applications were allowed to place their content files into the same folder, they might have certain file names in common. This could cause one SB1 application to replace content files of another and would produce unpredictable results, depending on the order in which the SB1 applications were deployed to a given device.

It is recommended that all content files for a given SB1 application be located in a unique subfolder on the SB1 that is dedicated for the use of that application. In addition, to make it easy to locate the content for an SB1 application, the subfolder for that SB1 application should be named based on the unique application name and should be placed under the \UserDrive\apps folder in the SB1. If appropriate, additional subfolders can also be created to further separate and organize the content files for an SB1 application, so long as those subfolders are located under the subfolder dedicated for use by that SB1 application.

For example, consider that three SB1 applications, with the unique application names of "myapp1", "myapp2", and "myapp3" are deployed to the same SB1. The recommended organization for the content files associated with these SB1 applications would be:

```
\UserDrive\
   apps\
      myapp1\
         myapp1.HTML
      myapp2\
         myapp2.HTML
      myapp3
         myapp3.HTML
   Config\
      Apps.json
      Config.js
      kbd-en-US.js
      Resources-en-US.js
```

Content File Location in Development Workstation

While it is not absolutely necessary that the location of the content files that make up an SB1 application be organized on the developer's workstation in the same manner that they are in the SB1, it is nonetheless recommended to do so. Further, to simplify the eventual packaging of an SB1 application, it is recommended to replicate on the workstation the entire folder structure in which the content files for an SB1 application will be stored on the SB1.

Following from the prior example, let's assume that a single developer is working on all three of the applications on the same workstation, but wants to ensure that the applications can be used freely, alone in combinations, on any given SB1. Further, let's assume that the developer wants to keep all his SB1 work under the folder D:\MyProjects\SB1. The recommended organization for the content files associated with these applications would be:

```
D:\MyProjects\SB1\
   myapp1\
      UserDrive\
         apps\
            myapp1\
               myapp1.HTML
   myapp2\
      UserDrive\
         apps\
            myapp2\
               myapp2.HTML
   myapp3\
      UserDrive\
         apps\
            myapp3
               myapp3.HTML
```

While the above organization might seem somewhat redundant, the reason for it will later become clear as it will simplify the packaging of the SB1 applications later and will also facilitate keeping the SB1 applications separate on a given Workstation.

Applications Versions

Inevitably, SB1 applications undergo modifications over time, resulting in a need to keep track of multiple distinct versions of one or more SB1 application. While many different schemes could be used to separate and identify multiple versions of a given SB1 application, certain approaches should be avoided since they would conflict with the goals of these guidelines. In particular, while it might at first seem like a good idea, it is likely unwise to include the application version as part of the unique application name. Such an approach would likely make it more difficult to manage application upgrades and downgrades on the SB1 and could result in inadvertently having multiple versions of the same application deployed to a single SB1, which likely is not a desirable outcome.

Instead, multiple versions of the same SB1 application should all have the same unique application name but should be distinguished from each other by an application version that is unique within the context of that SB1 application and meaningful to the developer and users of that application. All uses of the unique application name described above to label files and folders should be based on the unique application name without the application Version. The one exception is the top level organization for a given SB1 application on the Workstation, which would need to include the application Version if multiple versions of the same SB1 application will reside on the Workstation at the same time.

To continue with the prior examples, let's assume that we have two versions of each of three SB1 applications, and that the versions of these SB1 applications are identified by application versions “v1” and “v2”. Then, the recommended organization for the content files associated with these SB1 applications might be:

```
D:\MyProjects\SB1\
    myapp1-v1\
        UserDrive\
            apps\
                myapp1\
                    myapp1.HTML
    myapp1-v2\
        UserDrive\
            apps\
                myapp1\
                    myapp1.HTML
    myapp2-v1\
        UserDrive\
            apps\
                myapp2\
                    myapp2.HTML
    myapp2-v2\
        UserDrive\
            apps\
                myapp2\
                    myapp2.HTML
    myapp3-v1\
        UserDrive\
            apps\
                myapp3
                    myapp3.HTML
    myapp3-v2\
        UserDrive\
            apps\
                myapp3
                    myapp3.HTML
```

The above organization includes the application version with the application name as part of the folder name in the part of the folder structure that is not stored in the SB1. This ensures that the same folders in the SB1 are used for every version of a given SB1 application.

Or, alternately, another possible organization for the content files associated with those same SB1 applications might be:

```
D:\MyProjects\SB1\
   myapp1\
      v1\
         UserDrive\
            apps\
               myapp1\
                  myapp1.HTML
      v2\
         UserDrive\
            apps\
               myapp1\
                  myapp1.HTML
   myapp2\
      v1\
         UserDrive\
            apps\
               myapp2\
                  myapp2.HTML
      v2\
         UserDrive\
            apps\
               myapp2\
                  myapp2.HTML
   myapp3\
      v1\
         UserDrive\
            apps\
               myapp3\
                  myapp3.HTML
      v2\
         UserDrive\
            apps\
               myapp3\
                  myapp3.HTML
```

The above organization uses the application version to introduce an extra level of the folder structure in the part of the folder structure that is not stored in the SB1. This ensures that the same folders in the SB1 will be used for every version of a given SB1 application.

Memory Considerations

Devices supported by RhoElements span the entire range of the performance spectrum and therefore care should be taken when developing applications. Know the capabilities of the device when developing applications. Consider the following:

- JavaScript libraries such as **Sencha touch** or **JQuery Mobile** can use a significant amount of memory at runtime. The more JavaScript libraries loaded into the DOM the greater the RAM footprint of the web page.
- There are APIs available in the device to monitor the memory, including memory logs and a Memory API. Use these tools to get an understanding of the requirements for the application.
- Resources (including blogs and webinars) are available on *developer.motorolasolutions.com* to help create great looking, streamlined applications.
- Online performance tests for JavaScript and CSS, particularly those involving DOM manipulation will often be written to target desktop computers and may not run on all supported devices.
- Minimize application icon and image sizes. This saves network bandwidth and memory.
- Include only required JS and CSS files on a page to minimize its memory footprint and avoid unnecessarily wasting memory.
- Consider separating the JS and/or CSS files into multiple files.

SB1 Baseline Files and Baseline State

Certain files must be present in the *UserDrive* on the SB1 in order for SB1 applications stored on the *UserDrive* to be included in the behavior of the SB1 Shell. These files are generally independent of whatever applications are to be used and are common and shared by all applications. Because these files represent the base onto which applications can then be added, they are referred to as baseline files. The state that an SB1 is in, when the baseline files are present is called the SB1 baseline state. You can think of the SB1 baseline state as the application environment that must be present in order for SB1 applications to function.

Baseline files include configuration files that control the operation of the SB1 Shell and that define the applications that display on the Application screen. As a result, the baseline files typically vary, to some degree, from customer to customer. Some customers might want to have more than one set of baseline files, if they intend to deploy SB1s for multiple highly-specialized purposes.

The most important baseline file is the `config.js` file. If a valid `config.js` file is not present in the *\UserDrive\config* folder of the SB1 at boot, then all contents of the *UserDrive* is ignored by the SB1 Shell. In such a case, the SB1 Shell behaves according to the `config.js` file that is built into the SB1. This behavior is sometimes referred to as the default or fresh out of the box behavior of the SB1. For more information on the `config.js` file, see *config.js File on page 2-3*.

When a `config.js` file is initially placed into the *\UserDrive\config* folder of the SB1, and each time it is modified, a cold boot must be performed before the SB1 Shell begins using the new file. Whenever a bundle (used during staging and / or during management by MSP) is deployed to an SB1 to update its contents, a cold boot is automatically initiated at the end of the bundle. This ensures that any changes made by the bundle are activated for use by the SB1 Shell once the deployment of that bundle is completed.

Another important baseline file is the `apps.json` file. The SB1 Shell determines the set of application icons that are shown in the **Application** screen based on the content of an `apps.json` file. The location of the `apps.json` file that is used by the SB1 Shell is determined by the `config.js` file that is being used to control the behavior of the SB1 Shell. When a `config.js` file is placed into the *\UserDrive\config* folder of the SB1, it

should specify the location of the **apps.json** file that should be used. In most cases, the **apps.json** file specified by the **config.js** file also resides in the *\UserDrive\config* folder of the SB1.

A set of baseline files might include an empty **apps.json** file or might include a non-empty **apps.json** file that specifies a set of applications that are considered part of the baseline state of the SB1. If an empty **apps.json** file is used, it should actually contain a single empty list element and the set of baseline files need not include any application content files. If a non-empty **apps.json** file is used, then it should specify the desired set of applications and the set of baseline files would also need to include any application content files required by the specified applications. For more information on the format of an **apps.json** file, see *apps.json File on page 2-3*.

The following is an example of an empty **apps.json** file:

```
{
}
```

The following is an example of a non-empty **apps.json** file:

```
{
'name': 'App1',
'url': '/UserDrive/apps/myapp1/myapp1.HTML'
},
{
'name': 'App2',
'url': '/UserDrive/apps/myapp2/myapp2.HTML'
}
```

In most cases, the set of baseline files for an SB1 also includes additional files that are referenced, directly or indirectly, by the **config.js** file. These might include resource files, style sheets, image files, etc. Anything that is common and required to be part of the application environment for the SB1, as opposed to being part of an optional application that might or might not be added to that SB1, should be made part of the baseline state for that SB1.

Creating an SB1 Baseline

It is generally easier to start with a known working baseline state and customize it by changing the things you do not like than it would be to start from scratch and try to identify everything you might need. The sample ***Sb1SampleBaseline*** package has been provided as part of the MSP 4.2 Supplement for SB1. This sample package can be used as a starting point for defining a SB1 baseline state and it can then be used to create a SB1 Baseline Package to place any number of SB1s into that baseline state.

The ***Sb1SampleBaseline*** package is provided in a form that is ready-to-use. In most cases, however, it is likely that you will need or want to modify this sample to create your own custom baseline state. The best way to do that is to understand the application environment established by the sample baseline package and determine any ways in which it does not meet your needs. This generally involves the following parallel steps:

- Deploy the ***Sb1SampleBaseline*** package to an SB1 so you can experience the application environment that it establishes. This can be done by following the instructions for Deploying an SB1 Baseline Package to an SB1. Once you have an SB1 configured accordingly, explore the behavior of the SB1 to see how it does or does not meet your needs.

- Using the **Tools** > **Convert to Project** menu option of the MSP Package Builder, extract the contents of the ***Sb1SampleBaseline*** package onto a workstation. Exploring the extracted content, especially the `config.js` file, should help understand how the ***Sb1SampleBaseline*** package produces the behavior that it does and how that behavior might be modified by modifying selected content files.

If any changes in behavior are needed or wanted, the relevant extracted content files can be modified. Once all desired changes have been made, the Project created when the files were extracted can be used to create a new package with the new modified content. This new package can then be deployed to change the SB1 into the new modified Baseline State. This process can be repeated as many times as needed until the desired Baseline State is achieved.

Building an SB1 Baseline Package

Once you have a suitable *UserDrive* folder containing the desired baseline files on the workstation, you can build an SB1 Baseline package to deploy those baseline files to SB1s via the following process:

1. Launch the MSP Package Builder.
2. Open the Package Project (.MSPPROJ) file that was created when the ***Sb1SampleBaseline*** package contents were extracted.
3. Select the Files section of the Project and delete the entire *UserDrive* folder.
4. Drag the *UserDrive* folder containing the modified baseline files on the workstation into the **Client File System** pane of the project.
5. Select **File** > **Save** to save the modified Package Project (.MSPPROJ) file.
6. Select **Tools** > **Generate APF File** to make a package file containing the modified content.
7. Enter a version for the package to be created.

Deploying an SB1 Baseline Package

There are a variety of circumstances under which you might want to deploy an SB1 Baseline Package to an SB1. All are situations where you want to place the SB1 into the baseline state defined by that package. Some of the possible situations are:

- When an SB1 is in fresh out of the box state and needs to be prepared for production use by placing it into the baseline state defined by the package.
- When an SB1 is in fresh out of the box state and needs to be prepared for use in application development and testing by placing it into the baseline state defined by the package.
- When an SB1 is returned to production use and needs to be placed into the baseline state defined by the package.
- When an SB1 is returned to production use after it has been used for application development and testing and needs placed into the baseline state defined by the package.

Note that in some of the above cases, the SB1 is assumed to be in fresh out of the box state and in other cases the current state of the SB1 may or may not be known. Since the *UserDrive* on an SB1 is empty in fresh out of the box state, it would be possible to simplify the process of deploying an SB1 Baseline Package if it was known for certain that the SB1 was in the fresh out of the box state. But since it is generally necessary to deal with SB1s that may be in any state, a single more generic process, is generally simpler in the long run.

When the state of a SB1 is not known, we cannot know what content may be present in *UserDrive*. If we simply installed an SB1 Baseline Package, the new content being placed into the *UserDrive* would simply be merged into any old content that was already present in the *UserDrive*. Depending on the prior content, the end result

might or might not be what was desired. To ensure that the new content specified in the SB1 Baseline Package and only that new content, will be in the *UserDrive*, it is necessary to wipe the current contents of the *UserDrive* before deploying the new contents.

The SB1 *UserDrive* can be wiped by using the special SB1-specific ***WipeUserDrive*** package. When a bundle is created to install an SB1 Baseline Package, include an extra install package step in the same bundle to install the ***WipeUserDrive*** package just before the Install Package Step that installs the SB1 Baseline Package. When that bundle is deployed to an SB1, this ensures that when the SB1 Baseline Package is installed, it adds content to a freshly wiped *UserDrive*. This is safe even for SB1s that are in fresh out of the box state since it is benign to wipe a *UserDrive* that is already empty.

When creating a bundle to deploy an SB1 Baseline Package, you may want to install other packages. For example, you might want to install the ***SB1SaveCalibration*** package or the ***UpdateInProgress*** and ***EndUpdateInProgress*** packages. For more information on the use of those packages, refer to the *SB1 Integrator Guide*. When including an Install Package Step into a bundle to install the ***WipeUserDrive*** package, it is generally recommended to set the "Force Install" flag to "True". This ensures that the *UserDrive* folder is always wiped even if it has previously been wiped using the ***WipeUserDrive*** package.

Removing an SB1 Baseline Package

An SB1 Baseline package can be removed by placing an Uninstall Package Step into a bundle and deploying that bundle to the SB1. This could be a bundle that does nothing but uninstall the SB1 Baseline Package or a bundle that is also installing or uninstalling other packages. However, it is generally not recommended to remove an SB1 Baseline Package while SB1 application packages remain installed. This would likely render those applications non-functional since they depend on the Baseline State established by the SB1 Baseline package. The end result would likely be to cause the SB1 Shell to exhibit its default (out of the box) behavior since it no longer see the `config.js` file that was deployed by the SB1 Baseline Package. This would be a confusing situation since the *UserDrive* folder would not be empty, yet applications deployed to the *UserDrive* folder would be ignored.

The best approach is to first remove all SB1 application packages, then remove the SB1 Baseline Package. To prepare the SB1 for use again, you would again proceed as defined in *Deploying an SB1 Baseline Package on page 7-9*.

Developing and Testing an SB1 Application

Since application packages to deploy SB1 applications are expected to be deployed as add-ons to an SB1 that is in baseline state, the best approach is to place an SB1 into the baseline state in which the application is expected to run. Since a given SB1 application might need to run in SB1s that have different baseline states, it might be prudent to, at some point, test the application on SB1s that are in all relevant baseline states. If the various sets of SB1 baseline files are created properly, in accordance with the above guidelines, this should mostly be a formality to catch any mistakes. And at least initially, an application can likely be tested on any SB1 that is in any convenient baseline state.

The most efficient way of performing SB1 application development and testing is likely using an SB1 that is equipped with the optional Developer Back Housing. A developer can connect the SB1 to a workstation using a micro USB cable and access the *UserDrive* of the SB1 from the workstation. This permits the files associated with an SB1 application to be rapidly copied to the SB1 then tested.

While an SB1 is connected to the workstation, the *UserDrive* of the SB1 is dismounted and is completely inaccessible by all software running on the SB1. While in this state, the *UserDrive* of the SB1 appears as a drive on the workstation and the developer can copy files and folders to or from the SB1, delete files or folders from the SB1, etc. No attempt should be made to format the *UserDrive* or perform error correction on the *UserDrive* from the workstation as this could cause it to become damaged and could leave it unusable thereafter by all software running on the SB1.

When an SB1 is disconnected from the workstation, access to the *UserDrive* by the workstation is lost and the *UserDrive* of the SB1 is remounted and becomes accessible by all software running on the SB1. Applications that were running at the time the SB1 was connected to the workstation might or might not gracefully handle the loss of accessibility and later restoration of accessibility of the *UserDrive*. Also, the SB1 Shell does not begin to utilize content that was changed in the *UserDrive* until a cold boot is performed.

As described previously, the SB1 Shell determines the set of application icons that is shown in the Application screen based on the content of an **apps.json** file. When the goal is to test a single SB1 application, the easiest thing to do is to create a simple **apps.json** file that only launches the SB1 application being developed and tested. This **apps.json** file would contain a single section that describes the SB1 application being developed and tested. For more information on the format of an **apps.json** file, see *apps.json File on page 2-3*.

For example, an **apps.json** file to launch an SB1 application called **myapp1** that was developed according to the guidelines described above might look like:

```
{
“name”: “My First App”,
“url”: “/UserDrive/apps/myapp1/myapp1.HTML”
}
```

If the above **apps.json** file was copied into the *\UserDrive\config* folder of an SB1 that was in a suitable baseline state, then once a cold boot was performed on the SB1, the SB1 Shell Application screen would show a single application icon that, when selected, would attempt to launch the SB1 application **myapp1**. If the content files for that SB1 application was not part of the baseline state, then the application icon would not be able to launch the SB1 application because the content files for the application would not be present in the location identified by the “url” tag in the **apps.json** file.

To make this work as expected, the content files for the SB1 application would also need to be copied into the appropriate location in the SB1 before the cold boot. If the guidelines related to folder organization on the workstation were followed, then the *apps* folder for the SB1 application would be simply need to be copied from the workstation to the SB1. If the SB1 already contains an *apps* folder, the new content of the *apps* folder would be merged with the existing content. If the SB1 does already contain an *apps* folder, then the *apps* folder would be created and the appropriate content would be added. Having an *apps* folder for every SB1 application makes it easy to add or update the content of the SB1 application to an SB1 during development and testing.

Building an SB1 Application Package

Once an SB1 application has been successfully developed and tested it is generally desirable to create an SB1 application package to allow the application to easily be deployed to any SB1 that is in a suitable baseline state. Before using the MSP Package Builder to create such a package, one additional step must be performed.

As discussed earlier in the section Developing and Testing an SB1 application, an **apps.json** file was created and placed into the *\UserDrive\config* folder on the SB1 in order to direct the SB1 Shell to present an application icon in the **Application** screen. It was recommended to create a simple **apps.json** file containing just one section for the SB1 application. In addition to the reasons previously given for creating that simple **apps.json** file, it is even more useful when building an SB1 application package for the SB1 application.

The simple **apps.json** file for an SB1 application should be placed into the same folder on the workstation that contains the main.HTML file for that SB1 application. For example, let's consider the SB1 application “myapp1” from the previous examples. The **apps.json** file would be renamed to “myapp1.APP” and would be placed with the main .HTML file for the application, resulting in a workstation folder structure as shown below:

```
D:\MyProjects\SB1\
   myapp1\
      v1\
         UserDrive\
            apps\
               myapp1\
                  myapp1.HTML
                  myapp1.APP
```

When the .APP file is added to the package, the Rho application Template automatically detects, based on the file extension of .APP, that it is a file containing a single apps.json section for the SB1 application being added to the package.

When the an SB1 application package is installed on an SB1, the .APP file is processed and the single section it contains is added to the end of the existing (empty or non-empty) **`apps.json`** file contained within the *\UserDrive\config* folder on the SB1. The result of an install of the SB1 application package will thus be to both deploy the content files for the SB1 application and to extend the **`apps.json`** file to add an application icon for that SB1 application to the SB1 Shell **Application** screen.

If an SB1 application package is later uninstalled from the SB1, the .APP file is processed again and the single section it contains is located within existing **`apps.json`** file contained within the *\UserDrive\config* folder on the SB1 and is removed (leaving all other SB1 applications alone). The result of an uninstall of the SB1 application package will thus be to both remove the content files for the SB1 application and to reduce the **`apps.json`** file to the list element that added the application icon for that SB1 application from the SB1 Shell **Application** screen.

As you can see, that is a lot of free functionality that is gained simply by including an appropriate .APP file into the SB1 application package.

In some cases, it is desirable to have the application Icon for an SB1 application added to the front of the list of application icons instead of to the end of the list. This can be accomplished by using a .APP1 file instead of a .APP file. Simply using a file extension of .APP1 for the file instead of a file extension of .APP allows the Rho application Template to alter the behavior of the Package to place the application Icon at the front of the list instead of the end of the list.

To create an SB1 application package:

1. Launch the MSP Package Builder.
2. Select **File** > **New Project**.
3. Select the **RhoApplication** template.
4. Enter a name for the package (this should usually be based on the unique name of the application).
5. Select a location for the package (this is usually the workstation folder containing the *UserDrive* folder for the application).
6. Select the **Files** section of the Project.
7. Drag the *UserDrive* folder for the application on the workstation into the **Client File System** pane of the project.
8. Select **Tools** > **Generate APF File** to generate a package file containing the modified content.
9. Enter a version for the package to be created.

One key point to note is that step 7 takes advantage of the fact that a *UserDrive* folder exists on the workstation for each SB1 application. Having a complete *UserDrive* folder on the workstation allows all the content files needed for an SB1 application, including the .APP file, to be added to the package project by dragging and dropping one workstation folder.

Deploying an SB1 Application Package

An SB1 application package is intended to be installed to an SB1 that has been previously placed into a baseline state by installing an SB1 Baseline Package. Any number of SB1 application packages can be installed, in any order, to an SB1, so long as the SB1 Baseline Package is installed first. Additionally, a previously installed SB1 application package can be uninstalled at any time or optionally replaced by a different version of SB1 application package of the same package name.

When an SB1 application package is installed, the `apps.json` file in the *\UserDrive\config* folder is automatically modified to add the section for the SB1 application being installed to the end of the existing file. This causes the application icon for the new SB1 application to be added to the end of the list of applications previously specified in the `apps.json` file. Since the application icon for an SB1 application is added at the end of the list when an SB1 application package is installed, it is recommended to install SB1 application packages to an SB1 in the order you want the application icons to appear in the SB1 Shell **Application** screen.

To deploy an SB1 application to an SB1, simply include an Install Package Step for the SB1 application package for the SB1 application into a bundle that is deployed to the SB1. Since all SB1 applications are expected to be added to an SB1 that is in a suitable baseline state, an SB1 application package should only be installed an SB1 on which an SB1 baseline package is already installed. The SB1 Baseline Package and one or more SB1 applications packages could be installed to an SB1 in the same bundle, so long as the Install Package Step that installs the SB1 Baseline Package precedes the Install Package Steps that install any SB1 applications packages. SB1 applications packages could also be installed via separate bundles, so long as those bundles are not deployed until after the bundle that deploys the SB1 Baseline Package.

If some version of an SB1 application package is already installed on an SB1 and a different version of SB1 Baseline Package with the same package name is installed, then the previous version of the SB1 application package is first uninstalled and then the new version is installed. The uninstall of the previous version removes the SB1 application as described in the section below on Removing an SB1 application package from an SB1. The install of the new version proceeds as if the previous version were never present. The result replaces the prior version of the SB1 application with the new version, and to move the application icon for the SB1 application to the end of the list of application icons presented in the SB1 Shell **Application** screen.

A cold boot must be performed after any change is made to the *UserDrive* folder before the modified content determines the behavior of the SB1 Shell. A cold boot is automatically initiated at the end of every bundle deployed to an SB1. This ensures that any changes made by the bundle are activated for use by the SB1 Shell once the deployment of that bundle is completed.

When creating a bundle to deploy an SB1 application package, you may want to install other packages as well. For example, you might want to install the ***SB1SaveCalibration*** package or the ***UpdateInProgress*** and ***EndUpdateInProgress*** packages. For more information on the use of those Packages, see [TBD].

Removing an SB1 Application Package

An SB1 application that has been installed to an SB1 using an SB1 application package can be removed from the SB1 by uninstalling the SB1 application package from the SB1. An SB1 application package that is installed on an SB1 can be removed from the SB1 by placing an Uninstall Package Step into a bundle and deploying that bundle to the SB1. This could be a bundle that does nothing but uninstall the SB1 application package or any other bundle that is installing or uninstalling any other packages. However, it is recommended to remove all SB1 application packages from an SB1 before the SB1 Baseline Package is uninstalled. This prevents the baseline state on which applications depend from being removed while the SB1 applications are still installed.

When an SB1 application package is uninstalled, the **apps.json** file in *\UserDrive\config* is automatically modified the list element that added the application icon for the SB1 application being uninstalled from wherever it appeared in the existing file. This causes the application icon for the new SB1 application to be removed from the list of SB1 applications previously specified in the **apps.json** file. Since the application icon for an SB1 application is removed from wherever it might be in the list, SB1 application packages can be uninstalled in any order and need not be uninstalled in the reverse order in which they were installed.

As previously mentioned, a cold boot must be performed after any change is made to the *UserDrive* before the modified content determines the behavior of the SB1 Shell. A cold boot is automatically initiated at the end of every bundle deployed to an SB1. This ensures that any changes made by the bundle are activated for use by the SB1 Shell once the deployment of that bundle is completed.

CHAPTER 8 USING FONTS ON THE SB1

Introduction

This chapter describes various approaches to using non-standard fonts in RhoElements on the SB1. This allows the use of glyphs other than those found in the pre-installed fonts.

RhoElements Font System

The current version of RhoElements supports TrueType fonts (.ttf). It is necessary to locate an appropriate TrueType font for the SB1 in order to display Asian characters.

Utilizing Current Fonts

CSS gives the ability to set the font used in different HTML elements, classes or ids. If the TrueType font is in the correct font directory, then it can be referenced by name in CSS. The correct way to set a font in CSS is:

```
body
{
   font-family: 'Droid Sans', sans;
}
```

This command sets the default font family for the “body” element to be the “Droid Sans” font. If it cannot find this font, then it uses the default sans serif font. As the body element should enclose the entire visual mark-up within the HTML file, this means that the default font family for all the elements will be “Droid Sans”. If an element within the body element defines a different font-family to be used, then the default font will be changed for that element alone.

Utilizing New Fonts

There are two ways to utilize fonts currently not installed in the system, either by installing the fonts into the OS directory or by using the CSS3 feature: "@font-face." This section details instructions on how these approaches can be used together with where the font files should be stored to maximize the usability of the SB1.

Installing A New Font To a Local File System

To use a local font in RhoElements by the methods outlined above, the following steps need to be followed. Note that only .ttf extension fonts are supported:

1. Download the selected language font onto a desktop machine.
2. Create a .cpy file that copies the font file from the *\Userdrive\fonts* folder to the *\Windows* folder.

   \UserDrive\fonts\korean\gl_ce.ttf > *\Windows\gl_ce.ttf*
3. Deploy the fonts folder and the .cpy file onto the *\UserDrive* folder on the SB1. The .cpy file must be in the root of the *\UserDrive* folder using the Developer Back Housing (refer to the SB1 Integrator Guide for more information) or MSP.
4. Reboot the SB1. The language font is now available for use by SB1 web applications by using the method defined in *Utilizing Current Fonts on page 8-1*.

Font-Face/Webfonts

As part of the CSS3 specification, the @font-face feature is available in RhoElements and thus on the SB1. This allows the WebKit engine to render text with font files that can be stored remotely, or from a directory that is not the default OS font folder. This feature is commonly called WebFonts.

Font-face is used by adding a section to a web page's CSS file/section: An example loading a font from a local location:

```
@font-face
{
   font-family: 'Droid Sans';
   src: url('file:///UserDrive/Fonts/DroidSans.ttf');
}
```

Another example loading the fonts from a remote location:

```
@font-face
{
   font-family: 'Droid';
   src: url('http://www.example.com/DroidSans.ttf');
}
```

These examples add the Droid Sans font, enabling it to be used as in the section above.

For more information and options on font-face see the W3C's page on CSS3 fonts.

Transferring Fonts

Fonts can be transferred to the SB1 by the same methods as transferring the applications or config files. It is recommended to keep the font files on the SB1, as this increases the rendering speed of the application.

Fonts can be loaded from a remote location, but if the SB1s are in a partially connected environment this may cause display issues due to being unable to download the font files.

Font Licenses

Most fonts are associated with a license. Check the license of each font that is used in the application to make sure that it complies with the terms of the license agreement.

The fonts bundled with versions of Windows are licensed for that particular instance of Windows. You are able to use these fonts only on the device on which they were installed. If you want to use these fonts on devices that do not already have licenses for these fonts, new licenses are required to be purchased.

Other fonts, such as most of Google's Droid font family are licensed under the Apache License. This license is a free software license which allows the user of the software the freedom to use the software for any purpose, to distribute it, to modify it, and to distribute modified versions of the software, under the terms of the license along with some other terms.

A lot of fonts created by font foundries have commercial licenses. These must be purchased to legally use their font in your application. They often have restrictions which limit the use to one or a few devices.

There are many font foundries, shops and repositories which can supply fonts that can be used in web applications. Some examples are:

- www.ascenderfonts.com
- www.google.com/webfonts
- www.fontshop.com

CHAPTER 9 RUNNING OFF-LINE WEB APPLICATIONS

Introduction

Off-line web applications are web sites that continue to operate correctly without a network connection to their server. Making a web application available off-line allows the user to continue using the web application if the network connection is interrupted or if the user goes out of network range.

By utilizing new HTML5 standards, enabling current web applications to be used off-line or creating new applications has been made a simple task.

How to Deploy a Web Application

Making a web application ready for off-line usage can be as simple as writing a new file as part of your web site and adding an attribute to each of your web pages. This new file is called the “Cache Manifest” and it works as a download instruction list detailing all of the resources that make up the web application that should be downloaded and those that should not. The new attribute links each HTML page to the cache manifest, so that if the user visits any page of the application, an off-line version is always downloaded.

For more complex web applications, a strategy needs to be devised and put in place to handle data that needs to be sent back to a central server. With RhoElements’ Network Module combined with WebSQL, an application can be programmed to determine whether it has connection to its server, locally store gathered data in an SQLite database for later submission if connection is down, and submit the stored data to the server when connection is re-established.

Enabling the Cache Manifest in HTML

To enable the cache manifest and thus off-line applications in your HTML files a “manifest” attribute needs to be added to the <html> tag.

The manifest attribute informs the web browser where the cache manifest file is located, allowing the browser to start caching the web application.

The manifest attribute should be added to every HTML page within the web application.

Example

```
<!DOCTYPE html>
<html manifest="/cache.manifest">
...
```

Cache Manifest

The cache manifest is a file stored on the server that is referenced by each HTML page of the application. It contains an inventory of the files that make up the application together with information on whether it should be cached or not. Each off-line capable web application should have only one cache manifest.

Structure

The structure of the cache manifest file constitutes a header and three lists or file locations: the CACHE list, the NETWORK list and the FALLBACK list.

Header

The first line of a cache manifest must be the header:

```
CACHE MANIFEST
```

Cache

The cache section describes the location of all the resources that should be downloaded to enable the application run in off-line mode. Cache list is the default list.

```
CACHE:
index.html
default.emmp
scripts.js
styles.css
```

Network

The network section describes a list of files that should never be cached. This might be because they are dynamic server side scripts, so caching this file would defeat its purpose.

```
NETWORK:
getData.php
```

Fallback

The fallback section specifies which page to navigate to, should a page be visited whilst off-line that has not been cached for some reason (cached incorrectly/part of the NETWORK list/Not specified in the cache manifest).

```
FALLBACK:
/ fail.html
```

Example

This example redirects any non-cached or incorrectly cached files to the fail.html webpage.

```
CACHE MANIFEST
CACHE:
index.html
default.emmp
scripts.js
styles.css

NETWORK:
getData.php

FALLBACK:
/ fail.html
```

Server

For the cache manifest to be handled correctly, the web server that hosts the web application must provide the file with the content type:

```
text/cache-manifest
```

It is also good practice to make sure the cache manifest itself is not cacheable. Otherwise clients may read outdated, cached cache manifests and not receive an up-to-date version of the web application.

Tips

Revising the Cache Manifest

If you change one of the resources listed in the cache manifest, you would want browsers that cache your application to download the new file automatically. To accomplish this, every time a change has been made in the source files of your application, the cache manifest file should be modified to flag to browsers that the application needs to be downloaded again.

Once a browser comes across a cache manifest that is different from the last cache manifest that it downloaded for that application, it will re-download and replace all resources listed in the manifest.

One easy way of ensuring a difference in the cache manifest is by including a revision comment. Every time a file within the web application is changed, the revision number is incremented. For example:

```
CACHE MANIFEST
CACHE:
index.html
default.emmp
scripts.js
styles.css

FALLBACK:
/ fail.html

# Revision 14
```

Off-line Mode

To be able to make a web application useful while its off-line, data logic will need to be changed to manage download and upload data requests sent from the application that would normally be received by a remote server. This section includes some basic instructions on how to check whether the application is online or off-line through JavaScript, and how to synchronize data between a local SQLite database that is built into the browser, and remote servers.

If your online web application communicates with a server to access or send data, by only adding a cache manifest to your web application will cause it to not operate as expected in off-line mode. Changes need to be made in JavaScript to accommodate the difference in connection. For example, if a delivery person has delivered a parcel and needs to send the signature of the recipient to the server in the depot but is out of network range, using the same logic as when online will cause an error. The application needs to store the request until the connection is re-established and then send it.

To do this, the web application needs a reliable means to check whether the browser is online, and a way of storing data locally.

Discerning Connection Status

To be able to run different JavaScript code between off-line and online modes, there needs to be a mechanism to find out the status of the browsers connection.

As part of the HTML5 standard, the “navigator” JavaScript object has a parameter called “onLine”. The purpose of this field is to give JavaScript knowledge of whether the browser has an Internet connection. Unfortunately, no current browser has implemented this feature to reliably report the status of the browsers connection to the Internet. To remove this limitation, we have created an API in RhoElements that provides a way for a web application to access this information: the Network Module.

Network Module

With the Network Module, you are given a way of not only checking whether your device has connection to the Internet in general, but more importantly whether your device can connect to your server.

By using the Network Module, you can write connection status aware JavaScript code that can store data while off-line and synchronize with remote servers when connection is re-established.

For more information about the Network Module, see the RhoElements Help File.

WEBSQL

RhoElements implements the WebSQL specification that was until recently part of the HTML5 Draft. WebSQL provides an SQLite interface for managing client-side databases using SQL. This can be used within RhoElements, accessed via JavaScript, to store local data generated or used by the web application.

WebSQL is supported by many web browsers but was removed from the HTML5 Draft because all implementations used the same backend, SQLite. In order to form an independent standard more than one implementation would need to be used. Future releases of RhoElements will support alternative client based database web storage standards once they are included in the HTML5 specification.

Example of Using WEBSQL in RhoElements

For example, the function of the web application is to provide a delivery person a list of deliveries that they need to make today, combined with delivery recording capabilities that records the recipient’s signature and the time of delivery on the company’s servers.

When the application has connection to the company’s servers, a function in JavaScript can query the central server to download all records of the scheduled deliveries for the current day and application user. These records will be stored in a local SQLite database. When the user goes out of network range, they will still be able to see the list of scheduled deliveries that they need to make today because the application will detect that it is off-line, and return results from the local SQLite database instead of the central database back at the depot.

When the delivery person records a delivery and is out of range, the time details and signature image is saved to the local SQLite database. The next time the application gains network connection, the application synchronizes with the companies central servers, sending all pending information that has been cached in the SQLite database.

A detailed example can be found at http://www.html5rocks.com/en/tutorials/webdatabase/todo/.

More Information

Further information on off-line applications: http://diveintohtml5.org/offline.html

WebSQL specification: http://www.w3.org/TR/webdatabase/

Cache Manifest specification: http://www.w3.org/TR/html5/offline.html#manifests

SB1 smart badge - Best Practices & User Training

Scanning Recommendations

- Scanning with the SB1 smart badge is easy to use. Hold the SB1 in your hand with the top of the device pointed toward a bar code. With your thumb, press the scan button.
- The SB1 projects a red rectangular target for positioning over bar codes.
- Instead of aiming directly at the barcode (i.e. 90 degrees), it is best to tilt the device and scan at a slight angle to minimize reflection.
- Its omni-directional barcode reader allows the user to position this target at any angle - it doesn't need to align horizontally.
- Hold the device in such a manner that will NOT allow the Home Button to be accidentally hit while pressing the Scan trigger. Simultaneous pressing both the Scan Trigger and Home button is reserved for Device Reset and puts the device into Suspend mode.
- Successful decode distance from the bar code is dependent upon the type of bar code. Generally a distance in the range of 3 to 8 inches works best for most bar codes.
- If you have trouble scanning. Place the top of the SB1 against the bar code and pull away until a decode beep is heard.

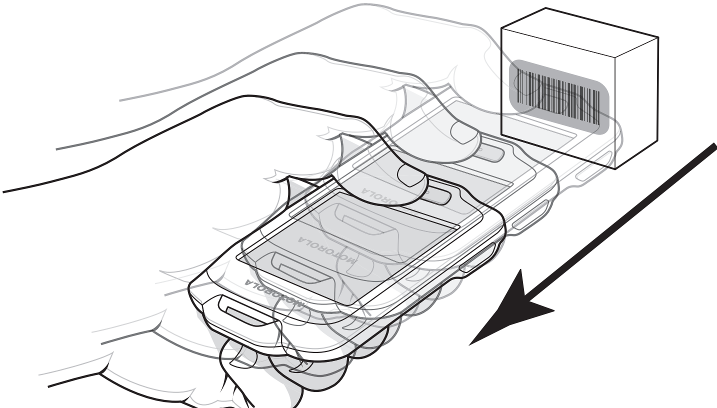


Figure 9-1 *Scanning Bar Codes*

Overall Usability

- To make a selection on the SB1 touch screen, it's sometimes best to use your finger nail vs. your full finger, especially for small selections. This is due to the SB1's highly sensitive resistive touch panel, which also allows the use of gloves.
- Touch the middle of any button on the screen to get the best response. You may find you have to press slightly harder than you would on a consumer smart phone display.
- Any touch on the screen generates a click sound even if it is not on a button or text field. This is normal.
- Be careful not to accidently press both the Scanner button and Home key at the same time as this puts the device into Suspend mode.
- If at any time the SB1 device appears unresponsive, first try touching the screen anywhere to hear a beep and bring the app into focus. If that doesn't work, press the Home key once or twice.
- By holding both buttons down together for 6 seconds, the device will reboot.

SB1 Accessories

- The lanyard is adjustable, and should be adjusted to the shortest length that is comfortable to minimize bouncing or risking the device falling off the J-Hook.
- The J-Hook has been designed for easy one-hand operation to take the SB1 device off for scanning and placing it back when not in use.
- The Belt Holster is designed to be worn on a belt or over the edge of slacks. It can also be used with the Arm Band.
  - This accessory is designed to protect the SB1 screen - the device screen should face inward when placed in the holster.
  - Also, the tether is designed to keep the SB1 connected to the employee so he/she does not lose or misplace it, while providing a long enough length to reach the farthest possible scanning points. Note the tether is neither strong enough nor short enough to keep the device from hitting the floor if dropped - that's not its purpose.
  - The SB1 in a belt holster may appear to reduce the loudness of beeper for notifications. The beeper duration and frequency can be modified if needed.

Network Connectivity and Coverage

- Sometimes a performance issue may appear to be due to the SB1 or application, but may actually be a result of a network issue. As a thin client device, the SB1 requires high quality, reliable WLAN connectivity between the application server and the SB1.
- MTU (Maximum Transmission Unit) settings may impact perceived device performance. Recommend setting is 1300 (current default is 1500). It is possible a setting less than 1300 is required in some customer environments, depending on the network patch between a server and an SB1. Also note that this MTU recommendation should be used for other devices (e.g. MC40, ET1) running the same apps as the SB1.
- Device will Beep on coverage lost and reconnection. Home Screen will indicate whether device Is connected or not
- If a user tries to use the device when coverage is lost, he may see an hour glass icon.

Configuration and Application Development

- Use application cache to reduce the amount of time lost on network transaction and application. The files are not downloaded or checked on each page.
- Resources on application cache can be found at:
  - http://diveintohtml5.info/offline.html
  - http://www.html5rocks.com/en/tutorials/appcache/beginner/
- Badge Mode can be disabled if desired.
- Beeper Notifications: You can change the beep duration and frequency settings to make the beeper more noticeable. This requires settings in the config.js file.
- Update the config.js on the user drive to set low memory to 5.
- Remove the white space in the JavaScript files. Use a tool such as yuicompressor.
  - Go to: http://yui.github.com/yuicompressor/
- Put all the sprite images in one image and then slice the image using CSS.
- All images should be grayscale png format. Do not use jpg format.

- Display a notification on the screen when performing functions that might take some time. This avoids the user from thinking the device locked up.
- When the user selects an input box, loads the keyboard and does not enter text, the input box may appear to disappear and the user cannot see it. So that the input box does not disappear, change the color of the text input box borders to black rather than grey so that it still shows through when the keyboard is closed.
- Minimize or avoid frequent page transitions. Avoid double redirects.
- Avoid apps that require a lot of keyboard entry. Instead of pop-up keyboard entry consider using scanning, voice input or selecting on-screen data which may be easier for the user
- Avoid moving graphics - these will not render well on the E Ink display.
- Use large buttons and input boxes for finger friendly usage. Extend the size of the touch area for each button where possible so that it is slightly larger than the displayed button to enhance touch response and accuracy.
- Consider a Single Page Architecture (SPA) that uses AJAX to dynamically change contents of the page. This will avoid unnecessary re-processing of JavaScript and other resources.
- Consider an alternative way of disabling the barcode scanner as noted in this blog instead of using the scanner.disable() method.
- Although any graphic will be rendered on the E Ink display, only 256 shades of grey are used. You may wish to resize/recolor any used images so that the file size is optimized.
- Avoid pages that require scrolling. Implement a paging mechanism instead.
- Avoid CSS3 properties like gradients, shadows and rgbA transparency.
- Use browser developer tools like Chrome Dev tools to detect if your page is performing unnecessary reflows/repaints. For more details refer https://developer.chrome.com/devtools/index.
- To ensure proper device hygiene and optimize memory usage, it is recommended that MSP or other MDM solution be setup to automatically reboot all SB1 devices at a set time and date once per week (i.e. Sunday night 2:00am).
- Important links for more updated info and best practices:
  - SB1 Developers Launch Pad: https://developer.motorolasolutions.com/community/technologies/mobile_computing/sb1
  - SB1 Software Development Toolkit: Includes sample applications, graphic assets, the Shell API, localization files and other supplemental information to be used as a reference and to help speed application development. Located on Support Central under SB1 or via this link: SB1 Software Development Toolkit Release
  - SB1 Manuals: SB1 User Manual, Programmers Guide, Integrators Guide.

INDEX

B

A

application naming 7-1
application services 2-1
Application Shared Library 1-2, 3-1
application timeout 2-5
application versions 7-4
apps.json 2-2, 2-3
asl module 3-1
asl.back 3-13
asl.badge 3-21
asl.events.fire 3-4
asl.events.subscribe 3-4
asl.exit 3-3
asl.keyboard 3-9
asl.minimize 3-4
asl.notify 3-6
asl.options 3-14
asl.run 3-2
asl.title 3-14
authentication services 2-1, 3-19

B

back button 3-13
baseline files 7-7
bullets vi

C

client applications 1-2
config.js 2-3
content file location 7-2
content file naming 7-2
conventions
notational vi
CSS 1-3

D

developer content 2-4
documentation updates vii

H

HTML 1-3

J

JavaScript 1-3

K

keyboard services 2-1, 3-9

M

messaging services 2-1, 3-18

N

native shell events 3-4
notification services 2-1, 3-6
NPAPI 3-16

O

onBackPressed 3-5
onCradleInsert 3-5
onCradleRemove 3-5
onCriticalBattery 3-5
onExit 3-5

onFocus 3-4
onFocusOut 3-5
onKill 3-5
onLoaded 3-5
onLowBattery 3-5
onOptionSelected 3-5
onSignalLost 3-5
onSignalRestored 3-5

R

resource services 2-1, 3-16

S

shared UI services 2-1, 3-13
system applications 1-1

T

title label 3-14
toolbox 2-4

U

updates, documentation vii

V

visual integration 2-2

W

window services 2-1, 3-22

Motorola Solutions, Inc.
1301 E. Algonquin Rd.
Schaumburg, IL 60196-1078, U.S.A.
http://www.motorolasolutions.com

MOTOROLA, MOTO, MOTOROLA SOLUTIONS and the Stylized M Logo are trademarks or registered trademarks of Motorola Trademark Holdings, LLC and are used under license. All other trademarks are the property of their respective owners.
© 2014 Motorola Solutions, Inc. All Rights Reserved.

72E-170991-02 Revision A - August 2014