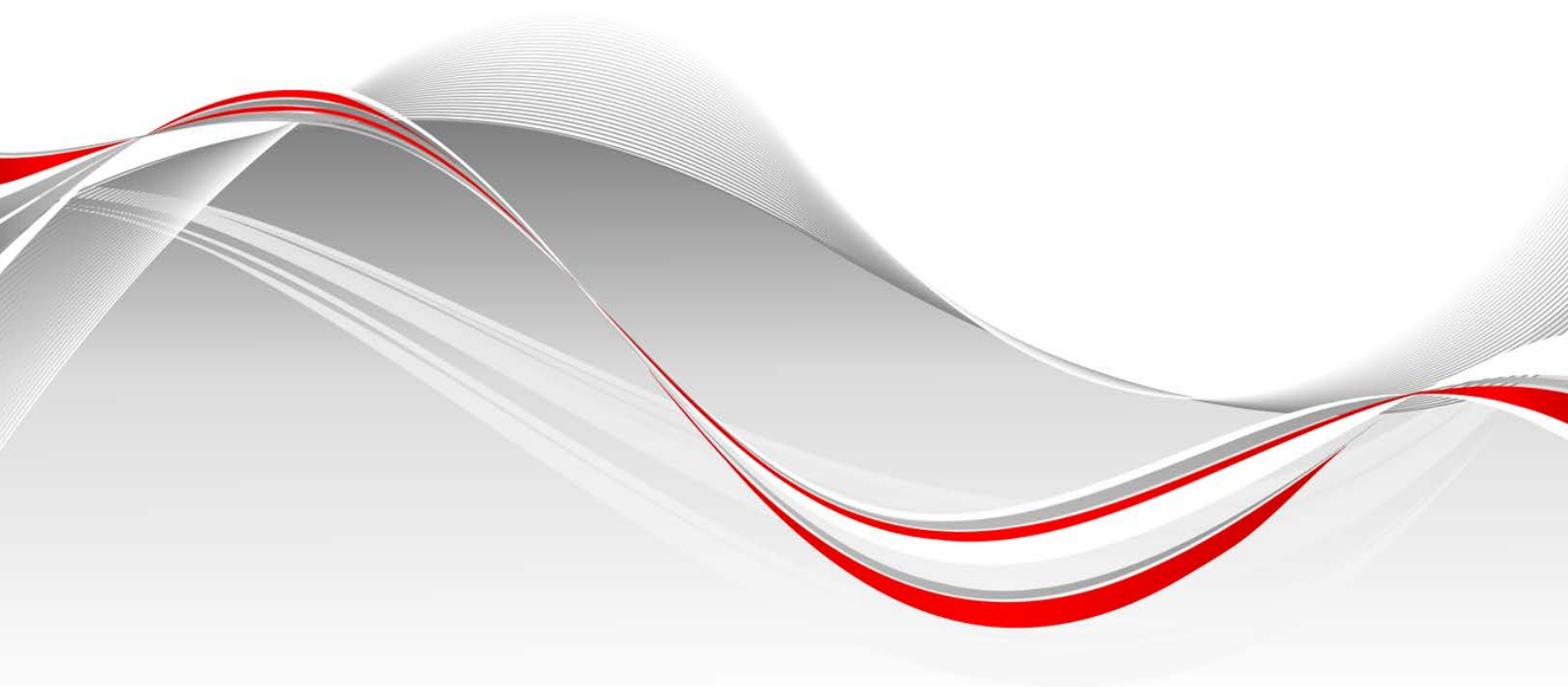




Software Management Client and Update Agent Integrator's Guide

SWM Client Version 5.7 / Update Agent Version 9.0

Doc. Rev. 1.0.12



Red Bend Software Management Solution Notices

vDirect Mobile® Notice

Copyright© 2005-2014, Red Bend Software. All Rights Reserved.

vRapid Mobile® Notice

Copyright© 1999-2014 Red Bend Software. All Rights Reserved.

Patented: www.redbend.com/red-bend-patents.pdf

Red Bend Software Management Center Notice

Copyright© 2008-2014, Red Bend Software. All Rights Reserved.

This Software is the property of Red Bend Ltd. and contains trade secrets, know-how, confidential information and other intellectual property of Red Bend Ltd.

vDirect Mobile®, vRapid Mobile®, Red Bend® and other Red Bend names, as well as the Red Bend Logo are trademarks or registered trademarks of Red Bend Ltd.

All other names and trademarks are the property of their respective owners.

The Products contain components owned by third parties. Copyright notices and terms under which such components are licensed can be found at the following URL, and are hereby incorporated by reference:

www.redbend.com/red-bend-legal-notices.pdf

Table of Contents

Part 1:	Introduction	8
1	Introduction	8
1.1	SWM Center	8
1.2	vDirect Mobile	8
1.3	Update Agent.....	9
1.3.1	Coexistence with Google's Recovery System (GOTA).....	9
1.4	Integration Overview	9
1.5	Feature Overview.....	10
1.6	This Document.....	11
1.7	Related Documentation	11
1.8	Support.....	12
2	OMA Device Management Overview	13
2.1	Device Management Features.....	13
2.2	Management Objects.....	13
Part 2:	SWM Client.....	15
3	Architecture.....	15
3.1	Application Layer	15
3.1.1	Device Integration Layer (DIL)	16
3.1.2	Business Logic Layer (BLL).....	17
3.1.3	Events	17
3.1.4	Event Streamer	18
3.2	DM Core Layer	18
3.3	Porting Layer.....	18
3.4	Program Flow.....	19
4	Installing and Rebuilding the SWM DM Client.....	21
4.1	Requirements	21
4.2	Delivery Structure	21
4.3	Installing the SWM Client on the Device.....	24
4.4	Integrating the SWM Client with the UA.....	24
4.4.1	Common Steps for Integrating the SWM Client with the UA	24
4.4.2	Additional Steps for Replacing GOTA with the UA	24
4.4.3	Additional Steps for Installing the UA to run Alongside GOTA	25
4.5	Modifying and Rebuilding the SWM DM Client.....	25
4.5.1	Configuring the UI.....	25
4.5.2	Configuring the DM Tree	25
4.5.3	Configuring DM Client Parameters	26
4.5.4	Editing the Source Code Using Eclipse.....	28
4.5.5	Rebuilding and Signing the Client using Eclipse IDE (Kepler).....	29

4.6	Registering for Google Cloud Messaging (GCM).....	32
4.7	Intents for Android Integration	32
4.7.1	Intents from SWM client.....	32
5	Configuring the DIL.....	34
5.1	DIL Class Overview	34
5.1.1	common/src/com/redbend/app	36
5.1.2	redbend/src/com/redbend/client	38
5.1.3	redbend/src/com/redbend/client/ui.....	42
5.1.4	redbend/src/com/redbend/client/uialerts.....	45
5.1.5	swm_common/src/com/redbend/swm_common	48
5.2	Event Streamer Overview.....	49
5.2.1	Event Streamer Deliverables	49
5.3	Initializing and Terminating the BLL.....	50
5.3.1	Event Streamer Initialization	50
5.3.2	Device Status Update.....	50
5.4	Sending and Receiving Events	50
5.4.1	Sending BL Events from the DIL.....	50
5.4.2	Receiving DIL Events in the DIL	51
5.5	Events	52
5.5.1	Business Logic Events.....	52
5.5.2	Device Integration Layer Events	56
5.6	Automatic Self-Registration.....	65
5.7	Silent Update and Installation	65
5.8	UA Handoff Flow	66
5.9	Device Administrator Permissions	68
6	SWM DM Client Porting Layer Functions.....	70
6.1	Native Porting Layer Functions.....	70
6.1.1	Device Information Functions.....	70
6.1.2	File Search Functions	70
6.1.3	Delivery Package Wrapper Function.....	71
6.1.4	Self-Update Function	71
6.2	Java Porting Layer Functions	71
6.2.1	iplGetAutoSelfRegDomainInfo.....	71
6.2.2	getDevModel	72
6.2.3	getManufacturer.....	72
6.2.4	getFwVersion	72
6.2.5	getDeviceId	73
6.2.6	getUserAgent.....	73
6.2.7	getDefaultValue	73
6.3	UA Native Porting Layer Functions	74

Part 3:	Update Agent	75
7	Red Bend Update Agent	75
7.1	UA Deliverables.....	76
7.1.1	UA Delivery Structure	76
7.1.2	init Related Files.....	77
7.1.3	Update Generation Files	77
7.2	Standard FOTA Update.....	77
7.3	Background Images Update.....	78
7.4	Recovery from Power Failure	78
7.5	Update Agent Self-Update	78
7.5.1	New Update Agent.....	79
7.5.2	Update Agent Delta Update.....	79
8	Integrating the UA.....	80
8.1	Configuring Partitions	80
8.1.1	Partitions List	80
8.1.2	eMMC Base Path.....	82
8.2	Integrating the UA.....	83
8.2.1	Modifying the Device Recovery System.....	83
8.2.2	Modifying the Device Main System	84
8.3	Implementing Update Package Read Functionality.....	85
8.4	Customizing the UI Display of the Boot Phase and Recovery.....	85
8.5	Rebuilding the Update Agent	86
9	Generating Updates.....	87
9.1	Defining Update Generator Configuration Files	87
9.1.1	Example: Configuration File for Update Generation	88
9.2	Configuration Files for Updating the Update Agent	89
9.2.1	Supplying a New Update Agent	89
9.2.2	Supplying an Update Agent Delta	89
10	Update Agent Parameters	91
11	Update Generator Parameters	95
11.1	Device-Level Parameters.....	95
11.2	General Partition Parameters.....	96
11.3	Custom Operations Parameters	98
12	Failsafe Operation	100
12.1	Integration for Failsafe Operation	100
12.2	Failsafe Coverage Table.....	101
13	Sample Configuration Files.....	102
14	Adding Custom Operations.....	103
14.1	Overview	103

14.2	Custom Update Types	103
14.2.1	Custom Update Calling Sequence.....	103
14.3	Defining a Custom Update during Delta Generation	104
14.3.1	Predefined Custom Operations	104
14.3.2	Sample Configuration File.....	105
14.4	Implementing a Custom Update in the UA	105
14.4.1	custom_ops.c.....	106
14.4.2	Using the Custom Operations API.....	107
14.5	Failsafe Design Principles	108
14.6	Using Custom Operations to Update the Bootloader Partition.....	109
14.6.1	Updating a Partition that is Executed before the OS.....	110
15	Deleting Data and the Cache (Factory Reset).....	112
16	Update Agent Log.....	113
17	Bootloader Update Example.....	114
17.1	Overview	114
17.2	The Original Update Methodology	114
17.2.1	Bootloader structure	114
17.2.2	Update Sequence.....	114
17.2.3	Failsafe Considerations.....	114
17.3	Red Bend Custom Operation Methodology	115
17.3.1	File Structure.....	115
17.3.2	Implementing setBootloaderFlag	115
17.3.3	Update Using Delta	118
17.4	Using the Predefined Custom Operation	119

Table of Tables

Table 3-1: Device Integration Layer Features.....	16
Table 4-1: Configuration Parameters.....	26
Table 5-1: App Classes	36
Table 5-2: Client Classes.....	38
Table 5-3: UI Classes	42
Table 5-4: UI Alerts Classes	45
Table 5-5: SWM Common Classes	48
Table 5-6: BL Events.....	52
Table 5-7: DIL Events.....	56

Table A-1: Update Agent Parameters.....	91
Table B-1: Update Generator – Device-Level Parameters	95
Table B-2: Update Generator – General Partition Parameters	96
Table B-3: Update Generator – Custom Operations Parameters.....	99
Table D-1: Power Failure Coverage	101

Table of Figures

Figure 1-1: SWM Client	10
Figure 3-1: SWM DM Client Layered Architecture	15
Figure 3-2: Application Layer Architecture	16
Figure 3-3: SWM DM Client Sequence Diagram.....	19
Figure 5-1: DIL Class Relationships	35
Figure 5-2: UA Handoff Flow	67
Figure 5-3: Device Administrator Dialog Box	69
Figure 7-1. UA on the device	75

Part 1: Introduction

1 Introduction

Red Bend's *Software Management (SWM) Client* is a set of software components on a mobile device that installs, removes, and updates software components on the device. The client receives instructions from Red Bend's SWM Center. The client can be pre-integrated onto the device by the OEM or downloaded by the end-user.

The SWM Client includes the *SWM DM Client*, a *vDirect Mobile DM Client* for Android, and an *Update Agent (UA)* that performs system-level installations on the device. You can replace Red Bend's Update Agent with your own. The downloaded client does not contain or require a UA.

1.1 SWM Center

The *SWM Center* is a complete software suite that provides a means for a Service Provider, original device manufacturer, or original equipment manufacturer to manage the software components on millions of end-user devices and deploy software and firmware updates over the air. The SWM Center is installed on a central server.

For more information, see *Red Bend FOTA and SWM Solution Description*.

1.2 vDirect Mobile

Red Bend Software's *vDirect Mobile*® provides a cost-effective solution for device manufacturers to quickly integrate Open Mobile Alliance (OMA) standards-based Device Management (DM) functionality into any mobile device or platform. OMA DM is used to remotely provision new subscribers, configure applications and network settings, manage software, and retrieve device information over the air.

Red Bend offers vDirect Mobile in the following configurations:

- **vDirect Mobile Framework:** A comprehensive toolset to build an OMA *DM Client*.
The standard framework is offered as native code (C). Most of the standard framework is platform-independent; the rest can be easily ported to any platform to work with any required business logic.
The Framework is also offered in a pre-built Android Edition that contains the standard framework together with additional Java interfaces for developing a DM Client on an Android device.
- **vDirect Mobile Extended Framework:** The vDirect Mobile Framework together with an Application Core Layer.
The Application Core Layer provides tools to interpret Red Bend's proprietary state machine language (RBSML), allowing you to quickly and easily create the business logic your application requires. RBSML significantly reduces the complexity and time-to-market of your DM Client.

- **vDirect Mobile DM Client:** A fully-operational DM Client for Android built on top of the vDirect Mobile Extended Framework.

The DM Client comes complete with an *Application Layer* built on the Extended Framework: *Business Logic* for one or more multiple mobile network operators (MNOs), and an Android *Device Integration Layer* (DIL) that manages all external interfaces, including the user-interface on the device.

The SWM DM Client implements all features required by several OMA standards and is ready to install with minimal configuration.

The DIL is provided with complete sources that you can modify (on Android) or use as a reference to help you build your own DIL on another platform.

For more information, see the *vDirect Mobile Product Description*.

1.3 Update Agent

A UA manages system-level installations that require a device reboot. After a reboot, the device launches the UA instead of the operating system to perform or complete the installation.

Red Bend offers a UA for Android, as well as an update installation library for safe, secure, and fast firmware installations. For more information on Red Bend's UA, see *Appendix A*.

The SWM Client can work with any UA that supports two API functions: to hand off processing and to return the result. These functions are described in this document.

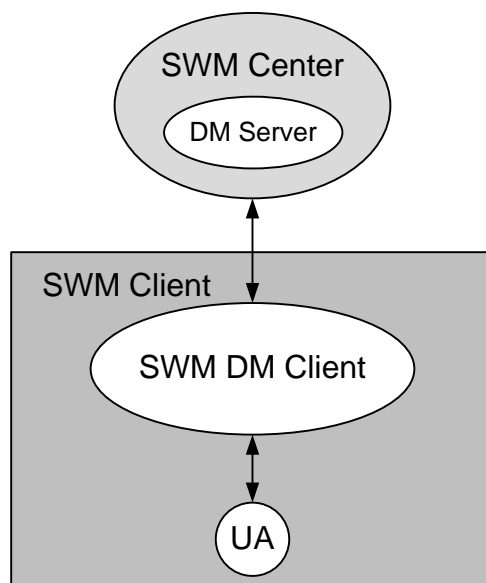
1.3.1 Coexistence with Google's Recovery System (GOTA)

The UA can be installed on the device in place of Google's Android (GOTA) recovery system (a program that is part of the device's standard recovery image) or can coexist together with GOTA on the device.

In the latter case, the SWM Client uses GOTA to invoke the SWM Client UA.

1.4 Integration Overview

The SWM Client must be downloaded as an APK or pre-integrated into the operating system on each device.

**Figure 1-1: SWM Client**

The SWM Client receives instructions from the SWM Center, via the DM Server, to install, remove, and update software components on the device. The SWM DM Client handles simple updates, such as standard APKs, by itself. The SWM DM Client calls the UA to perform system-level updates, such as embedded applications or firmware, which require a reboot of the device.

The SWM DM Client also provides a means for the SWM Center to query software component information on the device, including component versions, and to lock or wipe data from the device.

The SWM Client uses one or more OMA management objects—Firmware Update Management Object (FUMO), Software Component Management Object (SCOMO), and/or Lock and Wipe Management Object (LAWMO)—to download and install components, retrieve device and operation status, and perform other management operations. For more information about FUMO, SCOMO, LAWMO, and components, see the *vDirect Mobile Product Description*.

1.5 Feature Overview

The SWM DM Client implements the following features:

- **Firmware update over-the-air (FOTA/FUMO, pre-integrated client only):** Firmware update initiated by the DM Server, the DM Client, or the end-user.
- **Software component management over-the-air (SCOMO):** Software and/or firmware components installed, updated, or removed by the end-user or DM Server.
- **Device lock, unlock, and data wipe (LAWMO):** Remote device lock or unlock, or data wipe, in the event that the device is lost, stolen, or no longer supported by the carrier.
- **End-user-interaction modes:** Includes silent, end-user notification, and end-user confirmation.
- **End-user cancel/defer/resume:** In confirmation mode, the end-user may cancel or defer the update (indefinitely or for an enforced limited amount of time).

- **Automatic suspend/resume:** Updates may be suspended automatically if the device is receiving a call, out of range, out of service area, in airplane mode, or powered off. Updates resume automatically when applicable.
- **End-user/device/network initiation:** Updates may be initiated by the end-user (checking for updates), the device, or the DM Server.
- **Identity security**
- **Failsafe measures**
- **Integration with the SWM Center:** The SWM Client installs, updates, or removes software components according to the instructions within a Delivery Package (DP). Callbacks provide a means for the SWM Client to report progress information during the change of each component, or of the DP as a whole unit. After applying the DP, the SWM Client reports on the results to the SWM Center. The SWM Center can query the SWM Client for the results of each component in the DP.
The SWM Center can query the SWM Client for a list of all components in the device's inventory, or the attributes of a specific component.
- **Integration with the Update Agent (pre-integrated client only):** Installations are handed off to a UA. The SWM Client retrieves the status of the installation from the UA to send back to the SWM Center. When installed in parallel with GOTA, the SWM Client sends GOTA a small package that instructs GOTA to invoke the SWM Client's UA.

1.6 This Document

The first part of this document presents how to install and configure the SWM DM Client. If you want to customize or create your own version of the DIL, this document describes how to pass and receive events to and from the SWM DM Client's business logic, as well as the Porting Layer functions that you can change.

The second part of this document presents how to install and configure Red Bend's Update Agent.

This document is intended for engineers and support personnel who will install and/or configure the SWM Client and/or Update Agent.

1.7 Related Documentation

This section lists the documents related to this product.

- The SWM Center is provided with its own complete set of documentation.
- If you are using your own UA, and your UA uses Red Bend's vRapid Mobile Update Installer (UPI), see the *vRapid Mobile Integrator's Guide*.

Red Bend Software Product Documentation:

- *vDirect Mobile Product Description*
- *vRapid Mobile Product Description*
- *Red Bend FOTA and SWM Solution Description*
- *SWM Center Administrator's Guide*
- *vDirect Mobile Framework Integrator's Guide*

- *vDirect Mobile Extended Framework Integrator's Guide*
- *vDirect Mobile API Documentation*

Open Mobile Alliance Specification Documents:

- *OMA-Download-OTA-V1_0-20040625-A*
- *OMA-TS-DM_Notification-V1_2_1-20080617-A*
- *OMA-TS-DM_Protocol-V1_2_1-20080617-A*
- *OMA-TS-DM_RepPro-V1_2_1-20080617-A*
- *OMA-TS-DM_Security-V1_2_1-20080617-A*
- *OMA-TS-DM_StdObj-V1_2_1-20080617-A*
- *OMA-TS-DM_TNDS-V1_2-20070209-A*
- *OMA-TS-DM_TND-V1_2_1-20080617-A*
- *OMA-TS-DM_FUMO-V1_0_2-20090828-A*

1.8 Support

If you have a question or require further assistance, contact Red Bend customer support at support@redbend.com.

2 OMA Device Management Overview

The Open Mobile Alliance (OMA) is the leading industry forum for developing market-driven, interoperable mobile service enablers. OMA is designed to be the center of mobile service enabler specification work, aiding in the creation of interoperable services that will meet the needs of the end-user, across countries, operators, and mobile terminals. OMA member companies fall into four categories that define the various parts of the end-to-end value chain: Wireless Vendors, Information Technology Companies, Mobile Operators, and Application and Content Providers.

The purpose of OMA is to enable the growth of the market for the entire mobile industry by removing barriers to interoperability, and by supporting a seamless and easy to use mobile experience for end-users and a market environment that encourages competition through innovation and differentiation. The alliance works with several existing standardization bodies such as 3GPP, 3GPP2, and the WiMAX Forum. The alliance is not in competition with the existing standards but rather complements them by focusing on interoperability of services across markets, terminals, and operators.

For additional information, see <http://www.openmobilealliance.org/AboutOMA/Default.aspx>.

2.1 Device Management Features

OMA's Device Management Working Group specifies protocols and mechanisms for device management. These protocols and mechanisms enable robust management during the entire life cycle of the device and its applications over a variety of carriers.

Device management includes setting and configuring:

- Initial configuration information
- Subsequent installation and updates of persistent information
- Management information
- Events and alarms
- Configuration settings
- Operating parameters
- Software installation and parameters
- Software and firmware updates
- Application settings
- End-user preferences

The SWM DM Client is based on Red Bend's vDirect Mobile version 5.5.0, which handles the OMA DM 1.2 functionality.

2.2 Management Objects

Management objects (MOs) are enablers for agreed functionality between the DM Server and the DM Client.

The DM Working Group describes a few standard enablers. OMA DM is the core enabler, which defines the communication framework between a DM Server and a DM Client. Other enablers define the specialized data models for various management tasks.

Standard MOs include:

- **Firmware Update Management Object (FUMO, pre-integrated client only):** Over-the-air updating and installation of device monolithic firmware.
- **Software Component Management Object (SCOMO):** Over-the-air management of software components.
- **Lock and Wipe Management Object (LAWMO):** Over-the-air data management, including locking and unlocking the device and wiping the device's data.

Part 2: SWM Client

3 Architecture

This section describes the general architecture of the SWM DM Client.

The SWM DM Client comprises a set of components that are delivered as libraries and header files. These files contain all the parts of the Application Layer, vDirect Mobile Engine, and included MOs.

Some components require a platform-specific Porting Layer to integrate the component to the device platform. This layer is provided by Red Bend for some platforms; for other platforms this layer must be developed by the implementer. Red Bend provides prototypes for all Porting Layer functions.

The SWM DM Client is divided into three major layers:

- **The Application Layer:** The business logic, the user interface, and all other external interfaces
- **The DM Core Layer:** The MOs, Engine, and utilities
- **The Porting Layer:** Platform-specific functions that communicate directly with the platform

The SWM DM Client uses a relaxed-layered approach; any upper-level layer can access any lower-level layer directly, without passing through any layers in between.

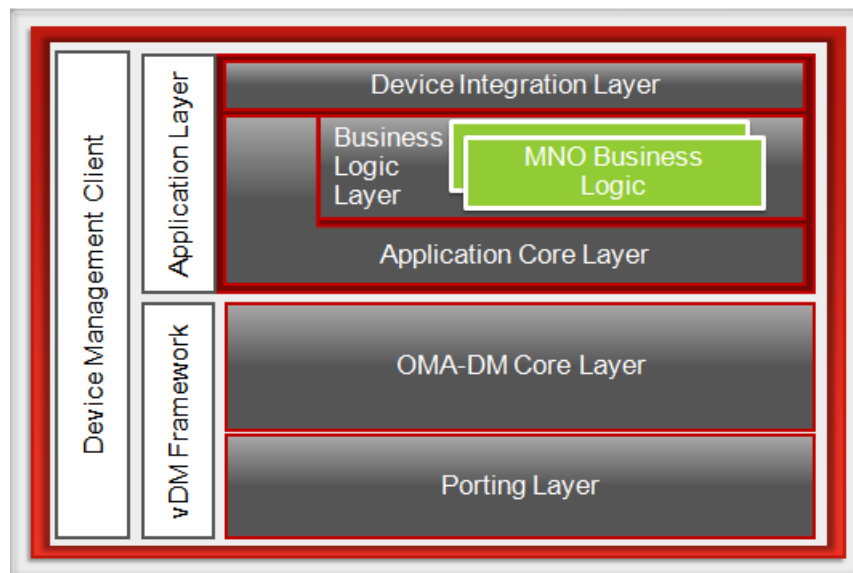


Figure 3-1: SWM DM Client Layered Architecture

3.1 Application Layer

The Application Layer includes the business logic, the user interface, and all other external interfaces.

The Application Layer comprises three parts: The Device Integration Layer (DIL), the Business Logic Layer (BLL), and the Event Streamer.

The following diagram presents an overview of the components of the Application Layer and their relation to the rest of the SWM DM Client.

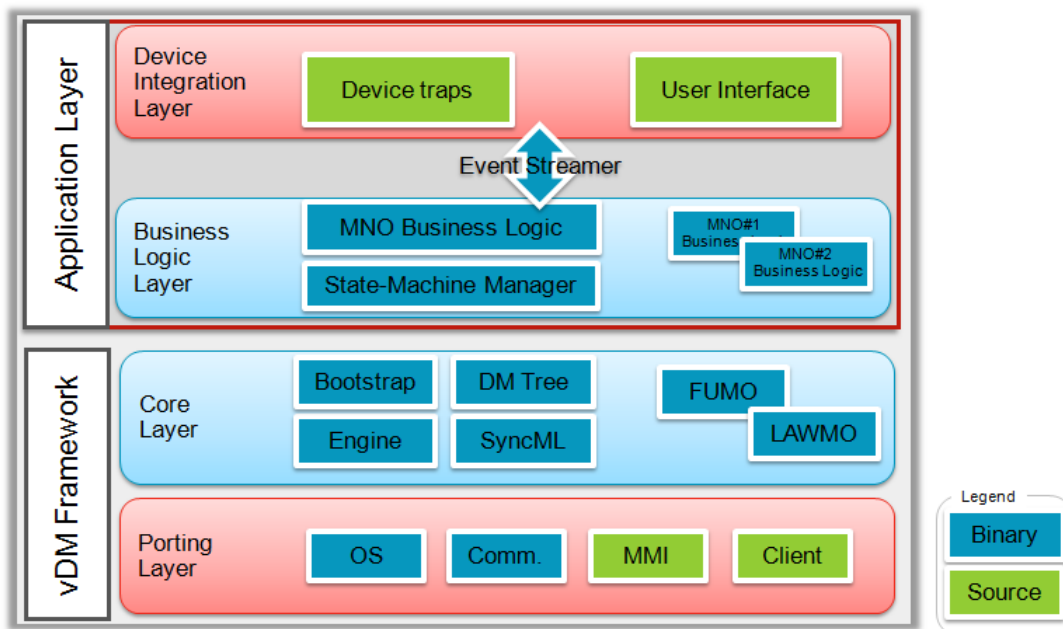


Figure 3-2: Application Layer Architecture

3.1.1 Device Integration Layer (DIL)

The DIL contains platform-dependent code required to handle events external to the SWM DM Client.

For instance:

- **The UI** displays information to the end-user and reads events (selections, key presses, and so on) generated by the end-user.
- **Platform traps** listen for external events, such as incoming phone calls or messages from the DM Server and, as required, generates BL events to send to the BLL.

The DIL features are listed in the following table.

Table 3-1: Device Integration Layer Features

DIL Component	Features
Communication	<ul style="list-style-type: none"> • Airplane Mode Notification • Roaming Notification • Wi-Fi Status Notification
Lock and Wipe	<ul style="list-style-type: none"> • Partially Lock • Fully Lock • Wipe

DIL Component	Features
User Interface	<ul style="list-style-type: none"> • Confirmation • Progress Bar • Text UI Alerts
Startup and Initiation	<ul style="list-style-type: none"> • WAP Push Receiver • Google Cloud Messaging for Android (GCM) • SMS Receiver • Boot Notification
MNO Business Logic	<ul style="list-style-type: none"> • Firmware Update • Software Update • Inventory Query

The DIL sends and receives events to and from the BLL using the Event Streamer.

Red Bend supplies a fully-functioning DIL implemented in Java for Android with the SWM DM Client.

3.1.2 Business Logic Layer (BLL)

The BLL is the core processing component of the Application Layer. The BLL receives BL events from the DIL and internal events from the other parts of the SWM DM Client. As per the instructions of the business logic, the BLL instructs vDirect Mobile to perform actions. The actions can include checking device information, performing an update, or sending a report to the DM Server. According to the results of these actions, the BLL sends DIL events to the DIL.

Some DIL events trigger changes to the device screen (UI). Other DIL events request information about the device, such as the battery level or the result of a DIL operation.

The main parts of the BLL are:

- Multiple **business logics** that control how the device operates in response to BL events. Several fully-functional business logics are included with the Application Layer.
- The **State Machine Manager (SMM)** passes received events to the business logics and processes the instructions from the business logics.

The DIL and BLL may exist as parts of a single application or may be two processes running on the same, or different, devices.

3.1.3 Events

The Application Layer is fully event-driven, activated by triggering events. Not everything that occurs on the device automatically triggers an event; the DIL and other parts of the SWM DM Client decide when to trigger BL and internal events and the business logics decides when to trigger DIL events.

3.1.3.1 Event Types

The Application Layer manages three types of events:

- **BL events:** BL events are generated by the DIL and sent to the BLL. The DIL generates BL events following end-user or device activity or external events such as an incoming message or low battery level. The DIL must inform the business logics about these events.
All BL events are queued in the BLL for processing.
- **Internal events:** Internal events are generated by other parts of the DM Client and sent to the BLL. Internal events are generated while the business logics are processing, such as during an actions or within a callback that the business logics invoked.
All internal events (like the BL events) are queued in the BLL for processing.
- **DIL events:** DIL events are generated by the business logics and sent to the DIL. A DIL event either produces a visible change to the end-user's screen (a new screen, an update to the progress bar, and so on) or triggers another action that must be performed by the platform. The DIL must handle these events.

Events that are not relevant to the device implementation, such as UI events for a device that has no UI, as well as internal events sent between the business logics appear in the log files but can be safely ignored when implementing the DIL, since they do not require any handling by the DIL.

3.1.3.2 Event Structure

Events contain two elements:

- **Event name:** Event names indicate the type of event and describe the event's purpose.
- **Variables:** Variables are attached to the event in an associative array. When receiving a BL or internal event, the variables associated with the event may be used to set state machine variables or be sent as parameters to actions performed by the BLL. When receiving a DIL event, the DIL uses the variables associated with the event to determine an action to perform or the change that will occur on the device screen.

3.1.4 Event Streamer

The DIL and BLL communicate using events handled by a platform-dependent Event Streamer on both sides. The Event Streamer is supplied by Red Bend.

If the DIL and BLL are running as independent processes, events are serialized before being sent between the two processes.

3.2 DM Core Layer

The DM Core Layer includes the Engine and extensions that implement various OMA protocols, such as OMA DM, OMA Download, and various management objects. All components of the DM Core Layer are platform-agnostic and use the API provided by the Porting Layer for platform related functionality.

3.3 Porting Layer

The Porting Layer contains API functions that are required by the Application and DM Core Layers for your specific platform. These API functions are mostly platform-specific.

3.4 Program Flow

The following diagram presents the standard flow of information as events pass between the different layers of the Application Layer and other parts of the SWM DM Client.

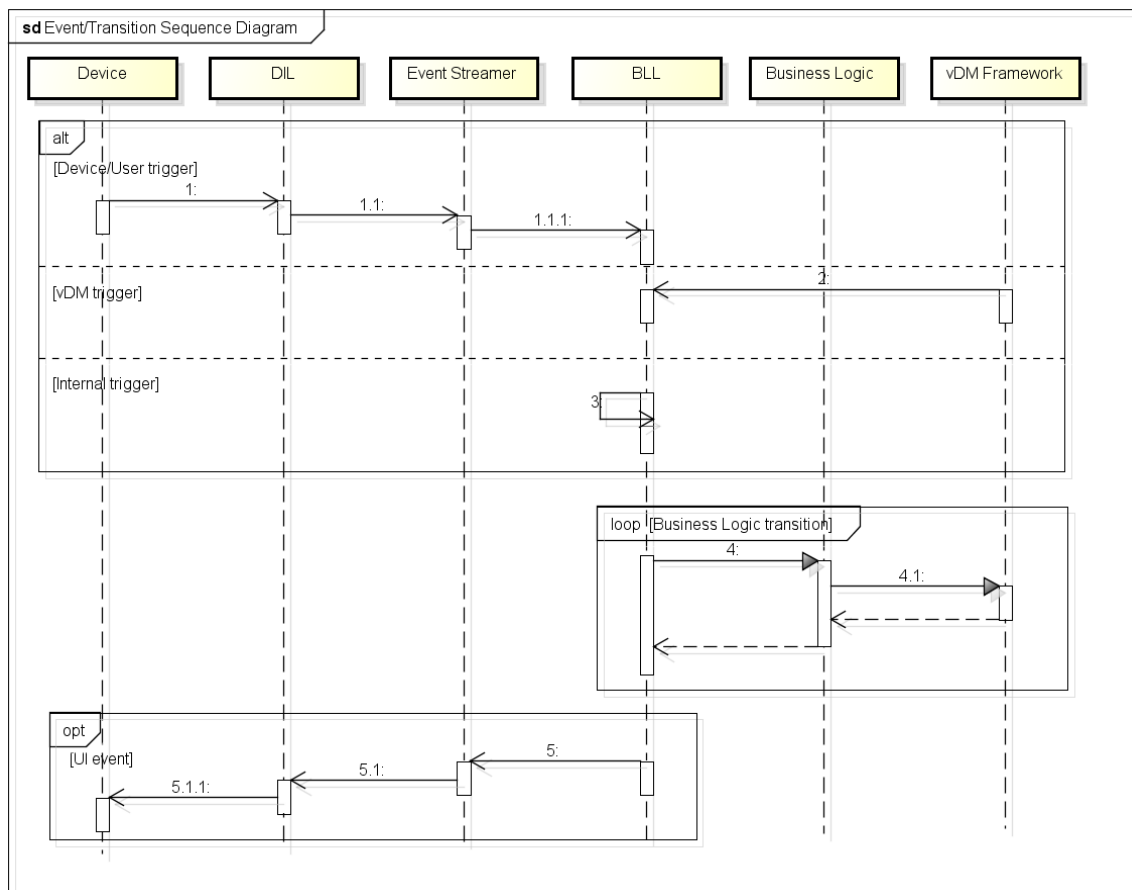


Figure 3-3: SWM DM Client Sequence Diagram

The flow is as follows:

- 1 A BL or internal event is triggered in one of three ways:**
 - As a result of an external real-world event (such as the end-user pressing a key or a message received from a DM Server), or in response to a request from the BLL, a BL event is triggered in the DIL (1).
 - i The DIL serializes the BL event to send it to the BLL (1.1).
 - ii The BLL de-serializes the BL event and places it in a queue (1.1.1).
 - Another part of the SWM DM Client invokes a callback in the BLL that triggers an internal event. The internal event is placed in the queue (2).
 - The BLL triggers a BL timeout event and places it in the queue (3).
- 2 The event is queued in the BLL.**
- 3 The business logic performs transitions:**

The BLL invokes business logic that may invoke actions in the DM Core Layer or Porting Layer as required (4.1).

This step may be repeated multiple times.

4 The business logic triggers a DIL event (optional):

- a The BLL serializes the DIL event to send to the DIL (5).
 - i The DIL de-serializes the DIL event and handles it (5.1).
 - ii The DIL event may trigger a screen to display to the end-user or require some information to be gathered from the device (5.1.1).

4 Installing and Rebuilding the SWM DM Client

A pre-built sample SWM DM Client APK for pre-installation is provided in the delivery package. This chapter presents how to install and/or rebuild the SWM DM Client APK, either as a pre-installable client or as a downloadable client. Basic configuration includes optionally modifying the UI elements, adding support for GOTA (if you are not replacing it with Red Bend's UA), and setting a few configuration parameters.

The downloadable client works only with user applications; it cannot manage embedded applications or firmware. To manage embedded applications and firmware, use the pre-installed client.

If you want to modify the DIL, or create your own, the delivery package includes Java sources that you can use to build your own APK. Before making any changes, see the remaining chapters in this document for important design requirements.

For information on installing Red Bend's UA, see *Integrating the UA*. To have the SWM Client coexist with GOTA, see *Additional Steps for Replacing GOTA with the UA*.

4.1 Requirements

Before installing or modifying the SWM client, you must first install Android SDK revision 22 or later. For the complete list of SDK system requirements, see *System Requirements* at <https://developer.android.com/sdk/index.html>.

The DIL requires the following permissions on the device:

- RECEIVE_BOOT_COMPLETED
- INTERNET
- ACCESS_NETWORK_STATE
- READ_PHONE_STATE
- RECEIVE_WAP_PUSH
- RECEIVE_SMS
- GET_TASKS
- WRITE_EXTERNAL_STORAGE
- INSTALL_PACKAGES
- DELETE_PACKAGES
- CHANGE_NETWORK_STATE
- ACCESS_WIFI_STATE

4.2 Delivery Structure

The SWM DM Client is delivered as a zip file; unzip the file to an empty directory. The delivery contains the following directories and files:

readme.txt	Readme, explaining manual installation and set up
-------------------	---

setup/	
com.redbend.client	An empty file that indicates to the SWM Center that the SWM client is installed.
rb_ua	An empty file that indicates to the SWM Center that the Update Agent is installed.
rb_recovery.fstab	Example Update Agent configuration file
rb_ua.conf	Example vRapid Mobile configuration file
<toolchain>/<target-platform>/	
rls/	
libdmacoapp.a	General functions for the DIL
libdmammi.a	MMI functions
libdma_jni.a	JNI functions used by the DIL
libsmm.so	A compiled shared object library comprising of all the other static libraries in this directory
libua_handoff_installer.a	Non-Porting Layer installation handoff functions
libvdmcomm.a	Communication functions
libvdmengine.a	Engine functions
libvdmfumo.a	FUMO operations functions
libvdmipc.a	IPC functions
libvdmlawmo.a	LAWMO operations functions
libvdmplat.a	Platform-specific Porting Layer functions
libvdmplclient.a	Client Porting Layer functions
libvdmscinv.a	SCOTA inventory operations functions
libvdmscomo.a	SCOTA general operations functions
libvdmsmm.a	SMM functions
libvdmsmmpl.a	SMM Porting Layer functions
libvdmswmcpldevice.a	Device Porting Layer functions
libvdmswmcpldir.a	Directory Porting Layer functions
libvdmswmcplua.a	Handoff Porting Layer functions
libvdmutil.a	Utility functions

<code>rls_dbginfo/</code>	The same files as the <code>rls/</code> directory compiled with the <code>rls_dbginfo</code> (show debug log prints) option
<code>platform/</code>	Porting Layer source files
<code>client/</code>	Source files for the libvdmpclient.a library
<code>device/</code>	Source files for the libvdmswmcpldevice.a library
<code>dir/</code>	Source files for the libvdmswmcpldir.a library
<code>ua/</code>	Source files for the libvdmswmcplua.a library
<code>common/</code>	A part of the application, built as library.
com.redbend.client.apk	Pre-built sample SWM Client application, built with the release option
<code>swm_common/</code>	A part of the application, built as library.
<code>redbend/</code>	Main Android application project directory (including source files of the DIL)
<code>assets/files/</code>	Files that are copied to the application directory
ignore_list	Internal configuration file
rb_ota_zip	Sample signed configuration zip file for using GOTA (configured for maguro and signed with a Google test key)
tree.xml	The DM Tree
<code>libs/</code>	Jars of Java project files for the Application and the native code smm.so file (compiled to show debug log prints)
<code>gota_config/</code>	Scripts and configurations used to generate the file needed for Google OTA (part of the Drop-In client)
build_config.sh	Script to create the signed configuration zip file (rb_ota_zip)
signapk.jar	Java classes used to sign the configuration zip file
<code>maguro/</code>	Sample configuration for maguro (Android Galaxy Nexus device)
<code>sdk/</code>	
<code>import/</code>	Porting Layer header files
<code>include/</code>	
<code>source/</code>	

4.3 Installing the SWM Client on the Device

For a downloadable client, provide the end-user a location (such as a URL) from which to download. No installation is required.

The distribution comes with a sample pre-built SWM DM Client APK that can be used as a pre-installed client for managing embedded applications and firmware.

To build a new version of the client, see *Modifying and Rebuilding the SWM DM Client*.

To pre-install the client:

- 1 Copy the APK to `/system/app` on the device.
- 2 Copy `libsmm.so` to `/system/lib` on the device.

You can test the installation by running a campaign on the device using the SWM Center. For more information, see the *SWM Center Administrator's Guide*.

4.4 Integrating the SWM Client with the UA

If the device won't be updating firmware or embedded applications – for example, if you are creating a downloadable client – skip this section.

There are two different procedures for integrating the SWM Client with the Update Agent:

- **Replace the recovery image of your device.** In this procedure, you remove Google's recovery system (GOTA) and replace it with Red Bend's UA.
- **Don't replace the recovery image,** and instead install Red Bend's UA alongside GOTA. Updates sent to the device are handed off to GOTA, which then hands them back to Red Bend's UA.

4.4.1 Common Steps for Integrating the SWM Client with the UA

Both procedures include the following common tasks:

- Copy the UA (**rb_ua**) executable from the vRapid Mobile distribution to `system/bin` on the device. See *Integrating the UA*.
- Configure **rb_recovery.fstab** and **rb_ua.conf** in the distribution and copy them to `/system/etc` on the device. For information about UA parameters for these files, see the *vRapid Mobile Integrator's Guide*.
- Create the directory `/system/SWM-Client/SYSAPK` on the device and copy the empty files **rb_ua** and **com.redbend.client** from `setup/` in the distribution to this directory.
- Create the directory `/data/redbend` on the device and change its permissions to 777. This directory is a placeholder for UA logs and handover.

4.4.2 Additional Steps for Replacing GOTA with the UA

For complete installation instructions, see *Integrating the UA*. The steps include:

- Modify **init.rc** (mount the file system and change the file, or create a file system and flash the device).

- Edit the DM Tree: change (or ensure that) **./DevDetail/Ext/RedBend/RecoveryType** to RB. If you change the tree, rebuild the client, as described in *Rebuilding and Signing the Client*.

4.4.3 Additional Steps for Installing the UA to run Alongside GOTA

In this scenario, the SWM client sends a signed file to GOTA along with the DP that is to be installed.

The signature on the signed file is a system certificate that is specific to the device model. Red Bend provides an example signed file and the sources used to create it in `gota_config/`. You must edit these files and rebuild the client if you change the keys.

To install the UA to run alongside GOTA:

- 1 Put the device keys in `gota_config/`.
- 2 Run the following script on the command line:

```
./build_config.sh <model> <public_key_file> <private_key_file>  
<output_file>
```

Where:

- `<model>` is the directory name for the model
 - `<public key>` and `<private key>` are the device model-specific keys of the system certificate
 - `<output file>` is the file path at which to create the new signed file.
- 3 Move the new signed file to `assets/files` (if it isn't already there).
 - 4 Edit the DM Tree: Change **./DevDetail/Ext/RedBend/RecoveryType** from RB to GOTA.
 - 5 Rebuild the client, as described in *Rebuilding and Signing the Client*.

4.5 Modifying and Rebuilding the SWM DM Client

The following sections present common scenarios that involve modifying the SWM Client. After any of these modifications, you must rebuild the client as described in *Rebuilding and Signing the Client*.

4.5.1 Configuring the UI

Message texts, icons, and screens are located in **dma/app/android/redbend/res**. Messages and screens are provided in English.

The DIL displays UI screens according to the DIL event ID (see *Device Integration Layer Events*).

If you change any UI elements, rebuild the client, as described in *Rebuilding and Signing the Client*.

4.5.2 Configuring the DM Tree

Much of the DM functionality, including server and client authentication, is controlled using the DM Tree **tree.xml**. For more information, see the *vDirect Mobile Framework Integrator's Guide*.

As an example of a change that you might make that involves changing the DM Tree, you can optimize RAM usage on the device by removing the **FUMO** branch of the tree if your client does not support FUMO or the **SCOMO** branch if your client does not support SCOMO.

Do not remove any nodes under the **Ext/RedBend** branch.

If you change the DM Tree, rebuild the client as described in *Rebuilding and Signing the Client*.

4.5.3 Configuring DM Client Parameters

The SWM DM Client provides a few configuration options in a separate configuration file. Changing these configuration options does not require you to rebuild the client. However, after changing the configuration file you must restart the client on the device. For more extensive configuration, you must change the DIL or Porting Layer functions, as described in the remaining chapters in this document.

To configure the SWM DM Client, create the file `/system/bin/dma_config.txt`. The format of the configuration file is as follows:

- Each line is a key-value pair, as follows:
`key=value`
- No spaces are allowed, even before and after the equal sign (=).
- No comments are allowed after the value.
- Lines may be empty.
- A comment line begins with the pound or hash sign (#). No spaces are allowed before the hash sign.

The following is an example of lines from a configuration file:

```
dp_path_file=DP_Path_File.txt
# Error logging; print only error messages
```

The following table presents the available parameters that are specific to the SWM DM Client. You can configure many additional standard configuration parameters. For more information about these parameters, see the *vDirect Mobile Framework Integrator's Guide*.

Table 4-1: Configuration Parameters

Parameter	Description	Value
dl_resume_max_counter	The maximum number of times to retry an interrupted download.	Integer The default is 10
NOTE: If you specify <code>dl_resume_max_counter</code> and <code>dl_resume_timeout</code> , the SWM DM Client retries while both values are valid		

Parameter	Description	Value
dl_resume_timeout	How long, in minutes, to retry an interrupted download. NOTE: If you specify <code>dl_resume_timeout</code> and <code>dl_resume_max_counter</code> , the SWM DM Client retries while both values are valid	Integer The default is 1440 (24 hours)
dp_path_file	The name of the file that contains the directory in which DPs are stored after they are downloaded. For example: <code>DP_Path_File.txt</code>	File name The file contains an absolute path to the directory used by the Porting Layer function <code>VDM_SWMC_PL_UA_handoff</code> The default is <code>/data/redbend/dp</code>
handoff_dir	Absolute path to the directory that contains the files specified by dp_path_file and ua_result_file . For example: <code>/system/bin</code>	Absolute path The default is <code>/data/redbend/workdir</code>
maxnetretries	The maximum number of times to try to reconnect following: <ul style="list-style-type: none"> • Socket read / write errors • TCP timeout: <ul style="list-style-type: none"> ◦ Host cannot be reached ◦ Connection refusal ◦ Unresolved address 	The number of <i>additional</i> times to retry (one retry will always be made) The default is 3

Parameter	Description	Value
report_persistency_max_counter	<p>The maximum number of times to retry uploading a report to the SWM Center.</p> <p>NOTE: If you specify <code>report_persistency_max_counter</code> and <code>report_persistency_timeout</code>, the SWM DM Client retries while both values are valid</p>	<p>Integer</p> <p>The default is 10</p>
report_persistency_timeout	<p>How long, in minutes, to retry uploading a report to the SWM Center.</p> <p>NOTE: If you specify <code>report_persistency_timeout</code> and <code>report_persistency_max_counter</code>, the SWM DM Client retries while both values are valid</p>	<p>Integer</p> <p>The default is 1440 (24 hours)</p>
ua_exec	<p>Absolute path to the UA executable.</p> <p>For example:</p> <pre>/system/bin/ua</pre>	<p>Absolute path to a file</p> <p>The default is <code>ua</code></p>
ua_result_file	<p>The name of the file into which the UA will write the result of a DP update.</p> <p>For example:</p> <pre>result.txt</pre>	<p>File name</p> <p>The default is <code>/data/redbend/result</code></p>
scom_battery_threshold	<p>The minimum percentage of the battery charge required for an installation to start.</p>	<p>Integer</p> <p>The default is 25</p>

4.5.4 Editing the Source Code Using Eclipse

Before editing the DM Client using Eclipse, install the following:

- Eclipse

- The Eclipse plug-in for Android
- Android API level 14

To edit the source code using Eclipse (Kepler Version):

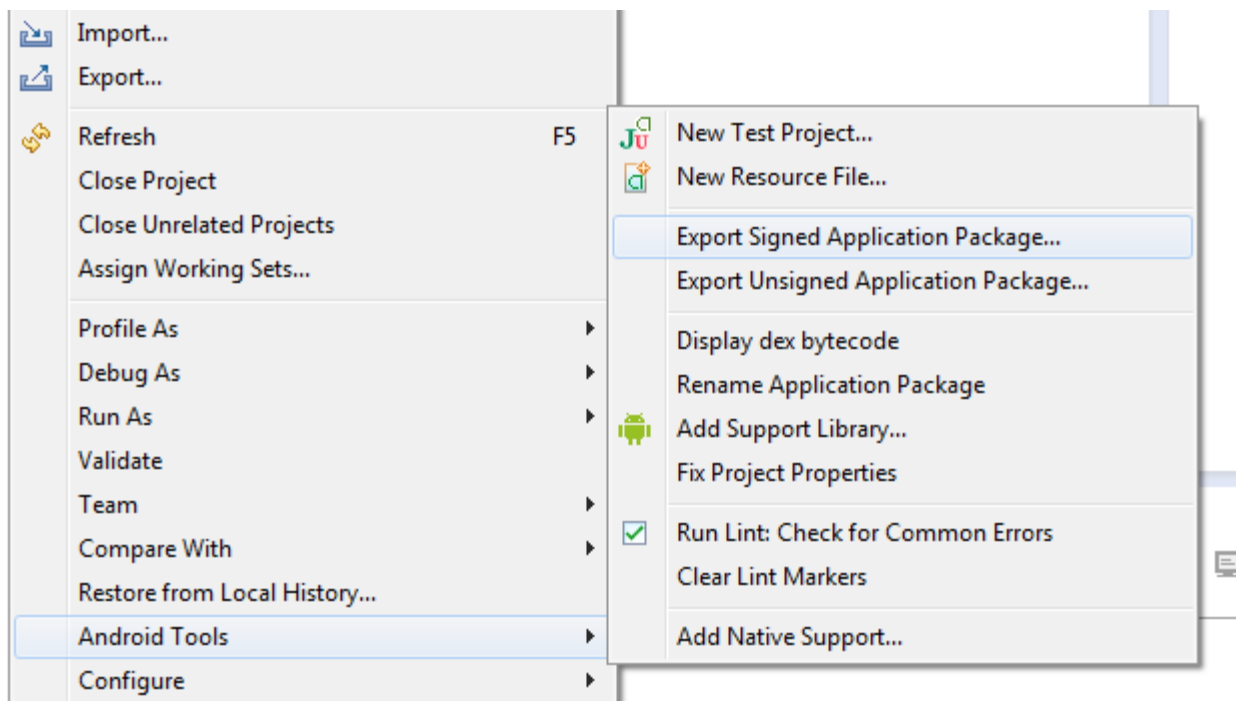
- 1 Open *Eclipse*, and select **File → New → Project...**
The **New Project** dialog box appears.
- 2 Select **Android → Android Project From Existing Code**, and click **Next**.
The **New Android Project** dialog box appears.
- 3 Browse to the main delivery folder and click **Finish**.
- 4 Select all three projects: `smw_common`, `common`, and `StartupActivity` and click **Finish**.

After editing the source code, rebuild the client as described in *Rebuilding and Signing the Client*.

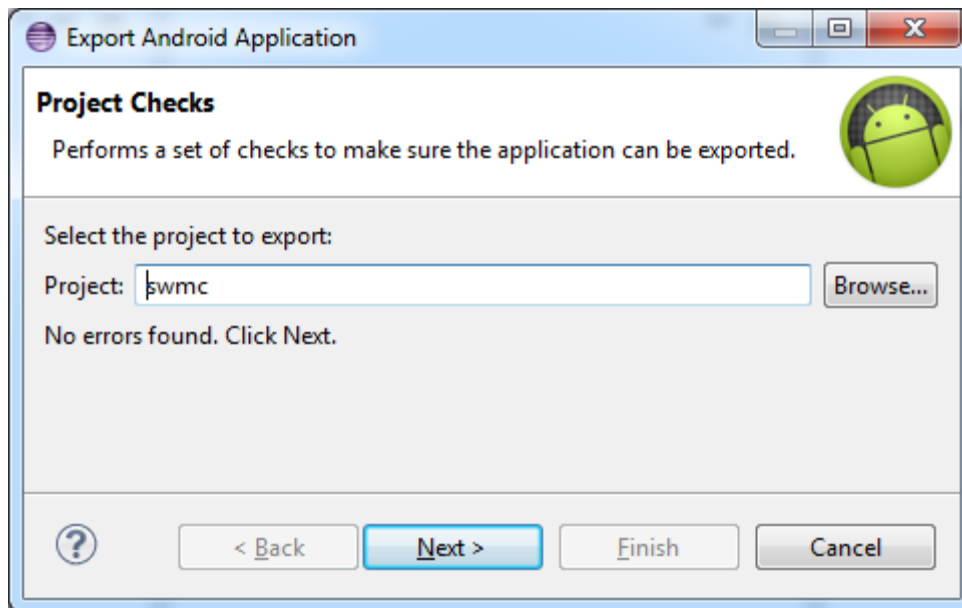
4.5.5 Rebuilding and Signing the Client using Eclipse IDE (Kepler)

To rebuild the SWM DM Client:

- 1 Make any of the changes described in this document and/or edit the source code using Eclipse or any other IDE.
- 2 Right-click the `StartupActivity` project and select **Android Tools → Export Signed Application Package...**

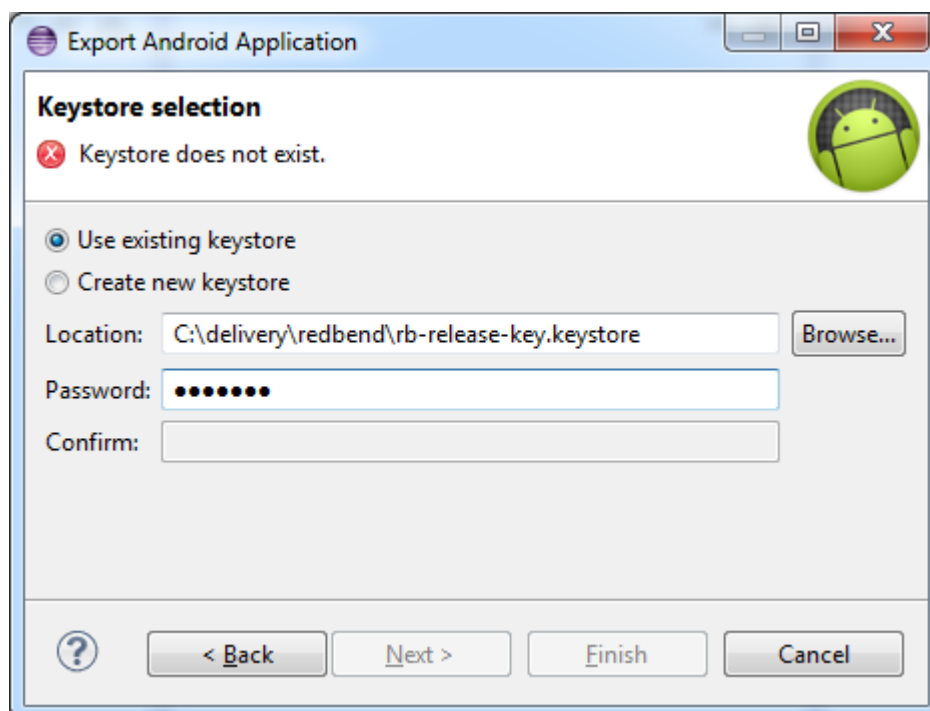


The **Project Checks** window appears.



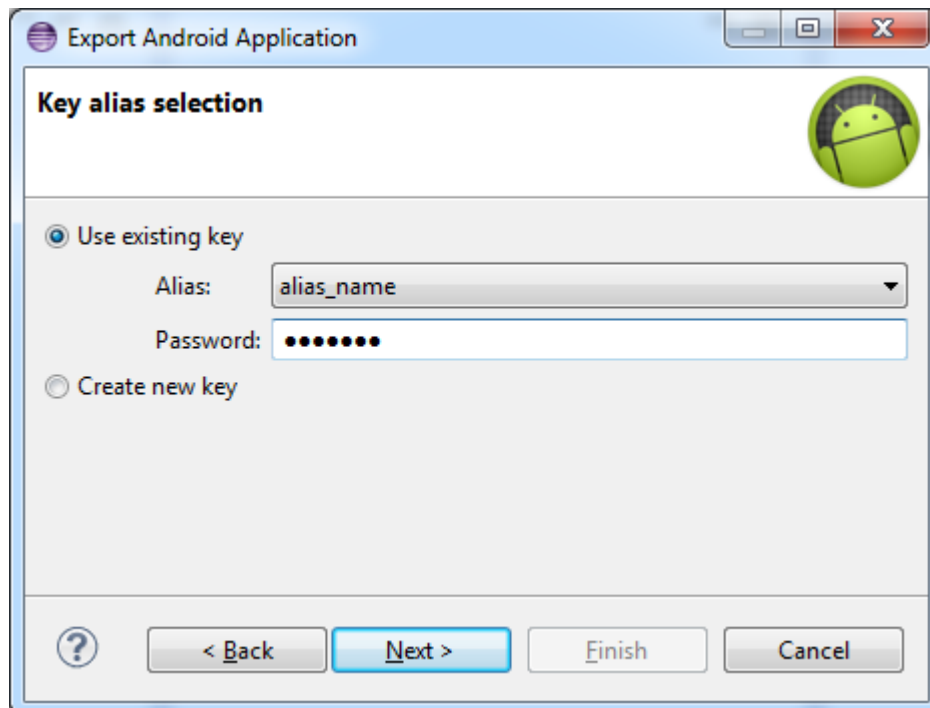
- 3 Enter `swmc` and click **Next**.

The **Keystore selection** window appears.

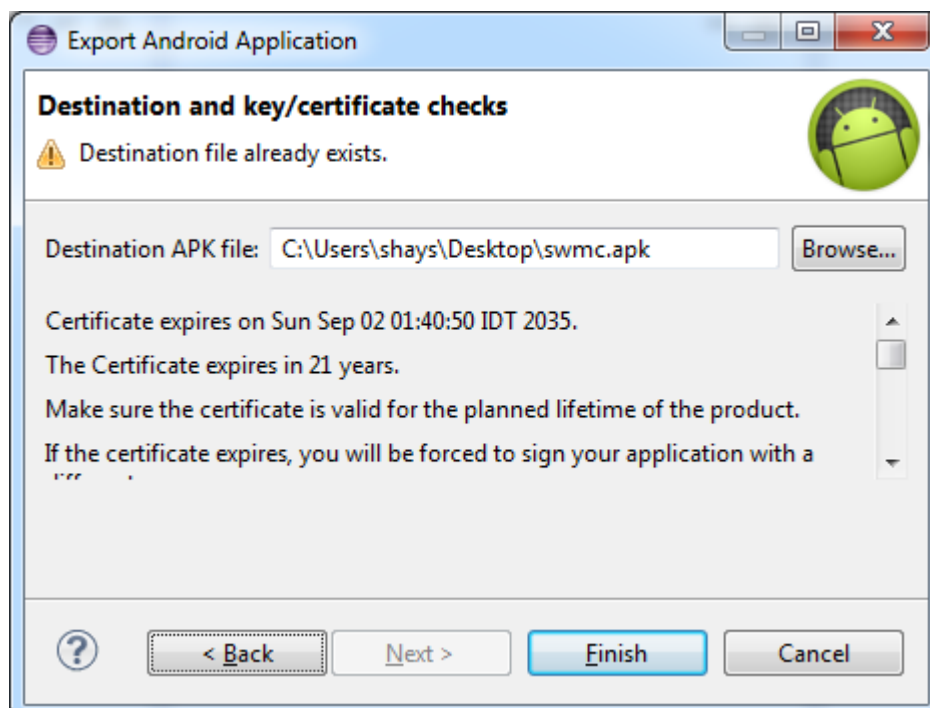


- 4 For **Location**, click **Browse**, and browse to the `redbend` folder and select the Red Bend keystore `rb-release-key.keystore` or a keystore that you have generated.
- 5 If you are using the Red Bend keystore, in **Password** enter: `redbend`. Otherwise, enter the keystore password.
- 6 Click **Next**.

The **Key alias selection** window appears.



- 7 If you are using the Red Bend keystore, select the `alias_name` alias. Otherwise, select the required alias from the list.
- 8 Enter the alias password in **Password** (for Red Bend, enter: `redbend`) and click **Next**.
The **Destination and key/certificate checks** window appears.



- 9 Browse to the location and name of the target APK.
- 10 Click **Finish**.

The client is rebuilt. After rebuilding, proceed with the installation as described in *Installing the SWM Client on the Device*.

4.6 Registering for Google Cloud Messaging (GCM)

The SWM Center can be configured to use the Google Cloud Messaging (GCM) service as a third-party gateway to send custom OMA DM notifications to the devices. The SWM Client includes support for GCM.

If the SWM Center uses GCM, each device must also be registered with the GCM service to receive OMA DM notifications from the SWM Center.

To register a device for GCM:

- For devices running Android 4.0.4 and above, nothing needs to be done. The SWM Client uses the default Google account to automatically register the device with the GCM service.
- Devices running Android versions prior to 4.0.4 do not have a Google account as a default. The end-user must manually set the Google account on the device. You must ensure that end-users perform this step.

4.7 Intents for Android Integration

The SWM Client receives the following intents to enable you to integrate the SWM Client with Android device settings.

- `SwmClient.CHECK_FOR_UPDATES_NOW`: Check if there is a new update for the device
- `SwmClient.ENABLE_PERIODIC_CHECK_FOR_UPDATES`: Enable periodic checking for updates
- `SwmClient.DISABLE_PERIODIC_CHECK_FOR_UPDATES`: Disable periodic checking for updates
- `SwmClient.CHANGE_PERIODIC_CHECK_FOR_UPDATES`: Change the interval of the periodic checking for updates, set the new interval in hours in the event variable `DMA_VAR_SCOMO_DEVINIT_NEW_POLLING_INTERVAL`.

Android example code:

```
Intent intent = new Intent();
intent.setAction("SwmClient.CHECK_FOR_UPDATES_NOW");
sendBroadcast(intent);
```

4.7.1 Intents from SWM client

The SWM Client sends broadcast intents to notify external Android components about the status of updates:

- `SwmClient.NEW_UPDATE_AVAILABLE`: Update available on the server
- `SwmClient.DMA_UPDATE_FINISHED`: Update finished successfully
- `SwmClient.UPDATE_FINISHED_FAILED`: Updated failed

If other applications wish to catch the above intents they must implement an Android receiver and add the intents to the receiver filter.

5 Configuring the DIL

This and the following chapter present how to integrate the SWM DM Client into your platform by configuring Red Bend's DIL or creating your own DIL. Any DIL implementation must send certain BL events to the BLL, handle certain DIL events received from the BLL, and implement certain Porting Layer functions required by the BLL and/or the Framework.

This chapter presents an overview of the DIL classes, describes how Red Bend's DIL implementation initializes the BLL and Event Streamer, and presents the list of events that must be sent from and received by the DIL. This chapter also presents an overview of key parts of the client execution flow.

5.1 DIL Class Overview

The DIL Java implementation contains two main packages:

- `com.redbend.app`
Contains the generic implementation of DIL logic, including:
 - Defining events and event variables
 - Initiating the SMM
 - Sending BL events and receiving DIL events to and from the SMM
 - Assigning class methods to handle incoming DIL events
 - Managing Android Tasks and Activities required by the DIL
 - Defining abstract Activity and broadcast receiver classes, which include functionalities that implement the above logic
- `com.redbend.client`
Contains specific DIL methods, including:
 - All classes that define the behavior when a certain DIL event is received:
 - Android Activity classes that define UI screens
 - Android broadcast receiver classes that define device traps that send BL events
 - Other general handlers, including end-user notifications
 - A service that extends the generic implementation (`com.redbend.app`), declares all DIL event handlers, and performs other initializations that are needed

The following partial class diagram presents the relationships between some of the main classes.

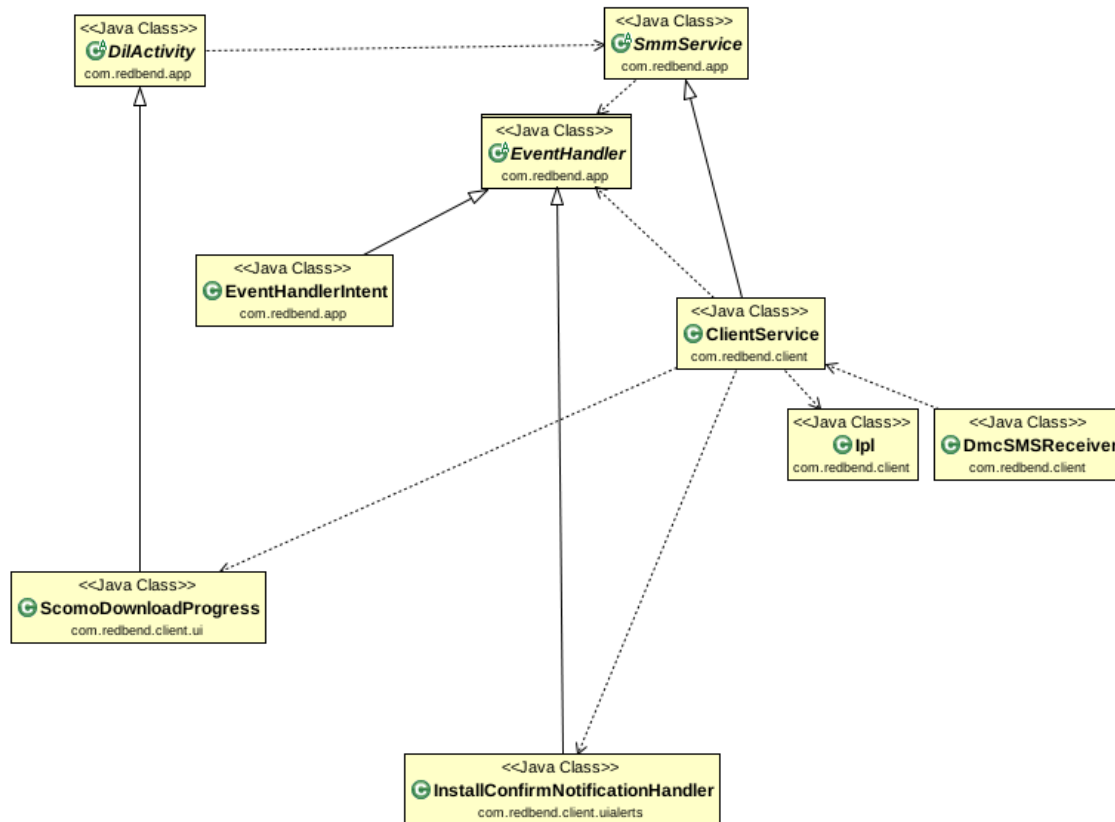


Figure 5-1: DIL Class Relationships

The classes in the diagram are as follows:

- **com.redbend.app:**

- `SmmService`

This class is an abstract implementation of Android Service. The class manages the DIL.

- `EventHandler`

This abstract class is common for all classes that receive one or more DIL events.

- `EventHandlerIntent`

This class is an implementation of `EventHandler`. This class starts an Android Activity when a DIL event is received.

- `DilActivity`

This class contains common logic for every Android Activity that is displayed as a result of a DIL event. An instance of the class is started from an Intent generated by `EventHandlerIntent`. This class communicates with `SmmService` to create Android Tasks.

`SmmService` communicates with `DilActivity` to know what's happening with every Activity so that the screens are handled properly. For example, the back button on the

FUMO download progress screen (an Activity) must not return to the FUMO download confirmation screen (another Activity).

- **com.redbend.client:**

- `ClientService`

This is a specific implementation of `SmmService` that declares all `EventHandler` instances.

- `Ipl`

This class includes Integration Porting Layer functions that you must implement.

- `DmcSMSReceiver`

This class is a specific implementation of `BroadcastReceiver`. The class receives and processes SMSs and creates BL events as required.

- `ScomoDownloadProgress`

This class is a specific implementation of `DilActivity`. The class handles DIL events about download progress during SCOMO.

- `InstallConfirmNotificationHandler`

This class is a specific implementation of `EventHandler`. The class implements how the Android Notification is displayed when the end-user is supposed to confirm an installation.

The following sections list all of the classes.

5.1.1 common/src/com/redbend/app

Table 5-1: App Classes

Class	Handles DIL Event	Description	Sends BL Events
<code>DilActivity</code>		Common logic for every Android Activity that is displayed as a result of a DIL event	
<code>Event</code>		The event class; used for both BL and DIL events	
<code>EventHandler</code>		Handles DIL events	
<code>EventHandlerContext</code>		Executes the event handler	
<code>EventHandlerIntent</code>		Starts an Android Activity when a DIL event is received	

Class	Handles DIL Event	Description	Sends BL Events
EventIntentService		Generates Android Intents when receiving DIL events, and generates BL events when receiving Android Intents	
EventMultiplexer		Invokes the associated handler (<code>EventHandler</code>) for an event	
EventVar		The event variable class An event object contains an <code>EventVar</code> list	
ExtNodesHandler		Handles read/write/exec to DM Tree nodes stored in external storage	
FlowManager		Logic of a flow: a flow is a set of event handlers that belong to the same task <code>SmmService</code> uses the <code>FlowManager</code> logic	
FlowUtils		Utilities (such as logging) to support <code>FlowManager</code>	
NetClient		Handles network communications (TCP/IP) Not used by the provided implementation of the SWM DM Client	
SmmReceive		Logic to send BL events to the BLL An Android broadcast receiver should inherit this class	

Class	Handles DIL Event	Description	Sends BL Events
SmmService		Manages the DIL An abstract implementation of the Android service	

5.1.2 redbend/src/com/redbend/client

Table 5-2: Client Classes

Class	Handles DIL Event	Description	Sends BL Events
BasicService		Sends and receives events using intents to and from the BLL Initializes the SMM service and sets up the relevant resources	
ClientService		Implements SmmService abstract class to handle events Sends and receives events, using intents to and from the BLL using BasicService Registers handlers for DIL events In addition, has listeners for the device state that generate BL events	DMA_MSG_PRODUCT_TYPE DMA_MSG_AUTO_SELF_REG_INFO DMA_MSG_STS_ROAMING DMA_MSG_STS_VOICE_CALL_START DMA_MSG_STS_VOICE_CALL_STOP
ConnectivityStateChangeReceiver		Handles network (carrier and Wi-Fi) connectivity changes	DMA_MSG_STS_MOBILE_DATA DMA_MSG_STS_WIFI

Class	Handles DIL Event	Description	Sends BL Events
DeviceUpdateReceiver		Handles OEM settings of device update commands: <ul style="list-style-type: none"> • Enable, disable, or change the periodic check for updates • Check for updates now 	DMA_MSG_SCOMO_DEVICE_INIT_POLLING_ENABLE DMA_MSG_SCOMO_DEVICE_INIT_POLLING_DISABLE DMA_MSG_SCOMO_DEVICE_INIT_POLLING_CHANGE DMA_MSG_SESS_INITIATOR_USER_SCOMO
DmcBootReceiver		Handles power on	DMA_MSG_STS_POWERED when the device powers on
DmcSMSReceiver		Simulates a WAP Push NOTE: This class (and its AndroidManifest.XML declaration) should be removed from a production system	DMA_MSG_NET_NIA when a WAP Push simulation is received via SMS
DmcWapReceiver		Handles a WAP Push	DMA_MSG_NET_NIA when a WAP Push is received The WAP Push data is added to DMA_VAR_NIA_MSG and DMA_VAR_NIA_ENCODED is set to 1 to indicate binary data
GetBatteryLevelHandler	DMA_MSG_GET_BATTERY_LEVEL	Sends the battery level to the business logic	DMA_MSG_BATTERY_LEVEL

Class	Handles DIL Event	Description	Sends BL Events
IntentBroadcaster	DMA_MSG_SCOMO_DL_CONFIRM_UI DMA_MSG_SCOMO_NOTIFY_DL_UI DMA_MSG_SCOMO_INSTALL_DONE	Sends intents (see <i>Intents for Android Integration</i>)	
Ipl		Gets the domain name and domain PIN from external storage, usually an SD card	
LawmoHandlerBase		Base abstract class for handling LAWMO events Implements common logic that includes checking for permission and sending a result event Each LAWMO operation implements this class	
LockingHandler	DMA_MSG_LAWMO_LOCK_LAUNCH	Fully lock device (LAWMO)	DMA_MSG_LAWMO_LOCK_ENDED_FAILURE DMA_MSG_LAWMO_LOCK_ENDED_SUCCESS
ManageSpaceActivity		Override Android's application clean data mechanism Invoked when the end-user taps Manage Space in Android's application settings menu	
ScomAlarmReceiver		Listens for polling alarm set by ScomAlarmSetter	DMA_MSG_DL_TIMESLOT_TIMEOUT when a polling alarm expires

Class	Handles DIL Event	Description	Sends BL Events
ScomoAlarmSetter	<code>DMA_MSG_SCOMO_SET_DL_TIMESLOT</code>	<p>Sets a polling alarm to be handled by ScomoAlarmReceiver</p> <p>The alarm time is set in <code>DMA_VAR_SCOMO_DOWNLOAD_TIME_SECONDS</code></p>	<code>DMA_MSG_SCOMO_SET_DL_TIMESLOT_DONE</code> when a polling alarm is set
SmmConstants		Sets some SMM constants	
StartDownload	<code>DMA_MSG_SCOMO_DL_INIT</code> (foreground only)	Moves download to background if not in user mode (checks <code>DMA_VAR_SCOMO_MODE</code>)	
StartupActivity		<p>Starts the application when the end-user taps the application icon.</p> <p>If Device Administrator permission is disabled displays the Device Administrator dialog box</p>	<code>DMA_MSG_SESS_INITIATOR_USER_SCOMO</code>
SwmStartupActivityBase		Base class for StartupActivity	
UnLockHandler	<code>DMA_MSG_LAWMO_UNLOCK_LAUNCH</code>	Unlocks device (LAWMO)	<code>DMA_MSG_LAWMO_UNLOCK_ENDED_FAILURE</code> <code>DMA_MSG_LAWMO_UNLOCK_ENDED_SUCCESS</code>
WipeAgentHandler	<code>DMA_MSG_LAWMO_WIPE_AGENT_LAUNCH</code>	Wipes device (LAWMO)	<code>DMA_MSG_LAWMO_WIPE_AGENT_ENDED_FAILURE</code> <code>DMA_MSG_LAWMO_WIPE_AGENT_ENDED_SUCCESS</code>

5.1.3 redbend/src/com/redbend/client/ui

Table 5-3: UI Classes

Class	Handles DIL Event	Description	Sends BL Events
BatteryLow	<code>DMA_MSG_SCOMO_INS_CHARGE_BATTERY_UI</code> (foreground only)	<p>Notifies the end-user that the update was interrupted by a low battery</p> <p>After user-confirmation, the business logic checks the battery level using <code>GetBatteryLevelHandler</code>. If the battery level is too low, this class handles foreground notification while the class <code>BatteryLowNotificationHandler</code> handles background notification</p> <p>The notification displays the required battery threshold (<code>DMA_VAR_BATTERY_THRESHOLD</code>).</p>	
ErrorHandler	<ul style="list-style-type: none"> <code>DMA_MSG_DM_ERROR_UI</code> <code>DMA_MSG_DL_ERROR_UI</code> <code>DMA_MSG_DL_INST_ERROR_UI</code> <code>DMA_MSG_SCOMO_DL_CONFIRM_UI</code> (user initiated and silent campaign only) 	Error handling, displays an error message	
InstallApk	<ul style="list-style-type: none"> <code>DMA_MSG_SCOMO_INSTALL_COMP_REQUEST</code> <code>DMA_MSG_SCOMO_REMOVE_COMP_REQUEST</code> <code>DMA_MSG_SCOMO_CANCEL_COMP_REQUEST</code> 	Manages APK installation / uninstallation	<code>DMA_MSG_SCOMO_INSTALL_COMP_RESULT</code> <code>DMA_MSG_SCOMO_REMOVE_COMP_RESULT</code>
LawmoLockResult	<ul style="list-style-type: none"> <code>DMA_MSG_LAWMO_LOCK_RESULT_SUCCESS</code> (foreground only) <code>DMA_MSG_LAWMO_LOCK_RESULT_FAILURE</code> (foreground only) 	<p>Notifies on fully lock result</p> <p>Extends <code>LawmoResultHandlerBase</code></p>	

Class	Handles DIL Event	Description	Sends BL Events
LawmoResultHandlerBase		Base class for: <ul style="list-style-type: none"> LawmoLockResult LawmoUnlockResult 	
LawmoUnlockResult	<ul style="list-style-type: none"> DMA_MSG_LAWMO_UNLOCK_RESULT_SUCCESS (foreground only) DMA_MSG_LAWMO_UNLOCK_RESULT_FAILURE (foreground only) 	Notifies on unlock result Extends LawmoResultHandlerBase	
LawmoWipeResult	<ul style="list-style-type: none"> DMA_MSG_LAWMO_WIPE_RESULT_SUCCESS (foreground only) DMA_MSG_LAWMO_WIPE_RESULT_FAILURE (foreground only) DMA_MSG_LAWMO_WIPE_RESULT_NOT_PERFORMED (foreground only) 	Notifies on wipe data operation result	
LoadingActivity	DMA_MSG_USER_SESSION_TRIGGERED (foreground only)	Notifies the end-user that the device is checking for new updates	
RecoveryReboot	DMA_MSG_SCOMO_REBOOT_REQUEST	Reboots to recovery mode	DMA_MSG_SCOMO_INSTALL_CANNOT_PROCEED DMA_MSG_SCOMO_VERIFICATION_FAILURE
ScomoConfirm	DMA_MSG_SCOMO_DL_CONFIRM_UI if DMA_VAR_SCOMO_QUIET=0	Prompts the end-user to confirm or reject the download	DMA_MSG_SCOMO_ACCEPT DMA_MSG_SCOMO_POSTPONE DMA_MSG_SCOMO_CANCEL

Class	Handles DIL Event	Description	Sends BL Events
ScomoConfirmProgressBase		Base class for: <ul style="list-style-type: none"> ScomoConfirm ScomoDownloadInterrupted ScomoDownloadProgress ScomoInstallConfirm ScomoInstallProgress 	
ScomoDownloadInterrupted	<ul style="list-style-type: none"> <code>DMA_MSG_SCOMO_DL_CANCELED_UI</code> (foreground only) <code>DMA_MSG_DNLD_FAILURE</code> (foreground only; on first non-silent retry only) 	Notifies the end-user that the download was interrupted	
ScomoDownloadProgress	<code>DMA_MSG_SCOMO_DL_PROGRESS</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code>	Displays download progress Displays a cancel button if not a critical update	
ScomoDownloadSuspend	<ul style="list-style-type: none"> <code>DMA_MSG_SCOMO_DL_SUSPEND_UI</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> (foreground only) <code>DMA_MSG_SCOMO_DL_SUSPEND_UI_FROM_ICON</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> 	Notifies the end-user that the download was suspended. The screen displays for <code>DMA_VAR_SCOMO_DOWNLOAD_TIME_SECONDS</code> seconds.	
ScomoInstallConfirm	<ul style="list-style-type: none"> <code>DMA_MSG_SCOMO_INSTALL_CONFIRM_UI</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> <code>DMA_MSG_SCOMO_INSTALL_CONFIRM_UI</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> and <code>DMA_VAR_DEVINIT_SESSION=1</code> (foreground only) 	Prompts the end-user to confirm or reject the installation	<code>DMA_MSG_SCOMO_ACCEPT</code> <code>DMA_MSG_SCOMO_POSTPONE</code> <code>DMA_MSG_SCOMO_CANCEL</code>
ScomoInstallDone	<code>DMA_MSG_SCOMO_INSTALL_DONE</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> (foreground only)	Notifies the end-user that the installation is complete	

Class	Handles DIL Event	Description	Sends BL Events
ScomInstallProgress	<code>DMA_MSG_SCOMO_INSTALL_PROGRESS_UI</code> if <code>DMA_VAR_SCOMO_QUIET=0</code> (foreground only)	Displays installation progress	
ScomPostponeConfirm	<code>DMA_MSG_SCOMO_POSTPONE_STATUS_UI</code>	<p>Notifies the end-user that a download (if <code>DMA_VAR_DURING_DL</code> is True) or installation (if <code>DMA_VAR_DURING_DL</code> is False) will start in <code>DMA_VAR_SCOMO_POSTPONE_PERIOD</code> minutes.</p>	

5.1.4 redbend/src/com/redbend/client/uialerts

Table 5-4: UI Alerts Classes

Class	Handles DIL Event	Description	Sends BL Events
BatteryLowNotificationHandler	<code>DMA_MSG_SCOMO_INSTALL_CHARGE_BATTERY_UI</code> (background only)	<p>Notifies the end-user that the update was interrupted by a low battery</p> <p>After user-confirmation, the business logic checks the battery level using <code>GetBatteryLevelHandler</code>. If the battery level is too low, this class handles background notification while the class <code>BatteryLow</code> handles foreground notification.</p>	

Class	Handles DIL Event	Description	Sends BL Events
ConfirmUIAlert	<code>DMA_MSG_UI_ALERT</code> if <code>DMA_VAR_UI_ALERT_TYPE=DMA_UI_ALERT_TYPE_CONFIRMATION</code> (foreground only)	<p>Displays confirmation message screen</p> <p>The screen displays the message <code>DMA_VAR_UI_ALERT_TEXT</code> and closes when the end-user taps OK or Cancel or after <code>DMA_VAR_UI_ALERT_MAXDT</code> seconds have passed</p> <p>The preselected value, if any, is defined by the value in <code>DMA_VAR_UI_ALERT_DEFAULT_CMD</code></p> <p>Used during self-registration</p>	<ul style="list-style-type: none"> <code>DMA_MSG_USER_ACCEPT</code> or <code>DMA_MSG_USER_CANCEL</code> if the end-user closes the message <code>DMA_MSG_TIMEOUT</code> if the message times out
DdTextHandler		<p>Parses the description field of a Download Descriptor</p> <p>The default description is defined in <code>DMA_VAR_FUMO_DP_DESCRIPTION</code></p>	
DownloadCompleteNotificationHandler	<code>DMA_MSG_SET_DL_COMPLETE_D_ICON</code>	Displays a download complete notification	
DownloadConfirmationNotificationHandler	<code>DMA_MSG_SCOMO_NOTIFY_DL_UI</code> if <code>DMA_VAR_SCOMO_QUIET=0</code> (background only)	Displays a notification prompting the end-user to that a download is available	<code>DMA_MSG_SCOMO_NOTIFY_DL</code>

Class	Handles DIL Event	Description	Sends BL Events
InfoUIAlert	<code>DMA_MSG_UI_ALERT</code> if <code>DMA_VAR_UI_ALERT_TYPE=DMA_UI_ALERT_TYPE_INFO</code> (foreground only)	Displays an information message screen The screen displays the message <code>DMA_VAR_UI_ALERT_TEXT</code> and closes when the end-user taps OK or after <code>DMA_VAR_UI_ALERT_MAXDT</code> seconds have passed	<ul style="list-style-type: none"> <code>DMA_MSG_USER_ACCEPT</code> if the end-user closes the message <code>DMA_MSG_TIMEOUT</code> if the message times out
InputUIAlert	<code>DMA_MSG_UI_ALERT</code> if <code>DMA_VAR_UI_ALERT_TYPE=DMA_UI_ALERT_TYPE_INPUT</code> (foreground only)	Displays a text entry message screen The screen displays an input field prefilled with the text value <code>DMA_VAR_UI_ALERT_TEXT</code> and closes when the end-user taps OK or after <code>DMA_VAR_UI_ALERT_MAXDT</code> seconds have passed Used during self-registration	<ul style="list-style-type: none"> <code>DMA_MSG_USER_ACCEPT</code> or <code>DMA_MSG_USER_CANCEL</code> if the end-user closes the message <code>DMA_MSG_TIMEOUT</code> if the message times out
InstallConfirmNotificationHandler	<code>DMA_MSG_SCOMO_INSTALL_CONFIRM_UI</code> if device is initiated and in non-silent mode (background only)	Displays an installation confirmation notification	
InstallDoneNotificationHandler	<code>DMA_MSG_SCOMO_INSTALL_DONE</code> (background only)	Displays an installation complete notification The installation result is defined in <code>DMA_VAR_SCOMO_RESULT</code>	
InterruptionNotificationHandler	<ul style="list-style-type: none"> <code>DMA_MSG_DNLD_FAILURE</code> if device is in non-silent mode (background only) <code>DMA_MSG_SCOMO_DL_CANCELLED_UI</code> (background only) 	Displays a notification that the download was interrupted	
LawmoLockResultNotification	<code>DMA_MSG_LAWMO_LOCK_RESULT_SUCCESS</code>	Displays a notification that the device was locked	

Class	Handles DIL Event	Description	Sends BL Events
LawmoUnlockResultNotification	<code>DMA_MSG_LAWMO_UNLOCK_RESULT_SUCCESS</code>	Displays a notification that the device was unlocked	
NotificationHandlerBase		Base class for: <ul style="list-style-type: none"> InstallDoneNotificationHandler ProgressNotificationHandler 	
ProgressNotificationHandler	<ul style="list-style-type: none"> <code>DMA_MSG_SCOMO_DL_PROGRESS</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> (background only) <code>DMA_MSG_SCOMO_INSTALL_PROGRESS</code> if <code>DMA_VAR_SCOMO_ISSILENT=0</code> (background only) 	<p>Displays a download progress notification, or handles a download failure or cancelation</p> <p>The current progress value is defined in <code>DMA_VAR_DL_PROGRESS</code> or <code>DMA_VAR_INSTALL_PROGRESS</code></p>	

5.1.5 swm_common/src/com/redbend/swm_common

Table 5-5: SWM Common Classes

Class	Handles DIL Event	Description	Sends BL Events
DmcDeviceAdminReceiver		Required for the DM operations related to LAWMO factory reset and lock operations	
GcmHandler	<code>DMA_MSG_GCM_REGISTRATION_DATA</code> <code>DMA_MSG_GCM_UNREGISTRATION_DATA</code>	<p>Parses GCM notification and, if required, sends a request for registration to GcmReceiver</p> <p>Implemented in <code>RBGcmHandler.java</code></p>	

Class	Handles DIL Event	Description	Sends BL Events
GcmReceiver		Manages GCM registration Receives GCM messages from Google	<ul style="list-style-type: none"> • DMA_MSG_NET_NOTIFICATION on GCM message. • DMA_MSG_NET_NOTIFICATION_REGIST on new registration • DMA_MSG_NET_NOTIFICATION_UNREGIST on unregistration
SmmCommonConstants		Sets some SCOMO constants	

5.2 Event Streamer Overview

The Event Streamer includes the functions required to encapsulate and send events between the two sub-layers of the Application Layer. The Event Streamer includes the functions on both sides that, when required, serialize and transport events between the two layers.

In Red Bend's Android implementation, the BLL is launched as a sub-thread of the DIL application. The DIL is written in platform-dependent Java while the BLL is written in native C. Using JNI, the DIL launches a separate thread that manages the BLL. This thread launches the BLL and provides a callback for returned DIL events.

To pass a BL event to the BLL, the DIL serializes the BL event and uses JNI to pass the serialized BL event to the BLL thread, which de-serializes the BL event and passes it to the BLL. To pass a DIL event to the DIL, the BLL invokes a callback that serializes the DIL event and calls Java using JNI. The DIL de-serializes the DIL event for processing.

The DIL and BLL can be joined using a variety of other implementations that are not covered in this guide.

5.2.1 Event Streamer Deliverables

The following files contain code associated with the Event Streamer:

- **Event.java:** Java event class. Includes (de)serialization of events.
- **EventVar.java:** Java event variables class. Includes (de)serialization of event variables.
- **BasicService.java:** Java class that sends serialized BL events to the BLL and receives serialized DIL events using JNI. This class receives DIL events from the BLL, wraps them inside Android intents, and broadcasts them to `ClientService.java`. In the reverse direction, this class receives intents from `ClientService.java`, unwraps the events, and passes them to the BLL.
- **dma_jni.c:** JNI functions that receive, de-serialize, and send BL events from the DIL to the BLL (using **dma_sm_exec.c**), and receive, serialize, and send DIL events from the BLL to the DIL (using a callback).

Events and the methods used to act on them are defined by the classes **Event.java** and **EventVar.java**.

5.3 Initializing and Terminating the BLL

The DIL must initialize and terminate the SMM of the BLL. The initialization is in **BasicService.java**, as follows:

```
initEngine(filesDir);
startSmm(deviceId, userAgent, deviceModel, deviceManufacturer);
```

5.3.1 Event Streamer Initialization

The SWM DM Client initializes the BLL side of the Event Streamer by calling the following function, passing a callback to process DIL events.

```
typedef void (VDM_SMM_sendUIEventFunc)(VDM_SMM_event_t* event_name);
int VDM_SMM_init(DMA_sendUIEventFunc sendFunc);
```

The DIL includes the classes `Event` and `EventVar` used to process the events in Java.

5.3.2 Device Status Update

On initialization, the DIL sends to the BLL the current network state using the following BL events:

- `DMA_MSG_STS_MOBILE_DATA` (in `ConnectivityStateChangeReceiver`)
- `DMA_MSG_STS_ROAMING` (in `ClientService`)

For more information on these events, see *Business Logic Events*.

5.4 Sending and Receiving Events

The following sections present an overview of how to send and receive events using the Event Streamer.

5.4.1 Sending BL Events from the DIL

The DIL must first construct the event. An event has a name and associated variables that are passed with the event. Variables include a variable name, integer value or string value.

```
Event event      = new Event(event_name) // Construct the event

EventVar var     = new EventVar(event_name,intval) // or
EventVar var     = new EventVar(event_name,strval)
                                     // Create event variable value

event_name.addVar(var)               // Add variable/value to event
```

The method used to send the event depends on the use case:

- To send an event originating from the Android platform, use `SmmReceive.sendEvent`:

```
SmmReceive sendEvent(context, class, event_name)
```

Where `context` is the context of the receiver required for transmitting the event, `class` is the class of the associated service (`ClientService`) that is required for transmission, and `event` is the event.

- To send an event originating from end-user input in a UI screen, use `DmActivity.sendEvent`:
`DmActivity sendEvent(event_name)`
- To send an event originating from a UI screen process, but not as a result of end-user interaction, use `SmmService.sendEvent`:
`SmmService sendEvent(event_name)`

The class that passes the BL event using JNI is a private method `SmmService.ipcSendEvent`. The non-serialized BL event (which is encapsulated in the `Event` class) is passed to the public method `SmmService.sendEvent`, which serializes the BL event and then calls the native `ipcSendEvent`. Using JNI, this executes the native function `Java_com_redbend_app_SmmService_ipcSendEvent` (in `dma_jni.c`).

5.4.2 Receiving DIL Events in the DIL

To handle DIL events, Red Bend's DIL first registers a handler for the event in `ClientService.eventHandlersRegister` as follows:

```
// Construct the event
Event event = new Event(event_name)
EventVar var = new EventVar(event_name,intval) // or
EventVar var = new EventVar(event_name,strval)
event_name.addVar(var)
...

// Define the handler:
// To handle a certain event by displaying an activity on the UI screen
// use:
event_handler = new EventHandlerIntent(context, ActivityClass.class);

// If you have a class FooHandler that implements the EventHandler
// interface, use:
event_handler = new FooHandler(...);

// Register the handler
registerHandler(flowId, event_name, ui_mode, event_handler);
```

Where `flowId` is a number representing a logical flow (all handlers belonging to the same logical flow have the same flow ID) and `ui_mode` is one of:

- `UI_MODE_FOREGROUND`: This handler will only handle events when the application is in the foreground.
- `UI_MODE_BACKGROUND`: This handler will only handle events when the application is in the background.
- `UI_MODE_BOTH_FG_AND_BG`: This handler will handle events regardless of whether the application is in the foreground or background.

The Java method `SmmService.recvEvent` de-serializes and handles the DIL event. The DIL event is de-serialized using the `Event` constructor that receives a byte array as a parameter.

5.5 Events

This section lists the events that must be sent by the DIL when required and handled by the DIL when received.

This section presents the BL and DIL events used by the business logic. The DIL must send the BL events and receive the DIL events listed in this section at the appropriate times in the business logic flow.

These events and events variables are no longer located in some files. Use the events or event variables by event name or event variable name directly.



NOTE: Beside the BL events and DIL events listed in the following tables, the application uses internal events for communication between the different parts of the BL. You may safely ignore these events if they appear in the logs.

5.5.1 Business Logic Events

The DIL must send the following BL events as described in the **Description** column for each event.

You can ignore events that are irrelevant to your specific implementation.

Table 5-6: BL Events

BL Event	Variables	Description
DMA_MSG_AUTO_SELF_REG_INFO	DMA_VAR_AUTO_SELF_REG_DOMAIN_NAME DMA_VAR_AUTO_SELF_REG_DOMAIN_PIN	The device is self-registering with the specified domain name and PIN.
DMA_MSG_BATTERY_LEVEL	DMA_VAR_BATTERY_LEVEL	The battery level; sent in response to DMA_MSG_GET_BATTERY_LEVEL .
DMA_MSG_DL_TIMESLOT_TIMEOUT		The timer set in DMA_MSG_SCOMO_SET_DL_TIMESLOT_DONE triggered.
DMA_MSG_LAWMO_LOCK_ENDED_FAILURE		The fully lock operation ended unsuccessfully.
DMA_MSG_LAWMO_LOCK_ENDED_SUCCESS		The fully lock operation ended successfully.
DMA_MSG_LAWMO_UNLOCK_ENDED_FAILURE		The unlock operation ended unsuccessfully.
DMA_MSG_LAWMO_UNLOCK_ENDED_SUCCESS		The unlock operation ended successfully.

BL Event	Variables	Description
DMA_MSG_LAWMO_WIPE_AGENT_ENDED_FAILURE		The wipe data operation ended unsuccessfully.
DMA_MSG_LAWMO_WIPE_AGENT_ENDED_SUCCESS		<p>The wipe data operation ended successfully.</p> <p>NOTE: The event is not sent from the BLL in this version of the SWM Client – this is a known limitation.</p>
DMA_MSG_NET_NIA	DMA_VAR_NIA_MSG DMA_VAR_NIA_ENCODED: <ul style="list-style-type: none"> • TRUE(1): Message is hex encoded • FALSE(0): Message is ASCII 	<p>An NIA was received from the DM Server.</p> <p>The BLL checks whether to start a server-initiated session.</p>
DMA_MSG_NET_NOTIF_REGIST	DMA_VAR_NOTIF_TYPE: The notification type, which is always GCM, Google cloud messaging DMA_VAR_NOTIF_REG_ID: GCM registration ID.	Register the device's preferred notification type and, if the type is GCM, indicate the device's GCM registration ID.
DMA_MSG_NET_NOTIF_UNREGIST		Unregister the device from its current preferred notification type.
DMA_MSG_NET_NOTIFICATION	DMA_VAR_FORCE_CANCEL: <ul style="list-style-type: none"> • TRUE (1): Cancel an in-progress DM or Download session before initiating a new DM session. • FALSE (0): Don't cancel an in-progress DM or Download session. 	Trigger a session to register the device's preferred notification type.

BL Event	Variables	Description
DMA_MSG_PRODUCT_TYPE	DMA_VAR_PRODUCT_TYPE: <ul style="list-style-type: none"> PRODUCT_SWMC_SYSTEM (0): The SWM Client came pre-installed PRODUCT_SWMC_DOWNLOADABLE (1): The SWM Client was installed (downloaded) by the end-user. 	Sends the SWM Client type; this is sent during initialization.
DMA_MSG_SCOMO_ACCEPT		The end-user confirmed a download or an installation when prompted to confirm.
DMA_MSG_SCOMO_CANCEL		The end-user rejected a download in progress or canceled a download when prompted to confirm.
DMA_MSG_SCOMO_DEVICE_POLLING_CHANGE	DMA_VAR_SCOMO_DEVICE_POLLING_INTERVAL <ul style="list-style-type: none"> 0: Periodic checking for updates is disabled Non-zero integer value: Number of hours between periodic session initiations 	The end-user requested to change the interval of periodic session initiations (device-initiated sessions).
DMA_MSG_SCOMO_DEVICE_POLLING_DISABLE		The end-user requested to disable periodic session initiations (device-initiated sessions).
DMA_MSG_SCOMO_DEVICE_POLLING_ENABLE		The end-user requested to enable periodic session initiations (device-initiated sessions).
DMA_MSG_SCOMO_INSTALL_CANNOT_PROCEED		Following a business logic-requested reboot, the installation could not be performed.
DMA_MSG_SCOMO_INSTALL_COMPLETED	DMA_VAR_SCOMO_INSTALL_COMPLETED_RESULT	An APK installation completed
DMA_MSG_SCOMO_NOTIFY_DL		The end-user pressed the “download available” notification. See DMA_MSG_SCOMO_NOTIFY_DL_UI .
DMA_MSG_SCOMO_POSTPONE		The end-user confirmed the postponement of a download or installation.

BL Event	Variables	Description
DMA_MSG_SCOMO_REMOVE_COMP_RESULT	DMA_VAR_SCOMO_REMOVE_COMP_RESULT	An APK uninstallation completed
DMA_MSG_SCOMO_SET_DL_TIMESLOT_DONE		The DIL set the timer to trigger in the next available time slot; sent in response to DMA_MSG_SCOMO_SET_DL_TIMESLOT .
DMA_MSG_SCOMO_VERIFICATION_FAILURE		Following a business logic-requested reboot, authentication of the configuration file failed.
DMA_MSG_SESS_INITIATOR_USER_SCOMO		<p>The end-user is attempting to check for updates.</p> <p>The BLL checks whether to start a user-initiated session.</p>
DMA_MSG_STS_MOBILE_DATA	DMA_VAR_STS_IS_MOBILE_DATA_CONNECTED: <ul style="list-style-type: none"> • TRUE (1) • FALSE (0) 	The device's connection to a network has changed.
DMA_MSG_STS_POWERED		The device was powered on.
DMA_MSG_STS_ROAMING	DMA_VAR_STS_IS_ROAMING: <ul style="list-style-type: none"> • TRUE (1) • FALSE (0) 	The device's roaming status has changed.
DMA_MSG_STS_VOICE_CALL_START		A voice call is now in progress.
DMA_MSG_STS_VOICE_CALL_STOP		A voice call has now completed.
DMA_MSG_STS_WIFI	DMA_VAR_STS_IS_WIFI_CONNECTED: <ul style="list-style-type: none"> • TRUE (1) • FALSE (0) 	The device's connection to a Wi-Fi point has changed.
DMA_MSG_TIMEOUT		The end-user failed to respond within the time limit when prompted.
DMA_MSG_USER_ACCEPT	DMA_VAR_UI_ALERT_TEXT	<p>The end-user tapped OK on a screen.</p> <p>The variable is included when the end-user was asked to enter text on an Input screen (DMA_UI_ALERT_TYPE_INPUT).</p>

BL Event	Variables	Description
DMA_MSG_USER_CANCEL		The end-user tapped Cancel on a screen.

5.5.2 Device Integration Layer Events

The DIL must handle the following DIL events received from the BLL as described in the **Description** column for each event. Most DIL events are used to update the device UI; some are used to retrieve device information, such as set a timer.

You can ignore events that are irrelevant to your specific implementation. For example, if the device has no UI, you may ignore all of these events.

Table 5-7: DIL Events

DIL Event	Variables	Description
DMA_MSG_DL_INST_ERROR_UI	DMA_VAR_ERROR: <ul style="list-style-type: none"> ERR_TYPE_USER_INTERACTION_TIMEOUT 	Display a screen indicating that there was a user interaction timeout.
DMA_MSG_DM_ABORTED_UI		Display a screen indicating that: <ul style="list-style-type: none"> There was a problem connecting to the server, or The end-user canceled when trying to connect to the server.
DMA_MSG_DM_COMPLETED_UI		DM session completed.
DMA_MSG_DM_ERROR_UI	DMA_VAR_ERROR: <ul style="list-style-type: none"> ERR_TYPE_DM_NO_PKG ERR_TYPE_FLOW_IN_PROGRESS ERR_TYPE_DM_GENERAL ERR_TYPE_ROAMING_OR_EMERGENCY ERR_TYPE_DM_NETWORK 	Display a screen indicating that there was a DM session error: <ul style="list-style-type: none"> No update was found. Cannot check for updates while another session is in progress. Could not trigger a check for updates, or the device was powered on while checking, or the end-user canceled the check while it was in progress. Could not trigger DM as roaming. Could not trigger DM as no network (for example, in airplane mode).
DMA_MSG_DM_STARTED_UI		DM session started.

DIL Event	Variables	Description
DMA_MSG_DNLD_FAILURE	DMA_VAR_ERROR: <ul style="list-style-type: none"> ERR_TYPE_DL_NETWORK ... DMA_VAR_DP_SIZE DMA_VAR_FUMO_DP_VENDOR DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_SCOMO_DP_X DMA_VAR_NETWORK_UI_REASONS DMA_VAR_USER_INIT <ul style="list-style-type: none"> TRUE: User-initiated FALSE: Not user-initiated 	Display a screen indicating that, before or during a download: <ul style="list-style-type: none"> The device is not within network range, or The device is not within range of a Wi-Fi point and a Wi-Fi point is required, or There was some other kind of error with the download.
DMA_MSG_GCM_REGISTRATION_DATA	DMA_VAR_NOTIF_TYPE: The notification type, which is always GCM, Google cloud messaging DMA_VAR_NOTIF_REG_ID: GCM registration ID. DMA_VAR_SENDER_ID: The GCM sender ID	Register with Google cloud messaging
DMA_MSG_GCM_UNREGISTRATION_DATA	DMA_VAR_NOTIF_TYPE: The notification type, which is always GCM, Google cloud messaging DMA_VAR_NOTIF_REG_ID: GCM registration ID. DMA_VAR_SENDER_ID: The GCM sender ID	Unregister with Google cloud messaging
DMA_MSG_GET_BATTERY_LEVEL		Return the current battery level The DIL should return DMA_MSG_BATTERY_LEVEL .
DMA_MSG_LAWMO_LOCK_LAUNCH		Start a fully lock operation.

DIL Event	Variables	Description
DMA_MSG_LAWMO_LOCK_RESULT_FAILURE	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the fully lock operation failed.
DMA_MSG_LAWMO_LOCK_RESULT_SUCCESS	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the fully lock operation succeeded.
DMA_MSG_LAWMO_UNLOCK_LAUNCH		Start an unlock operation.
DMA_MSG_LAWMO_UNLOCK_RESULT_FAILURE	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the unlock operation failed.
DMA_MSG_LAWMO_UNLOCK_RESULT_SUCCESS	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the unlock operation succeeded.
DMA_MSG_LAWMO_WIPE_AGENT_LAUNCH	DMA_VAR_LAWMO_WIPE_LIST	Start a wipe data operation.
DMA_MSG_LAWMO_WIPE_RESULT_FAILURE	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the wipe data operation failed.
DMA_MSG_LAWMO_WIPE_RESULT_NOT_PERFORMED	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the wipe data operation was not performed.
DMA_MSG_LAWMO_WIPE_RESULT_SUCCESS	DMA_VAR_LAWMO_RESULT	Display a screen indicating that the wipe data operation succeeded.
DMA_MSG_SCOMO_DL_CANCELED_UI	DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_SCOMO_DP_X	Display a screen indicating that the end-user rejected a download in progress or canceled a download when prompted to confirm.

DIL Event	Variables	Description
DMA_MSG_SCOMO_DL_CONFIG_UI	DMA_VAR_DP_SIZE DMA_VAR_FUMO_DP_VENDOR DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_SCOMO_DP_X DMA_VAR_SCOMO_CRITICAL: <ul style="list-style-type: none"> 0: Not a critical update 1: Critical update DMA_VAR_SCOMO_MODE: <ul style="list-style-type: none"> SCOMO_MODE_DEFAULT SCOMO_MODE_USER DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> 0: Not a silent update 1: Silent update 	Display a screen to prompt the end-user to confirm a download. If a silent update, return DMA_MSG_USER_ACCEPT without displaying a screen.
DMA_MSG_SCOMO_DL_INIT	DMA_VAR_SCOMO_MODE: <ul style="list-style-type: none"> SCOMO_MODE_DEFAULT SCOMO_MODE_USER 	If not a silent update, display a screen indicating that the download is starting.

DIL Event	Variables	Description
DMA_MSG_SCOMO_DL_PROGRESS	DMA_VAR_DL_PROGRESS DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_SCOMO_DP_X DMA_VAR_SCOMO_MODE: <ul style="list-style-type: none"> SCOMO_MODE_DEFAULT SCOMO_MODE_USER DMA_VAR_SCOMO_CRITICAL: <ul style="list-style-type: none"> 0: Not a critical update 1: Critical update DMA_VAR_IS_FUMO_OPERATION: <ul style="list-style-type: none"> 0: SCOMO operation 1: FOTA operation DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> 0: Not a silent update 1: Silent update 	If not a silent update, display a screen indicating the download progress.
DMA_MSG_SCOMO_DL_SUSPEND_UI	DMA_VAR_SCOMO_DOWNLOAD_TIME_SECONDS	Display a screen indicating the download has been suspended.
DMA_MSG_SCOMO_INS_CHARGE_BATTERY_UI	DMA_VAR_BATTERY_THRESHOLD DMA_VAR_SCOMO_ISSILENT	If the battery level is not above or equal the threshold, and this is not a silent update, display a screen indicating that the end-user needs to charge the battery.

DIL Event	Variables	Description
DMA_MSG_SCOMO_INS_CONFIRM_UI	<p>DMA_VAR_INS_CONFIRM_TIMER_SECONDS: Screen timeout in seconds</p> <p>DMA_VAR_SCOMO_CRITICAL:</p> <ul style="list-style-type: none"> 0: Not a critical update 1: Critical update <p>DMA_VAR_FUMO_DP_DESCRIPTION</p> <p>DMA_VAR_SCOMO_ISSILENT:</p> <ul style="list-style-type: none"> 0: Not a silent update 1: Silent update <p>DMA_VAR_DP_INFO_URL</p> <p>DMA_VAR_SCOMO_MODE:</p> <ul style="list-style-type: none"> SCOMO_MODE_DEFAULT SCOMO_MODE_USER <p>DMA_VAR_IS_POSTPONE_ENABLED:</p> <ul style="list-style-type: none"> 0: End-user cannot postpone 1: End-user can postpone <p>DMA_VAR_SCOMO_IS_NEED_REBOOT:</p> <ul style="list-style-type: none"> 0: Device will not reboot 1: Device must reboot 	<p>Display a screen to prompt the end-user to confirm an installation.</p> <p>The screen does not provide an option to cancel the installation.</p> <p>If a silent update, return DMA_MSG_USER_ACCEPT without displaying a screen.</p>
DMA_MSG_SCOMO_CANCEL_COMP_REQUEST		<p>Cancel a request to install or uninstall an APK.</p> <p>Typically sent as a result of a timeout.</p>
DMA_MSG_SCOMO_INSTALL_COMP_REQUEST	<p>DMA_VAR_SCOMO_COMP_FILE:</p> <p>The file name</p>	Install an APK.

DIL Event	Variables	Description
<code>DMA_MSG_SCOMO_INSTALL_DONE</code>	DMA_VAR_SCOMO_RESULT DMA_VAR_OPERATION_TYPE: <ul style="list-style-type: none"> 0: SCOMO operation 1: FOTA operation DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_SCOMO_DC_ID: package name DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> 0: Not a silent update 1: Silent update 	If not a silent update, display a screen indicating that an installation has completed.
<code>DMA_MSG_SCOMO_INSTALL_PROGRESS_UI</code>	DMA_VAR_SCOMO_INSTALL_PROGRESS DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> 0: Not a silent update 1: Silent update DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_OPERATION_TYPE: <ul style="list-style-type: none"> 0: SCOMO operation 1: FOTA operation DMA_VAR_DP_SIZE	Display a screen indicating the installation progress.

DIL Event	Variables	Description
<code>DMA_MSG_SCOMO_NOTIFY_DL_UI</code>	DMA_VAR_FUMO_DP_DESCRIPTION DMA_VAR_DP_INFO_URL DMA_VAR_SCOMO_MODE: <ul style="list-style-type: none"> • SCOMO_MODE_DEFAULT • SCOMO_MODE_USER DMA_VAR_SCOMO_CRITICAL: <ul style="list-style-type: none"> • 0: Not a critical update • 1: Critical update DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> • 0: Not a silent update • 1: Silent update 	If not a silent update, display a screen indicating that an update is available to download and install. The update was initiated by the server.
<code>DMA_MSG_SCOMO_POSTPONE_STATUS_UI</code>	DMA_VAR_SCOMO_POSTPONE_PERIOD DMA_VAR_DURING_DL: <ul style="list-style-type: none"> • 0: Not during download • 1: During download 	Display a screen indicating the postpone status.
<code>DMA_MSG_SCOMO_REBOOT_REQUEST</code>	DMA_VAR_SCOMO_ISSILENT: <ul style="list-style-type: none"> • 0: Not a silent update • 1: Silent update DMA_VAR_SCOMO_UPDATE_CONFIG_FILE: <ul style="list-style-type: none"> • If the variable exists, it contains the path to the signed configuration zip file that is passed to GOTA 	If the update is not silent, display a screen indicating that the update will continue after a reboot.
<code>DMA_MSG_SCOMO_REMOVE_COMPONENT_REQUEST</code>	DMA_VAR_SCOMO_COMP_NAME: The APK	Uninstall an APK.
<code>DMA_MSG_SCOMO_SET_DL_TIMER_SLOT</code>	DMA_VAR_SCOMO_DOWNLOAD_TIMER_SECONDS	Set the timer indicating the time (in seconds) after which to start the download.
<code>DMA_MSG_SET_DL_COMPLETED_ICON</code>		Display a notification that the download has completed.

DIL Event	Variables	Description
DMA_MSG_UI_ALERT	<p>DMA_VAR_UI_ALERT_TYPE:</p> <ul style="list-style-type: none"> DMA_UI_ALERT_TYPE_INFO (1) DMA_UI_ALERT_TYPE_CONFIRMATION (2) DMA_UI_ALERT_TYPE_INPUT (3) DMA_UI_ALERT_TYPE_CHOICE (4) <p><i>For DMA_UI_ALERT_TYPE_INFO, additional variables include:</i></p> <p>DMA_VAR_UI_ALERT_TEXT</p> <p>DMA_VAR_UI_ALERT_MINDT</p> <p>DMA_VAR_UI_ALERT_MAXDT</p> <p><i>For DMA_UI_ALERT_TYPE_CONFIRMATION, additional variables include:</i></p> <p>DMA_VAR_UI_ALERT_TEXT</p> <p>DMA_VAR_UI_ALERT_MINDT</p> <p>DMA_VAR_UI_ALERT_MAXDT</p> <p>DMA_VAR_UI_ALERT_DEFAULT_CMD</p> <p><i>For DMA_UI_ALERT_TYPE_INPUT, additional variables include:</i></p> <p>DMA_VAR_UI_ALERT_TEXT</p> <p>DMA_VAR_UI_ALERT_MINDT</p> <p>DMA_VAR_UI_ALERT_MAXDT</p> <p>DMA_VAR_UI_ALERT_DEFVAL</p> <p>DMA_VAR_UI_ALERT_MAXLEN</p> <p>DMA_VAR_UI_ALERT_INPUTTYPE</p> <p>DMA_VAR_UI_ALERT_ECHOTYPE</p> <p><i>For</i></p>	<p>Display a screen of some kind to the end-user.</p> <ul style="list-style-type: none"> Information (OK button only) Confirmation (OK and Cancel) Text input (input field, OK, and Cancel) Selection (single or multiple options, OK, and Cancel) <p>All screens include heading text and a minimum and maximum time for display.</p> <p>The confirmation screen includes a default selection.</p> <p>The text input screen has a default text, indicates data and echo type and has a maximum length.</p> <p>The selection screen has default selections and indicates if multiple options may be selected.</p>

DIL Event	Variables	Description
	<i>DMA_UI_ALERT_TYPE_CHOICE,</i> <i>additional variables include:</i> DMA_VAR_UI_ALERT_TEXT DMA_VAR_UI_ALERT_MINDT DMA_VAR_UI_ALERT_MAXDT DMA_VAR_UI_ALERT_ITEMS DMA_VAR_UI_ALERT_ITEMSCOUNT DMA_VAR_UI_ALERT_DEFSEL DMA_VAR_UI_ALERT_MULTIPLE	
<i>DMA_MSG_USER_SESSION_TRIGGERED</i>		User triggered a new DM session, display a screen indicating that the checking for new updates.

5.6 Automatic Self-Registration

The DIL can be set to automatically self-register the device to a domain in `ClientService.onCreate`. Default registration credentials (domain name and PIN) can be set in the DM Tree.

To change the credentials without having rebuild the client, the DIL also retrieves credentials by calling `Ipl.iplGetAutoSelfRegDomainInfo`, passing an empty array to store the credentials. This function reads the information from the file `<sdcard_directory>/private/Credentials.txt`. The first line in the file must be the domain name, and the second line must be the PIN.

After retrieving the self-registration information, the DIL sends the BL event `DMA_MSG_AUTO_SELF_REG_INFO` with the registration information to the BLL. The BLL updates the DM Tree with the credentials. If `Credentials.txt` does not exist, then the SWM client sends the default values in the DM tree.



NOTE: The self-registration information is retrieved only on client service creation, meaning that the device must be rebooted after the file is pushed (or the application process can be killed).

5.7 Silent Update and Installation

The DIL handles DIL events marked for *silent installation* (`DMA_VAR_SCOMO_ISSILENT=1`) without notifying the end-user. The DIL must return an "accept" BL event on behalf of the end-user

(DMA_MSG_USER_ACCEPT) when required, as if the end-user had pressed **OK** on a notification screen.

For example, the following code snippets from `ClientService.eventHandlersRegister()` define two of the event handlers.

- **Non silent install:** If the DIL event `DMA_MSG_SCOMO_DL_CONFIRM_UI` is received, **and** it is **not** marked for silent installation (`DMA_VAR_SCOMO_ISSILENT=0`), then handle the event with `ScomConfirm`.

```
registerHandler(
    1,
    new Event("DMA_MSG_SCOMO_DL_CONFIRM_UI").addVar(new
        EventVar("DMA_VAR_SCOMO_ISSILENT", 0)),
    UI_MODE_BOTH_FG_AND_BG,
    new EventHandlerIntent(this, ScomConfirm.class)
);
```

- **Silent install:** If the DIL event `DMA_MSG_SCOMO_DL_CONFIRM_UI` is received, **and** it is marked for silent installation (`DMA_VAR_SCOMO_ISSILENT=1`), then return `DMA_MSG_SCOMO_ACCEPT` on behalf of the end-user.

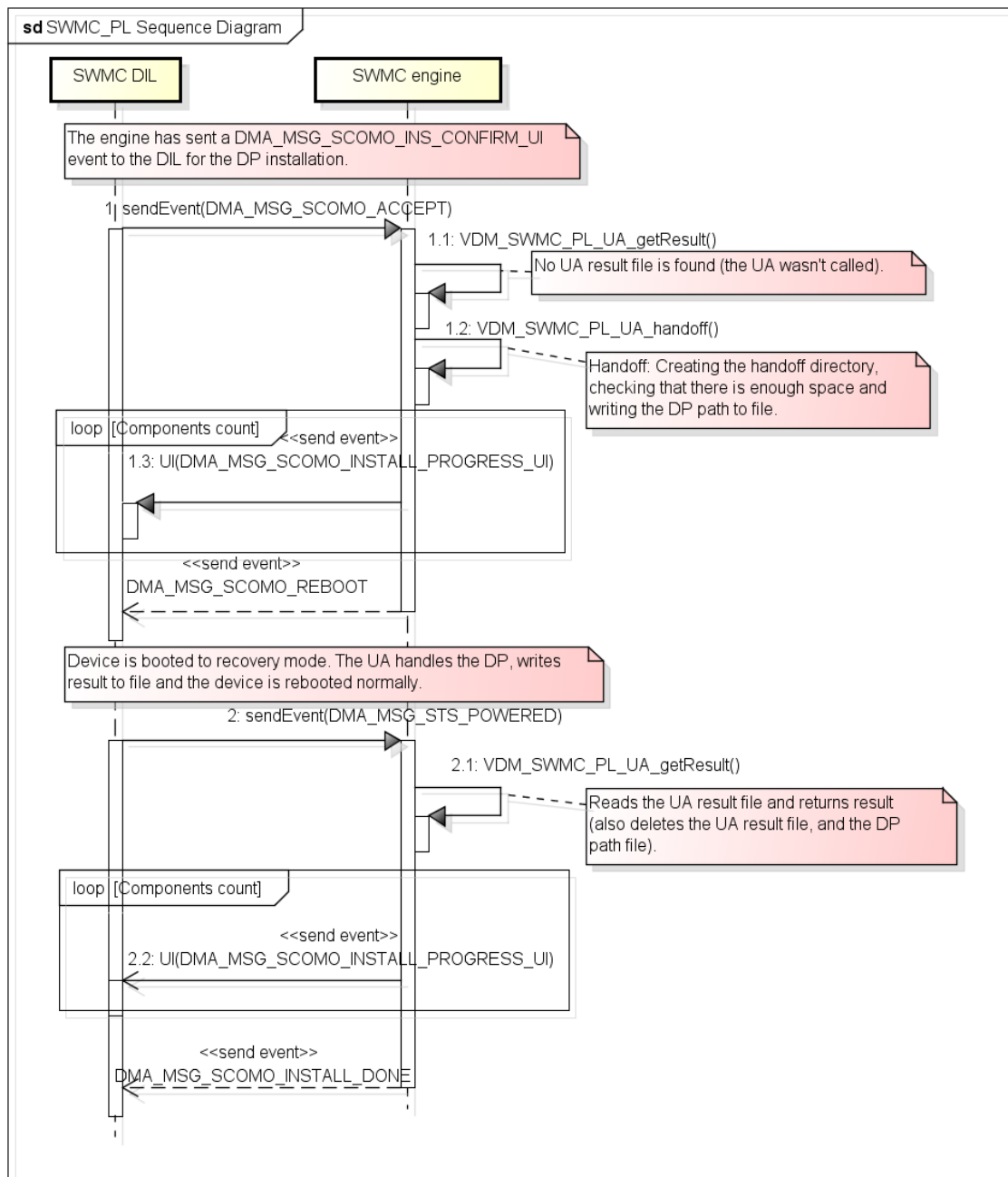
```
registerHandler(
    1,
    new Event("DMA_MSG_SCOMO_DL_CONFIRM_UI").addVar(new
        EventVar("DMA_VAR_SCOMO_ISSILENT", 1)),
    UI_MODE_BOTH_FG_AND_BG,
    new EventHandler(this) {
        @Override
        protected void genericHandler(Event ev) {
            sendEvent(new Event("DMA_MSG_SCOMO_ACCEPT"));
        }
    }
);
```

To change this functionality, remove the second event handler and the variable requirement (`.addVar(new EventVar("DMA_VAR_SCOMO_ISSILENT", 0))`) in the first handler.

Do this for all locations where alternate handling is performed for `DMA_VAR_SCOMO_ISSILENT`.

5.8 UA Handoff Flow

The following diagram presents the flow as the SWM DM Client hands off to, and reads results from, the UA. The process requires you to implement the two Porting Layer functions described in *UA Porting Layer Functions*.

**Figure 5-2: UA Handoff Flow**

In the above diagram, *SWMC* is the SWM DM Client, which contains the *DIL* and the *engine* (the *BLL* and *vDM* Extended Framework).

The flow is as follows:

- 1 After the end-user confirms the installation, the DIL sends the BL event `DMA_MSG_SCOMO_ACCEPT` to the BLL.
 - a The engine checks if the installation was already performed by looking for a result file (`VDM_SWMC_PL_UA_getResult`).

- b If a result file was not found, the engine writes information to a handoff directory, so that the UA can read the information, and invokes the UA (`VDM_SWMC_PL_UA_handoff`). The information includes the path to the downloaded package to install.
 - c While the installation proceeds, the engine provides progress information to the DIL by repeatedly sending the DIL event `DMA_MSG_SCOMO_INSTALL_PROGRESS_UI`. Eventually the engine sends the DIL event `DMA_MSG_SCOMO_REBOOT_REQUEST` to the DIL asking the DIL to reboot the device to recovery mode to complete the installation. After rebooting, the device reboots again into standard mode.
- 2 After a reboot into standard mode, the DIL sends the BL event `DMA_MSG_STS_POWERED` to the BLL.
 - a The state machine knows that the device is in the middle of an installation, so the engine checks for the results of the installation (`VDM_SWMC_PL_UA_getResult`).
 - b If there is additional installation to complete, the engine provides progress information to the DIL by repeatedly sending the DIL event `DMA_MSG_SCOMO_INSTALL_PROGRESS_UI`. When the installation is complete, the engine sends the DIL event `DMA_MSG_SCOMO_INSTALL_DONE` to the DIL.

5.9 Device Administrator Permissions

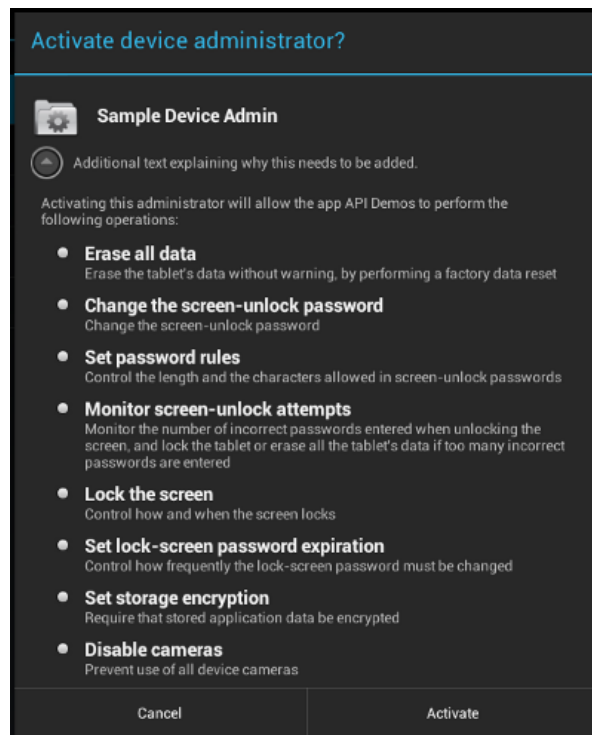
The client performs tasks that require administrative privileges, such as:

- Locking the device
- Performing a factory reset on the device

When the SWM Client is started during device initialization, the SWM Client activates these permissions without asking the end-user. To do this, the client must be signed with the device's public and private keys.

If the end-user later tries to remove administrator permissions from the client, the client overrides this on the next reboot. So if someone steals the device, the SWM Center will be able to lock or wipe data from the device on the next reboot.

If the client is not signed, you must enable these permissions in the Security / Device Administrator settings menu. Otherwise, the client requests the user to enable these privileges by displaying the Device Administrator dialog box when the end-user taps the SWM Client icon.

**Figure 5-3: Device Administrator Dialog Box**

6 SWM DM Client Porting Layer Functions

The SWM DM Client is delivered with a complete set of Porting Layer functions for Android. The Client Porting Layer Functions (see the *Client Porting Layer Functions* chapter of the *vDirect Mobile Framework Integrator's Guide*) are delivered in source format and can be modified as required.

This chapter presents some additional Porting Layer functions delivered in source format that can be modified.

6.1 Native Porting Layer Functions

6.1.1 Device Information Functions

These functions, defined in **vdm_swmc_pl_device.h**, retrieve basic device information.

```
// Get device model
SWM_Error VDM_SWMC_PL_Device_getModel(
    UTF8CStr    outModel,
    IU32        *ioModelSize);

// Get device vendor
SWM_Error VDM_SWMC_PL_Device_getManufacturer(
    UTF8CStr    outMan,
    IU32        *ioManSize);

// Get current firmware version
SWM_Error VDM_SWMC_PL_Device_getFWVersion(
    UTF8CStr    outFWVersion,
    IU32        *ioFWVersionSize);
```

6.1.2 File Search Functions

These functions, defined in **vdm_swmc_pl_dir.h**, allow the client to search for a file that matches a search string (may include * and ? wildcards).

```
// Create a handle to a list of files in a directory
SWM_Error VDM_SWMC_PL_Dir_create(
    void        **outHandle,
    UTF8CStr    inPath);

// Get next file name in the above handle
SWM_Error VDM_SWMC_PL_Dir_getNextFile(
    void        *inHandle,
    UTF8CStr    outBuffer,
    IU32        *ioBufferLen);

// Close the above handle
VDM_SWMC_PL_Dir_destroy(void* inHandle);
```

6.1.3 Delivery Package Wrapper Function

This function, defined in **vdm_swmc_pl_dp.h**, validates the signature of the DP.

```
SWM_Error VDM_SWMC_PL_Dp_validateExternalSignatureDp(
    const char    *inPath,
    IU32          *outOffset);
```

6.1.4 Self-Update Function

This function, defined in **vdm_swmc_pl_self_upgrade.h**, is called by the engine for each DM Tree node that must be added to the DM Tree after a self-update. This function calls the Java equivalent *getDefaultValue*.

```
// Get values of new nodes after self-update
SWM_Error VDM_SWMC_PL_SelfUpgrade_getNodeDefaultValue(
    const char    *inUri,
    char          *outDefaultValue,
    int           *ioValueSize);
```

6.2 Java Porting Layer Functions

The following IPL functions are in `android\redbend\src\com\redbend\client`.

6.2.1 iplGetAutoSelfRegDomainInfo

Description

Get self-registration credentials from external file **Credentials.txt** (see *Automatic Self-Registration*) and, if they exist, set them in the DM Tree.

Declaration

```
public static int iplGetAutoSelfRegDomainInfo(String
    [] autoSelfRegDomainInfo)
```

Parameters

Parameter	Description
autoSelfRegDomainInfo	An array of two elements: a domain name and a PIN.

Return Values

Value	Description
-1	Error
0	Success

6.2.2 getDevModel

Description

Get device model.

Declaration

```
public static String getDevModel()
```

Parameters

None

Return Values

Value	Description
String	The device model

6.2.3 getManufacturer

Description

Get device vendor.

Declaration

```
public static String getManufacturer()
```

Parameters

None

Return Values

Value	Description
String	The device vendor

6.2.4 getFwVersion

Description

Get firmware version.

Declaration

```
public static String getFwVersion()
```

Parameters

None

Return Values

Value	Description
String	The firmware version

6.2.5 `getDeviceId`

Description

Get device ID: The IMEI for a device with SIM card, or the MAC address (for tablets, for examples).

Declaration

```
public static String getDeviceId(Context ctx)
```

Parameters

Parameter	Description
ctx	Service context

Return Values

Value	Description
String	The device ID

6.2.6 `getUserAgent`

Description

Get device User Agent, which is used in the user agent header during HTTP transactions.

Declaration

```
public static String getUserAgent(Context ctx)
```

Parameters

Parameter	Description
ctx	Service context

Return Values

Value	Description
String	The device User Agent

6.2.7 `getDefaultValue`

Description

Add a DM Tree node. Called by the engine for each DM Tree node that must be added to the DM Tree after a self-update.

Declaration

```
public static String getDefaultValue(String Uri)
```

Parameters

Parameter	Description
Uri	<p>The DM Tree node to set, one of:</p> <ul style="list-style-type: none"> ./Ext/RedBend/BootupPollingInterval: Sets the time after a reboot before the first device-initiated DM session. The time is set to 60 (minutes). ./Ext/RedBend/RecoveryPollingInterval: Sets the polling interval after a failed Download or DM session to 1440 (minutes). ./Ext/RedBend/UserInteractionTimeoutInterval: Sets the timeout after no end-user response to 1440 (minutes). ./Ext/RedBend/PostponePeriod: Sets the time to postpone a download or installation after the end-user requests to postpone the action. The time is set to 60 (minutes). ./Ext/RedBend/PostponeMaxTimes: Sets the number of times the user can postpone a download or installation to 3 (times).

Return Values

Value	Description
-1	Error
0	Success

6.3 UA Native Porting Layer Functions

These functions, defined in **vdm_swmc_pl_ua.h**, provide a means for the SWM DM Client to send and receive information to and from the UA.

The SWM DM Client uses **VDM_SWMC_PL_UA_handoff** to set the location on the device to which DP files are stored after they are downloaded. The location is stored in a file. The UA can then use this information to perform an update after a reboot.

```
SWM_Error VDM_SWMC_PL_UA_handoff(
    UTF8CStr *inDpPath,          // Path to DP storage
    UTF8CStr *inDpPathFile,      // File in which to store the path
    UTF8CStr *inHandoffDir);     // Path in which to store the above file
```

After an update, the SWM DM Client uses **VDM_SWMC_PL_UA_getResult** to get the result and remove the DP file.

```
SWM_Error VDM_SWMC_PL_UA_getResult(
    IU32      *outUpdateResult,    // Pointer to file to store result
    char      *inResultFile,       // File which currently stores result;
                                   // removed by this function
    char      *inDpPathFile);     // DP file to remove
```

Part 3: Update Agent

7 Red Bend Update Agent

Red Bend's UA can execute firmware and embedded application update operations on an Android device. The UA is capable of performing failsafe firmware updates using a wide range of FOTA modes. The UA may run from either the main system or the recovery system.

Using Red Bend's UA means that OEMs do not need to:

- Invest effort in a design process
- Implement Porting Layer functions
- Implement the Update Agent logic (including the self-update logic)
- Implement the handoff logic from the SWM DM Client

The UA is an out-of-the-box implementation that covers many time-consuming OEM tasks.

The main benefits of the UA are:

- Easy integration on the device
- Multi-process support for parallelized firmware updating
- Reduced device downtime during the firmware update
- Firmware update using a single reboot when running from the main system
- Smallest delta size on the market
- Failsafe operation
- Small footprint

The UA wraps Red Bend's vRapid Mobile Update Installer (UPI). An instance of the UPI must be located in both the main system and the recovery system.

The following figure shows the layout of the UA on the device.

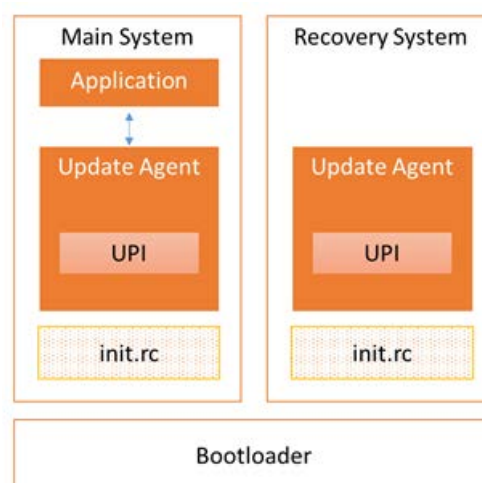


Figure 7-1. UA on the device

7.1 UA Deliverables

The deliverables for UA are:

- Update generation environment:
 - vRapid Mobile Update Generator (command-line executable) for creating the update packages (updates), with a configuration file template covering all delta-update types that may need to be included in the package
- Update Agent – Binary file in either executable or library (with Main) form:
 - Executable: If encryption of the update is not required by the OEM
 - Library: If you require updates that are either encrypted or in a customized format, you must implement an interface for parsing the update and link the update with the UA library
 - Reference code that implements the read function (see *Implementing Update Package Read Functionality*)
 - Reference code that implements basic custom operations (see *Adding Custom Operations*)
- Modified `init.c` and `init.rc` files for a Nexus S device: Instructions explaining how to modify these files for other Android devices are provided in the technical documentation
- A sample UA configuration file

7.1.1 UA Delivery Structure

This section presents the delivery structure of the UA.

```

├─ android_ndk443
│   └─ android_ndk_r7_ics
│       └─ rls - Binary static libraries and UA executable:
│           ├── rb_ua - UA executable
│           ├── libfuse.a
│           ├── librb_ua.a
│           ├── librb_ua_pl.a
│           ├── librb_ua_utils.a
│           ├── libupi.a
│           └─ libutils.a
├─ docs - API Reference documentation
│   └─ html
│       ├── index.html - Entry point the API reference documentation
│       └─ ... other documentation files
└─ sdk - Update Agent SDK:
    ├── Android.mk - Android makefile
    ├── main.c - main, implementing call to the UA entry point
    ├── get_delta.c - Reference implementation of the get delta API
    ├── rb_ua_ui.c - UA sample interface with Android recovery UI
    ├── custom_ops.c - File in which Custom Operations are implemented
    └─ include
  
```

```
| | | common.h
| | | RB_UaApi.h - Porting Layer that can optionally be implemented
| | | RB_UaCustomApi.h - API to be used in Custom Operations
| | | custom_ops.h - Header file for implementing Custom Operations
| | | RB_vRM_Errors.h
| | | RB_vRM_Update.h
| | └─ recovery - recovery UI source code
| |   | default_recovery_ui.c
| |   | recovery_ui.h
| |   └─ ui.c
```

7.1.2 init Related Files

```
| └─ Android
|   | fstab.tuna.jb.diff
|   | init.gb.diff
|   | init.ics.diff
|   | init.jb.diff
|   | init.rc
|   | init.tuna.rc.ics.diff
|   | init.tuna.rc.jb.diff
|   └─ README
```

7.1.3 Update Generation Files

```
| └─ CreateDp.zip
| └─ linux
|   | └─ 32
|   |   └─ vRapidMobileCMD-Linux.exe
| └─ windows
|   | └─ 32
|   |   └─ vRapidMobileCMD.exe
```

7.2 Standard FOTA Update

Standard FOTA update solutions require restarting the device into a special update mode where the device is not operational, performing the update, and then starting the device again into the main system.

The update steps are:

- 1 The update is downloaded into the device and saved in a location accessible by the UA when running in the recovery system.
- 2 The device is restarted into the recovery system.
- 3 When the recovery system starts, the UA is invoked from the **init.rc** script.
- 4 The UA locates the update and validates it.
- 5 The UA performs a scout operation on all the components available in the update.

- 6 If there is a scout failure on any of the components, no update is performed.
- 7 The UA applies the firmware and embedded application updates.
- 8 If a **recovery.img** update is part of the update, its update is not performed now. The UA flags this information and handles it at step 12.
- 9 The UA records the update results in a file.
- 10 The UA restarts the device into the main system.
- 11 When the main system starts, the UA is invoked from the **init.<hardware>.rc** script.
- 12 If a **recovery.img** update is part of the update (the UA finds the flag it set at step 8), **recovery.img** is updated.
- 13 The UA exits.
- 14 The device is available for use by the end-user.



NOTE: When the device boots into main system, the UA is started from **init.rc** only to check if there is a flag indicating that a **recovery.img** update is required. If there is no flag, the UA exits without performing any operation.

7.3 Background Images Update

The UA can implement a *background images update* to update the firmware while the device is operational (unlike a standard FOTA update), minimizing the time that the end-user cannot use the device and requiring only one restart of the device.

Background images update enables scouting of the update target and image update online while all of the file system update is offline.

To update firmware:

- 1 **Pre-Boot Phase.** The scout operation is executed in this phase. In addition, images are also updated. The device is fully operational during the pre-boot phase.
- 2 **Boot Phase.** Update processing is completed during this phase. The device is not operational during the boot phase.

7.4 Recovery from Power Failure

If a power failure halts the update, the device being updated is not negatively impacted. The UA ensures that the device always boots from the correct system (main or recovery) when power is restored to the device following a power failure. Following the reboot, the UA executes regularly, detects its state, and executes failsafe operation steps.

In order to ensure failsafe operation in all update types, integration is required in both **recovery.img** and **boot.img**. For more information, see *Failsafe Operation*.

7.5 Update Agent Self-Update

The UA, which is a statically-linked executable, can be updated.

The UA *self-update* is failsafe. A UA self-update is performed first; the update of all device components is performed by the updated UA.

The UA executable should be packed as an independent part of the update. To ensure failsafe operation, the UA must **not** be updated via the `/system` partition update. For more information, see *Configuration Files for Updating the Update Agent*.

7.5.1 New Update Agent

Whenever the format of the generated update changes (which can occur only with a major UPG version release), a device's existing UA cannot read or process the update internal deltas. In this case, the UA must be updated as full-release file, not as a delta from the old UA.

To create and apply a new UA:

- 1 The UPG generates an update that provides a new UA executable as a standalone component (no delta is calculated for the UA). Optionally, the component can be compressed.
See *Configuration Files for Updating the Update Agent* for the configuration file definitions that instruct the UPG to include a new UA when it generates an update.
- 2 Before beginning an update, the current UA reads the new UA and extracts the new UA to a temporary location.
- 3 The new UA copies itself back to the original UA location.
- 4 The current UA executes the new UA to complete the update operation (the new UA updates all other components).

The new UA uses the configuration file of the current UA.

7.5.2 Update Agent Delta Update

Between minor updates of the UPG, the UA can self-update using a delta update.

Use this method when:

- There are changes to the UA that do not change the update structure.
- A new Custom Operation is added (see *Adding Custom Operations*)

To create and apply a delta update to the UA:

- 1 The UPG generates an update that provides a delta for the UA as a standalone component.
See *Configuration Files for Updating the Update Agent* for the configuration file definitions that instruct the UPG to include a UA delta when it generates an update.
- 2 Before beginning an update, the current UA updates itself to a temporary location.
- 3 The updated UA copies itself back to the original UA location.
- 4 The current UA executes the updated UA to complete the update operation (the updated UA updates all other components).

The updated UA uses the configuration file of the current UA.

8 Integrating the UA

Before integrating the UA, ensure that you have the following:

- A Linux build environment containing relevant Android source code.
See <http://source.android.com/source/initializing.html> for a complete guide on how to set the build environment.
- Ability to manipulate main system and recovery system images.

8.1 Configuring Partitions

The UA can update any partition on the device. Partition recognition can be done using either of the following parameters:

- `partitions_list`
- `emmc_base_path`

At least one of these parameters must be supplied. For information about these parameters, see *Update Agent Parameters*.



NOTE: It is recommended to use `partitions_list` and not `emmc_base_path`, see *File Systems in emmc_base_path*.

8.1.1 Partitions List

The file given as the partitions list should contain a table in TSV (tab-separated values) format with the following fields:

- **MountPoint:** The path in which a file system device should be mounted. For an image device, this parameter is ignored. However, it is used as the default value for the partition name (see *Partition Naming*).
- **Type:** The type of the device. Supported devices include:
 - For file system partitions: `yaffs2`, `ext4`, `vfat`
 - For image partitions: `emmc`, `mtb`
- **Device:** The path of the physical device that should be updated (usually in `/dev/`).
- **Options:** Extra options in the format `<key>=<value>`. The supported options are:
 - `name`: Indicates the partition name in the delta (see *Partition Naming*)
 - Other options are ignored

The following is an example of a partitions list file.

#	MountPoint	Type	Device	Options
	/sdcard	vfat	/dev/block/sda1	
	/system	ext4	/dev/block/platform/<dev_name>/by-name/system	
	/cache	ext4	/dev/block/platform/<dev_name>/by-name/cache	
	/data	ext4	/dev/block/platform/<dev_name>/by-name/userdata	name=userdata
	/misc	emmc	/dev/block/platform/<dev_name>/by-name/misc	


```
/boot      emmc    /dev/block/platform/<dev_name>/by-name/boot
/recovery  emmc    /dev/block/platform/<dev_name>/by-name/recovery
```



NOTE: From Android 4.3, the default `recovery.fstab` file structure is similar to the `fstab.<hardware>` structure.

Enable the `part_list_fstab_format` parameter to support the `fstab` format.

The `name` parameter from the old file structure can be used in the Android 4.3 `fstab` structure and should be concatenated to the `fs_mgr_flags` column.

8.1.1.1 Partition Naming

The UA matches physical partitions to updates according to the `PartitionName` tag in the UPG configuration file.

The matching is performed as follows:

- 1 The default partition name is the mount point, excluding its preceding '/'. For example, in the line:

```
/boot emmc /dev/block/platform/<dev_name>/by-name/aboot
```

The partition name of the device `/dev/block/platform/<dev_name>/by-name/aboot` is `boot`.

- 2 To override this value, add a `name` field. For example, in the line:

```
/data ext4 /dev/block/platform/<dev_name>/by-name/userdata name=userdata
```

The partition name of the device

`/dev/block/platform/<dev_name>/by-name/userdata` is `userdata`.

8.1.1.2 System Partition

When performing a background update in the main OS, you must first mount the system device to a temporary location (`/rb_system`), and then the UA mounts the system device to its expected location (`/system`). In such a case, the expected location should be given in the partitions list.

That is, match the `/system` mount point with the system device, even though it is first mounted to `/rb_system`.

8.1.1.3 MTD Devices

An MTD device is specified by the `mtd` type. MTD devices are not matched by a full path to the device. Instead, use the declarations that appear in the `/proc/mtd` file. This file matches devices to partition name. The following is an example of an MTD file.

```
dev:      size  erasesize  name
mtd0: 00200000 00040000 "bootloader"
mtd1: 00140000 00040000 "misc"
mtd2: 00800000 00040000 "boot"
mtd3: 00800000 00040000 "recovery"
mtd4: 1d580000 00040000 "cache"
```

For MTD devices, specify the device name, as it appears in this file, as the device of the partition. For example, the line in `partitions_list` that declares the boot partition in the preceding example is:

#	MountPoint	Type	Device	Options
	/boot	mtd	boot	



NOTE: Recognition of MTD devices is possible only via `partitions_list`, not via `emmc_base_path`.

8.1.2 eMMC Base Path

Physical eMMC devices usually appear under `/dev/block/mmcblk0p<num>`

In every Android device, the eMMC devices are also presented in `/dev/block/platform/<dev_type>/by_name/` as a symbolic link. For example:

```
root@android:/ # ls -l /dev/block/platform/omap/omap_hsmmc.0/by-name
lrwxrwxrwx root root 2013-12-05 19:03 boot -> /dev/block/mmcblk0p7
lrwxrwxrwx root root 2013-12-05 19:03 cache -> /dev/block/mmcblk0p11
lrwxrwxrwx root root 2013-12-05 19:03 misc -> /dev/block/mmcblk0p5
lrwxrwxrwx root root 2013-12-05 19:03 radio -> /dev/block/mmcblk0p9
lrwxrwxrwx root root 2013-12-05 19:03 recovery -> /dev/block/mmcblk0p8
lrwxrwxrwx root root 2013-12-05 19:03 sbl -> /dev/block/mmcblk0p2
lrwxrwxrwx root root 2013-12-05 19:03 system -> /dev/block/mmcblk0p10
lrwxrwxrwx root root 2013-12-05 19:03 userdata -> /dev/block/mmcblk0p12
lrwxrwxrwx root root 2013-12-05 19:03 xloader -> /dev/block/mmcblk0p1
```

This path can be passed as a parameter to the UA service; all devices contained in this path are marked as eMMC devices and the symbolic link name is the partition name. For example, the link to the boot image is equivalent to the following line in the partitions list in the previous example:

```
/boot emmc /dev/block/mmcblk0p7
```



NOTE: When using this parameter you cannot declare a different partition name; you must use the symbolic link name.

8.1.2.1 File Systems in `emmc_base_path`

File system devices that are located in `emmc_base_path` are treated as follows:

- If the device is currently mounted, it is stored as a file system partition and its type is recognized according to the mount type.
- If the device is not currently mounted, the UA cannot identify it as a file system partition, so it cannot be updated.



NOTE: It is recommended to use `partitions_list` and not `emmc_base_path`, due to this difficulty in identifying file system partitions. If `emmc_base_path` is used, make sure to mount all file systems prior to starting the UA.

8.2 Integrating the UA

Changes must be made to **recovery.img**, where most of the update is done, and to **boot.img**, where the update of **recovery.img** is done. (**recovery.img** must be updated from the main system and not from the recovery system to ensure successful completion of the update if there is a power failure.)

8.2.1 Modifying the Device Recovery System

To perform a standard FOTA update in a device, the main integration task is to start the UA from the recovery system. Do this by modifying the device's *recovery system image init script*, **init.rc**. The **init.rc** syntax is described in http://www.kandroid.org/online-pdk/guide/bring_up.html; see *Android Init Language*.

To modify the recovery system image:

- 1 In the **init.rc** script file in the `on boot` section, mount the system and userdata partitions in read/write mode.

For example :

```
mount ext4 /dev/block/platform/omap/omap_hsmmc.0/by-name/system
/system wait rw
mount ext4 /dev/block/platform/omap/omap_hsmmc.0/by-
name/userdata /data wait rw
```

- 2 In the **init.rc** script file, add the `rb_ua` service:

```
service rb_ua <UA executable> <parameter list>
    oneshot
    user root
    group root
```

Where:

- `<UA executable>` is the full path to the UA executable.
This executable must reside under the `/system` directory and be statically linked.
- `<parameter list>` is one or more parameters to be used by the UA. For the available parameters to use with the UA, see *Update Agent Parameters*.

- 3 In the **init.rc** script file, prevent the native recovery executable from starting.

Usually the native executable is defined as a service with the name `recovery`. If it exists, remove or comment out the declaration of the service.

- Before modification:

```
service recovery /sbin/recovery
```

- Modified declaration:

```
#service recovery /sbin/recovery
```

- 4 In the configuration file for running the UA, add the following lines:

```
update_flavor=std
in_recovery_kernel=1
```

Example of the modifications applied to the `on boot` section of the recovery system image init script:

- The `on boot` section, prior to modification:

```
on boot
    ifup lo
    hostname localhost
    domainname localdomain
    class_start default
```

- The modified `on boot` section:

```
on boot
    ifup lo
    hostname localhost
    domainname localdomain
    mount ext4 /dev/block/platform/omap/omap_hsmmc.0/by-name/system
                                                /system wait rw
    mount ext4 /dev/block/platform/omap/omap_hsmmc.0/by-name/userdata
                                                /data wait rw
    class_start default
```

8.2.2 Modifying the Device Main System

If **recovery.img** needs to be updated, you must modify the *main system image init script*. **recovery.img** is updated after the device is updated and started into the main system.

To modify the main system image:

- 1 In either the **init.rc** script file or the **init.<hardware>.rc** script file, add the `rb_ua` service:

```
service rb_ua <UA executable> <parameter list>
    oneshot
    user root
    group root
```

- 2 Modify the `on fs` section of the boot image script file by adding the following line that starts the `rb_ua` service:

```
start rb_ua
```

- 3 In the configuration file for running the UA, add the following line:

```
update_flavor=std
```



NOTE: This must be a different configuration file, not the one used in the recovery image.

Example of the modifications applied to the `on fs` section of the main system image init script:

- The `on fs` section, prior to modification:

```
mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/system
/system wait ro
mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/userdata
/data wait noatime nosuid nodev
```

- The modified `on fs` section:

```
mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/system
/system wait ro
mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/userdata
/data wait noatime nosuid nodev
start rb_ua
```

8.3 Implementing Update Package Read Functionality

If you want to replace the implementation of reading the update (for example, for decrypting an update), implement the following functions:

- `RB_UaOpenDelta`
- `RB_UaGetDelta`
- `RB_UaCloseDelta`

The vRapid Mobile deliverables include a reference implementation of these three functions that handle an update in an unencrypted file.

For detailed information about implementing these functions, see the *vRapid Mobile API Reference*.

8.4 Customizing the UI Display of the Boot Phase and Recovery

You can customize the graphics that are presented on the device UI as background to the progress indication by replacing the following files in the `/res/images` directory in **recovery.img** and (for background update) **boot.img**. (The `/res/images` directory might not exist in **boot.img**; if it does not, copy it from **recovery.img**.)

- **icon_installing.png**
- **progress_empty.png**
- **progress_fill.png**

You can replace the default implementation of the UI by implementing the following functions:

- `RB_Ua_UiInit`
- `RB_Ua_UiExit`
- `RB_Ua_UiPrint`
- `RB_Ua_UiProgress`

The vRapid Mobile deliverables include a reference implementation of these four functions that handle recovery/boot UI interaction.



NOTE: In Red Bend's reference implementation, `RB_Ua_UiExit` calls the `ui_exit` function, which is not implemented in Android vanilla source code; the `ui_exit` function should clean up all UI resources that were allocated by `RB_Ua_UiInit`. A reference implementation of this function is also provided in vRapid Mobile deliverables.

8.5 Rebuilding the Update Agent

To re-implement any of the default functions mentioned in the previous section:

- 1 Prepare and build a full Android source environment.
See <http://source.android.com/source/index.html> for details on downloading and building an environment.

- 2 Extract the delivery package and go to the UA SDK directory.

- 3 Set up the environment:

```
$ export TOP=<full_path_to_your_Android_code>
$ export
RB_LIBS=<relative_path_to_RB_libs_in_the_delivery_package>
# For example: export RB_LIBS=../android_ndk443/android_ndk_r7_ics/rls/
$ source $TOP/build/envsetup.sh
```

- 4 Modify or re-implement the code.

- 5 Rebuild the UA:

```
$ mm
```

The built UA is created at `$TOP/out/target/product/generic/system/bin/rb_ua`

9 Generating Updates

Use the UPG, delivered with the UA, to create updates. The UPG requires an XML configuration file as input.

A UPG operation requires the use of the CreateDP utility. Extract the CreateDP utility archive, delivered with the UA, into the UPG's working directory. Make sure to retain the CreateDP structure, `CreateDp/Lib`.

9.1 Defining Update Generator Configuration Files

When defining an XML configuration file for input to the UPG, be aware that:

- To modify the location of the CreateDP package, use the optional `<CreateDp>` tag in the configuration file and place it at the top level, which is before the first `<Partition>` tag. For example: `<CreateDp>/path_to_package/CreateDp</CreateDp>`
- The configuration file's `<DeviceOS>` tag must contain the value `Android`.
- The required content for the XML configuration file was simplified, that is, many parameters have default values. Override recommended values by specifying replacement values in the configuration file.

The main default values for update generation are:

- `<RamSize> 0x1000000 (16MB)`
- `<ImageType> abi (for image partitions)`
- `<ComponentDeltaFileName> AndroidUpdate.mld`
- `<UseAndroidFileAttr> 1`
- Any partition that is recognized on the device is updatable. In the partition section, you must supply at least the following elements:
 - `PartitionName`: The name that is assigned to the partition on the device (see *Configuring Partitions*)
 - `SourceVersion`: The source version of the delta
 - `TargetVersion`: The target version of the delta

The UPG identifies whether the partition is a file system partition or an image partition according to the following rules:

- If `SourceVersion` and `TargetVersion` are directories, then the partition must be a file system partition.
- If `SourceVersion` and `TargetVersion` are files, then the partition is probably an image partition.
- If you are creating a delta between two files and the delta is to be applied on a file system partition, explicitly specify `<PartitionType>PT_FS</PartitionType>`.
- If the recovery partition is updated, its partition name must be `recovery`, since it is handled differently than other images.
- File attributes are, by default, automatically assigned according to the Android file attributes mapping and, therefore, no file attribute input file needs to be supplied.

For information about file attributes and their mappings, see the explanations in the code of the `fs_config` tool at https://android.googlesource.com/platform/build/+/-/jb-dev/tools/fs_config/fs_config.c.

- You can add Custom Operations, such as formatting file system partition. See *Adding Custom Operations*.
- To assign a non-default name to the update, use the optional `<ComponentDeltaFilename>` tag in the configuration file and place it at the top level, which is before the first `<Partition>` tag.
- For an image partition, the default image type is `zImage`. Use the configuration file `<ImageType>` tag to use another image type (for example, `lzo` or `standard`). For more information about available image types, see the *vRapid Mobile Integrators Guide*.

To create an update, execute the following command:

```
vRapidMobileCMD-linux.exe gen /configuration_file=<filename.xml>
```

For further information about using the vRapid Mobile UPG, see the *vRapid Mobile Integrator's Guide*.

The following example is a configuration file for generating an update and defines the boot, system, recovery, and radio partitions for update, and supplies a new UA (see *Configuration Files for Updating the Update Agent*).

9.1.1 Example: Configuration File for Update Generation

```
<vrml>
  <DeviceOS>Android</DeviceOS>
  <Partition>
    <PartitionName>boot</PartitionName>
    <SourceVersion>images/src/boot.img</SourceVersion>
    <TargetVersion>images/dest/boot.img</TargetVersion>
  </Partition>
  <Partition>
    <PartitionName>system</PartitionName>
    <SourceVersion>images/src/system</SourceVersion>
    <TargetVersion>images/dest/system</TargetVersion>
  </Partition>
  <Partition>
    <PartitionName>recovery</PartitionName>
    <SourceVersion>images/src/recovery.img</SourceVersion>
    <TargetVersion>images/dest/recovery.img</TargetVersion>
  </Partition>
  <Partition>
    <PartitionName>radio</PartitionName>
    <ImageType>standard</ImageType>
    <SourceVersion>images/src/radio.img</SourceVersion>
    <TargetVersion>images/dest/radio.img</TargetVersion>
  </Partition>
```



```
<Partition>
  <PartitionName>update_agent</PartitionName>
<TargetVersion>images/dest/system/bin/rb_ua</TargetVersion>
</Partition>
</vrm>
```

9.2 Configuration Files for Updating the Update Agent

As discussed in *Update Agent Self-Update*, the UPG can generate an update that includes a UA self-update

9.2.1 Supplying a New Update Agent

When the UPG must generate an update that supplies a new UA, the configuration file must define the UA to be supplied in its own partition section, as follows:

- Assign the value `update_agent` to the `<PartitionName>` tag
- Do not assign a `<SourceVersion>` tag
- Assign the `<TargetVersion>` tag the valid path to the new UA

The target device requires this new UA in order to manage the other updates after a major UPG protocol version release that changes the format of the update and makes the existing UA incompatible with the update to be generated—the existing UA cannot read the format of the updates (see *New Update Agent*).

The following example is a configuration file for generating an update that supplies a new UA.

```
<vrm>
  <DeviceOS>Android</DeviceOS>
  <Partition>
    ...
  </Partition>
  <Partition>
    <PartitionName>update_agent</PartitionName>
    <TargetVersion>new_ua</TargetVersion>
  </Partition>
</vrm>
```

Optionally, the UA can be compressed.

```
<Partition>
  <PartitionName>update_agent</PartitionName>
  <TargetVersion>new_ua</TargetVersion>
  <Compress>1</Compress>
</Partition>
```

9.2.2 Supplying an Update Agent Delta

If the UA must be updated between major UPG protocol version releases, a delta update can be used (see *Update Agent Delta Update*).

The following example is a configuration file for generating an update that supplies a UA delta.

```
<vrn>
  <DeviceOS>Android</DeviceOS>
  <Partition>
    ...
  </Partition>
  <Partition>
    <PartitionName>update_agent</PartitionName>
    <SourceVersion>old_ua</SourceVersion>
    <TargetVersion>new_ua</TargetVersion>
  </Partition>
</vrn>
```

10 Update Agent Parameters

To integrate the UA, modify either the device's **init.rc** script or its **init.<hardware>.rc** script to include the UA service **rb_ua**, which runs the UA on the device. This appendix describes the parameters that are available for use when running the UA.

- You can provide these parameters in a configuration file, via the command line, or using a combination of both.
- When parameters are provided via the command line, they are preceded by **--** (or by **-** for the short form), with a space before the value.
- When parameters are provided in a configuration file, they are provided without any preceding characters, with **=** between the parameter name and value.

Parameters are optional unless stated otherwise. The following table describes the parameters.

Table A-1: Update Agent Parameters

Parameter	Description
<code>--config_file <file></code> <code>-c <file></code>	<p>The full path of a configuration file containing UA parameters.</p> <p>The information in the configuration file must be in the form <key>=<value>. Any UA parameters may be included in the configuration file.</p> <p>The value of a parameter provided via the command line overrides the same parameter in the configuration file.</p>
<code>--work_dir <directory></code> <code>-w <directory></code>	<p>(Mandatory) The working directory path to be used by the UA to store temporary data during the update.</p>
<code>--log_path <file></code> <code>-l <file></code>	<p>The path of the UA's log file.</p> <ul style="list-style-type: none"> • If the path is kernel, the UA writes to the kernel log. (This log is read using dmesg.) • If the log file name is prefixed with debug, the log level is set to debug. • If the parameter is not included, no log is created. <p>NOTE: When the UA starts, it creates a new log file. If a log file already exists in the given path, it is backed up as <path>.prev.1. Up to four historical log files are saved (<path>.prev.[1-4]).</p>

Parameter	Description
<code>--log_child <flag></code>	<p>A flag that specifies whether to create a separate log file for each child process.</p> <p>The log files are named <code><log_path>.<pid></code></p>
<code>--delta_path <file></code> <code>-d <file></code>	<p>(Mandatory) The path of the update file to use.</p>
<code>--update_dir <directory></code>	<p>The directory in which the system partition is mounted at the init script. This parameter is ignored when operating in the recovery system.</p> <p>The parameter has a default value of <code>/rb_system</code>.</p>
<code>--partitions_list</code> <code><part_list>:<part_list>...</code>	<p>A list of one or more partition lists that are used to match physical partitions on the device to partition names in the delta.</p> <p>The first file is used. If it does not exist, the second file is used, and so on. If none of them exists, this parameter is ignored.</p> <p>The parameter has a default value of <code>/recovery.fstab:/etc/recovery.fstab</code> (which are the default locations for this file in recovery.img).</p> <p>There is a relation between <code>--emmc_base_path</code> and <code>--partitions_list</code>. See <i>Configuring Partitions</i> for more information.</p>
<code>--part_list_fstab_format</code> <code><flag></code>	<p>A flag that specifies whether to parse the partitions list as an <code>fstab</code> file structure (the structure that was introduced in Android 4.3).</p> <p>The parameter has a default value of 0 (parse as a <code>/etc/recovery.fstab</code> structure).</p>

Parameter	Description
<code>--emmc_base_path <path></code> <code>-p <path></code>	<p>The path to the directory that contains the eMMC devices by-name. Android automatically creates such a directory; you can find it by running the <code>mount</code> command on the device. Check which device is mounted on <code>/system</code>. Usually, the path is <code>/dev/block/platform/omap/omap_hsmmc.<number>/by-name</code>.</p> <p>There is a relation between <code>--emmc_base_path</code> and <code>--partitions_list</code>. See <i>Configuring Partitions</i> for more information.</p>
<code>--set_boot_to_recovery <flag></code>	<p>A flag that specifies whether to boot into the recovery system, in the event of a power failure during boot.img update.</p> <p>For demonstration purposes, set this flag to 0.</p> <p>The parameter has a default value of 1 (boot into the recovery system).</p>
<code>--in_recovery_kernel <flag></code>	<p>A flag that specifies whether to run the UA in the recovery system.</p> <p>The parameter has a default value of 0 (do not run in the recovery system).</p>
<code>--result_file <path></code>	<p>(Mandatory for standard FOTA update)</p> <p>The path to the directory where a text file is created to hold the final result of the update.</p> <p>The file is created when the update finishes and must be deleted by the calling application. The file contains 0 for success and an error code otherwise.</p> <p>The parameter has a default value of " ".</p>

Parameter	Description
<code>--max_num_process <number></code>	<p>The maximum number of processes to run in parallel in a multi-process file system update.</p> <p>A value of 0 disables multi-processing.</p> <p>A value of 1 leaves the parent and sub-process mechanism in place even though only a single process is handling the update.</p> <p>The parameter has default value of 8.</p>
<code>--recovery_command <path></code>	<p>The path to a file containing the factory reset command. The file is created by the Android API that handles factory reset.</p> <p>The parameter has default value of <code>/cache/recovery/command</code>.</p>

It is very convenient to pass parameters to the service using a configuration file. For example:

```
service rb_ua /system/bin/rb_ua --config_file=/system/bin/config.txt
    oneshot
    user root
    group root
    socket RBCTL stream <permission> <uid> <gid>
```

Where the configuration file **config.txt** contains:

```
work_dir=/data/bgu/work
logfile=debug:/data/bgu/ualog.txt
delta_path=/data/bgu/delta
```

11 Update Generator Parameters

The UPG configuration file parameters are described in the *vRapid Mobile Integrator's Guide*.

To create an update for the UA, the <DeviceOS> parameter must be specified with the value `Android`. Once this is done some parameters are given default values and some parameters become mandatory. These changes are described in the following tables.

11.1 Device-Level Parameters

The following UPG configuration file parameters control the update generation process.

Table B-1: Update Generator – Device-Level Parameters

Parameter	Description	Mandatory	Value
DeviceOS	Specifies the device OS name for the output of the UPG.	Y	The value must be: <code>Android</code>
ComponentDelta FileName	Specifies the file name for the output of the UPG.	N	Output file name The parameter has default value of <code>AndroidUpdate.mld</code>
CreateDp	The CreatedDP utility archive, provided as part of the delivery, should be extracted in the UPG working directory. Its file structure should be retained (<code>CreateDp/Lib</code>). If the utility was installed elsewhere, its location can be specified using this parameter. NOTE: The output of the UPG is in Red Bend Deployment Package (DP) format. This format should be opaque for the user creating the update; the user can later pack it in an additional package (including wrapping it in a DP).	N	Path to CreatedP The parameter has default value of <code>./CreatedP</code> (That is, under the current working directory)

Parameter	Description	Mandatory	Value
ReFlashDelta	Specifies that the update generation is for re-flash delta. This parameter is only applicable to Android. This parameter is obsolete; use a Custom Operation (see <i>Adding Custom Operations</i>).	N	<ul style="list-style-type: none"> 0: Default value 1: Insert the <code>format</code> Custom Operation
RamSize	Specifies the amount of device RAM to be used for the update.	N	Default is: 0x1000000 (16MB)

11.2 General Partition Parameters

The following UPG configuration file parameters control the partition attributes during the UPG process and are applicable to both image and file system updates.

Table B-2: Update Generator – General Partition Parameters

Parameter	Description	Mandatory	Value
PartitionType	Specifies the device partition type. Do not modify the default value assigned to this parameter. The UPG automatically assigns it a value. <i>See Defining Update Generator Configuration Files</i> for more information.	N	<ul style="list-style-type: none"> The parameter has default value of <code>PT_FS</code> for file system partitions The parameter has default value of <code>PT_FOTA</code> for image partitions
PartitionName	Specifies the device partition name.	Y	<ul style="list-style-type: none"> Partition name on the device. Special values: <ul style="list-style-type: none"> <code>recovery</code> Android recovery image <code>update_agent</code> UA self-update

Parameter	Description	Mandatory	Value
SourceVersion	The path of the partition image or the root directory holding file system source version data located in the relevant partition.	Y	Partition root directory path for source version
TargetVersion	The path of the partition image or the root directory holding file system target version data located in the relevant partition.	Y	Partition root directory path for target version
Compress	A flag indicating whether to compress the new UA component in the update. (Only relevant if the PartitionName tag has a value of update_agent.) NOTE: Do not set this flag for a UA delta update.	N	<ul style="list-style-type: none"> 0: Do not compress (default value) 1: Compress
ImageType	The compression method to use. This parameter is relevant for image partitions only . NOTE: The ImageType value is case-sensitive.	N	<p>The parameter has default value of zImage</p> <p>For information about available image types (for example, lzo or standard), see the <i>vRapid Mobile Integrators Guide</i>.</p>

Parameter	Description	Mandatory	Value
UseAndroidFileAttr	<p>A flag indicating whether to use Android file attributes mapping.</p> <p>If this parameter is used, there is no need to provide attribute files using the <code>SourceAttributesFile</code> and <code>TargetAttributesFile</code> parameters (for information about these parameters, see the <i>vRapid Mobile Integrators Guide</i>).</p>	N	The parameter has default value of 1 (collect file attributes)
SectorSize	<p>The sector size of the partition.</p> <p>The sector size for an image must be a divisor of the actual image size on the device.</p> <p>NOTE: For eMMC-based devices the sector size is determined by the UA and is set to 0x10000 (64kB).</p>	N	The parameter has default value of 0x40000 (256kB)



NOTE: A configuration file may mention only some partitions to create a partial delta.

11.3 Custom Operations Parameters

The following UPG configuration file parameters control the partition attributes during the UPG process and are applicable to both image and file system updates.

The following UPG configuration file parameters control any Custom Operations that are to be run when applying the delta. The optional `<CustomOperation>` section can be added inside any `<Partition>` tag. See *Adding Custom Operations* for more information.

Table B-3: Update Generator – Custom Operations Parameters

Parameter	Description	Mandatory	Value
Operation	<p>The Custom Operation to perform. The operation name must match the name of an operation defined in the UA.</p> <p>Optionally, arguments may be added after the operation name.</p>	Y	<p>The format is:</p> <p>name [arg1 [...]]</p>
Type	Specifies the Custom Operation type.	Y	<ul style="list-style-type: none"> • Pre: Perform the custom operation before starting the update of the partition • Update: Perform the custom operation instead of the normal delta update • Post: Perform the custom operation after the update of the partition finishes successfully <p>Note that these values are not related to the phases of background image updates.</p>
Type	A file that contains data to be used in the operation.	N	The path to the data file

12 Failsafe Operation

If a power failure halts image partition update processing, the device being updated is not negatively impacted. When power is restored to the device, it executes the following, failsafe operation steps:

- If the power failure occurs during the pre-boot phase of an image partition update, the device:
 - Boots into recovery system
 - Recovers from the failure and updates the remaining images
 - Reboots and continues with the background update
- If the power failure occurs during pre-update processing of a file system partition or the recovery partition, the device:
 - Boots into main system
 - Recovers from the failure
 - Continues with the pre-update processing
- If the power failure occurs during either boot phase or post-boot phase, the device:
 - Boots into main system
 - Recovers from the failure
 - Continues with the background update

In order to ensure failsafe operation when updating **boot.img**, integration is required in **recovery.img**.

12.1 Integration for Failsafe Operation

The UA service must start on recovery startup to ensure failsafe operation. Accomplish this by modifying the recovery **init.rc** script.

To modify the device recovery image script for failsafe operation:

- 1 In the **init.rc** script file, add the `rb_ua` service as described in step [] of []
- 2 In the configuration file for running the UA, add the following line:

```
in_recovery_kernel=1
```

12.2 Failsafe Coverage Table

The following table describes failsafe coverage for the update process.

Table D-1: Power Failure Coverage

Phase		Recovery Flag	State	Boot in case of power failure	Failsafe Update Flow
Update in recovery system	Scout	On	N/A	Recovery system	<ol style="list-style-type: none">1 Old version2 Detect images or file system status and resume the update
	File system update				
	Image partitions update				
Finalize Update	recovery.img	Off	N/A	Main system	<ol style="list-style-type: none">1 New version2 Restart recovery image update

13 Sample Configuration Files

```
work_dir=/data/install/workdir
log_path=/data/install/mylog1
delta_path=/data/install/delta
update_dir=/system
mount_point=/system
partitions_list=/etc/recovery.fstab # Note: This is the default value
result_file=/data/install/result
in_recovery_kernel=1
update_flavor=std
```

When using this mode, the UA must be integrated in **boot.img** in case **recovery.img** needs to be updated as well.

The configuration file for **boot.img** is the same except for the modification of the following line:

```
in_recovery_kernel=0
```



NOTE: When integrating the UA to the main system, `/etc/recovery.fstab` is a path that is located in **recovery.img**; this file must be copied to a location **that is** accessible by the main system (or you must specify a different partition list). In the main system `/etc` is a symbolic link to `/system/etc`.

14 Adding Custom Operations

14.1 Overview

The standard update operation includes using the UPG to generate an update and applying the update on an Android device using the UA.

However, the full update process may include Custom Operations that cannot be performed using the update. For example:

- Unlock or decrypt a partition before it is updated, and lock or encrypt it afterwards.
- Ensure failsafe operation that is relevant to the specific device. For example, the bootloader must be updated and a failsafe update is ensured by setting a flag in the secondary bootloader telling the device to launch from a different bootloader. In this case, setting (and unsetting) the flag should be implemented as a Custom Operation.
- Update a device in a special manner. For example, if a file system partition should be formatted instead of updated, this can be implemented as a Custom Operation (previously, this was done using the `ReFlashDelta` tag).

Custom Operations are implemented in the UA and called according to their appearance in the UPG configuration file.

14.2 Custom Update Types

A custom update is executed on a specific partition.

There are three types of Custom Operations:

- **Pre-update operation:** A Custom Operation that runs before updating the partition.
- **Update handler operation:** A Custom Operation that runs instead of running the regular update using the delta.
- **Post-update operation:** A Custom Operation that runs after the partition has updated successfully.

14.2.1 Custom Update Calling Sequence

Every Custom Operation is called twice – once during the scout phase and once during the actual update phase. The action parameter that is passed to the Custom Operation function indicates if the operation is a scout operation or an update operation.

At each phase, the operation must perform the relevant action:

- During the scout phase, the custom update must not change any data on the partition. If an error value is returned, the UA treats it as a scout error (a mismatch between the device version and the delta source version).
- If an error value is returned during the update the UA halts the update process, which can result in bricking the device—all possible errors must be determined during the scout phase.
- It is strongly recommended that partition-specific Custom Operations do not affect the data of any other updated partitions. However, the integrator may perform actions on other partitions that are not being updated (e.g., remove files in the `/data` partition after the

system partition update is completed, to avoid using invalid data with the new system image).

Calling sequence when performing a background update

When running in standard FOTA update mode or when updating images in any mode, each partition is processed twice – once for the scout phase and once for the update. Similarly, Custom Operations are called with a scout action when scouting and with an update action when updating.

When updating the system partition in background update mode, the calling sequence is as follows:

- 1 The Custom Operations are called with the scout action when scouting.
- 2 If there are any pre-update operations to be performed by the UA (that is, updating to a PUL), no Custom Operation is called for this operation.
- 3 After the device reboots, the update continues. The Custom Operations are called during the update of the real file system.
- 4 For a full background update, the post-update Custom Operations are called after the post-update phase of the update completes.

14.3 Defining a Custom Update during Delta Generation

To define a Custom Operation during the delta generation, add a `<CustomOperation>` section to the UPG configuration file. Add this section inside the `<partition>` section of the corresponding partition.

Every `<CustomOperation>` section must contain the following elements:

- **Type:** This can have the values `pre`, `update`, or `post`.
- **Operation:** The operation name and additional parameters to be passed to the operation on the device. For more information, see *Implementing a Custom Update in the UA*.

You can also add a file to be used as data to the operation. The data is accessible only from the Custom Operation that declared it. To add data to a Custom Operation, add a `Data` element that contains the path of the data file.

When specifying an `update` type operation, there is no need to declare `SourceVersion` and `TargetVersion` since the update operation replaces the delta update operation. However, as described in section [], the partition type is determined according to `SourceVersion` and `TargetVersion`. As a result, when specifying an `update` type Custom Operation, `PartitionType` must be set explicitly. If `SourceVersion` and `TargetVersion` are given, their values are ignored.

14.3.1 Predefined Custom Operations

Red Bend supplies two predefined Custom Operations. Use the `operation` element and call the operation with the correct arguments. The operations are:

- **format:** Formats a file system partition to the `ext4` file system. A single parameter can be given that indicates the name of the partition to be formatted. If the parameter is not given, the partition that contains the operation is formatted.

This Custom Operation replaces the `ReflashDelta` tag. To delete the contents of a file system partition and replace it with a new version, do all of the following:

- Declare the pre-update Custom Operation `format`, to delete the old image.
- Set `SourceVersion` to an empty folder.
- Set `TargetVersion` to the target image.
- `writeData`: Writes simple data to a specified partition. See *Using the Predefined Custom Operation* for an example of using this operation.

14.3.2 Sample Configuration File

The following is an example of a configuration file that defines several types of Custom Operations.

```
<vrn>
  <DeviceOs>Android</DeviceOs>
  <Partition>
    <!-- Bring full release of system partition. This is done by preparing
    delta from empty folder to the target version, and calling pre-operation
    'format'. -->
    <PartitionName>system</PartitionName>
    <SourceVersion>/tmp/empty</SourceVersion>
    <TargetVersion>/QA/Images/bgu/image04/system</TargetVersion>
    <CustomOperation>
      <Type>pre</Type>
      <Operation>format</Operation>
    </CustomOperation>
  </Partition>

  <Partition>
    <!-- Custom Operation, with data supplied -->
    <PartitionName>cache</PartitionName>
    <PartitionType>PT_FS</PartitionType>
    <CustomOperation>
      <Type>Update</Type>
      <Operation>HelloWorld have_data</Operation>
      <Data>/tmp/some_data</Data>
    </CustomOperation>
    <!-- SourceVersion and TargetVersion are not supplied, since we do
    not generate delta.-->
  </Partition>
```

14.4 Implementing a Custom Update in the UA

The delivery includes the following files, which can be used to implement Custom Operations:

- **custom_ops.h**: A header file that contains the function type of the Custom Operation and the header of `GetCustomOperations()` (that registers the Custom Operations functions) that must be implemented. This file should not be changed.

- **RB_UaCustomApi.h:** A header file that contains the API that can be used by Custom Operations. This file should not be changed.
- **custom_ops.c:** The file that contains the implementation of the Custom Operations. The file contains a reference HelloWorld operation and should be changed if new Custom Operations are needed.

14.4.1 custom_ops.c

This file contains a sample HelloWorld implementation and also an implementation of GetCustomOperations() that can be extended. Additional functions may be added if required.



NOTE: GetCustomOperations() may be called several times, so its implementation should be trivial. That is, it should return a pointer to a static array. The last parameter of the array must be a struct, RB_UaCustomOperation_t, for which all fields are NULLed.

In the following example, there is an implementation of a Custom Operation called HelloWorld. When the operation is called during the update process, the function HelloWorldFunc is called. The linkage is specified by the RB_UaCustomOperation_t[] array, which is returned from the UA call to GetCustomOperations().

The function HelloWorldFunc is called twice (if it appears in the UPG configuration file):

- The first time, the action parameter has a value of RB_UA_ACTION_SCOUT: The function performs the scout operation.
- The second time, the action parameter has a value of RB_UA_ACTION_UPDATE: The function performs the actual update.
- The implementation should ignore other values for the action parameter.

```
#include "custom_ops.h"
#include "RB_UaCustomApi.h" // Needed in order to use API in Custom
Operations

#ifdef NULL
#define NULL ((void*)0)
#endif

int HelloWorldFunc(const char *part_name, const char *part_path,
CustomOpAction_t action, int argc, char *const argv[], void *rb_ua_ctx)
{
    switch (action)
    {
        // The following operation occurs during the scout stage. No change
        // to the real file system should occur at this stage.
        // If an error is returned, the whole update fails, and no change
        // occurs.
        case RB_UA_ACTION_SCOUT:
            RB_UaLogPrint(rb_ua_ctx, "%s: Scouting\n", __func__);
            break;
    }
}
```

```
// The following operation occurs during the update stage. This
// happens after all the partitions were scouted and match the
// source in the delta.
case RB_UA_ACTION_UPDATE:
    RB_UaLogPrint(rb_ua_ctx, "%s: Hello partition %s!\n", __func__,
        part_name);
    break;

// Note: In the future, there may be more actions. 'default'
// section must be provided
default:
    RB_UaLogPrint(rb_ua_ctx, "Hello world! unknown operation\n");
}
return 0;
}

static RB_UaCustomOperation_t RB_UaCustomOperations[] = {
    { "HelloWorld", HelloWorldFunc },
    // Add your functions here, in the same format

    // Must be last
    { NULL, NULL }
};

RB_UaCustomOperation_t *GetCustomOperations()
{
    return RB_UaCustomOperations;
}
```

14.4.2 Using the Custom Operations API

The UA supplies a variety of APIs to be used when implementing Custom Operations. The APIs cover the following services:

- **Logging services:** The ability to write to the UA log file.
- **Data access services:** These services supply access to the data file specified in the UPG configuration file.

Those APIs are declared in **RB_UaCustomApi.h**. Some of the API functions have predefined error values. In case of other errors, the UA log contains more information about the cause of the error.

Logging Services

- **RB_UaLogPrint(void *rb_ua_ctx, const char *format, ...):** This function can be used to print a message the UA log file (if it exists).

Data Access Services

You can send a file to be used as data to the Custom Operation. The UA supplies the following API functions to access this data:

- `RB_UaOpenData(void *rb_ua_ctx, char *id)`: Opens the data file for access.
- `RB_UaGetData(void *rb_ua_ctx, unsigned long offset, unsigned long size, void *buf)`: Reads data from the data file.
- `RB_UaCloseData(void *rb_ua_ctx)`: Closes the data file.
- `RB_UaGetDataSize(void *rb_ua_ctx, unsigned long *data_size)`: Returns the size of the data file.

For more information about API functions, see the *vRapid Mobile API Reference*.

14.5 Failsafe Design Principles

Red Bend has expended much effort and thought in the design of the UPI to ensure that the update operation successfully updates the device, even in the case of power failure.

The UA enables failsafe operation using the following guidelines:

- The UA treats every partition update as a single update unit. That is, if a pre-update Custom Operation completes successfully and then there is a power failure during the update process, when the device reboots the UA calls the pre-update Custom Operation. This is necessary in case the pre-update operation performs a run-time preparation, such as changing the mount type to R/W from Read-Only.
This also applies during a post-update operation; a power failure can occur immediately after the update operation completes, but before the UA stores its current state in a file. When the device reboots, the whole partition update process restarts.
- If a power failure occurs after the update process of a different partition starts, the UA does not call the update operation of the first partition.

To ensure that the custom update succeeds, the following rules must be applied when implementing a Custom Operation function:

- All checks that can reject the delta must be performed during the scout phase. This includes, but is not limited to:
 - Verifying the device's version and hardware
 - Verifying the source image to be changed

There is no need to check the integrity of the sent data file; this is handled by the UA.

- Remember that a power failure may occur at any point of the custom update.
 - Implement every portion so that it can be run again, even if the operation has already completed. For example:
 - To delete or move a file: Delete or move the file if it exists, but do not fail if the file does not exist.
 - To change data in a file based on its content (e.g. making free space in the middle of a file): Modify the source of the data in an atomic operation. That is, do not change the file directly: copy it to a temporary file and then rename the file to its original location. This way, if a power failure occurs while changing the file, the original file is not harmed.
 - If it is not possible to re-perform an operation, save the current state to permanent storage and read it at the beginning of the Custom Operation. Make sure to evaluate

this option carefully since it may introduce other issues: It is possible that after the termination of the current operation, a power failure occurs and the UA calls the operation again.

14.6 Using Custom Operations to Update the Bootloader Partition

The bootloader partition requires special treatment so that its update is failsafe. This is because it has a unique role in the device boot sequence: the bootloader is responsible for initiating the OS. If there is a power failure during the bootloader update, the device cannot boot. Since the UA resides in the OS (both main and recovery), it is impossible to re-run it to recover from the power failure.



NOTE: In some device architectures there is more than one partition that is loaded prior to the OS. This section relates to all such partitions.

Before updating the bootloader, you should verify how the bootloader partition can be updated in a safe manner. This is very device-specific. Possibilities include, but are not limited to:

- There is a secondary bootloader that is used to launch the OS if the primary bootloader is in the middle of an update. It is invoked automatically when the device recognizes that the primary bootloader is unstable.
In this case, no special Custom Operation is needed.
- There is secondary bootloader: a flag must be set in flash to indicate that it is the active bootloader.
In this case:
 - Add a pre-update operation to set this flag.
 - Add a post-update operation to reset the flag.
- There is no secondary bootloader. The update cannot be performed by the UA. Instead, it should be performed by a process in one of partitions that are booted prior to the bootloader
In this case, declare a Custom Operation. The scout operation can be performed in the recovery kernel (by passing the source signature as a parameter to the Custom Operation and verifying that it matches the bootloader source signature). The update operation can include handing off the needed delta (passed as data to the Custom Operation) to a predefined location, signaling that there is a pending update, and rebooting the device. After the device reboots, the dedicated partition that performs the update applies the delta, writes the result of the update, and reboots the device.

After rebooting, the new bootloader is invoked and the UA runs again. The UA recognizes that it was rebooted in middle of an update and calls the Custom Operation of the bootloader again. The Custom Operation should check the result value and return this value to mark whether the update was successful or not. An example of such function is presented in the following section.

14.6.1 Updating a Partition that is Executed before the OS

To implement the hand-off of a delta to an update process in another partition, use configuration files similar to the examples presented in this section.

In the UPG configuration file, add an update to the bootloader partition.

```
<Partition>
  <!-- Update bootloader by using proprietary Update Agent -->
  <PartitionName>bootloader</PartitionName>
  <PartitionType>PT_FOTA</PartitionType>
  <CustomOperation>
    <Type>Update</Type>
    <Operation>UpdateBootloader 0xdabad00</Operation>
    <Data>bootloaderDelta</Data>
  </CustomOperation>
</Partition>
```

In the UA, add a Custom Operation. (The following code sample is a skeleton; complete it with needed functions, error handling, and other changes.)

```
int UpdateBootloaderFunc(const char *part_name, const char *part_path,
    CustomOpAction_t action, int argc, char *const argv[],
    void *rb_ua_ctx)
{
    switch (action)
    {
        // During Scout, make sure the signature matches
        case RB_UA_ACTION_SCOUT:
            RB_UaLogPrint(rb_ua_ctx, "%s: Scouting %s\n", __func__,
part_name);
            validateSignature(part_path, argv[1]); // Implemented by
customer
            break;

        // Perform handoff. It is called again after the update is done
        case RB_UA_ACTION_UPDATE:
            if (hasResultFile())
            {
                int res = getUpdateResult();
                RB_UaLogPrint(rb_ua_ctx, "%s: Update %s done, res is
%d!\n",
__func__, part_name, res);

                // Need to remove result file, so next updates will not
                // mistake to think that update was already performed.
                // However, in case of power failure after the result is
                // removed, the update will occur again. The underlying
                // Update process should be able to handle such case.
                removeResultFile();
            }
            return res;
    }
}
```

```
    }
    else
    {
        long data_size;
        char *delta;

        // Extract the bootloader delta from the UA delta
        RB_UaOpenData(rb_ua_ctx, NULL));
        RB_UaGetDataSize(rb_ua_ctx, &data_size);
        delta = malloc(data_size);
        RB_UaGetData(rb_ua_ctx, 0, data_size, delta);

        // Perform the handoff. This function also reboots the
device.

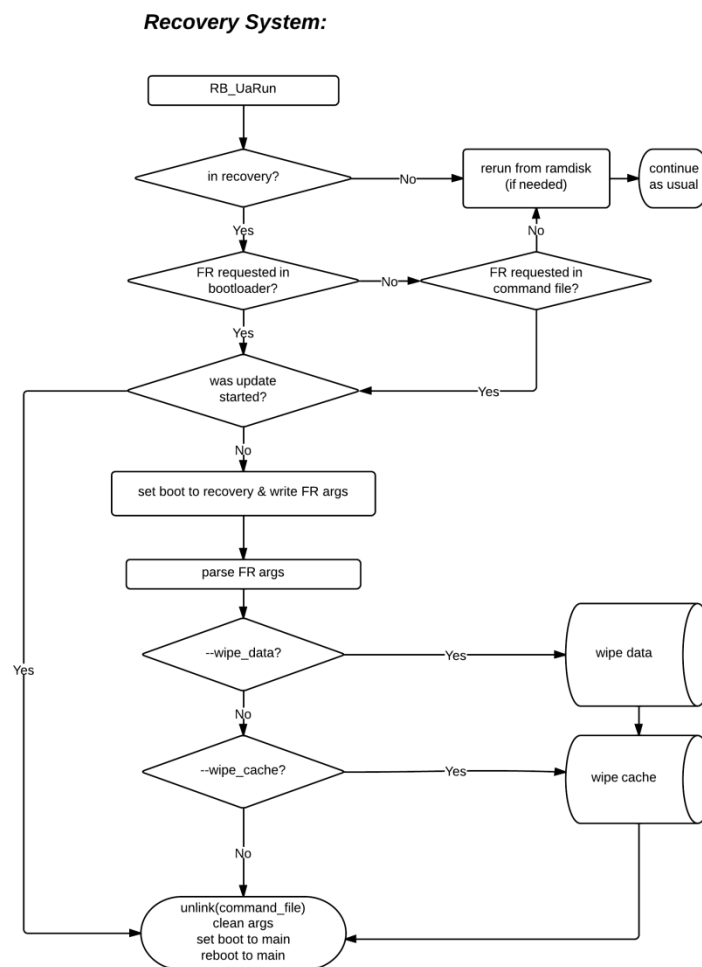
        doHandOff(delta, data_size);
        // Should not reach here.
    }
    break;

    // Note: In the future, there may be more actions. 'default'
// section must be provided
default:
    break;
}

return 0;
}
```

15 Deleting Data and the Cache (Factory Reset)

Android supports a Factory Reset operation that wipes all data and the cache. In the main system, when the end-user selects **Factory Data Reset** from the **Settings** menu, the Android API writes an FR command to `/cache/recovery/command` and immediately reboots to the recovery system. When the UA starts to run in the recovery system it checks for an FR command; if it finds an FR command and no update was started, it formats the data and cache partitions.



To change the path to the FR command file, use the UA `--recovery_command` parameter.

16 Update Agent Log

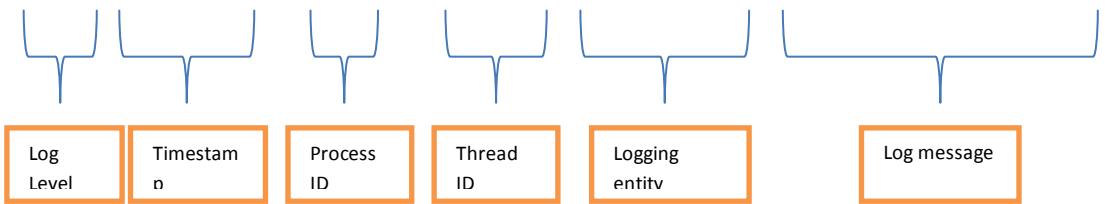
The UA writes its log to a file whose path is provided by the parameter `log_path`. If this parameter is omitted, the UA does not write to any log file.

Each line in the log consists of:

- **Log Level:** A value between 0 and 10, where 0 indicates the highest importance (errors and main flow) and 10 the lowest (usually debug messages)
- **Timestamp:** Time in seconds since epoch
- **Process ID:** Mainly useful in multi-processor devices
- **Thread ID:** Mainly useful in multi-processor devices
- **Logging entity:** The part of the UA that created this log message
- **Log message**

For example:

< 00 > [2260178] [pid 111 tid 724332] Redbend UA: Calling RB_vRM_Update



Log Level	Timestamp	Process ID	Thread ID	Logging entity	Log message
-----------	-----------	------------	-----------	----------------	-------------

17 Bootloader Update Example

17.1 Overview

This example demonstrates how to migrate a bootloader update from an existing methodology to using Red Bend delta methodology.

The example uses a Custom Operation (see *Adding Custom Operations*).

17.2 The Original Update Methodology

The example uses the LGE Mako device (LG Nexus) methodology.

17.2.1 Bootloader structure

The Mako bootloader comprises six separate partitions: **sbl1**, **sbl2**, **sbl3**, **tz**, **rpm**, and **aboot**.

All the partitions, except **sbl1**, have backup partitions: **sbl2b**, **sbl3b**, **tz**, **rpmb**, and **abootb**.

17.2.2 Update Sequence

The input to the update generator is a single file, `bootloader.img`, which contains images of all the partitions.

- 1 The update generator splits **bootloader.img** into separate images. Every image is stored as a file in the package.
For example: **bootloader.sbl2.img**, **bootloader.tz.img**, etc.
- 2 The update is sent to the device.
- 3 All other (non-bootloader) deltas are applied.
- 4 Before updating the bootloader, a 32 byte file is written to the **misc** partition. This file contains `updating-bootloader` and is padded with NULLs.
- 5 Each image is extracted to its target location.
For example, **bootloader.aboot.img** is extracted to `/dev/block/platform/msm_sdcc.1/by-name/aboot` (the mappings are in **recovery.fstab**).
- 6 After the update finishes, a 32 byte file is written to the **misc** partition. This file contains only NULLs and overwrites the file created in step 4.
- 7 The boot images are backed up to **sbl2b**, **abootb**, etc., if such partitions exist on the target device.

17.2.3 Failsafe Considerations

When the device starts up, **sbl1** is invoked (this partition has no backup partition, so it should not be updated on released devices). It reads the **updating-bootloader** flag; if the flag is not set, it uses

the original set of bootloader partitions, otherwise it uses the backup set (the partitions whose names end with **b**).

If a power failure occurs when the partitions are being updated, the device uses the old backup set to boot to recovery and continue the update.

If a power failure occurs after the bootloader update finishes, the new bootloader partitions are used to boot the device. The update is not completed so the device boots to recovery and the update continues until it finishes.

17.3 Red Bend Custom Operation Methodology

This example is for vRapid Mobile version 8.3 and higher. Future releases of vRapid Mobile will simplify this methodology.

The original method, described in the previous section, does not update using a delta, but uses full-release of the images. Red Bend recommends using a delta whenever possible, since this provides a faster update time.

17.3.1 File Structure

The following files are assumed to exist.

Partition	Source	Target
sbl1	source/bootloader.sbl1.img	target/bootloader.sbl1.img
sbl2	source/bootloader.sbl2.img	target /bootloader.sbl2.img
sbl3	source/bootloader.sbl3.img	target /bootloader.sbl3.img
tz	source/bootloader.tz.img	target /bootloader.tz.img
rpm	source/bootloader.rpm.img	target /bootloader.rpm.img
aboot	source/bootloader.aboot.img	target /bootloader.aboot.img

Bootloader Flags

File used to set the flag	bootloader-flag.txt
File used to unset the flag	bootloader-flag-clear.txt

In this example, the update can be done using a predefined Custom Operation. This is described in the following section (see *Using the Predefined Custom Operation*). This section shows how to implement the update using a Custom Operation that can be altered for other devices.

17.3.2 Implementing setBootloaderFlag

The bootloader update requires a new Custom Operation (`setBootloaderFlag`) in the Update Agent.

This Custom Operation receives a file as its data and writes it to a given partition. It receives the device path as a parameter.

The function is implemented in the **custom_ops.c** file (see *custom_ops.c*) and references the function in the declaration of `RB_UaCustomOperations()`.

The following additions are required in **custom_ops.c**.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>

// Extract the given data, and write to the path given as an argument
int setBootloaderFlagFunc(const char *part_name, const char *part_path,
    CustomOpAction_t action, int argc, char *const argv[], void *rb_ua_ctx)
{
    int fd = -1;
    int ret = 1; // Non-zero value indicates error
    unsigned long dataSize;
    struct stat statBuf;
    char *dataBuf = NULL;

    // Make sure we have data and argument
    ret = RB_UaOpenData(rb_ua_ctx, NULL);
    if (ret)
    {
        RB_UaLogPrint(rb_ua_ctx, "Data was not given to this operation\n");
        return 1;
    }

    if (argc != 2 || stat(argv[1], &statBuf))
    {
        RB_UaLogPrint(rb_ua_ctx,
            "Target partition was not given or does not exist\n");
        ret = 1;
        goto end;
    }

    // On scout, just verify we have the parameter and data
    if (action == RB_UA_ACTION_SCOUT)
        goto end;

    // Get data size, read it and write it to the given partition
    // NOTE: No need to use RB_UaLogPrint() in this error handling, since the
    // Custom Operation API already prints the error to the log
    ret = RB_UaGetDataSize(rb_ua_ctx, &dataSize);
    if (ret)
        goto end;
```

```
dataBuf = (char *)malloc(dataSize);
if (!dataBuf)
    goto end;

ret = RB_UaGetData(rb_ua_ctx, 0, dataSize, dataBuf);
if (ret)
    goto end;

fd = open(argv[1], O_RDWR);
if (fd == -1)
{
    RB_UaLogPrint("Cannot open %s, %s\n", argv[1], strerror(errno));
    ret = -1;
    goto end;
}

ret = write(fd, dataBuf, dataSize);
if (ret != dataSize)
{
    RB_UaLogPrint("Write to %s failed, %s\n", argv[1], strerror(errno));
    goto end;
}

// Make sure to sync the flag before updating the bootloader itself
ret = fsync(fd);
if (ret)
{
    RB_UaLogPrint("fsync failed, %s\n", strerror(errno));
    goto end;
}

end:
if (fd != -1)
    close(fd);

if (dataBuf)
    free(dataBuf);

RB_UaCloseData(rb_ua_ctx);

return ret;
}
```

The following reference must be added in the declaration of `RB_UaCustomOperations()`.

```
static RB_UaCustomOperation_t RB_UaCustomOperations[] = {
    { "HelloWorld", HelloWorldFunc },

    // Put here your functions, in the same format
    { "setBootloaderFlag", setBootloaderFlagFunc },
}
```

```
// Must be last
{ NULL, NULL }
};
```

17.3.3 Update Using Delta

The next step is to generate a regular Red Bend delta to apply on all the bootloader partitions including the flags to be set in the correct places.

Use the following UPG configuration file.

```
<vrm>
  <DeviceOs>Android</DeviceOs>
  <CreateDp>/QA/tests/RB_tools/bgu/CreateDp</CreateDp>

  <!--
    The order of the image updates in the config file fits to the order
    during the update. Note that FileSystems and images are updated
    separately, so there is no guarantee that image will be updated prior
    to fileSystem, even if it appears before it in this configuration file
  -->

  <!-- Update here the system and boot partitions (or any other partition -->

  <Partition>
    <!-- Set bootloader flag before updating first bootloader partition -->
    <CustomOperation>
      <type>pre</type>
      <operation>setBootloaderFlag /dev/block/platform/msm_sdcc.1/by-
name/misc</operation>
      <data>bootloader-flag.txt</data>
    </CustomOperation>
    <PartitionName>sb11</PartitionName>
    <SourceVersion>source/bootloader.sb11.img</SourceVersion>
    <TargetVersion>target/bootloader.sb11.img</TargetVersion>
  </Partition>

  <Partition>
    <PartitionName>sb12</PartitionName>
    <SourceVersion>source/bootloader.sb12.img</SourceVersion>
    <TargetVersion>target/bootloader.sb12.img</TargetVersion>
  </Partition>

  <!-- ... -->

  <!-- Last bootloader partition -->
  <Partition>
    <PartitionName>aboot</PartitionName>
    <SourceVersion>source/bootloader.aboot.img</SourceVersion>
    <TargetVersion>target/bootloader.aboot.img</TargetVersion>
    <!-- After updating last bootloader, declare post operation to clear the
      bootloader flag -->
```

```

    <CustomOperation>
      <type>post</type>
      <operation>setBootloaderFlag /dev/block/platform/msm_sdcc.1/by-name/misc
</operation>
      <data>bootloader-flag-clear.txt</data>
    </CustomOperation>
  </Partition>

  <!-- And now, update the backup bootloader partitions -->
  <Partition>
    <PartitionName>sbl2b</PartitionName>
    <SourceVersion>source/bootloader.sbl2.img</SourceVersion>
    <TargetVersion>target/bootloader.sbl2.img</TargetVersion>
  </Partition>

  <Partition>
    <PartitionName>sbl3b</PartitionName>
    <SourceVersion>source/bootloader.sbl3.img</SourceVersion>
    <TargetVersion>target/bootloader.sbl3.img</TargetVersion>
  </Partition>

  <!-- ... -->

</vrm>

```

In this delta, after all the images have been updated, the update of the **sbl1** partition starts. The bootloader flag is set and the update of all the bootloader partitions starts.

After the bootloader partitions are updated, the bootloader flag is reset (as a post-update operation on the **aboot** partition, which is the last partition in this example). Now the backup partitions can be update safely. If a power failure occurs, the regular partitions are used to boot the device.

17.4 Using the Predefined Custom Operation

If simple data is to be written to a partition defined in the UA `partitions_list` parameter, the predefined `writeData` operation can be used (see *Predefined Custom Operations*). This operation writes the given data to the specific partition. To use this operation, no change is required in the UA.

The UPG configuration file should be the same as in the previous section, except for the lines in the `CustomOperation` sections.

```

<!-- Set bootloader flag before updating first bootloader partition -->
  <CustomOperation>
    <type>pre</type>
    <operation>writeData misc</operation>
    <data>bootloader-flag.txt</data>
  </CustomOperation>

  <!-- After updating last bootloader, declare post operation to clear the
    bootloader flag -->

```

```
<CustomOperation>
  <type>post</type>
  <operation>writeData misc</operation>
  <data>bootloader-flag-clear.txt</data>
</CustomOperation>
```


Terms and Abbreviations

Term	Description
APN	Access Point Name
Application Core Layer	An Application Layer sub-layer that contains the SMM
Application Layer	<p>The part of the SWM Client containing the business logic and managing communication external to the device, such as networking and the UI</p> <p>The Application Layer is built on the Framework and contains the Application Core Layer, BLL, and DIL sub-layers.</p>
aufs	advanced multi layered unification filesystem
BBM	Bad Block Management
BBT	Bad Block Table
BL Event	<p>An event typically generated by the end-user or another external source, such as an incoming call or a message received from a DM Server</p> <p>The DIL passes BL events to the BLL. BL events are queued before they are processed by the business logic.</p> <p>Certain BL events are generated by the Framework internally or by the DIL in response to requests (DIL events) from the BLL.</p>
Boot Phase	When the UA operates using background update, the firmware update phase that occurs during a device restart
boot.img	The image of the device main system
Business Logic	<p>One or more state machines within the BLL</p> <p>The business logic responds to BL events, instructs the Engine to perform actions, and generates internal and DIL events.</p>
Business Logic Layer (BLL)	A mostly platform-independent Application Layer sub-layer that contains the business logic and one side of the Event Streamer
CI	Client-initiated: a communication with the DM Server initiated by the end-user (user-initiated) or by the device (device-initiated)
cramfs	Compressed read-only Linux file system

Term	Description
CRC	Cyclic Redundancy Check
DD	Download Descriptor: a small packet that contains information about the name, size, and location of a DP that should be downloaded
Device Integration Layer (DIL)	A platform-dependent Application Layer sub-layer that contains the UI and platform traps
DIL event	<p>An event that typically results in a UI screen presented to the end-user</p> <p>The BLL generates DIL events and sends them to the DIL. Some DIL events are handled directly by the DIL without any change in the UI.</p>
DL	Download
DLOTA	Download Over-the-Air
DM	Device Management
DM	Device Management
DM Core Layer	The OMA-DM Core Layer contains the Engine and other main parts of the vDirect Mobile Framework
DMC	Device Management Client
DP	Delivery Package
DP	<p>Deployment Package</p> <p>A file containing one or more software components, applications, or updates that is downloaded, via OMA-DM, to devices being updated</p>
Engine	<p>Red Bend's vDirect Mobile Engine</p> <p>The Engine contains the core library of the vDirect Mobile Framework</p>
Event Streamer	<p>A process of communication between the DIL and BLL</p> <p>Communication is typically based on a TCP/IP sockets or APIs.</p>
Extended Framework	Red Bend's vDirect Mobile Framework with the Application Core Layer
FOTA	Firmware Over-the-Air
FOTA	Firmware Over-the-Air

Term	Description
Framework	<p>Red Bend's vDirect Mobile Framework</p> <p>The Framework contains the OMA-DM Core and Porting Layers</p> <p>The Application Layer communicates with the Framework using a comprehensive list of API and callback functions.</p>
FUMO	Firmware Update Management Object
FUSE	Filesystem in Userspace
GCM	Google Cloud Messaging
GOTA	Google's recovery system
IPC	Inter-Process Communication
IPL	Integration Point Layer
IPL	Integration Porting Layer
JFFS	Journaling Flash File System
LAWMO	Lock And Wipe Management Object
LFS	Linear File System
LZO	A lossless data compression algorithm
M2M	<p>Machine-to-Machine</p> <p>M2M implies resource-constrained devices with different characteristics than mobile devices</p>
Main System	The device main system, loaded from boot.img
MBB	Mobile Broadband
MMI	Man Machine Interface
MMU	Memory Management Unit
MO	Management Object
MSM	<p>Mobile Software Management</p> <p>Remotely deploying and manipulating software assets on mobile devices</p>

Term	Description
NI	Network Initiated: a communication with the DM Server initiated by the DM Server (also called server-initiated)
NIA	Network-Initiated Alert
NSS	Network Shared Secret
OMA	Open Mobile Alliance
OMA	Open Mobile Alliance
OTA	Over-the-Air
Platform Traps	DIL functionality that handle external and device generated events, such as incoming phone calls and messages from the DM Server These events are passed as BL events to the BLL.
Porting Layer	A platform-dependent Framework layer that contains functions used by the Application Layer and OMA DM Core Layer
Post-Boot Phase	When the UA operates using a full background update, the firmware update phase that occurs after a device restart
Pre-Boot Phase	When the UA operates using background update, the firmware update phase that occurs before a device restart
PUL	Pre-update location Any device location where updated files are temporarily stored during the intermediate phases of the full background update process
R/O	Read-Only
R/W	Read-Write
RBBM	Red Bend Bad Block Management Solution
Recovery System	The device recovery system, loaded from recovery.img
recovery.img	The image of the device recovery system
ROFS	Read-Only File System

Term	Description
ROMFS	ROM File System A Red Bend mechanism that provides a means for the UPG to update a file system as an image
SCOMO	Software Component Management Object
Scout	The operation of verifying that the flash or file content matches the source version used when generating the update
SD card	Secure Digital card A non-volatile memory card format used in portable devices, such as mobile phones , digital cameras , GPS navigation devices , and tablet computers
State Machine Manager (SMM)	Libraries for selecting and implementing the business logic; part of the Application Core Layer
SWM	Software Management A Red Bend solution for managing software components on mobile devices that may include applications and firmware
UA	Update Agent Device-resident software component that calls the UPI to perform updates
UPG	Update Generator vRapid Mobile product that generates updates of device firmware
UPI	Update Installer Device-resident vRapid Mobile component that installs updates on a device
User Interface (UI)	A DIL component that passes UI screens to the end-user, and passes information entered by the end-user to the BLL as BL events
vDM	Red Bend's vDirect Mobile
YAFFS	Yet Another Flash File System