Qualcomm Technologies International, Ltd.

ADK 4.3 Audio Sink Application Software Upgrade

User Guide

80-CF420-1 Rev. AA

November 6, 2017

Qualcomm Technologies International, Ltd. (formerly known as Cambridge Silicon Radio Limited) is a company registered in England and
Wales with a registered office at: Churchill House, Cambridge Business Park, Cowley Road, Cambridge, CB4 0WZ, United Kingdom.
Registered Number: 3665875 | VAT number: GB787433096

Revision history

Revision	Date	Description
AA	November 2017	Original publication of this document. Alternative document number CS-00406555-UG. Branched from CS-00328886-UG for ADK 4.3.

Contents

Tables

Figures

1 Upgrade overview

The upgrade feature gives customers the ability to download and update the software on CSR devices. It enables upgrades of components both in external serial flash and internal flash appropriate for the QTIL device. Examples of upgrades include:

- New audio prompt data files in external serial flash
- Updated VM application and/or firmware in internal flash
- Modification of PS Keys

The upgrade feature permits these elements to be upgraded in a single process by downloading a single file, containing different upgrade components into external serial flash, and then performing the upgrade operation.

Upgrade of components in external serial flash partitions is managed by the upgrade library directly, upgrade of components in internal flash are managed using the standard QTIL DFU mechanism, under the control of the firmware loader.

The upgrade feature has been designed to work with audio streaming using the Dual Mode Topology (DMT) features available in Bluetooth v 4.1. A2DP audio can be streamed from a handset to a device over a BR/EDR connection, in conjunction with an upgrade over Bluetooth low energy technology. This is known as trickle-mode, and can be used when the bandwidth available for the upgrade is limited. The upgrade process is robust to interruptions during the download, and can resume from link-loss of the device. When the upgrade file is downloaded to the device, it can be used to upgrade the device.

The upgrade application is provided as a VM library, which can be integrated with existing VM applications. The Sink application provided with ADKs has already been integrated as an example.

The upgrade VM library implements a new upgrade protocol, to perform the download and control the execution of an upgrade. The library interacts with Qualcomm® BlueCore™ firmware to manage, write and erase external serial flash partitions, and to perform DFU operations to update internal flash where required.

The Upgrade application is designed to use:

- Bluetooth Classic (BR/EDR)
- Bluetooth low energy technology

NOTE In future, it is planned to add more transports.

Bluetooth connectivity is provided to the upgrade library by GAIA, using new GAIA commands, which carry the upgrade protocol. This beta version of the upgrade application supports upgrade over Bluetooth low energy technology. Developers can use their own transports instead of GAIA if necessary, interfacing to the standard transport API of the upgrade library.

The upgrade library presents a simple interface to VM applications, after initialization the library largely runs autonomously. Future extensions before final release will provide more controls to the VM application, enabling more involvement in an upgrade process if required.

QTIL has implemented the upgrade protocol in an iOS application **GaiaControl**, which enables upgrade of QTIL devices over a Bluetooth low energy technology transport, using GAIA.

This document provides information required by developers to use the upgrade feature. Details are provided for configuring external serial flash to be compatible with upgrades, and information on creating upgrade files that can be used with this new feature. It also provides a step-by-step guide to using the iOS **GaiaControl** application to perform an upgrade.

2 Setup a development board for upgrade

2.1 Setup for upgrade - getting started

This section describes how to setup a development board for an upgrade and how to create an upgrade file that is ready to be sent to a device. Initial setup is covered in the *ADK 4.0 Sink Application Starter Guide*.

Requirements

- Development boards CSR8670, CSR8675, or final hardware with external serial flash memory installed.
- ADK 4.0 or later installation.
- Input files to upgrade.
- ADK tools in the **<ADK>\tools\bin** folder that are accessible from the command line.

2.2 Flash a new board for upgrade

The DFU upgrade validates the contents of the DFU file against an SHA public key that has been inserted into the loader partition. The loader and stack supplied with an ADK are unsigned therefore it is necessary to generate a SHA key pair and sign the firmware stack with the private key and insert the public key into the loader.

Prepare loader and stack for upgrade

The unsigned loader and stack are included in the ADK installation in **<ADK>\firmware\assisted \unified\<platform>**. For CSR8670 is **gordon** and for CSR8675 it is **rick**.

Run the following steps from the same location as `loader_unsigned.xdv` file. Alternatively copy <platform>`loader_unsigned.xdv/.xpv`< and `stack_unsigned.xdv/.xpv` to another folder and run the commands from there.

generates a private key file called `dfu.private.key` and a public key file called `dfu.public.key`.

1. Generate a key pair using the `dfukeygenerate` command that is called system key. The command:

   □
   ```
   $ dfukeygenerate -o dfu
   ```

2. Insert the public key into the loader using the `dfukeyinsert` command:

   □
   ```
   $ dfukeyinsert -v -o loader_signed -l loader_unsigned.xdv -ks
   dfu.public.key
   ```

3. Sign the content of the stack software area using the `dfusign` command.

   □ `$ dfusign -v -o stack_signed -s stack_unsigned.xpv -ks dfu.private.key`

4. Use **BlueFlash** to flash the signed loader and onto the board:

   a. Start **BlueFlash**

   b. Click **Stop Processor**

   c. Click **Flash Erase** and confirm that you want to erase the full chip

   d. Click **Choose File** and select `loader_signed.xpv`

   e. Press **Download**

   f. Click **Choose File** and select `stack_signed.xpv`

   g. Click **Download**

   h. Click **Start Processor**

   NOTE Using **BlueFlash** is the only way to get the loader partition onto a device.

Setup PS Keys

See the *ADK 4.0 Sink Application Starter Guide* for the basic setup of PS Keys.

Partition external serial flash memory

An upgrade uses the external serial flash memory to store the upgrade file contents until they can be committed to complete the upgrade. The steps required to partition the external serial flash memory so that it is ready for an upgrade are.

This operation completely erases the content of the external serial flash memory. The internal flash area containing firmware and application is unaffected.

1. Open the command line and execute:

   □ `nvscmd <transport> erase`

   Where `<transport>` can be:

   □ `-usb <port>`
   To indicate the USB-SPI port. Where:

   – `<port>` = the USB-SPI port. For example, `–usb 0` or `–usb nnnnnn` (where `nnnnnn` is the serial number of the USB-SPI converter).
   or

   □ `-lpt <port>`To indicate a LPT-SPI port. Where:

   – `<port>`= LPT-SPI multiport e.g. `–lpt 1` to select LPT1.

   – `-mul <port>` to indicate a SPI multiport.

   – `<port>` = LPT-SPI multiport (a number from 0 upwards; typically upto15). For example –`mul 2`.

2. Using a text editor, create the file `upgrade_partition_file.ptn` using the example code:

```
# Sample partition file for ADK
# Allows testing of all ADK forms of OTAU
0, 700K, RS, (erase)        # Logical 0: Partition for DFU
1, 32K, RO, (erase)        # Logical 1: Audio Prompt partition #1,1
2, 32K, RO, (erase)        # Logical 1: Audio Prompt partition #1,2
3, 16K, RO, (erase)        # Logical 2: Audio Prompt partition #2,1
4, 16K, RO, (erase)        # Logical 2: Audio Prompt partition #2,2
5, 64K, RS, (erase)        # Logical 3: Test partition #1
6, 64K, RS, (erase)        # Logical 3: Test partition #2
7, 8K, RS, (erase)        # Logical 4: Single-banked test partition
```

NOTE The extension of the file is important and must be `.ptn`.

3. Using the file created execute:

□ `nvscmd <transport> burn upgrade_partition_file.ptn all`

Setup xIDE

1. Open the default Sink application, that is `sink.xiw`.
2. In **Project Properties -> Build System -> Firmware Image** set the full path without the `.xpv`/`.xdv`/`.xuv` extension where the signed stack is located (created in Prepare loader and stack for upgrade).
   For example, if the stack file name is `stack_signed.xpv` and it is at **c:\work\firmware\signed**, the pathname entered should be:

```
c:\work\firmware\signed\stack_signed
```

3. Select **Rebuild** and **Run** from the menu bar.

2.3 Create an upgrade file

The command line tools necessary to create the upgrade file are located in the **<ADK install>\tools \bin** folder of the ADK installation.

2.3.1 Create an application upgrade DFU file

To create a valid upgrade file, use the same key pair created in Flash a new board for upgrade, or create a new key pair that will be called the application key.

NOTE If using the previous key pair change any reference to `app.public/private.key` to `dfu.public/private.key`.

Creating DFU file with VM application content

1. To create a new key pair, use the command:

```
$ dfukeygenerate -o app
```

This generates the `app.public.key` and `app.private.key`.

2. Create a file to identify the hardware to prevent the DFU upgrade from being applied to inappropriate hardware. Use Notepad or any text editor to create a file called `stack.psr` containing the hardware type key:
   - □ For CSR8670: `&F002 = 0000 0000 0000 1000`
   - □ For CSR8675: `&F002 = 0000 0000 0000 2000`
3. Insert the application public key into `stack.psr` using `dfukeyinsert`:

   ```
   $ dfukeyinsert -v -o sps -ps stack.psr -ka app.public.key
   ```

   This generates `sps.psr`.
4. For the DFU upgrade the file needs to be signed with the system private key created in the section Prepare loader and stack for upgrade. The following command line creates `sps.stack.psr:`

   ```
   $ dfusign -v -o sps -ps sps.psr -ks dfu.private.key
   ```

5. Sign the application with the application private key:

   ```
   $ dfusign -v -o image_signed -h image.fs -ka app.private.key
   ```

   NOTE `image.fs` is generated by **xIDE** when it builds a VM application. It is put into the same folder as the application project file, e.g. **<ADK>\apps\sink** for the sink application.
6. Use `dfubuild` to create a DFU upgrade file from the files generated above.

   ```
   dfubuild -v -pedantic -t 2 -f upgrade_file.dfu -uv 0xffff -up 0xffff -ui
   "DFU OTA upgrade" -p3 . sps.stack.psr . -h image_signed.fs
   ```

   Where:
   - □ `-v` := Run in verbose mode, displaying detailed information about the actions performed.
   - □ `-pedantic` = Fail if redundant files have been specified.
   - □ `-t 2` = Generate DFU file “2002” format
   - □ `-uv 0xffff -up 0xffff` = Instructs **dfubuild** to ignore USB parameters as this file will be sent over-the-air
   - □ `-ui "CSR OTA upgrade"` = The long information string to include in the DFU file
   - □ `-p3 . sps.stack.psr .` = PSR files listed as triples. Missing files are replaced by a period (.).
   - □ `-h image_signed.fs` = The application file system.

Contents of DFU File

Figure 2-1 shows the contents of the DFU file and how the public/private key pairs were used to sign it.

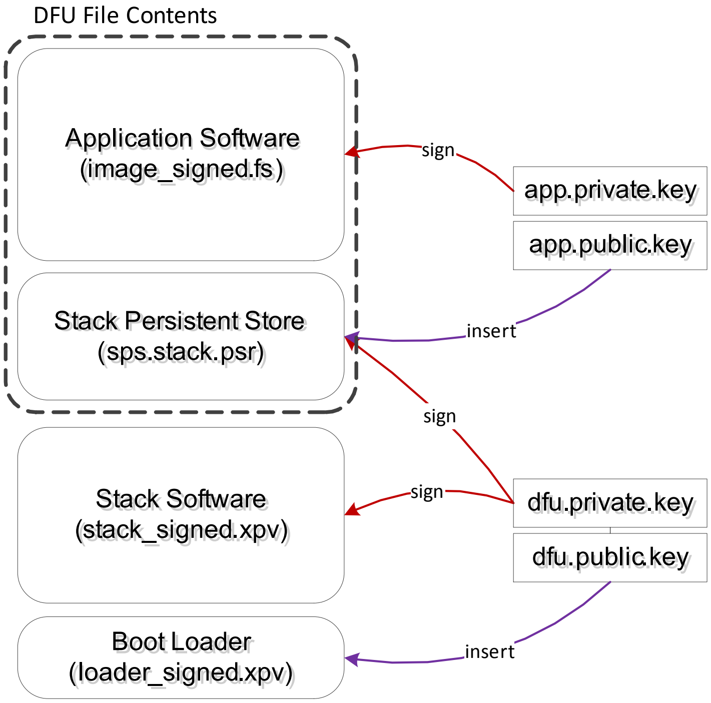

DFU File Contents
Application Software (image_signed.fs)
Stack Persistent Store (sps.stack.psr)
Stack Software (stack_signed.xpv)
Boot Loader (loader_signed.xpv)
sign
insert
sign
sign
insert
app.private.key
app.public.key
dfu.private.key
dfu.public.key

Figure 2-1 Keys Used to Sign DFU File

2.3.2 Create an upgrade file

An upgrade file can contain one or more partitions containing different data types, e.g. a DFU upgrade or voice prompt data. In this example, the upgrade file contains only the DFU file created in Create an application upgrade DFU file.

1. Using a text editor, create the file `upgrade_partition.upd`. This file describes the version of the software embedded into the upgrade file as well as the location and partition type to be upgraded. The content of this file should look like the following example:

```
# ADK upgrade requires an empty signature appended to the end of the file.
add_empty_signature
# Set the upgrade version and previous version(s)
# that are compatible to upgrade from. The minor
# version can be '*' to act as a wildcard.
upgrade_version 2.1
compatible_upgrade 0.*
compatible_upgrade 1.*
compatible_upgrade 2.*
# Set the ps config version and previous version(s)
# that are compatible to upgrade from
ps_config_version 2
ps_prev_config_version 0
ps_prev_config_version 1
# list all partition starting from index 0 including partition type
# <partition number> <partition type> <full path filename>
# DFU file with file system
0 1 upgrade_partition_dfu.xuv
```

Description of the file content:

- `add_empty_signature`: This is used to create a partitioned file with an empty signature. The ADK does not use a signature to validate the contents of the upgrade file but it is still required to add empty padding where the signature would be.
- `upgrade_version 2.1` : Defines the version of the software embedded in this upgrade file.
- `compatible_upgrade`: Defines what other versions are compatible with the content of this upgrade file.
- `ps_config_version`: Defines the current PS configuration version.
- `ps_prev_config_version`: Defines what other versions of PS configuration are compatible with the current PS configuration version.
- `0 1 upgrade_partition_dfu.xuv`: File that represents each partition for upgrade. In the example the first digit 0 is the partition number and the second digit 1 is the partition type (see Table 2-1), followed by the file name in XUV format. More partitions can be added as in the example:

  ```
  0 1 partition_file1.xuv
  1 3 partition_file2.xuv
  ```

Table 2-1 lists the partition types supported.

Table 2-1 Partition types

Partition Type	Value	Description
`UPGRADE_FW_IF_PARTITION_TYPE_EXE`	0	VM Executable Partition (not used in ADK)
`UPGRADE_FW_IF_PARTITION_TYPE_DFU`	1	DFU File Partition
`UPGRADE_FW_IF_PARTITION_TYPE_CONFIG`	2	PSFS Configuration Data (not used in ADK)
`UPGRADE_FW_IF_PARTITION_TYPE_DATA`	3	Standard Data on a read only (RO) partition
`UPGRADE_FW_IF_PARTITION_TYPE_DATA_RAW_SERIAL`	4	Standard Data on a raw serial (RS) partition

2. The DFU is in binary format and needs to be converted to XUV:

   ```
   XUV2BIN -e upgrade_file.dfu upgrade_partition_dfu.xuv
   ```

   Where:
   `upgrade_file.dfu` = The input DFU file in binary format
   `upgrade_partition_dfu.xuv` = DFU file converted to XUV format.

3. Create the upgrade file in XUV format using `UpgradeFileGen` using the created `upgrade_partition.upd` file:

   ```
   UpgradeFileGen upgrade_partition.upd upgrade_file.xuv
   ```

   Where:

`upgrade_partition.upd` = The Upgrade Partition Data file
`upgrade_file.xuv` = The output upgrade file XUV.

4. Convert the generated XUV upgrade file into binary before sending to the device:

   ```
   XUV2BIN -d upgrade_file.xuv upgrade_file.bin
   ```

   Where:
   `upgrade_file.xuv` = The input upgrade partitioned file in XUV format
   `upgrade_file.bin` = The final upgrade file in binary format to be sent to the device.

3 External serial flash partitioning

Upgrade requires an external serial flash partition layout which matches the logical partition mapping table passed to `UpgradeInit()` in the VM application.

From a partition layout point of view there are two types of partitions: DFU and data.

- In the case of DFU only one partition in external serial flash is needed.
- Data partitions must be banked (there must be two physical partitions).

This is because DFU is downloaded to external serial flash then copied to the internal flash, whereas for data, one partition in each bank is currently in use by the VM application and another one is used for storing a new version of data.

Partition management is handled by the upgrade library. However there are two assumptions made:

1. Partitions to be used for downloading upgrade data are erased in the factory.
2. Factory defaults for partitions used by the VM application should be that the first partition in a bank is used and second one is left empty.

Example partition table:

```
# Sample partition file for ADK
# Allows testing of all ADK forms of OTAU
0, 700K, RS, (erase)        # Logical 0: Partition for DFU
1, 32K, RO, (erase)        # Logical 1: Audio Prompt partition #1,1
2, 32K, RO, (erase)        # Logical 1: Audio Prompt partition #1,2
3, 16K, RO, (erase)        # Logical 2: Audio Prompt partition #2,1
4, 16K, RO, (erase)        # Logical 2: Audio Prompt partition #2,2
5, 64K, RS, (erase)        # Logical 3: Test partition #1
6, 64K, RS, (erase)        # Logical 3: Test partition #2
7, 8K, RS, (erase)        # Logical 4: Single-banked test partition
```

The partition table file is used by **nvscmd** tool to set the partition structure in a flash chip.

The format of a partition table definition is: physical partition number, size in 16-bit words (K=1024), type of a partition; file to be downloaded to a partition.

Example of matching logical partition mapping table:

```
static const UPGRADE_UPGRADABLE_PARTITION_T logicalPartitions[]
             = {UPGRADE_PARTITION_SINGLE(0x1000,DFU),
                UPGRADE_PARTITION_DOUBLE(0x1001,0x1002,MOUNTED),
                UPGRADE_PARTITION_DOUBLE(0x1003,0x1004,MOUNTED),
                UPGRADE_PARTITION_DOUBLE(0x1005,0x1006,UNMOUNTED),
                UPGRADE_PARTITION_SINGLE(0x1007,KEEP_ERASED)};
```

Each row in the table defines a logical bank of two physical partitions. The fields are defined as:

- Least significant four bits is a physical partition number of first partition in logical bank.
- Reserved.
- Least significant four bits is a physical partition number of second partition in logical bank.
- Partition arrangement:
  - □ `UPGRADE_LOGICAL_BANKING_SINGLE_KEEP_ERASED`: Indicates a partition, that is not part of a pair, and that is always erased after an upgrade completes.
  - □ `UPGRADE_LOGICAL_BANKING_DFU`: Partition used for DFU (synonym for `UPGRADE_LOGICAL_BANKING_SINGLE_KEEP_ERASED`).
  - □ `UPGRADE_LOGICAL_BANKING_DOUBLE_BANK_UNMOUNTED`: A partition pair that is not used for a file system. These partitions are of type raw serial.
  - □ `UPGRADE_LOGICAL_BANKING_DOUBLE_BANK_MOUNTED`: A partition pair that is used in a file system. These partitions are read only (although they can be upgraded). Following an upgrade of these partitions the file system table is updated so that the newly updated partition is active.
  - □ `UPGRADE_LOGICAL_BANKING_SINGLE_ERASE_TO_UPDATE`: A partition, that is not part of a pair, and is not kept erased by the upgrade library. This type of partition is not supported at present.

4 VM application upgrade library integration

The upgrade library fully manages the upgrade process but it requires integration with the VM application to communicate with an upgrade host and to synchronize at certain points in the process that will potentially disrupt other application functions. For example, before an upgrade is committed the device must be rebooted and this stops any audio that is currently playing.

This section details how a VM application uses the upgrade library. The interface between the application and the upgrade library is defined in `upgrade.h`.

4.1 Initialization

The upgrade library is initialized by calling `UpgradeInit()` with the parameters described in Table 4-1. It must be initialized before any other service to allow it to change the PS store and erase one or more partitions when an upgrade is committed.

If an upgrade is committed after other services are started, it may lead to bad configuration of the PS store or interruption of those services due to accessing the external flash device.

4.1.1 Parameters

Table 4-1 Upgrade library initialization parameters

Parameter	Description
`appTask`	The main VM application Task.
`dataPskey, dataPskeyStart`	Upgrade needs to store some persistent state in the PS store but the PS Key to use is set by the VM application. The data can also be offset from the start of the key, to allow for other VM application-specific data to share the same PS Key. `UpgradeInit()` checks that there is enough space in the PS Key given the offset and size of the data the upgrade library needs to store.

`logicalPartitions,`
`numPartitions`
	The VM application must tell the upgrade library what the layout and organization of the partitions on the external flash are and pass them in these parameters. The partition table format is detailed in External serial flash partitioning.

`power_mode`	The initial power mode of the system, it should be set to `UPGRADE_INIT_POWER_MANAGEMENT`. See for more detail of this parameter.
`dev_variant`	An ASCII string that defines the current VM application variant. This string is checked against the variant set in the header of an upgrade file and if they do not match the upgrade is rejected. The variant check is disabled if `dev_variant` is `NULL`.

Table 4-1 Upgrade library initialization parameters (cont.)

Parameter	Description
`init_perm`	Sets the initial upgrade permission state. See Upgrade permission mechanism for more detail.

`init_version,`
`init_config_version`
	Set the initial upgrade version and configuration version on a factory-flashed device. Any upgrade afterwards sets the upgrade version and configuration version with the values from the upgrade file header.

4.1.2 Upgrade messages

After `UpgradeInit()` is called the upgrade library sends two messages to the VM application. Table 4-2 explains these messages.

Table 4-2 Upgrade library initialization messages

Message	Description	Notes
`UPGRADE_INIT_CFM`	Sent by the upgrade library to signal whether `UpgradeInit()` was successful.	Returns either success or failure.
`UPGRADE_RESTARTED_IND`	Sent during upgrade initialization to let the VM application know if a restart occurred due to an upgrade and if reconnection to a host is required.	If a restart is requested by the upgrade library, then it is the responsibility of the VM application to configure itself so that an upgrade host can reconnect and complete the upgrade. For the best user experience, it is assumed that the host should be able to reconnect without any extra user input.

4.2 Upgrade transport integration

The upgrade library has a generalized API to allow any transport to connect to it and pass data to and from an upgrade client. The transport is responsible for connecting to the upgrade client, for example pairing or setting up a secure connection.

When an upgrade client connects to a transport the transport must connect to the upgrade library before sending any data packets. This is required so that the upgrade library knows which transport to send outgoing data to.

When a client disconnects from a transport the transport must also disconnect from the upgrade library. This is true in both normal and link-loss scenarios.

Upgrade protocol messages are contained within the data packets sent over a transport. The transport only needs to forward the packets to and from the upgrade client. It does not need to know the contents of the data packets.

The upgrade library can handle messages split up over multiple data packets. The maximum size of a message is controlled by the upgrade library so it should never run out of memory when reconstructing a message from multiple data packets.

4.2.1 Upgrade client restrictions

Only one upgrade client is supported at a time. A second upgrade client trying to connect receives an error message and is not connected.

4.2.2 Upgrade transport API

Table 4-3 lists the upgrade functions that must be used by a transport to communicate with the upgrade library.

Table 4-3 Upgrade library transport functions

Function	Description
`UpgradeTransportConnectRequest()`	Request to connect a transport to the upgrade library. The result is returned in `UPGRADE_TRANSPORT_CONNECT_CFM`.
`UpgradeTransportDisconnectRequest()`	Request to disconnect a transport from the upgrade library. The result is returned in `UPGRADE_TRANSPORT_DISCONNECT_CFM`.
`UpgradeProcessDataRequest()`	Pass a data packet from a transport to the upgrade library. If the transport was not registered via `UpgradeTransportConnectRequest()` the data will be ignored. The result of the processing of the data packets is returned in `UPGRADE_TRANSPORT_DATA_CFM_T`.

Table 4-4 lists the messages sent from the upgrade library to a transport.

Table 4-4 Upgrade library transport messages

Message	Description
`UPGRADE_TRANSPORT_CONNECT_CFM_T`	Response to `UpgradeTransportConnectRequest()`. Status contains: ■ `upgrade_status_success` if transport connected ok. ■ `upgrade_status_already_connected_warning` if another transport is connected. ■ `upgrade_status_unexpected_error` if the upgrade library is not initialized.
`UPGRADE_TRANSPORT_DISCONNECT_CFM_T`	Response to `UpgradeTransportDisconnectRequest()`. Always returns `upgrade_status_success`.

Table 4-4 Upgrade library transport messages (cont.)

Message	Description
`UPGRADE_TRANSPORT_DATA_CFM_T`	Response to `UpgradeProcessDataRequest()`. Contains how much of the data packet was processed by the upgrade library.
`UPGRADE_TRANSPORT_DATA_IND_T`	Sent from the upgrade library to send a data packet to the upgrade client. Contains a pointer to the data and the size. The data packet is assumed to always be sent to the client so no reply needs to be sent to the upgrade library.

The sequence diagram below shows the relationship between an upgrade host, the VM application, a transport and the upgrade library.

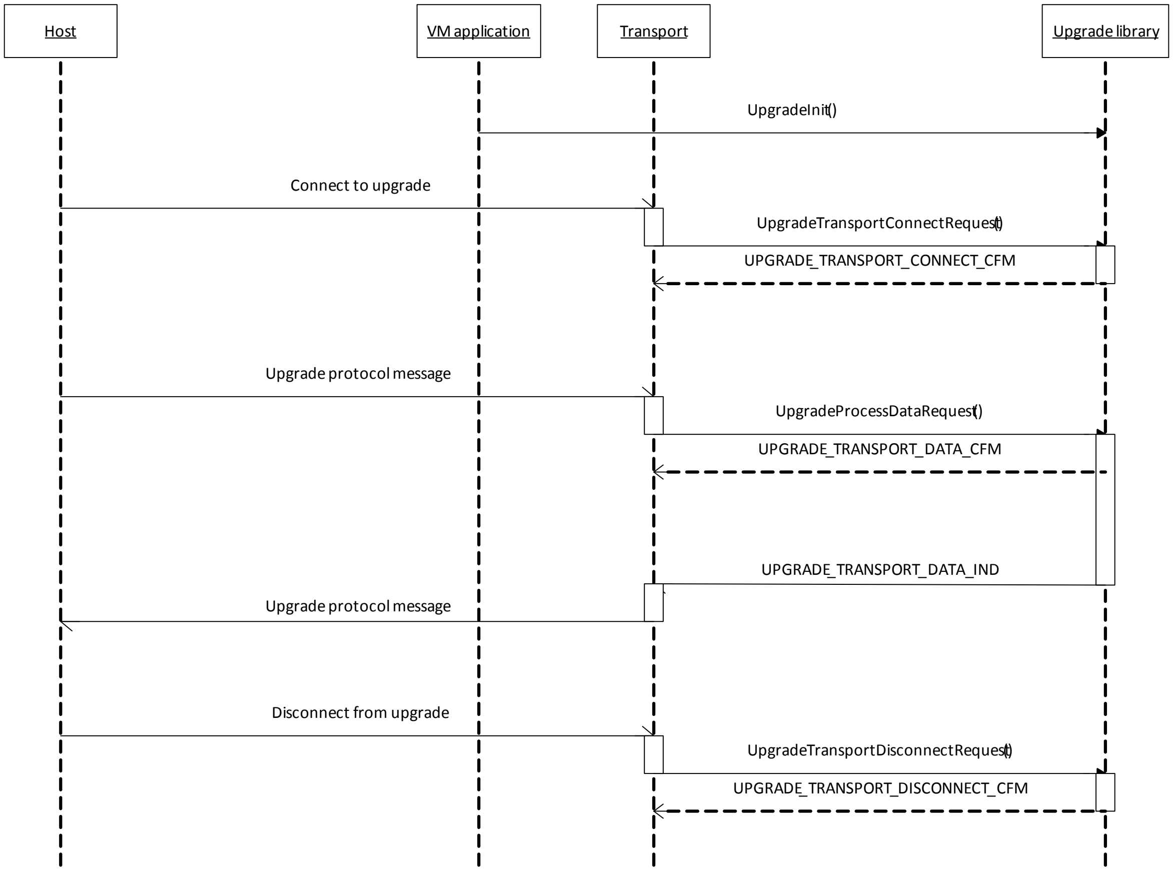

Host
VM application
Transport
Upgrade library
UpgradeInit()
Connect to upgrade
UpgradeTransportConnectRequest()
UPGRADE_TRANSPORT_CONNECT_CFM
Upgrade protocol message
UpgradeProcessDataRequest()
UPGRADE_TRANSPORT_DATA_CFM
UPGRADE_TRANSPORT_DATA_IND
Upgrade protocol message
Disconnect from upgrade
UpgradeTransportDisconnectRequest()
UPGRADE_TRANSPORT_DISCONNECT_CFM

Figure 4-1 Upgrade transport sequence diagram

4.3 Upgrade permission mechanism

During an upgrade, the upgrade library may at certain stages want to perform an action that either disrupts a service (for example interrupt audio playback) or reboot the device.

The permission mechanism allows the VM application control over when these actions occur. The initial permission state is set in `UpgradeInit()` and it can be changed at any time by calling `UpgradePermit()` with the new state. However, if an upgrade is currently in progress, `UpgradePermit()` returns an error.

4.3.1 Upgrade states

Table 4-5 Upgrade library permission types

State	Description
`upgrade_perm_no`	Upgrades are not permitted. Any host request to upgrade is rejected.
`upgrade_perm_assume_yes`	Upgrades are permitted. The upgrade library always assumes it is allowed to perform the actions it needs. The VM application is never notified.
`upgrade_perm_always_ask`	Upgrades are permitted. The upgrade library always asks the VM application for permission before performing an action that may affect the rest of the system.

4.3.2 Upgrade "Always Ask" implementation

If the permission state is set to `upgrade_perm_always_ask`, the VM application is sent permission messages from the upgrade library. The VM application must reply to these messages by calling functions defined in `upgrade.h`. Table 4-6 lists the permission messages and expected responses.

Table 4-6 Upgrade library permission messages

Message	Description	Response
`UPGRADE_START_IND`	Sent to the VM app when a host sends an upgrade request to the device.	`UpgradeStartResponse(TRUE)` to allow the upgrade to proceed. `UpgradeStartResponse(FALSE)` to deny the upgrade request.
`UPGRADE_APPLY_IND`	Sent to the VM app when upgrade lib wants to apply an upgrade. This requires a reboot of the device.	`UpgradeApplyResponse(0)`to apply the upgrade immediately. `UpgradeApplyResponse(time)` to postpone applying the upgrade for `time` ms. After this period the upgrade library sends another `UPGRADE_APPLY_IND`.
`UPGRADE_BLOCKING_IND`	Sent to the VM app when upgrade lib wants to block the system for an extended time to erase external flash partitions.	`UpgradeBlockingResponse(0)` to block the system immediately. `UpgradeBlockingResponse(time)` to postpone the blocking for `time` ms. After this period the upgrade library sends another `UPGRADE_BLOCKING_IND`.

4.4 Upgrade DFU loader message

An upgrade file may contain a DFU file and if this is the case, the DFU file is applied by the firmware.

The success or failure to apply the DFU file is sent to the VM application from the firmware in a `MESSAGE_DFU_SQIF_STATUS` message. The VM application must pass this message on to the upgrade library using the `UpgradeDfuStatus()` function so that it knows if the upgrade was successful or not. Failure to do this means that the upgrade may not complete successfully.

4.5 Power management during upgrade

If the VM application is running on a battery powered device, it can tell the upgrade library when the battery level is too low to permit an upgrade to happen using the `UpgradePowerManagementSetState()` function.

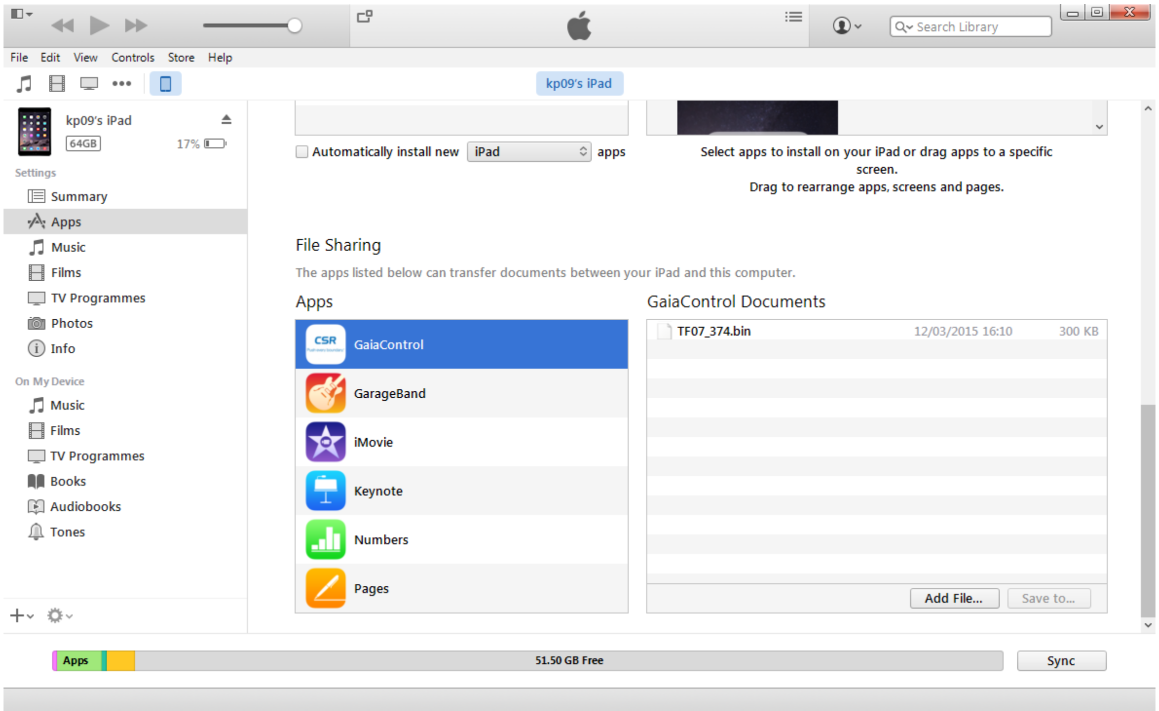
If the battery level is too low `UpgradePowerManagementSetState()` should be called with `upgrade_battery_low` and then:

- Any upgrade currently in progress is paused and a `low battery` error sent to the upgrade host.
- Any new upgrade request is rejected with a `low battery` error.
- If a charger is attached or the battery level is otherwise ok, then calling `UpgradePowerManagementSetState()` with `upgrade_battery_ok` or `upgrade_charger_connected` permits upgrades to start or resume again.

5 Upgrade with iOS GaiaControl app

With the **GaiaControl** app installed on an iOS device:

1. Add an upgrade file to the device using iTunes:



□ Automatically install new iPad apps

Select apps to install on your iPad or drag apps to a specific screen.
Drag to rearrange apps, screens and pages.

File Sharing

The apps listed below can transfer documents between your iPad and this computer.

Apps

GaiaControl Documents

TF07_374.bin	12/03/2015 16:10	300 KB

Figure 5-1 Adding upgrade files in iTunes

2. Tap to pen the **GaiaControl** app:

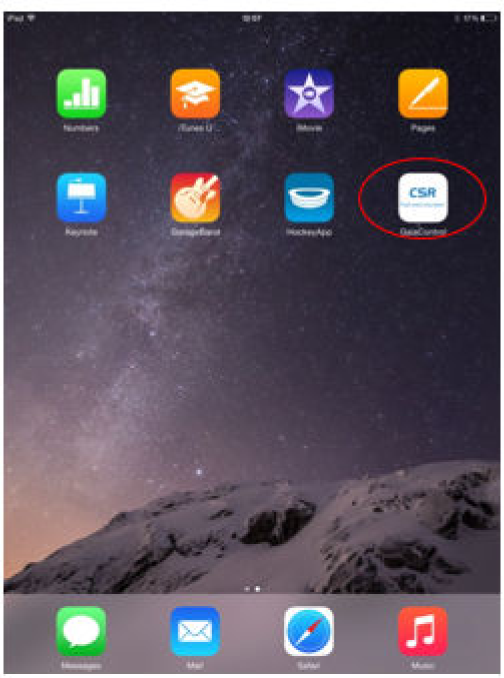
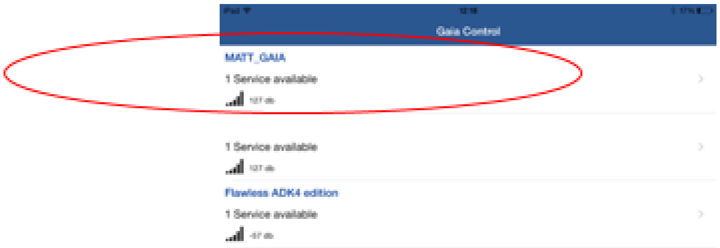


Figure 5-2 GaiaControl Open Application

3. **GaiaControl** scans for available Bluetooth low energy technology devices. Select a device to be upgraded:



Gaia Control

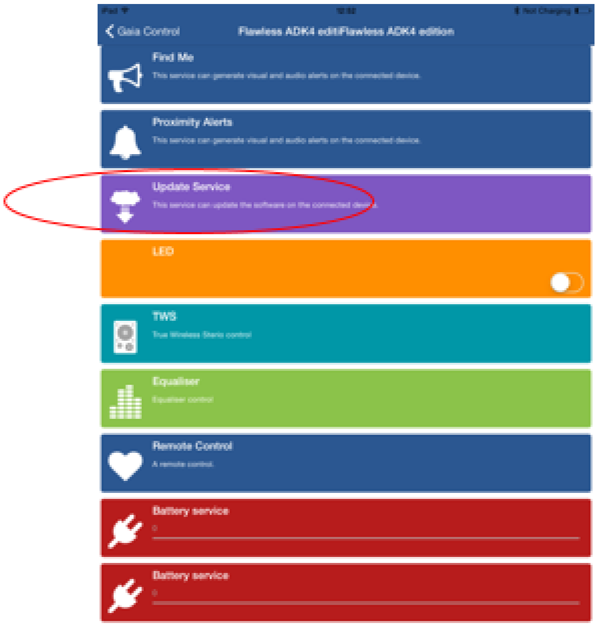
MATT_GAIA
1 Service available
127 db

1 Service available
127 db

Flawless ADK4 edition
1 Service available
-57 db

Figure 5-3 GaiaControl select device

4. The device advertises which Bluetooth low energy technology services it supports. Select **Update Service**:



Find Me
This service can generate visual and audio alerts on the connected device.

Proximity Alerts
This service can generate visual and audio alerts on the connected device.

Update Service
This service can update the software on the connected device.

LED

TWS
True Wireless Stereo control

Equaliser
Equaliser control

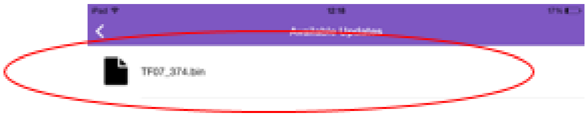
Remote Control
A remote control.

Battery service
0

Battery service
0

Figure 5-4 GaiaControl select service

5. The one or more upgrade files that were uploaded from iTunes are listed. Select the upgrade files required:

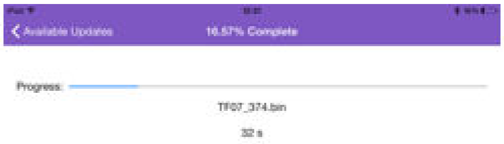


TF07_374.bin

Figure 5-5 GaiaControl select upgrade file or files

NOTE Unlike in the android app, the iOS app dynamically checks if RWCP is enabled and supported through GAIA commands. If it is enabled the upgrade process proceeds using RWCP.

6. The upgrade starts automatically and a progress bar page is displayed:



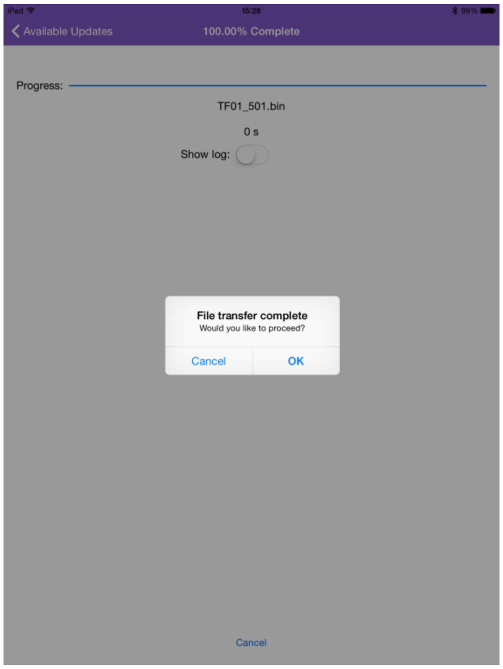
Progress:

TF07_374.bin

32 s

Figure 5-6 GaiaControl download progress bar

7. When the upgrade file has been transferred the user can tap **OK** to proceed or **Cancel** to quit the upgrade.



100.00% Complete

Progress:

TF01_501.bin

0 s

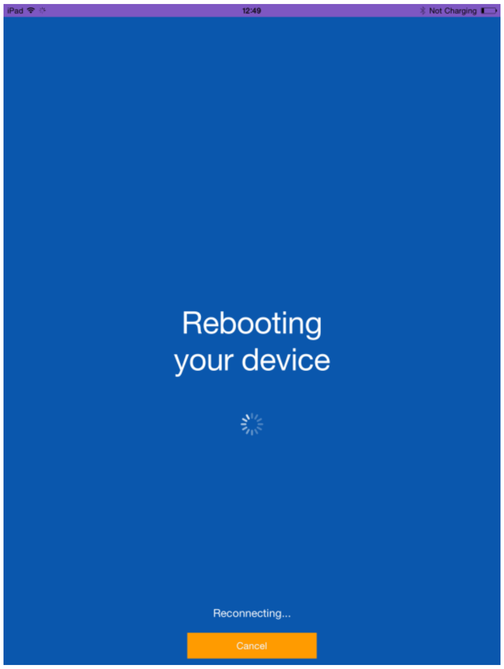
Show log:

File transfer complete

Would you like to proceed?

Figure 5-7   GaiaControl File Transfer Complete

8. If **OK** is selected the device reboots. This may take a few seconds, **GaiaControl** then automatically reconnects to complete the upgrade.

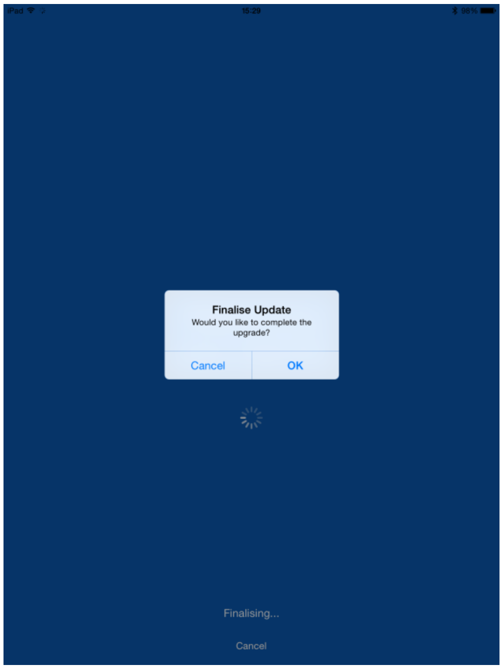


Rebooting
your device

Reconnecting...

Figure 5-8   GaiaControl rebooting

9. If the upgrade file does not contain a DFU partition the application asks if the upgrade should be applied:

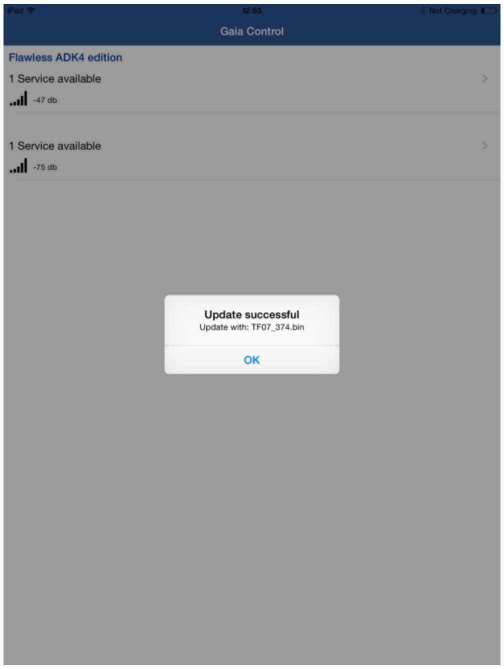


Finalise Update

Would you like to complete the upgrade?

Figure 5-9 GaiaControl confirmation

10. When the upgrade is committed **GaiaControl** confirms that it was successful:



Update successful

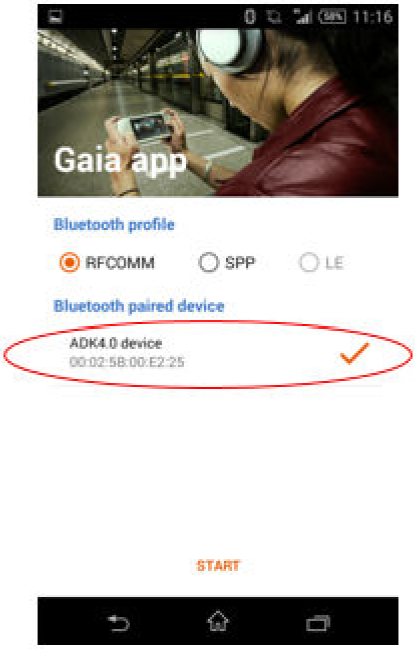
Update with: TF07_374.bin

Figure 5-10 GaiaControl upgrade complete

6 Upgrade with Android Gaia app

With the **Gaia app** installed on an android device copy an upgrade file **to VMUPGRADE** folder on the device's internal memory. To use **Gaia app** devices must be paired**. VMUPGRADE** should be created in the root of the android device's Internal Storage file system.

1. Select a device to be upgraded:



Bluetooth profile

● RFCOMM ○ SPP ○ LE

Bluetooth paired device

ADK4.0 device
00:02:5B:00:E2:25 ✓

Figure 6-1 Gaia select device

2. Select **Update** service:

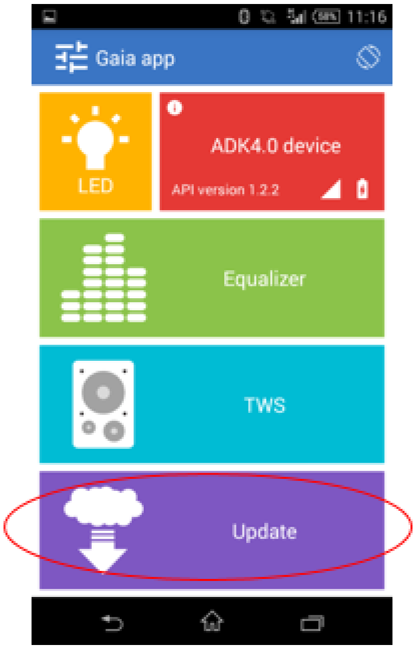


Figure 6-2 Gaia Select Service

3. Select an upgrade file and tap **UPDATE** button:

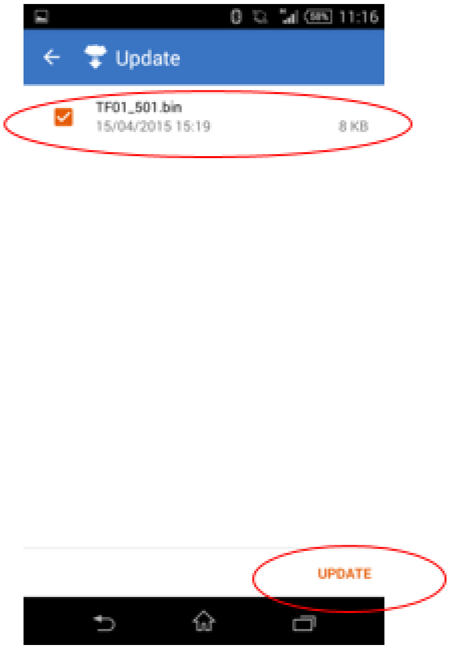
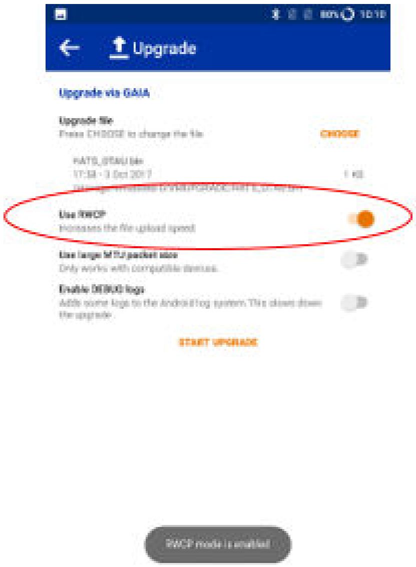


Figure 6-3 Gaia select upgrade file

4. Before starting the download, RWCP, a protocol used to speed up the download, can either be enabled or disabled, see Figure 6-4



Upgrade

Upgrade via GAIA

Upgrade file
Press CHOOSE to change the file — CHOOSE

HATS_OTAU.bin
17:58 - 3 Oct 2017 — 1 KB
[illegible]

Use RWCP
Increases the file upload speed

Use large MTU packet size
Only works with compatible devices.

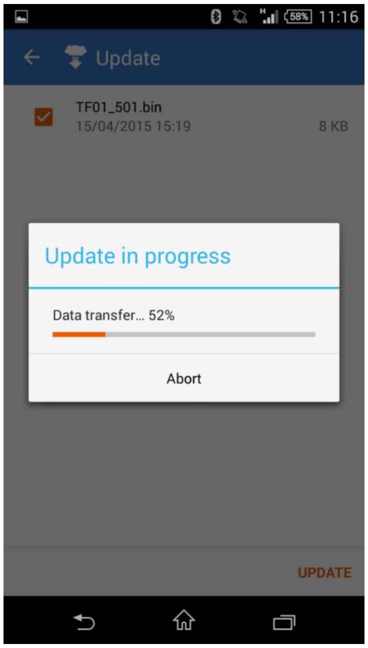
Enable DEBUG logs
Adds some logs to the Android log system. This slows down the upgrade.

START UPGRADE

RWCP mode is enabled

Figure 6-4 Gaia enable RWCP

5. Download of the upgrade file is in progress:



TF01_501.bin
15/04/2015 15:19 — 8 KB

Update in progress

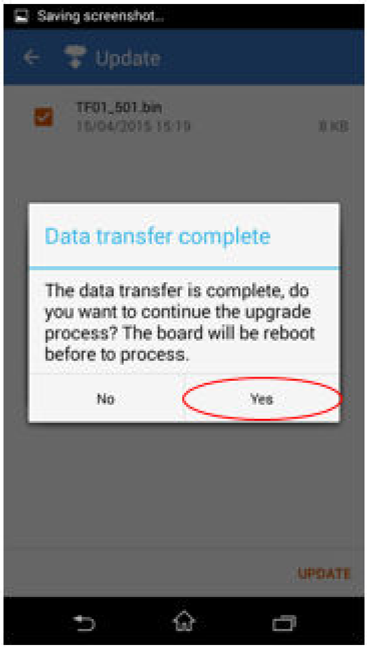
Data transfer… 52%

Abort

UPDATE

Figure 6-5 Gaia upgrade progress bar

6. When download is complete select **Yes**:

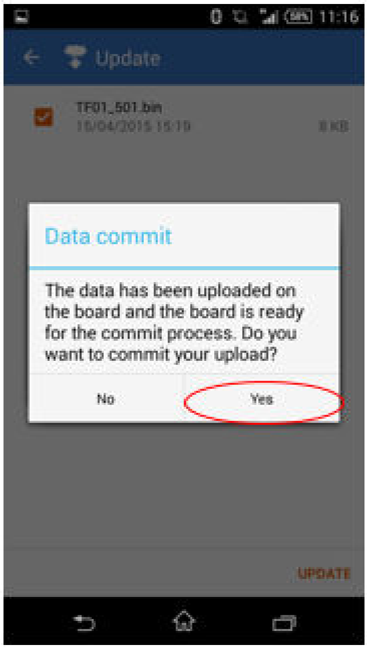


Data transfer complete

The data transfer is complete, do you want to continue the upgrade process? The board will be reboot before to process.

Figure 6-6 Gaia data transfer complete

7. When upgrade is ready to commit select **Yes**:

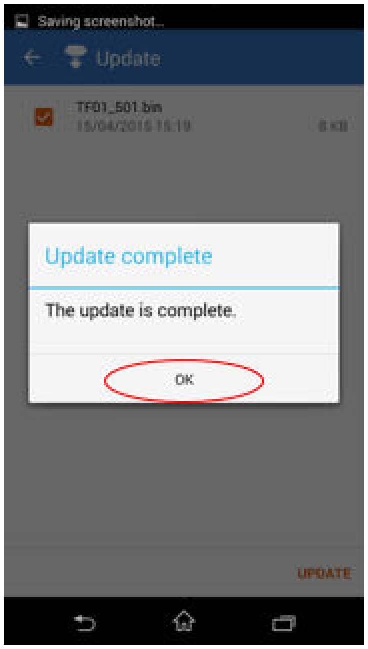


Data commit

The data has been uploaded on the board and the board is ready for the commit process. Do you want to commit your upload?

Figure 6-7 Gaia upgrade commit

8. Upgrade is completed:



☑ TF01_501.bin
15/04/2015 15:19 8 KB

Update complete

The update is complete.

Figure 6-8 Gaia upgrade complete

9. Tap **OK**.

Document references

Document	Reference
ADK 4.0 Sink Application Starter Guide	80-CE757-1/CS-00328898-UG

Terms and definitions

Term	Definition
A2DP	Advanced Audio Distribution Profile
ADK	Audio Development Kit
ANCS	Apple Notification Centre Service
BlueCore	Group term for the range of QTIL Bluetooth wireless technology chips
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
BR/EDR	Basic Rate/Enhanced Data Rate
DFU	Device Firmware Upgrade
DMT	Dual Mode Topology
GAIA	Generic Application Interface Architecture
GATT	Generic Attribute Profile
HID	Human Interface Device Profile
OTA	Over The Air
PS	Permanent Store
QTIL	Qualcomm Technologies International, Ltd.
RWCP	Reliable Write Command Protocol
SQIF	Serial Quad I/O Flash, a nonvolatile memory technology
VM	Virtual Machine
XUV	An ASCII hex representation of a binary file