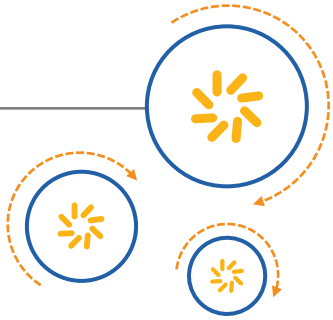
Qualcomm Technologies, Inc.



Qualcomm® Snapdragon Flight ™

Developer Guide

80-P8822-2 Rev. A

October 26, 2016

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	October 2016	Initial release

Contents

Figures

Tables

1 Introduction

This document is the developer guide for the Qualcomm® Snapdragon Flight™ software.

1.1 Acronyms, abbreviations, and terms

Table 1-1 provides definitions for the acronyms, abbreviations, and terms used in this document.

Table 1-1 Acronyms, abbreviations, and terms

Term	Definition
adb	Android debug bridge
aDSP	Application digital signal processor
API	Application programming interface
ARM	Advanced RISC machines
CTS	Clear to send
DSP	Digital signal processor
DSPAL	DSP abstraction layer
EIS	Electronic image stabilization
I2C	Inter-integrated circuit
IMU	Inertial measurement unit
RTS	Receive to send
SOC	System on chip
SPI	Serial peripheral interface
TZ	Trust zone
UART	Universal asynchronous receiver transmitter
VIO	Visual inertial odometry

1.2 References

The references listed in Table 1-2 provide additional information about topics discussed in this document.

Table 1-2 References

Title	DCN or URL
Qualcomm Snapdragon Flight User Guide	80-P8822-1
Qualcomm Snapdragon Linux Camera Interface: Programming Manual	80-P3945-1
Snapdragon Flight Documentation on GitHub	GitHub link
Snapdragon Flight Information on Qualcomm Developer Network	QDN link

2 ARM SDK

2.1 What is the software development kit?

The software development kit (SDK) is a cross-development toolkit for the advanced RISC machines (ARM) apps processor in the Snapdragon system on chip (SOC). The SDK allows developers to compile and link applications meant for the ARM processor on their x86-64 based desktop, and then to deploy them on-target.

Once installed, the SDK provides:

- The gcc cross-toolchain, which is downloaded from the Linaro servers
- The include files and libraries required to compile and link user space applications that run on the apps processor
- The environment setup script environment-setup-<target> needed by the cross-toolchain

2.2 How to install the software development kit

Complete the installation of the qrlSDK package that is part of the Snapdragon Flight platform.

Refer to the Release Notes for instructions on how to generate the qrlSDK.

1. Find the qrlSDK.tgz file at build/tmp-eglibc/deploy/sdk in your build tree.
2. Untar the file in the desired location.
3. Run the installer from the untarred files, giving it the source of the SDK and the destination of where to install it.

The recommendation is for the developer to install the SDK in a suitable location such as /opt/data/qrlSDK, /local/mnt/qrlSDK, or /usr/local/qrlSDK.

Example:

Install the tarball at `/path/to/qrlSDK.tgz` at the location `/dir/to/install/sdk`.

```
> ls /path/to/qrlSDK.tgz
qrlSDK.tgz
> ls /dir/to/install/sdk
ls: cannot access /dir/to/install/sdk: No such file or directory
> mkdir -p /dir/to/extract/sdk
> tar zxf /path/to/qrlSDK.tgz -C /dir/to/extract/sdk
> /dir/to/extract/sdk/qrlSDKInstaller.sh -s /dir/to/extract/sdk -d
/dir/to/install/sdk
```

2.3 How to compile and run the sample app

A sample app is included in the SDK tarball.

To compile the included sample app:

```
> cd /dir/to/extract/sdk/sample
> source /dir/to/install/sdk/environment-setup-<target>
> make
```

Use the command adb push to push the app to the target and run it.

2.4 What does the environment setup actually do?

The environment setup sets environment variables like CC, CFLAGS, LDFLAGS, so you can source this file, then use a Makefile for the build. The environment variables provide the variables $CC, $CFLAGS, etc., to the Makefile, making it simple to compile/link your code.

The environment setup also includes passing --sysroot=xxx to the gcc so the include file and libraries used on the target can be linked with your code.

NOTE: Sourcing the SDK's environment setup script renders the user's shell unusable for normal operations other than cross-compilation for the Snapdragon Flight apps processor applications. For instance, the adb will not work. It is suggested that users open another shell to do other shell operations.

3 Applications Processor Inertial Measurement Unit API

This chapter describes the API and libraries required to obtain the inertial measurement unit (IMU) data at the applications processor from the aDSP independent of the flight stack running on aDSP.

Currently, the common interface supports obtaining the following information:

- Raw IMU data.
- Bias compensated IMU data
- Attitude estimation

This is an API only. The data and API implementation are dependent on the flight stack being run.

3.1 Background

In the current platform, the IMU data is received by the aDSP. This release provides an aDSP driver that accesses the IMU data. The IMU data obtained on the aDSP are not directly accessible by the applications processor. There are various use cases where this may be useful, such as electronic image stabilization (EIS), and visual inertial odometry (VIO).

To access IMU data on apps processor, we provide a dynamic module and set of APIs to access IMU data through the FastRPC mechanism.

The IMU data consist of the following:

- Timestamp
- Temperature in Celsius
- Linear acceleration in all three axes in IMU frame coordinates in units of g/s^2
- Angular velocity in all three axes (x, y, and z) in IMU frame coordinates in units of rad/sec

The IMU data obtained are independent of the flight stack (PX4, Qualcomm® Snapdragon Navigator™, or no flight stack). The frame reference is the same as the IMU on the board. To convert the data to the frame reference shown in Figure 3-1, call the *GetImuFrameToBodyFrameRotationMatrix()* API.

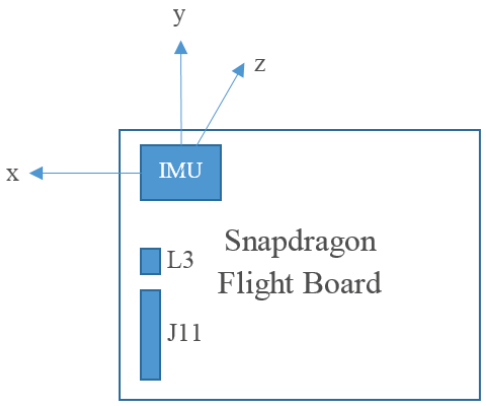


Figure 3-1 IMU data frame reference

3.2 API

The sensor_imu_attitude_api provides the APIs to initialize the interface and get the IMU data onto the applications processor. The API assumes the MPU9x50 driver is initialized and generates the IMU data either by the flight stack or a standalone app.

The timestamp of the data reported by the API is synchronized to the applications processor. The clock reference to use is provided as a parameter to the Initialize function.

The required header file and the libraries are located in the Hexagon flight controller add-on (qcom_flight_controller_hexagon_sdk_add_on.zip file). Contact the support team to identify the location of this file in the Snapdragon Flight software release.

NOTE: The API was updated from C++ to a C-Style API in order to enable custom vendors to provide different implementations of the API for clients.

Table 3-1 lists the files required for development.

Table 3-1 Files required for development

Filename	Location in ZIP file	Description
sensor_imu_api.h	flight_controller/krait/inc	Header file to obtain the IMU data.
sensor_datatypes.h	flight_controller/krait/inc	Header file to include the data types required to access IMU.
libsensor_imu.so	flight_controller/krait/libs	Dynamic library that provides the service of getting the IMU data.

3.2.1 sensor_handle* sensor_imu_attitude_api_get_instance()

This function gets the instance/handle for the API. This should be called first in order to get access to other APIs.

3.2.2 char* sensor_imu_attitude_api_get_version()

Returns the API version.

3.2.3 int16_t sensor_imu_attitude_api_initialize()

This function initializes the communication from the apps processor and the aDSP using the FastRPC mechanism, while also loading the respective OpenDSP dynamic module.

This function also takes the clock reference to use to synchronize the aDSP data to the applications processor's clock.

3.2.4 int16_t sensor_imu_attitude_api_terminate()

This function terminates the FastRPC link to the aDSP. This API must be called at the end of the application running on the applications processor or when the application is exited.

This API ensures proper release of aDSP resources upon the exit of the application.

3.2.5 int16_t sensor_imu_attitude_api_wait_on_driver_init()

This function ensures the successful initialization of the IMU driver from either the flight stack or a standalone app. This function must be invoked after the API is initialized to block calls.

3.2.6 int16_t sensor_imu_attitude_api_get_imu_raw()

This function gets the IMU data from the aDSP. Refer to the sensor_imu_api.h header file for the actual parameters passed to the function.

This is a blocking call if there is no data available. On the aDSP, the maximum number of samples buffered is 100. If the buffer is full, the oldest data is overwritten. The buffer limitation is based on the MPU9x50 driver provided in the qcom_flight_controller_hexagon_sdk_add_on.

This function returns all the IMU samples buffered from the time it was previously invoked.

The IMU data returned by this API are raw values with no calibration offsets applied. The frame reference is in the co-ordinates of the MPU9250 (see Figure 3-1).

3.2.7 int16_t sensor_imu_attitude_api_get_mpu_driver_settings()

After the MPU9x50 driver initialization, this API provides options to obtain the accel/gyro sample rate in Hz, compass sample rate in Hz (-1 is returned if the compass is not enabled), and the accel/gyro LPF setting in Hz.

This API should be called after the *sensor_imu_attitude_api_initialize()* and *sensor_imu_attitude_api_wait_on_driver_init()* API calls.

3.2.8 int16_t sensor_imu_attitude_api_get_bias_compensated_imu()

This provides the IMU data with bias compensation applied to the raw IMU data. This data is dependent on a flight stack running on the aDSP. See the header file for additional information on using the API. The returned IMU data is in the MPU9250 frame reference (see Figure 3-1).

3.2.9 int16_t sensor_imu_attitude_api_get_attitude()

This provides the attitude estimates as reported by the flight stack buffered since the last call to the API. This is dependent on the flight stack running on the aDSP. The attitude is represented as a 3x3 matrix. Refer to the header files for additional information.

3.2.10 bool sensor_imu_attitude_api_is_flight_stack_enabled()

This is a utility interface to check if a flight stack is running on the aDSP. This may be used before calling any of the flight stack-dependent APIs listed above.

3.2.11 int16_t sensor_imu_attitude_api_imu_frame_to_body_frame_rotation_matrix()

This is also a utility API to facilitate the translation of the raw IMU frame reference to Snapdragon Flight Body Frame. This is a 3x3 matrix.

To get the translation, do the following:

IMUbody_frame(1x3) = IMUraw(1x3) * ImuFrameToBodyFrameRotaionMatrix(3x3)

If additional translations are needed based on bodyFrame, then the new translation can be combined by doing the following:

FinalImuTranslation(3x3) = ImuFrameToBodyFrameRotationMatrix(3x3) * NewAdditionalTranslaton(3x3)

NOTE: All of the above are 3x3 matrices.

3.2.12 DataType: sensor_imu

The *sensor_imu* structure provides the details of the IMU data returned by aDSP. For more details, see *flight_controller/krait/inc/sensor_datatypes.h.*

3.2.13 DataType: sensor_rotation_matrix

This is the matrix used to store/report the attitude information. For more details, see *flight_controller/krait/inc/sensor_datatypes.h.*

3.3 Sample standalone app

The release includes a sample app *imu_app* to generate imu data at a desired sample rate (found in the flight_controller\krait\apps of the qcom_flight_controller_hexagon_sdk_addon.zip file).

```
Usage:
imu_app [-s <sample_rate>] [-f] [-h|--help]
[-s <sample_rate>] --> range is [0,4] inclusive: Default=2
                   0  == 100 Hz
                   1  == 200 Hz
                   2  == 500 Hz
                   3  == 1000 Hz
                   4  == 8000 Hz
[-f] --> emulate running a flight stack: Default no flight stack.
         This does not run a flight stack, but supports the SensorIMU
         interface by generating the random bias offsets.  The Raw Imu
         interface is not changed. This option will do the following:
            - use dummy bias offsets
         If real/correct bias offsets are needed, use with a valid
          flight stack like Snapdragon Navigator or PX4.
```

Other MPUx50 driver settings set by app as default:

```
gyro_lpf = MPU9X50_GYRO_LPF_184HZ,
acc_lpf  = MPU9X50_ACC_LPF_184HZ,
gyro_fsr = MPU9X50_GYRO_FSR_2000DPS,
acc_fsr  = MPU9X50_ACC_FSR_16G,
gyro_sample_rate = MPU9x50_SAMPLE_RATE_1000HZ,
compass_enabled = true
compass_sample_rate = MPU9x50_COMPASS_SAMPLE_RATE_100HZ
fifo_enabled = false
```

Usage example:

```
> imu_app –s 2
```

NOTE: The standalone app must not be executed concurrently with the flight stack. It might produce unexpected results.

3.4 Build a test application

The qrlSDK from the release is required. For details, see Chapter 2.

1. Call the test source code file: imu_test.cpp
2. Write a simple make file as follows:

```
EAGLE_ADDON_SRC ?= /opt/eagle_addon/flight_controller/
EAGLE_ADDON ?= ${EAGLE_ADDON_SRC}/krait/libs
EAGLE_INCS ?= ${EAGLE_ADDON_SRC}/krait/inc
sensor_imu_test: imu_test.cpp
${CXX} -o imu_test  -D__STDC_FORMAT_MACROS  -I${EAGLE_INCS} -
L${EAGLE_ADDON} imu_test.cpp -lpthread -lrt -lsensor_imu
```

NOTE: This assumes the qcom_flight_controller_hexagon_sdk_add_on.zip is installed in the /opt/eagle_addon folder. Replace with the actual path used.

3. Source the qrlSDK environment path (see Section 2.3). Then run the following:
   ```
   make
   ```
4. Push the test application to the target. ADB or SCP can be used. The following is an example using adb:
   ```
   adb push imu_test /home/linaro/
   ```
5. SSH/ADB to the target and run the application:
   ```
   adb shell
   cd /home/Linaro
   ./imu_test
   ```

3.5 Released artifacts

Table 3-2 Apps processor artifacts

File	Description	Location on target	Location in flight controller add-on
SensorImu.hpp [*Deprecated*]	Header file for the Sensor API.	/usr/include/sensor-imu	flight_controller/krait/inc
sensor_imu_api.h	Header file for the sensor IMU/Attitude	/usr/include/sensor-imu/	flight_controller/krait/inc
sensor_datatypes.h	Header file that includes the data types for the API.	/usr/include/sensor-imu	flight_controller/krait/inc
libsensor_imu.so	Dynamic library for the implementation of the API on apps processor.	/usr/lib	flight_controller/krait/libs
imu_app	Standalone application to allow IMU data to be obtained when there is no flight stack running.	/usr/bin	flight_controller/krait/apps

Table 3-3 aDSP artifacts

File	Description	Location on target	Location in flight controller add-on
libsensor_imu_skel.so	aDSP implementation of the API.	/usr/share/data/adsp	flight_controller/hexagon/libs
libmpu9x50.so	MPU driver for getting IMU data. This driver is also used by the flight stack.	/usr/share/data/adsp	flight_controller/hexagon/libs

3.6 Example developer use-case to get raw IMU data

This section provides an example work flow to get raw IMU samples.

1. Get API handle.

Example code:

```
sensor_handle* api_handle = 0;
api_handle = sensor_imu_attitude_api_get_instance();
if( api_handle == 0 )
{
     std::cout << “Error Getting the API handle for Sensor IMU API” <<
std::endl;
    exit();// or bail out.
}
```

2. Initialize the driver/server.

   Example code:

```
int16_t ret_code = 0;
ret_code = sensor_imu_attitude_api_initialize( api_handle,
SENSOR_CLOCK_TYPE_REALTIME );
if( ret_code != 0 )
{
    Std::cout << “Error initializing the API” << std::endl;
   Exit(); // or bail out.
}
```

3. Wait for the MPU driver to start.

   Since IMU data is dependent on starting/initializing the MPU9x50 chip, it is recommended to wait on this API call. This is optional.

   Example code:

```
int16_t ret_code = 0;
ret_code = sensor_imu_attitude_api_wait_on_driver_init( api_handle );
if( ret_code != 0 )
{
    Std::cout << “Error wait for driver init” << std::endl;
   Exit(); // or bail out.
}
```

4. Get the IMU Raw data.

   After completing the above steps successfully, get the IMU data by calling sensor_imu_attitude_api_get_raw_imu api.

   Example code:

```
int16_t ret_code = 0;
sensor_imu imu_data[100]; // 100 is used as an example.
int32_t returned_sample_count = 0;
ret_code = sensor_imu_attitude_api_get_raw_imu( api_handle, imu_data,
100, &returned_sample_count  );
if( ret_code != 0 )
{
    Std::cout << “Error getting raw imu data” << std::endl;
   Exit(); // or bail out.
}
```

5. Call API terminate call to exit gracefully.

   This should be called at application exit or when access to the API is no longer required.

   Example code:

```
int16_t ret_code = 0;
ret_code = sensor_imu_attitude_api_terminate( api_handle  );
if( ret_code != 0 )
{
    Std::cout << “Error calling api terminate” << std::endl;
}
```

4 Camera API

For more information on how to interface with the Snapdragon camera device, see *Qualcomm Snapdragon Camera Interface: Programming Manual* (80-P3945-1).

5 Software Update

The software update feature refers to support for updating the system using update packages (zip files).

This feature is useful for providing maintenance releases when devices are already deployed in the field and for pushing out new features, security fixes, or bug fixes.

5.1 Leveraging Android

This feature leverages the recovery framework from Android by integrating it into our bitbake build system and Linux OS.

The following Android documentation covers much of the functionality of the feature:

- http://s.android.com/devices/tech/ota/index.html
- http://s.android.com/devices/tech/ota/tools.html
- http://s.android.com/devices/tech/ota/inside_packages.html
- http://s.android.com/devices/tech/ota/device_code.html
- http://s.android.com/devices/tech/ota/sign_builds.html

All UI-based recovery features have been disabled and a command-line-only version of the tools is used.

5.2 Partition layout

Table 5-1 shows the three new partitions for the partition table.

Table 5-1 Modified partition table

Name	Size	Description
Update	512 MB	Used for storing the update package, command file for recovery, and recovery logs.
Factory	512 MB	Used for storing an update package that installs the complete original factory image; it includes Linaro's rootfs.
Misc	30 KB	Used for bootloader commands. LK reads this and looks for a recovery command. If so, then it boots to recovery. Used for fail-safe during recovery update; recovery writes to this block so if the update fails to finish, the LK boots to recovery again. The block is erased after the update finishes.

5.3 Target files

Updates are generated from a zipped output of the build called target files. The folder structure is defined by Android for historical reasons, which includes all the contents from the build needed to recreate the same image. This file needs to be saved for each release so it is possible to create update packages afterwards.

Table 5-2 shows the folder structure.

Table 5-2 Folder structure

Folder	Description
BOOTABLE_IMAGES	Contains boot.img and recovery.img
META	Contains some meta files with information about the device and list of files from ext4 partitions
OTA	Contains the static updater and applypatch binaries
QCOM FIRMWARE	Contains contents of cache.img taken from /lib/firmware in the sysroot. firmware files (added for QR-Linux)
RADIO	Named for historical reasons and contains all non-Linux partitions, including LK, modem, TZ, and others. It also includes filesmap, which is a file with the mapping for the partition corresponding to each file
RECOVERY	Contains the recovery.fstab file. This file is parsed by the recovery binary for mounting the partitions during the update process. It has its own format derived from Android for historical reasons.
SYSTEM	Contains the contents of the system.img (debian packages)
LINARO-ROOTFS	Contains the contents of the Linaro rootfs userdata.img, which is used for factory reset

5.3.1 How to generate target files

The target files zip is created by the target_files task from the qrl-binaries recipe.

Run the following:

```
MACHINE=eagle8074 bitbake qrl-binaries -c target_files
```

The file eagle8074-target_files.zip is in the images output directory.

5.4 Update package

This is also described at http://s.android.com/devices/tech/ota/tools.html.

The update package is generated by running a Python script that processes the target files zip. The script unzips the eagle8074-target_files.zip file and processes all entries to generate an optionally signed, update zip package (eagle8074-ota.zip). A delta update zip package is generated based on two different target_files. The delta update zip package can only be applied on a specific original version number to the target version.

The main Python script is called ota_from_target_files. It has the following main options:

- **n** – Omit the timestamp prereq (checked from build.prop). Normally, the update can be installed only on a build with an earlier timestamp; this option overrides that.
- **d** – Device storage type, which is always MMC.

- **s** – Script with device-specific code used to handle flashing of QTI partitions from the RADIO folder.
- **p** – Additional path used to find binaries for execution by the script. It is appended to the system PATH variable.
- **v** – Verbose, print more debug logs.
- **signapk_path** – Path to the signapk.jar file, which is used for signing the package.
- **k** – Path to the private key to be used for signing the package.

Table 5-3 shows the structure of the full update zip file.

Table 5-3 eagle8074-ota.zip for Snapdragon Flight file structure

File	Description
firmware-update	Contains the files from RADIO folder (all NHLOS mbn/img)
qcom-firmware	Contains the contents of QCOM-FIRMWARE folder; /lib/firmware from the build
recovery	Contains the install-recovery.sh script for multistage upgrades and recovery-from-boot.p, a patch to construct recovery.img from boot.img; both images share the same zImage and most of the ramdisk, so the patch is used to save space
system	Contains the contents of the SYSTEM folder; includes all debian packages and build.prop
boot.img	Contains the kernel bootable image from the BOOTABLE_IMAGES folder
META-INF	Folder containing metadata about the update package
\- com\google\android	Contains the update-binary from the OTA folder; updater, and the update-script generated with help from edify.py using the Edify language (see http://s.android.com/devices/tech/ota/inside_packages.html#edify-syntax); update-binary reads the update-script and executes functions based on a function map
\- com\android	Contains metadata and certificates

5.5 Keys and signing

Public and private key pairs are used to sign and authenticate the update zip packages. Keys can be created using the openssl tool. The zip files are signed using Android's SignApk.jar file and are verified using the MinCrypt library. The whole procedure is also explained at http://www.londatiga.net/it/how-to-sign-apk-zip-files/.

The recovery RAMDisk contains a file called otacerts with all the certificates to be used for verifying signatures. This file is generated by the DumpKey.jar tool from Android, based on the keys specified by the user.

5.5.1 How to generate keys

Each key comes in two files.

- Certificate – Has the extension .x509.pem
- Private key – Has the extension .pk8

The private key must be kept secret and is needed to sign a package. The key may itself be protected by a password. The certificate contains only the public half of the key, so it can be distributed widely. It is used to verify a package has been signed by the corresponding private key.

The recovery update uses 2048-bit RSA keys with public exponent 3. Generate certificate and private key pairs using the openssl tool at openssl.org.

```
# generate RSA key
% openssl genrsa -3 -out temp.pem 2048

# create a certificate with the public part of the key
% openssl req -new -x509 -key temp.pem -out releasekey.x509.pem \
-days 10000 \
-subj '/C=US/ST=California/L=San Narciso/O=Yoyodyne, Inc./OU=Yoyodyne
Mobility/CN=Yoyodyne/emailAddress=yoyodyne@example.com'
# create a PKCS#8-formatted version of the private key
% openssl pkcs8 -in temp.pem -topk8 -outform DER -out releasekey.pk8 -
nocrypt
# securely delete the temp.pem file
% shred --remove temp.pem
```

The openssl pkcs8 command shown creates a .pk8 file with no password, suitable for use with the build system. To create a .pk8 secured with a password (which should be done for all actual release keys), replace the -nocrypt argument with -passout stdin. Openssl encrypts the private key with a password read from the standard input. No prompt is printed, so if stdin is the terminal, the program will appear to hang, though it is waiting for you to enter a password. Other values can be used for the -passout argument to read the password from other locations (see the openssl documentation).

The temp.pem intermediate file contains the private key without any kind of password protection, so dispose of it thoughtfully when generating release keys. The GNUshred utility may not be effective on network or journaled file systems. Use a working directory located in a RAMDisk (for example, a tmpfs partition) when generating keys to ensure the intermediates are not inadvertently exposed.

5.5.2 SignApk.jar

This is the same tool used for signing Android APK files. APKs are zip files with a special structure and are signed with an .x509.pem and .pk8 key pair. OTA packages are signed using this .jar by the ota_from_target_files tool automatically when the -k option is given.

It is also possible to sign the package manually using the following command:

```
java -jar signapk.jar certificate.pem key.pk8 update.zip signed-update.zip
```

5.5.3 DumpKey.jar

The update package is verified by recovery based on the keys loaded in the RAMDisk. These keys are stored in a file in /res/keys. They use a special format created by the dumpkey.jar tool (see http://opengrok.qualcomm.com/source/xref/LA.BR.1/system/core/libmincrypt/tools/).

The keys file is created using the following command:

```
java -jar dumpkey.jar <list of keys (.pem)>
```

The tool uses a specific encoding understood only by the MinCrypt library (http://opengrok.qualcomm.com/source/xref/LA.BR.1/system/core/libmincrypt/).

MinCrypt is used by recovery to verify the update package signature against all keys present in /res/keys.

5.6 Recovery image

Recovery is run as part of a bootable image with the same zImage kernel image and a modified Linaro's RAMDisk.

1. The following is performed to initialize in recovery.img:
2. Mount /dev, /tmp, /sys and other default system mounts.
3. Create links for all partitions by-name under /dev/block/platform/MSM_sdcc.1/by-name/.
4. Copy /system/build.prop to /tmp for use during recovery.
5. Run the recovery binary without arguments. Arguments are read from the command file.
6. Print the recovery log.
7. Check the result from recovery and reboots on success. Open a shell if failed.

NOTE: There is known bug with the shell in Recovery mode where it gets locked if it idles. If this occurs, restart the putty or similar serial console.

5.7 Recovery application

For the recovery app initialization:

1. Set all buffer output (stdio/stderr) to /tmp/recovery.log.
2. Read the arguments from the command file.
3. If a package was given, then proceed; if not, then return error.
   This causes it to stay in /bin/sh.
4. Write the reboot-recovery message in the bootloader control block (/misc partition). This causes the bootloader to reboot to recovery and retry it indefinitely until the bootloader control block is cleared.
5. Parse the updater-script and copy the updater-binary from the package into /tmp.
6. Execute the updater-binary.

For finalization:

7. Check the updater-binary return value.
8. Clear the bootloader control block (/misc) to reboot to the main system.
9. If a failure occurs, print the error message and return a failure value.
10. If successful, write the success message to the update partition. This causes post install to run. Copy logs to /cache/recovery/ and remove the command file.

5.8 Updater-binary

Updater-binary is used for processing the updater-script. The update-script is written in the Edify language (see http://s.android.com/devices/tech/ota/inside_packages.html#edify-syntax). Each call in the language is mapped to a function in install.c using a function table. The updater-binary parses the script into a tree and then executes every function from each node, step-by-step.

For example:

- FormatFn – Used for formatting partitions
- PackageExtractDirFn – Used for extracting directories from the zip file
- PackageExtractFileFn – Used to flash raw partitions and extract the partition file (for example, sbl1.mbn) into the partition dev block

Perform these steps for the full update package:

1. Verify if the device machine name matches the machine name this package is for.
2. Format the system partition.
3. Mount the system partition.
4. Format the cache partition.
5. Mount the cache partition.
6. Extract the recovery from the zip into /system (used for updating the recovery image).
7. Extract the system from the zip into /system.
8. Extract the cache from the zip into /cache.
9. Set permissions for the system and cache.
10. Extract boot.img to the partition dev block.
11. Extract all NON-HLOS images to the partition dev blocks.
12. Unmount the cache and system.

The following is an example taken from an actual update package

```
getprop("ro.product.device") == "eagle8074" || abort("This package is for
\"eagle8074\" devices; this is a \"" + getprop("ro.product.device") +
"\".");
show_progress(0.500000, 0);
format("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/system",
"0", "/system");
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/system",
"/system");
format("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/cache", "0",
"/qcom-firmware");
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/cache",
"/qcom-firmware");
package_extract_dir("recovery", "/system");
package_extract_dir("system", "/system");
package_extract_dir("qcom-firmware", "/qcom-firmware");
```

```
set_perm_recursive(0, 0, 0775, 0644, "/system");
set_perm(0, 0, 0755, "/system");
set_perm(0, 0, 0664, "/system/build.prop");
set_perm_recursive(0, 0, 0775, 0544, "/system/etc");
set_perm_recursive(0, 0, 0755, 0644, "/qcom-firmware");
set_perm_recursive(0, 0, 0775, 0644, "/qcom-firmware/ar3k");
set_perm(0, 0, 0755, "/qcom-firmware/venus.b00");
set_perm(0, 0, 0755, "/qcom-firmware/venus.b01");
set_perm(0, 0, 0755, "/qcom-firmware/venus.b02");
set_perm(0, 0, 0755, "/qcom-firmware/venus.b03");
set_perm(0, 0, 0755, "/qcom-firmware/venus.b04");
set_perm(0, 0, 0755, "/qcom-firmware/venus.mbn");
set_perm(0, 0, 0755, "/qcom-firmware/venus.mdt");
show_progress(0.200000, 0);
show_progress(0.200000, 10);
package_extract_file("boot.img", "/dev/block/platform/msm_sdcc.1/by-
name/boot");
show_progress(0.100000, 0);
# ---- radio update tasks ----
ui_print("Patching firmware images...");
package_extract_file("firmware-update/tz.mbn",
"/dev/block/platform/msm_sdcc.1/by-name/tz");
package_extract_file("firmware-update/sbl1.mbn",
"/dev/block/platform/msm_sdcc.1/by-name/sbl1");
package_extract_file("firmware-update/sdi.mbn",
"/dev/block/platform/msm_sdcc.1/by-name/dbi");
package_extract_file("firmware-update/rpm.mbn",
"/dev/block/platform/msm_sdcc.1/by-name/rpm");
package_extract_file("firmware-update/emmc_appsboot.mbn",
"/dev/block/platform/msm_sdcc.1/by-name/aboot");
package_extract_file("firmware-update/NON-HLOS.bin",
"/dev/block/platform/msm_sdcc.1/by-name/modem");
unmount("/qcom-firmware");
unmount("/system");
```

The system prints logs and sends commands back to the recovery binary via a pipe stream. Logs are printed out to the same recovery log and a failure return value is generated if any of the steps fail.

Most steps should be recoverable if a power failure occurs, except for some very short intervals.

5.9 Post-update

Recovery saves the update result in the log folder. The qrlUpdate upstart job checks for this file and runs the post-update procedure if this was the first boot after the successful update.

The script mounts the system image and installs any present deb file. It overwrites files if the deb was already installed before. The Wi-Fi MAC configuration script runs again and copies the MAC from the persist partition, ensuring the MAC is not modified.

The system reboots another time after qrlUpdate finishes and the system runs the new version afterward. The version update can be checked by reading /etc/qrl-version.

5.10 Update hooks in bitbake

Developers should use pkg_preinst and pkg_postinst shell tasks in their bitbake recipes for handling the update of their packages. Pkg_preinst can be used to back up userdata from configuration files and pkg_postinst can be used to restore it. For example, pkg_preinst can move or delete files before the actual package installation happens.

Responsibility for managing each individual package update requirement is left to the developer in the bitbake recipe. The recovery update process will not handle this.

Examples of hooks are shown at http://www.yoctoproject.org/docs/1.8/dev-manual/dev-manual.html#new-recipe-post-installation-scripts.

5.11 Factory reset

Factory reset is the act of restoring the board state to its original state when it was first flashed in the factory or by the developer. The userdata partition is erased during the process and all the contents from the factory package are extracted to that partition. The effect is that the eMMC has the same contents after the factory reset as when it was first flashed via USB.

The contents from the userdata partition are copied over to the recipe work directory during the userdata generation process. They are then used by the factory-reset task to populate the target files for factory reset and are added to the factory reset package. The factory reset package is packaged into the factory image. The factory reset procedure then mounts the factory partition and installs the package from there. The rest of the procedure is the same as the software update.

The main recipe files for a factory reset are as follows:

- oe-core\meta-qti\scripts\mkuserimage.sh – Script to generate userdata image; also populates the sysroot with the contents of the image.
- oe-core\meta-qr-linux\recipes-core\images\qrl-binaries.bb – do_factory_image – Runs the OTA script to generate the factory package and then runs make_ext4fs to create the factory image.

6 Additional Information

6.1 Snapdragon Flight documentation on GitHub

See https://github.com/ATLFlight/ATLFlightDocs/ for additional information.

6.2 Snapdragon Flight information on QDN

See https://developer.qualcomm.com/hardware/snapdragon-flight for additional information