

CC3200 SimpleLink™ Wi-Fi® and Internet-of-Things Solution, a single-chip wireless MCU

Over-The-Air (OTA) Update

TABLE OF CONTENTS

1	PURPOSE AND SCOPE.....	3
2	REFERENCES.....	4
3	OVERVIEW OF OTA ENABLERS	6
3.1	COMPONENTS	6
3.2	SUMMING IT ALL	12
4	SAMPLE APPLICATION (PACKAGED WITH CC3200 SDK).....	21
4.1	SOURCE FILES BRIEFLY EXPLAINED	21
4.2	CREATING DROPBOX API APPLICATION	23
4.3	CONFIGURING THE APPLICATION FOR NEW DROPBOX ACCOUNT.....	24
4.4	USAGE	24
5	ADDING OTA FEATURE TO EXISTING MCU APPLICATION – REFERENCE.....	29
5.1	CREATING DROPBOX API APPLICATION	29
5.2	GETTING CCS SETTINGS IN ORDER	29
5.3	ADDING / LINKING OTA COMPONENTS WITH GET_WEATHER APPLICATION	31
5.4	CONFIGURING THE DEFAULT SFLASH CONTENTS.....	34
5.5	UPLOADING THE NEW IMAGE TO CLOUD.....	37
6	MOVING TO OTHER FILE HOSTING SERVICES	37
6.1	PORTING OTA LIBRARY TO OTHER SERVERS	37
7	LIMITATIONS/KNOWN ISSUES	46
8	APPENDIX	47
8.1	DETAILED REPRESENTATION OF THE OTA FLOW	47
8.2	GENERIC GUIDELINES FOR ENABLING OTA ON CC3100/CC3200	51

1 Purpose and Scope

This application note provides detailed information on adding “Over the Air (OTA) updates” capability for user applications to be hosted in the MCU sub-system of CC3200 device.

Basic premise of the following reference is to help MCU application developers integrate the reference OTA library with his application. Integrating the OTA service would enable in-system upgrades of the MCU application, other vendor files and CC3200 Firmware releases made available by Texas Instruments.

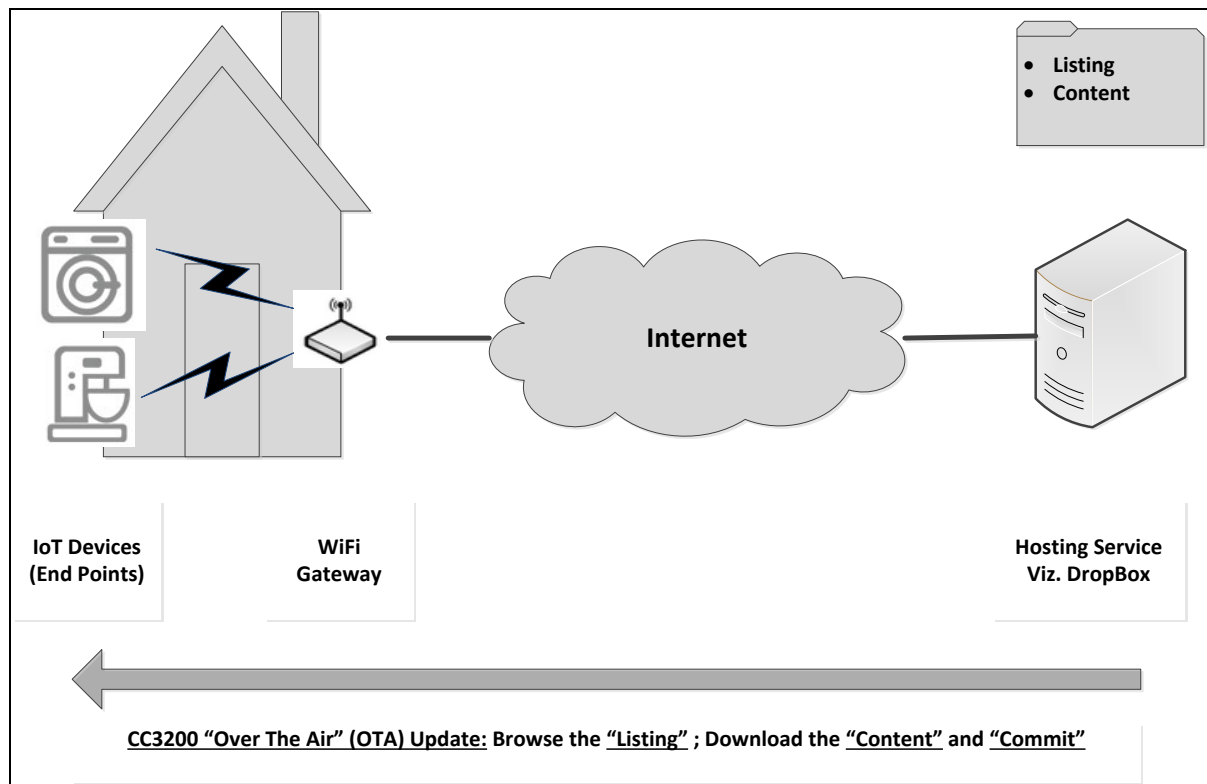


Figure 1 : CC3200 OTA: Basic Flow

These upgrades happen on the wireless network, and the library itself is oriented to scan the vendor’s element listing on *DropBox* [A file hosting provider – more details in subsequent sections], download / commit these elements to the CC3200 Serial FLASH.

How to use this Document:

This Application Note consists of five sections

- **Overview:** Developer can refer to this section to get a high level familiarity with OTA enablers included in CC3200 version 1.1.

- **Introduction to “ota_update” example:** This section would introduce the reference example included in the SDK package. This should help developers experience a few key capabilities like “rollback” of Application Image
- **Adding OTA Service to an Existing Application:** Developers can refer to this section for a step by step procedure integrating OTA services with their existing application
- **Working with other File Hosting Services:** As mentioned earlier the OTA solution is tuned to operate with DropBox service. This section attempts to provide specific pointers to the MCU application developers to modify OTA Library for inter-working with other hosting services. A case study to support Exosite cloud service is also provided.
- **Detailed Flow / Transactions:** Detailed ladder diagrams have been provided in an Appendix for users who wish to understand the transactions better

So if you are looking to get started immediately to link OTA and FLC libraries with your existing application, then jump to straightaway to **Section-5**.

For a better understanding of the OTA enablers and some key concepts viz. Commit/Rollback visit **Sections – 3**.

Section-4 would come handy understand the example provided in SDK. This would be nice place to experiment with features like application rollback

Section-6 would be useful reference, if you decided to use a different cloud service provider

2 References

Term	Comments
SDK	Refers to CC3200 Software Development Kit http://www.ti.com/tool/cc3200sdk
TI Service Pack	Refers to Firmware Package released by TI (pertains to updates associated with WiFi and Networking Blocks) http://www.ti.com/tool/cc3200sdk
OTA Client	Over The Air Client - this refers to a sub-module in the OTA library that is responsible for stabling the initial connection with the hosting server to check for updates and the associated listing
CDN Client	Content Delivery Network Client – this refers to a sub-module in the OTA library that is responsible for actually Downloading the elements CDN comes into picture after OTA Client receives the complete listing from the

	hosting server
FLC	File Commit – this refers to a module in the OTA solution that gets invoked after CDN does its Job. This module helps with basic checks on the new image and allows either a “Commit” or a “roll back”
Commit	This is the action of the user application accepting the update via the FLC library
Rollback	This is the action of the user application not accepting the update via the FLC library and requesting a revert to previous (or Factory Default) application image
Uniflash	Tool for programming the CC3200 serial flash. See the CC3100 & CC3200 UniFlash Guide .
CSP	Cloud Service Provider (viz. DropBox)
SFLASH	Serial Flash connected to CC3200. CC3200’s File System resides in this storage

3 Overview of OTA Enablers

At a high level, the OTA enablers, when linked with the MCU application (and triggered):

- Connect with DropBox service and look for an update / Listing
- Download / Transfer the content to SFLASH connected to CC3200 device
- Provide a reference to check the MCU application and allow either to “Commit” or “Roll back”

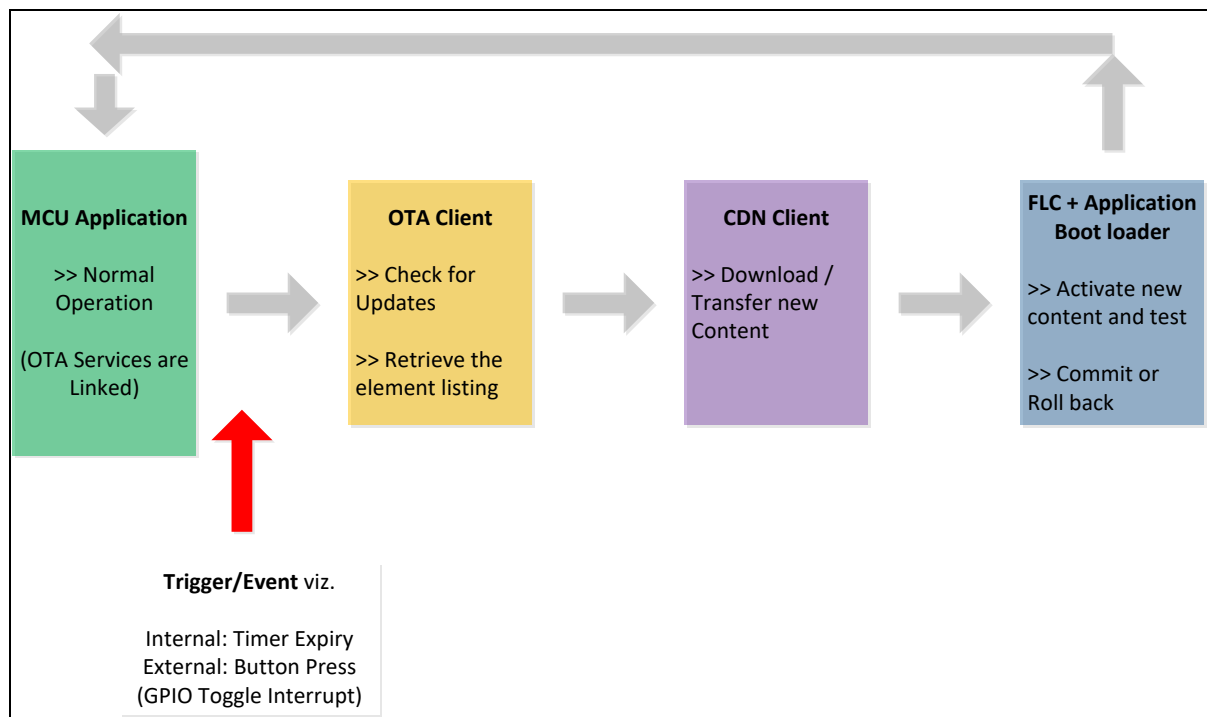


Figure 2 : Concept

In this section, we would have a quick look at the components and sum it all up with a high level flow description

3.1 Components

OTA Enablers included in CC3200 SDK provide different services / references and Figure – 3 attempts to capture there functionalities and utility.

These enablers are:

- OTA Library
- File Commit Library
- MCU Application Boot Loader (a.k.a. Secondary Boot Loader)
- Reference Example

OTA Library (ota.a)	
Location (in SDK Distribution)	\cc3200-sdk\simplelink_extlib\ota
Source	Available
Build	IDE projects and pre-built libraries available CCS - \cc3200-sdk\simplelink_extlib\ota\ccs\Release\ota.a IAR- \cc3200-sdk\simplelink_extlib\ota\ewarm\Release\exe\ota.a GCC - \cc3200-sdk\simplelink_extlib\ota\gcc\exe\ota.a
API Description	\cc3200-sdk\doc\ CC3200 Simplelink OTA Extlib API User's Guide.chm
Functionality	<p>OTA Library can be split in two components.</p> <p>OTA Client is responsible for connecting to the hosting server/service and specifically look for updates in folder specified with the MCU application. In the event of the updates being available, the Listing is queried. OTA Client's communication with the hosting service should comply with the Hosting service's protocol and APIs (for example the reference in our package uses DropBox API)</p> <p>CDN Client is responsible for transferring the Content from the hosting service and deploys a standard HTTP/REST flow.</p>

File Commit Library (flc.a)	
Location (in SDK Distribution)	\cc3200-sdk\simplelink_extlib\flc
Source	Available
Build	IDE projects and pre-built libraries available CCS - \cc3200-sdk\simplelink_extlib\flc\ccs\Release\flc.a IAR- \cc3200-sdk\simplelink_extlib\flc\ewarm\Release\exe\flc.a GCC - \cc3200-sdk\simplelink_extlib\flc\gcc\exe\flc.a
API Description	\cc3200-sdk\doc\ CC3200 Simplelink OTA Extlib API User's Guide.chm
Functionality	<p>FLC provide services for:</p> <ul style="list-style-type: none"> - Accesses to the SFLASH file system (Open, Read, Write, Close) - Managing the MCU image commit process: <ul style="list-style-type: none"> o Uses /sys/mcubootinfo.bin file to identify active image (1, 2) and image status (TESTING, TESTREADY, NOTEST). o Selects the image (file in SFLASH) that needs to be updated with the content that CDN transfers o Allows testing/evaluation the new image by setting TESTREADY and signaling reboot. o Commits the new image when indicated by host application.

MCU Application [a.k.a. Secondary] Boot Loader (application_bootloader.bin)	
Location (in SDK Distribution)	\cc3200-sdk\example\application_bootloader
Source	Available
Build	IDE projects and pre-built images available CCS - \cc3200-sdk\example\application_bootloader\ccs\Release\ IAR- \cc3200-sdk\example\application_bootloader\ewarm\Release\Exe\ GCC - \cc3200-sdk\example\application_bootloader\gcc\exe\
API Description	NA
Functionality	<p>Let us start with an understanding of why we need this component.</p> <p>MCU's Primary Boot Loader executes from the ROM (hence cannot be modified) and loads user's application image residing in Serial FLASH to the MCU's ROM and transfers the execution control.</p> <p>User's Image in the TI FileSystem (SFLASH) is tagged as "/sys/mcuimg.bin". So if we were not concerned about OTA, this would</p>

specifically be the user image.

For OTA, we require two stage boot loading. So the MCU's Primary boot loader would load /sys/mcuimg.bin which would be MCU's Secondary boot loader.

Given the above, the Secondary Boot Loader could be used as an effective tool to load and execute binary images from the SFLASH storage after making a choice from multiple binary images available in the storage. This is the key for providing a back-up mechanism for the MCU application images (something that could enable a rollback, if the new update with OTA has a problem)

Note: Application Boot Loader provided in the SDK is an example with capabilities/Functionality listed below. Source code is made available in the SDK in case the developer wishes to alter the behavior.

Focus of this specific example is to enable the developer to create a roll-back option. Specifically this implementation can manage three user images and would be responsible for selecting one of the following three application images

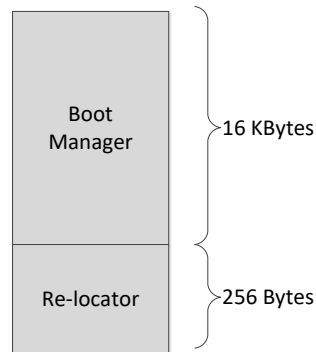
Remember /sys/mcuimg.bin would be the secondary boot loader itself and needs to be flashed in the SFLASH at production.

/sys/mcuimg1.bin	Factory Default (This is something that would be flashed in the system during the production flow for the end equipment)
/sys/mcuimg2.bin	Application Image (Updated)
/sys/mcuimg3.bin	Application Image (Updated)

Key points:

- The secondary boot-loader on execution will auto relocate to the SRAM base area "0x20000000 to 0x20003FFF" before loading the application images; this ensures that actual application can still start at 0x20004000.
- Application boot loader loads application images from SFLASH using simple link NVMEM API's.
- Application boot loader refers to the boot info file (/sys/mcubootinfo.bin) to keep track of the latest and best known image, see "Application image selection and rollback logic" section for more details.

- If the bootinfo file doesn't exist in the file system it is auto generated by application bootloader by scanning the file system for bootable images in the following order, if none exists, the booting is aborted:
 1. Factory Default i.e. /sys/mcuimg1.bin
 2. App Image 1 i.e. /sys/mcuimg2.bin
 3. App Image 2 i.e. /sys/mcuimg3.bin
- Application Boot Loader image is a concatenation of two binaries (termed as re-locator and boot manager)



Source Files briefly explained

- main.c : Contains the core logic for application bootloader
- startup_*.c : Start up file for respective IDEs
- udma_if.c : UDMA driver wrapper API implementation

Usage

This application can be compiled using CCS and IAR. Project files are provided for each in their directories.

This application uses an "Optimized" version of the "simplelink" library. The following section lists the steps for compiling this application

IAR

1. Open the simplelink work space from cc3200-sdk\simplelink\ewarm\simplelink.eww
2. From the workspace select NON_OS project configuration

	<ol style="list-style-type: none"> Go to project->Options->C/C++ Compiler->Optimizations->Level and set it to "Medium" Go to project->options->Library Builder and make sure the output file name is changed to simplelink_opt.a". Build the project configuration. Open cc3200-sdk\example\application_bootloader\ewarm\bootmgr.eww Rebuild the project. This will generate application_bootloader\ewarm\application_bootloader.bin <p>CCS</p> <ol style="list-style-type: none"> Import the simple link project to CCS workspace Right click the project. Go to Build Configurations->Set Active. Select NON_OS. Right click the project->Properties. Go to CCS build->ARM Compiler->Optimization. Set optimization level to "3 Interprocedure Optimizations" Go to CCS build->ARM Archiver->Basic Options. Set output file name to "\${ProjName}_opt.a" Rebuild the project Import "bootmgr" from cc3200-sdk\example\application_bootloader\ccs Rebuild the project. <p>GCC</p> <ol style="list-style-type: none"> Execute the below command on Cygwin in the simplelink/gcc <p style="text-align: center;"><i>make -f Makefile_opt target=NONOS</i></p> <p>See Section-4 section for details to use this bootloader along with OTA Application.</p>
--	--

Reference Example – "ota_update"	
Location (in SDK Distribution)	\cc3200-sdk\example\ota_update
Source	Available
Build	IDE projects and pre-built libraries available
API	NA

Description	
Functionality	Please Refer to Section#4

3.2 Summing it all

Before we summarize the high level flow let us once review the process of setting the content with the CSP (with DropBox in this reference)

3.2.1 Creating the DropBox API Application

- Create a Dropbox account and login
- Go to <https://www.dropbox.com/developers/apps/create> and choose “Dropbox API App”
- Choose “Files and DataStores” and “Yes my app only needs access to files it creates”
- Provide a name for the App and click “Create APP” button
- You’ll be redirected to App settings page. Scroll down to “Generate access token” and click generate. Copy and save the generated token. Please note that this “access token” is absolutely critical for the embedded application to communicate with DropBox service.
- Go to <https://www.dropbox.com/home/Apps>
- Click on the application name
- Create a new folder and give it some meaningful name. This name will be used to identify the CC3200 application version and must be less than 20 characters. The name should include vendor id, product id and the SW version.
for example: “TI_CC3200_WTHR01”

3.2.2 Nomenclature of the files to be hosted on DropBox

Please note that rules specifically related to file nomenclature are quite important as the listing of the files is the only cue the OTA client has regarding the further procedure

The folder nomenclature as mentioned earlier is quite flexible and developer can exercise his discretion. In the following the folder nomenclature is extracted from the reference example we have in the SDK but again to reiterate, the file nomenclature is rather rigid

The files stored on the cloud should be in the following format

`/VidVV_PidPP_VerRRXX/fAA_sys_filename.ext`

The directory or folder `/VidVV_PidPP_VerRR_X.X.X.X`

Note: This is as per our example; developer can choose the name as long as it does not exceed 20 characters

- VV – Vendor id number
- PP – Product id number
- RR – Application version
- X.X.X.X – Service Pack version (4 decimal values as appears in NWP version)

The filename `fAA_sys_filename.ext`

Note: This scheme is exploited by OTA Client, developer is expected to name his files in the cloud accordingly

- fAA – File Flags
 - f - File prefix
- AA - File flags bitmap :
 - 01 - The file is secured
 - 02 - The file is secured with signature
 - 04 - The file is secured with certificate
 - 08 - Don't convert `_sys_` into `/sys/` for SFLASH
 - 10 - Use external storage instead of SFLASH
 - 20 - Reserved.
 - 40 - NWP should be reset after this download
 - 80 - MCU should be reset after this download
- sys - Interpreted as the folder in TI's embedded filesystem (hosted in Serial Flash). So this tag would be converted to `/sys/` directory in TI's SFLASH (embedded) file system
- ext
 - signature - .sig, filename must be the name of the secured file
 - certificate - .cer, filename must be the name of the secured file

Example:

- Assuming the current the device Service Pack (SP) version is 2.7.0.0, and the `OTA_VENDOR_STRING` (that is defined in the Application) is `Vid01_Pid33_Ver18`, then a new service pack (for example: SP_2.8.0.0) will be retrieved from the following CDN location:
`Vid01_Pid33_Ver18_2.7.0.0/f43_sys_servicepack.ucf`

-
- Now that the service pack was updated, the next OTA will be checking for updates in Vid01_Pid33_Ver18_2.8.0.0/. The following is an example for MCU Image update:
Vid01_Pid33_Ver18_2.8.0.0/f80_sys_mcuimgA.bin
 - Assuming the OTA_VENDOR_STRING was updated in the latest MCU image (e.g. to Ver19), then the next update (SP or MCU image) will be taken from:
Vid01_Pid33_Ver19_2.8.0.0/

3.2.3 High Level Flow

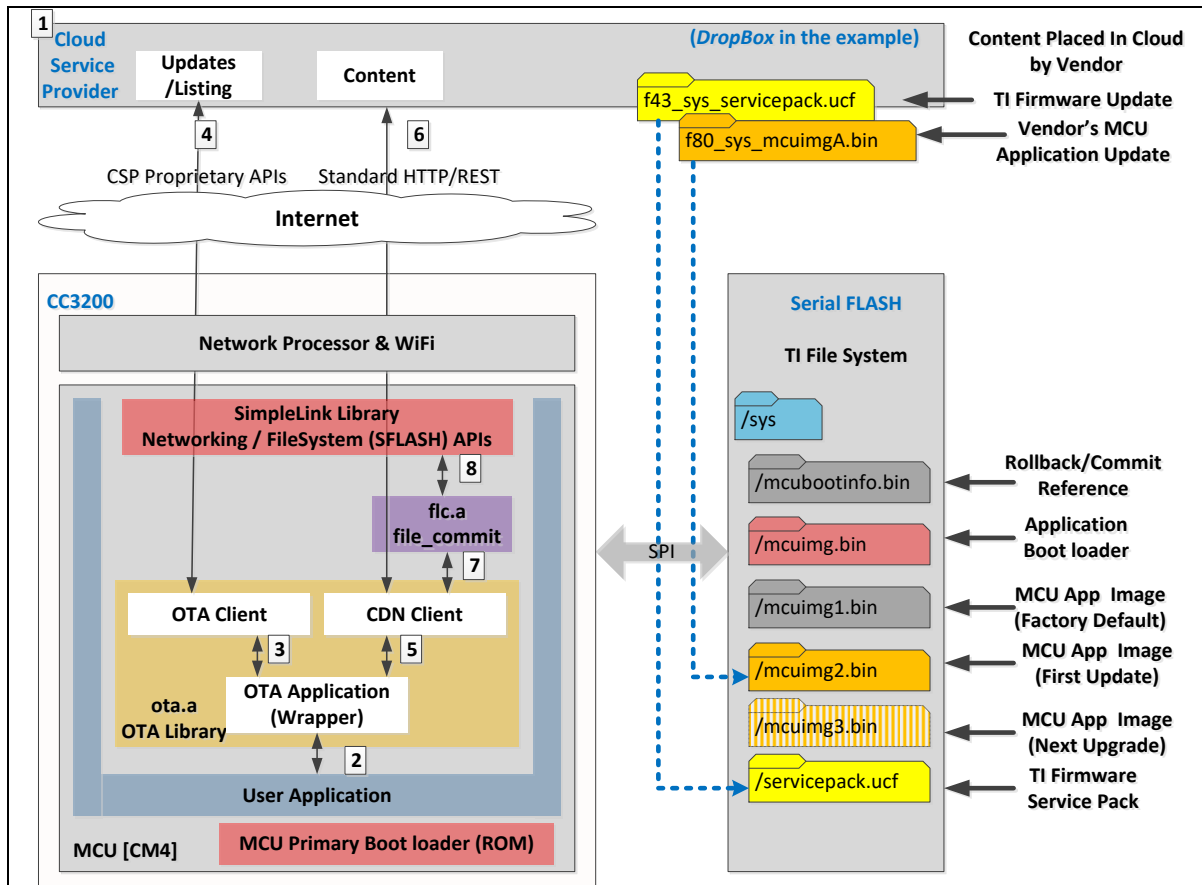


Figure 3 : High Level Flow

1. We have seen earlier the pointers to get the CSP account up and getting the content set up
2. User application invokes the OTA process on an event for example periodically on expiry of a timer or an interrupt tied to a GPIO transition where in the GPIO gets associated with a physically button on the end equipment or the board
3. OTA client send "update_check" request with vendor id , ask OTA server for a list of resources
4. OTA Server based vendor id sends back the list to resources to update
5. OTA client send "metadata" request with next resource id, asking for specific resource information
6. OTA server sends back the CDN domain and path to the resource
7. OTA app call CDN Client to download the resource to the File system (or to external storage)
8. CDN client, using HTTP requests, downloads the file in chunks into the File storage

9. Steps from 4 to 8 are repeated until each resource in the list is updated.
10. OTA returns DOWNLOAD_DONE to user application along with reset MCU and/or NWP flag.
11. User application activates the commit process.

3.2.4 Application Image Selection and Rollback Logic

Let us start with a description of relevant files in TI FileSystem (hosted in SFLASH) from the OTA perspective.

`"/sys/mcuimg.bin"`

This file would host the Application Boot Loader [a.k.a. MCU's Secondary boot loader]. This file should be flashed at production. Remember that MCU's ROM (Primary) Boot loader recognizes only this file and for OTA services this file would host the image corresponding to the secondary boot loader in order to manage/load the following files

`"/sys/mcuimg1.bin"`

This file would host the baseline Factory default user application image. This file should be flashed at production. In our example, we are propagating the notion where this image would always be retained in the file system and could supersede any further updates via OTA, if they do not perform to the desired expectation.

`"/sys/mcuimg2.bin" and "/sys/mcuimg3.bin"`

Now these are images which would be transferred in-system via OTA. Purpose of having two of these is to support a "rollback" feature

`"/sys/mcubootinfo.bin"`

This file is updated by the FLC library and is referred to by the MCU's Application (secondary) boot loader to implement commit/rollback feature. If this file is flashed during production then the application bootloader generates this file on the first power up cycle based on the following rules

If the mcubootinfo.bin file doesn't exist on the file system it is auto generated by this bootloader by scanning the file system for bootable images in the following order, if none exists, the booting is aborted:

- Factory Default - /sys/mcuimg1.bin
- App Image 1 - /sys/mcuimg2.bin
- App Image 2 - /sys/mcuimg3.bin

If user wants to use all the three images, then flash the first application binary as /sys/mcuimg1.bin. In case user wants to use only 2 images (no Factory default) then flash the first application binary as /sys/mcuimg2.bin. Note that if there exists an old /sys/mcubootinfo.bin file and you want to start a fresh with a new image programming via Uniflash, make sure to delete the /sys/mcubootinfo.bin file.

Rollback

This file has the following info

Table 1 : Application image selection and rollback logic

ACTIVE_IMAGE_FILE_NAME (sBootInfo_t.ucActiveImg)	File name of current active image, This can have the values 0 - /sys/mcuimg1.bin – Factory default image 1 - /sys/mcuimg2.bin 2 - /sys/mcuimg3.bin
OTAU_MCU_IMAGE_STATE (sBootInfo_t.ullImgStatus)	Status of the OTAU image 0xABCDDCBA - IMG_STATUS_NOTEST 0x56788765 - IMG_STATUS_TESTREADY 0x12344321 - IMG_STATUS_TESTING

If this file does not exist, the application bootloader creates it based on following priority order:

- If /sys/mcuimg1.bin exists, active image is set to this else
 - If /sys/mcuimg2.bin exists, active image is set to this else
 - If /sys/mcuimg3.bin exists active image is set to this else
 - Bootimg is aborted
- Reads the parameter “**ACTIVE_IMAGE_FILE_NAME**” and “**OTAU_MCU_IMAGE_STATE**” from this file /sys/mcubootinfo.bin.
 - If OTAU_MCU_IMAGE_STATE is equal to **IMG_STATUS_NOTEST**. Launch the contents of the file “**ACTIVE_IMAGE_FILE_NAME**” for execution.
 - If OTAU_MCU_IMAGE_STATE is equal to **IMG_STATUS_TESTING**, It means image previously tested has not been committed. Set the parameter “**OTAU_MCU_IMAGE_STATE**” to “**IMG_STATUS_NOTEST**” and launch the contents of the file “**ACTIVE_IMAGE_FILE_NAME**”

for execution. This step essentially ensures rollback to previous image in case of new image test failure.

4. If **OTAU_MCU_IMAGE_STATE** is equal to **IMG_STATUS_TESTREADY**. Set the parameter “**OTAU_MCU_IMAGE_STATE**” to “**IMG_STATUS_TESTING**” and launch the contents of the file “**NON_ACTIVE_IMAGE_FILE_NAME**” for execution. **ACTIVE_IMAGE_FILE_NAME** and **NON_ACTIVE_IMAGE_FILE_NAME** would share the below relationship

Table 2 : Active and In-Active Images

Active Image	In Active Image
/sys/mcuimg1.bin	/sys/mcuimg2.bin
/sys/mcuimg2.bin	/sys/mcuimg3.bin
/sys/mcuimg3.bin	/sys/mcuimg2.bin

Evolution of Serial Flash Contents with Commit or Rollback – A few Scenarios

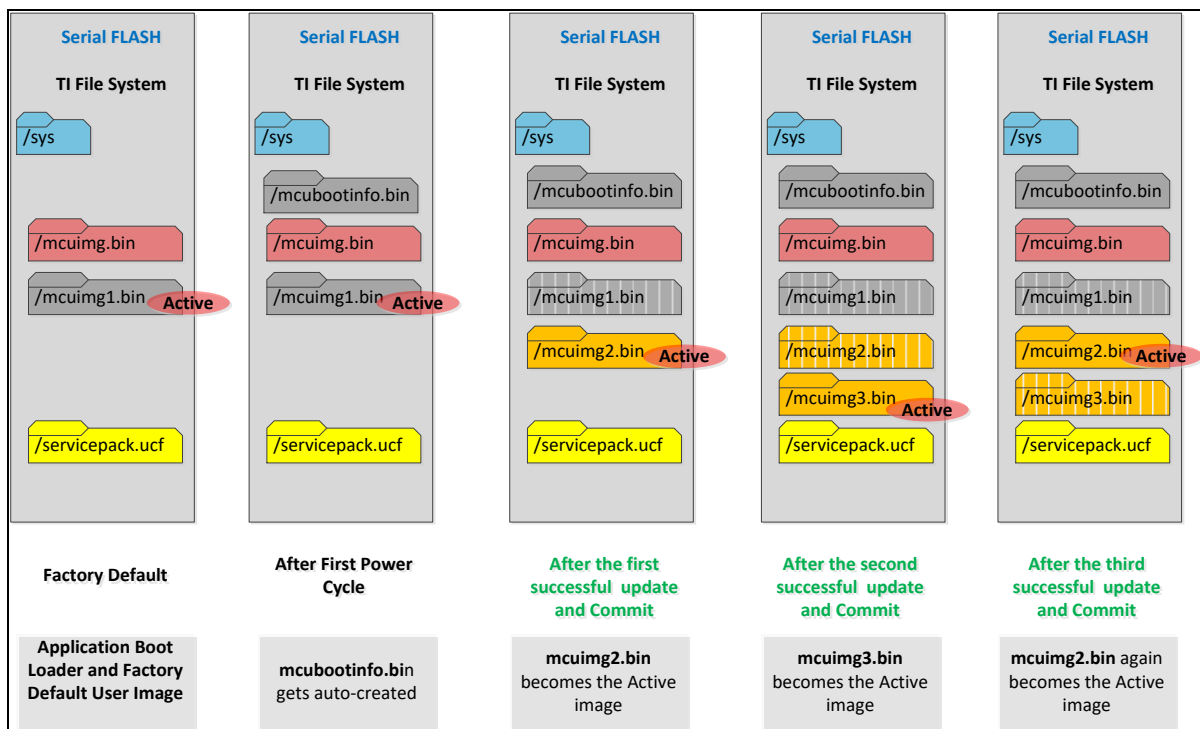
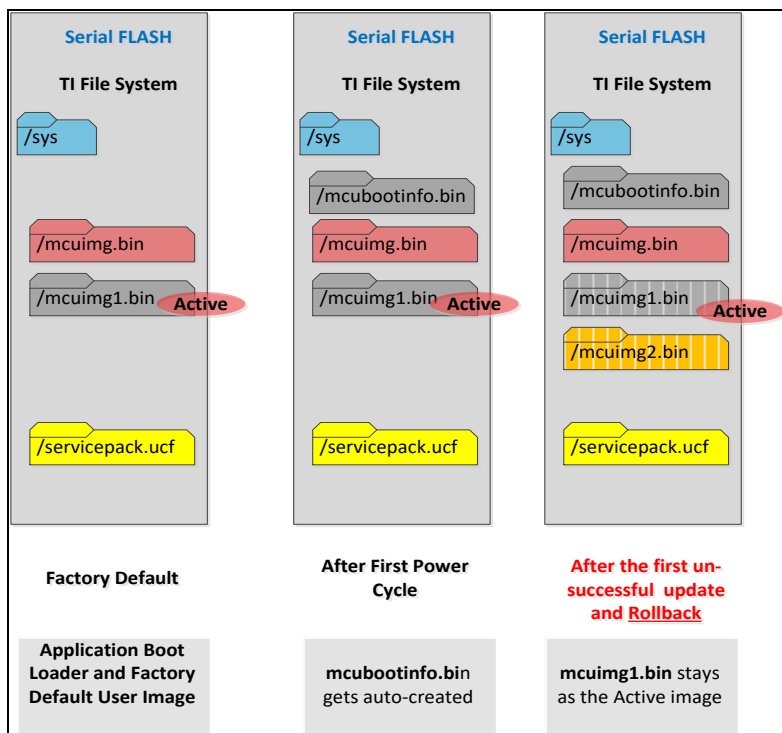


Figure 4 : Flow of Successful Installation and Commit



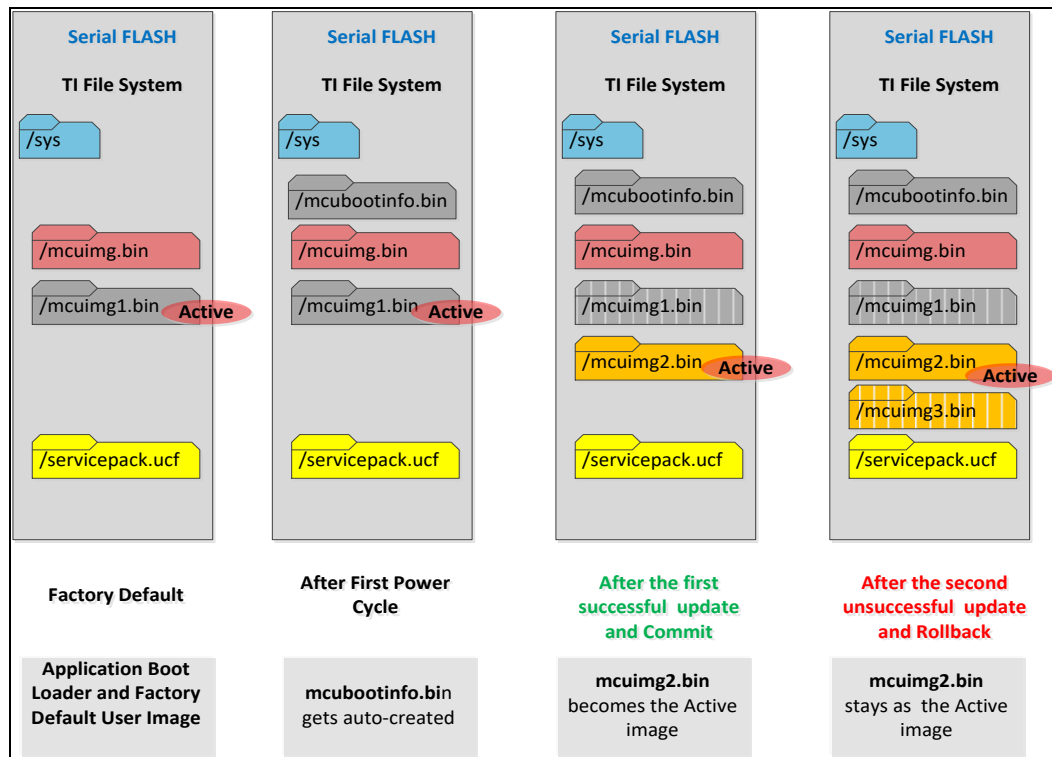


Figure 5 : Flow of un-Successful Installation and rollback

Note: At this point we would like to summarize a few points again

- User application (that gets downloaded or the factory default one as well) has the responsibility of including an acceptability test/criteria (viz..Connectivity to a WiFi AP or Just getting the timer interrupts etc..)
- Based on passing or failure of the above criteria, user application would use FLC API to select (or rather point to) the intended image i.e.
 - Commit to Newly downloaded one or
 - Rollback to Previous Image or Factory Default.
- FLC Library translates the decision to “`/sys/mcubootinfo.bin`” in the Serial Flash FileSystem.
- Application (secondary) Boot Loader can pick the intended MCU application image based on commit or rollback information stored by FLC library in `/sys/mcubootinfo.bin`

4 Sample Application (Packaged with CC3200 SDK)

This application focuses on showcasing CC3200's ability to receive firmware update and/or any related files over the internet enabled Wi-Fi interface. The example uses Dropbox API App platform to store and distribute the OTA update files.

An APP on Dropbox API platform can be looked at as a network accessible drive where user contents are arranged as a tree of files and/or folders. The OTA library expects a folder at the top level which is pointed to via VendorString (the folder name on Dropbox), set during OTA initialization. This top level folder should contain the files to be updated directly and no folders. The OTA library also puts some restriction on the file names (see File Naming Convention for OTA on Dropbox section). File(s) with other name pattern will be rejected.

The VendorString can be constructed in a variety of ways and as an example this application constructs it by appending the Service Pack version to a macro **OTA_VENDOR_STRING** defined in otaconfig.h file.

Assuming the current service pack running on the device holds the version number **2.1.0.87.31.0.0.4.1.1.5.3.3** and **OTA_VENDOR_STRING** is defined as **Vid01_Pid00_Ver01**, the application constructs the VendorString by appending the first 4 bytes (shown in red) to the macro i.e. **Vid01_Pid00_Ver01_2.1.0.87**. This folder on Dropbox should contain all the files that need to be updated. If left empty OTA library assumes a NO_UPDATE condition.

Note: Application bootloader and the OTA library can be compiled with different flavor than the default pre-build binaries. The configuration enables a more optimized and faster boot of the OTA application. A detailed description on enabling the feature is provided as part Appendix section 8.3 at the end of the document

4.1 Source Files briefly explained

- **NON-OS** – Directory holding non-os based implementation of the application
 - main.c – Contains the core logic for the application
 - net.c – Wrapper function implementation for required SL_HOST APIs
 - otaconfig.h – Contains OTA server configuration details
 - pinmux.c – Auto generated file pin muxing on CC3200 LP boards
 - task.c – Implements non-os tasking
- **OS** – Directory holding os based implementation of the application
 - main.c – Contains the core logic for the application
 - net.c – Wrapper function implementation for required SL_HOST APIs
 - otaconfig.h – Contains OTA server configuration details

-
- pinmux.c - Auto generated file pin muxing on CC3200 LP boards

4.2 Creating Dropbox API application

Note: Assuming service pack version **2.1.0.87**.31.0.0.4.1.1.5.3.3 is installed and next it is getting updated to version **2.1.0.88**.31.0.0.4.1.1.5.3.3

1. Create an account with Dropbox and login
2. Go to <https://www.dropbox.com/developers/apps/create> and choose “Dropbox API app”
3. Choose “Files and Datastores” and “Yes My app only needs access to files it creates”.
4. Provide a suitable name for the APP and click “Create APP” button
5. You will be redirected to Apps setting page. Scroll down to “**Generated access token**” and click generate. Copy and save the generated token.
6. Go to <https://www.dropbox.com/home/Apps>
7. Click on the application name
8. Create a new folder and name it “Vid01_Pid00_Ver00_**2.1.0.87**”. This folder will contain only Service pack of version **2.1.0.88**.31.0.0.4.1.1.5.3.3
9. Create another folder and name it “Vid01_Pid00_Ver00_**2.1.0.88**”. This folder will contain Application and other files corresponding to App version **01**.
10. Create another folder and name it “Vid01_Pid00_Ver**01**_**2.1.0.88**”. This folder will contain next update. If no next update, leave the folder empty.
11. The rationale behind steps 8 to 10 is to ensure that Service pack gets updated first and then other files like MCU image are updated. To start with device has Application files of version 00 and service pack of version **87**. In order to update the device to Application files of version **01** and service pack of version **88**, copy new service pack **88** into the folder “Vid01_Pid00_Ver00_**2.1.0.87**” and all other application files corresponding to version 01 into the folder “Vid01_Pid00_Ver00_**2.1.0.88**”. After completion of the update Application will start to point to “Vid01_Pid00_Ver**01**_**2.1.0.88**”

4.3 Configuring the application for new Dropbox account

Note: Assuming service pack version **2.1.0.87**.31.0.0.4.1.1.5.3.3 is installed and next it is getting updated to version **2.1.0.88**.31.0.0.4.1.1.5.3.3

1. Open *otaconfig.h*
2. Update the following Parameters
 - a. **OTA_SERVER_APP_TOKEN** - Set this to the token generated in the previous steps
 - b. **OTA_VENDOR_STRING** - Set this to Vid01_Pid00_Ver01
3. Compile and upload the .bin file into "Vid01_Pid00_Ver00_2.1.0.88" folder on Dropbox server
4. Rename it "f80_sys_mcuimgA.bin".
5. Update the following parameters again
 - a. **OTA_VENDOR_STRING** - Set this to Vid01_Pid00_Ver00
6. Open *example\common\common.h*, this file contains macros related to the network AP to which this application will try to connect. Set the following parameters to match the AP
 - a. **SSID_NAME** - AP SSID Name
 - b. **SECURITY_TYPE** - Security type
 - c. **SECURITY_KEY** - Password, if any

Note: Parameters/macros defined in common.h file are shared among all the SDK examples.

7. Compile and upload the .bin file into "Vid01_Pid00_Ver01_2.1.0.88" folder on Dropbox server
8. Rename it "f80_sys_mcuimgA.bin".

4.4 Usage

4.4.1 Setting up OTA Application for LP with CC3200 SDK

4.4.1.1 Basic Setup

1. Copy *simplelink_extLib* directory into the root directory of the SDK, for example:
C:\ti\CC3200SDK\cc3200-sdk
2. Copy "ota_update" and "application_bootloader" directory into the examples directory, for example: C:\ti\CC3200SDK\cc3200-sdk\example.
3. Create and setup Dropbox account (see sections below).

4.4.1.2 Flashing

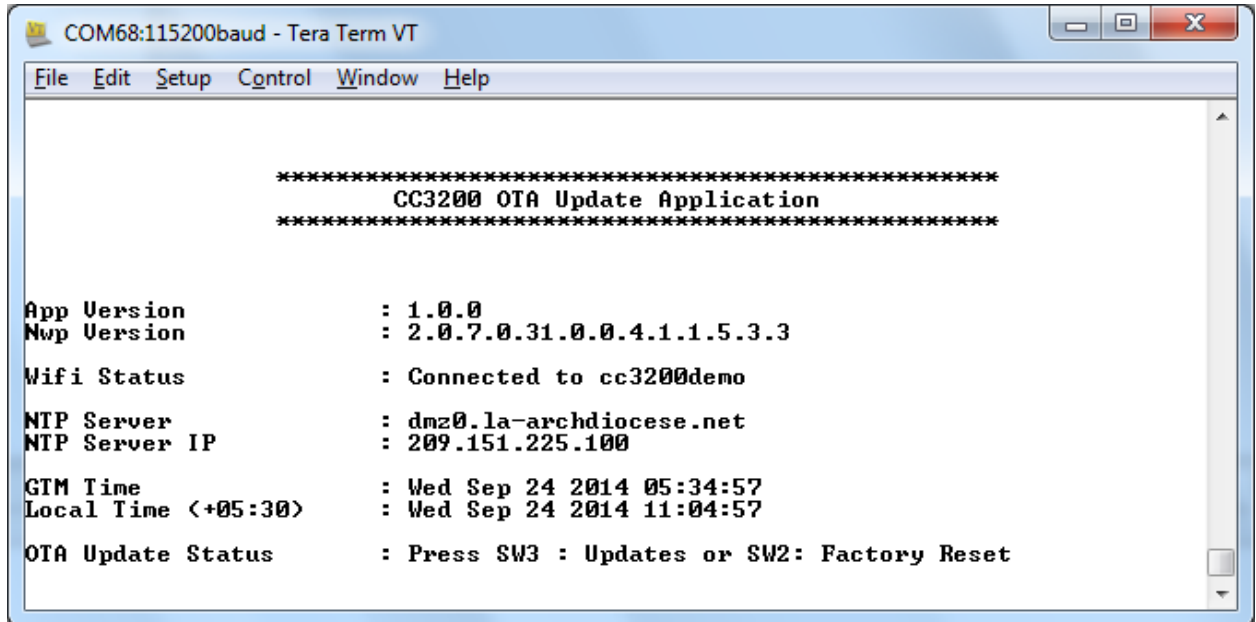
1. Open Uniflash tool for CC3xxx
2. Make sure LP is in UARTLOAD mode i.e. **SOP2** jumper mounted.
3. Format the sFlash.
4. Program the service pack.
5. Go to File->New Configuration. Choose the connection and board and click ok
6. Set “/sys/mcuimg.bin” URL to point “application_bootloader.bin”
7. Select “Erase”, “Update” and “Verify”
8. Add a new file and rename it to “/sys/mcuimg1.bin”. If no factory default is required rename it to “/sys/mcuimg2.bin”. Set the URL of this file to point to the OTA application binary, set the URL of this file to point to the OTA application binary, say “ota_update_nonos.bin” if using NON-OS example.
9. Select “Erase”, “Update” and “Verify” for this new file.
10. Set the COM port to LP’s com port number
11. Press Program

4.4.1.3 Running

1. Remove SOP2 jumper
2. Connect to COM port via Tera-term or Hyper Terminal with following configuration
 - a. **Port:** Enumerated COM port
 - b. **Baud rate:** 115200
 - c. **Data:** 8 bit
 - d. **Parity:** None
 - e. **Stop:** 1 bit
 - f. **Flow control:** None
3. Press reset.

4.4.2 Triggering OTA Update

1. Wait for the Application to completely boot. Observer the app version



COM68:115200baud - Tera Term VT

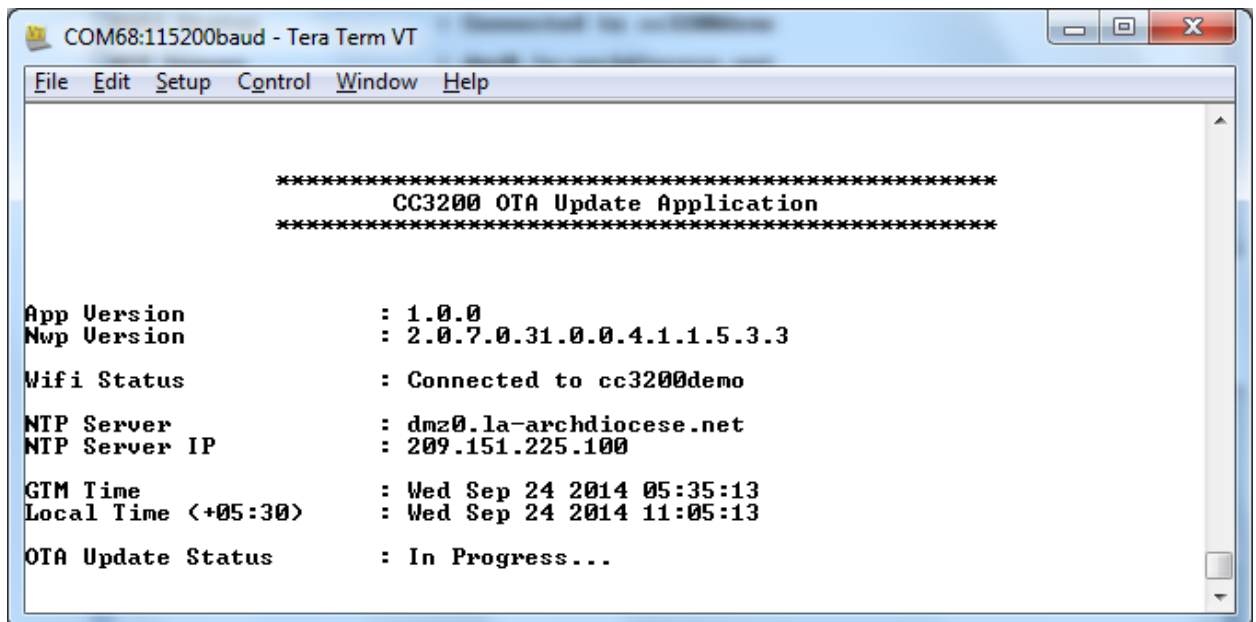
File Edit Setup Control Window Help

```

*****
CC3200 OTA Update Application
*****

App Version           : 1.0.0
Nwp Version           : 2.0.7.0.31.0.0.4.1.1.5.3.3
Wifi Status           : Connected to cc3200demo
NTP Server             : dmz0.la-archdiocese.net
NTP Server IP          : 209.151.225.100
GTM Time               : Wed Sep 24 2014 05:34:57
Local Time (<+05:30)  : Wed Sep 24 2014 11:04:57
OTA Update Status      : Press SW3 : Updates or SW2: Factory Reset
  
```

2. Press SW3 on LP to start the update. Observer the “OTA Update Status” is now “In Progress”



COM68:115200baud - Tera Term VT

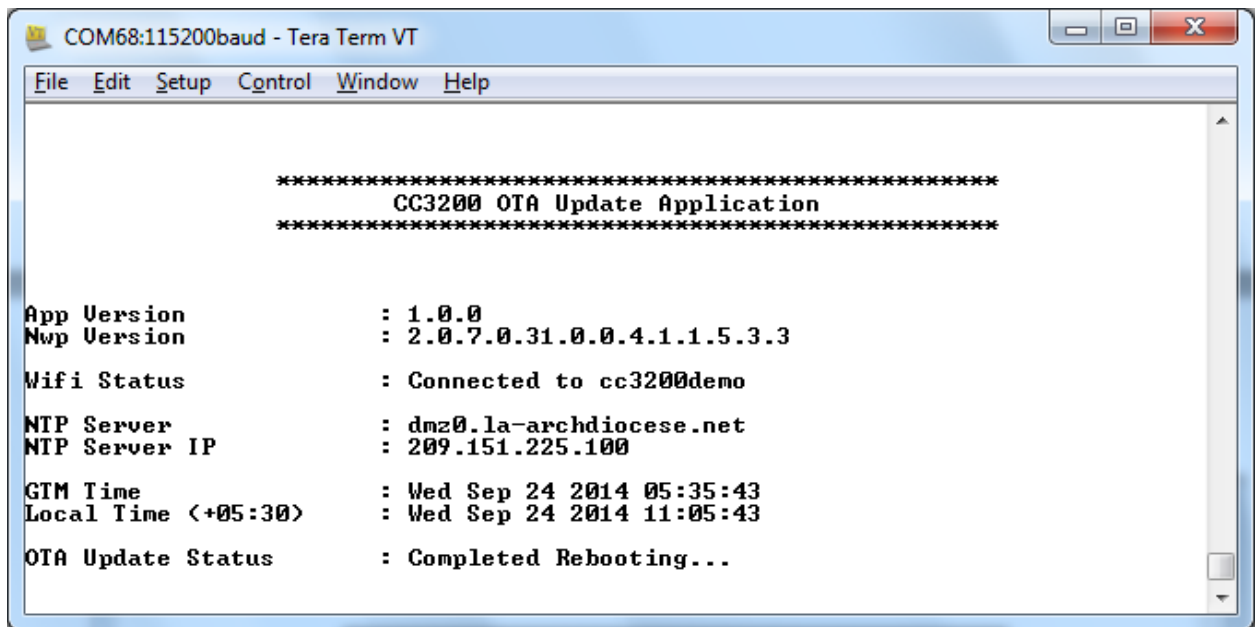
File Edit Setup Control Window Help

```

*****
CC3200 OTA Update Application
*****

App Version           : 1.0.0
Nwp Version           : 2.0.7.0.31.0.0.4.1.1.5.3.3
Wifi Status           : Connected to cc3200demo
NTP Server             : dmz0.la-archdiocese.net
NTP Server IP          : 209.151.225.100
GTM Time               : Wed Sep 24 2014 05:35:13
Local Time (<+05:30)  : Wed Sep 24 2014 11:05:13
OTA Update Status      : In Progress...
  
```

3. Wait for OTA to complete. Application will reboot itself



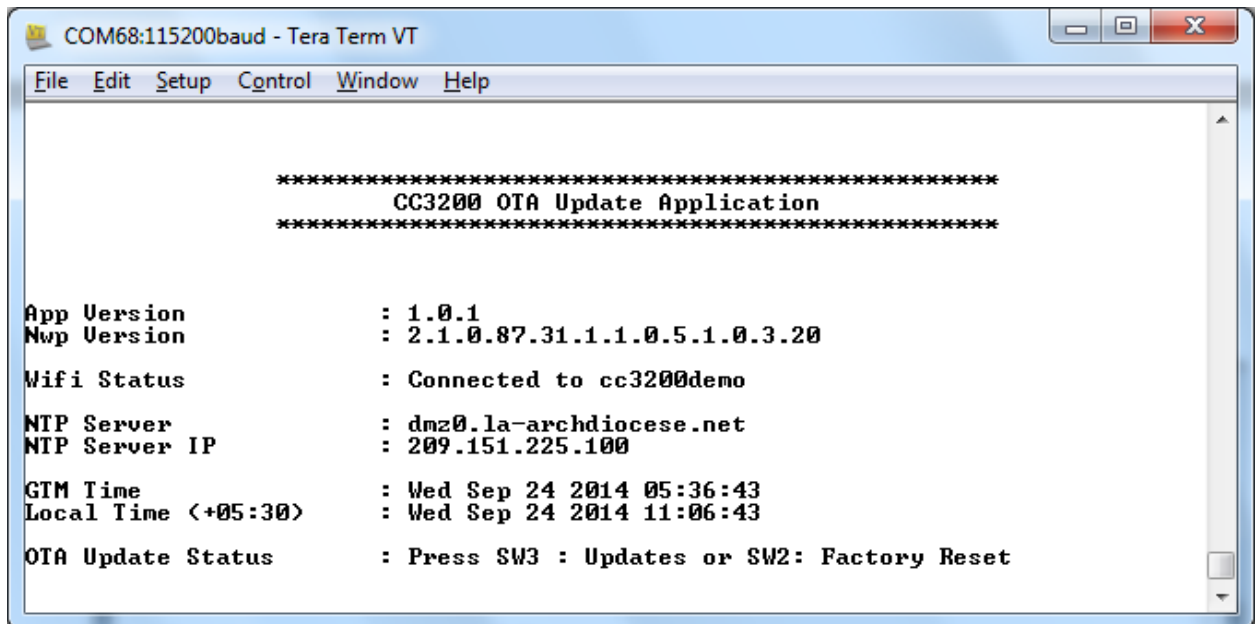
```

COM68:115200baud - Tera Term VT
File Edit Setup Control Window Help

*****
CC3200 OTA Update Application
*****

App Version           : 1.0.0
Nwp Version           : 2.0.7.0.31.0.0.4.1.1.5.3.3
Wifi Status           : Connected to cc3200demo
NTP Server             : dmz0.la-archdiocese.net
NTP Server IP         : 209.151.225.100
GTM Time              : Wed Sep 24 2014 05:35:43
Local Time (<+05:30) : Wed Sep 24 2014 11:05:43
OTA Update Status     : Completed Rebooting...
  
```

4. Observer App version after reboot



```

COM68:115200baud - Tera Term VT
File Edit Setup Control Window Help

*****
CC3200 OTA Update Application
*****

App Version           : 1.0.1
Nwp Version           : 2.1.0.87.31.1.1.0.5.1.0.3.20
Wifi Status           : Connected to cc3200demo
NTP Server             : dmz0.la-archdiocese.net
NTP Server IP         : 209.151.225.100
GTM Time              : Wed Sep 24 2014 05:36:43
Local Time (<+05:30) : Wed Sep 24 2014 11:06:43
OTA Update Status     : Press SW3 : Updates or SW2: Factory Reset
  
```

Note: Nwp version will be different from one show based on the service packs used.

4.4.3 Sequence diagram

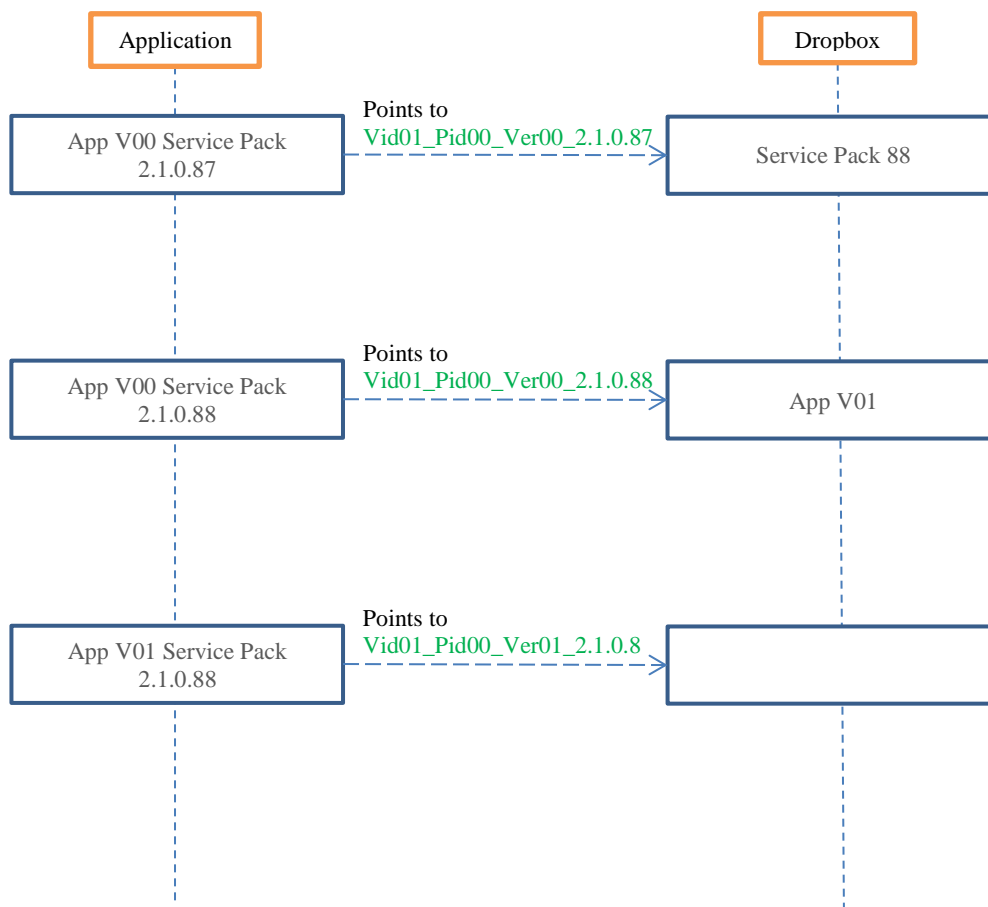


Figure 2: OTA Application Update Sequence

5 Adding OTA Feature to Existing MCU Application – Reference

This section would be useful for developers who would like quickly add OTA feature to their existing application. We have assumed that the baseline application uses TI's Code Composer Studio as development environment. Procedure is extendible to IAR as well.

Specifically here we have picked the GET_WEATHER application and illustrated the steps one might have to follow to add OTA feature (GET_WEATHER is a sample application available in CC3200 SDK distribution - \cc3200-sdk\example\get_weather).

5.1 Creating DropBox API Application

Refer to section “3.2.1” – Do retain the “Access Token”

5.2 Getting CCS Settings in Order

- Open CCS
- Import get_weather project
- Go to project properties\ ARM Linker\ File Search Path
- Add the following paths to the library search path
 1. "\${CC3200_SDK_ROOT}/simplelink_extlib/ota/ccs/release"
 2. "\${CC3200_SDK_ROOT}/simplelink_extlib/flc/ccs/release"
- Add the following libraries to the include libraries:
 1. "ota.a"
 2. "flc.a"
- Go to project properties\ ARM Compiler\ Include Options
 1. Add the following path to the include search path:
"\${CC3200_SDK_ROOT}/simplelink_extlib/include"

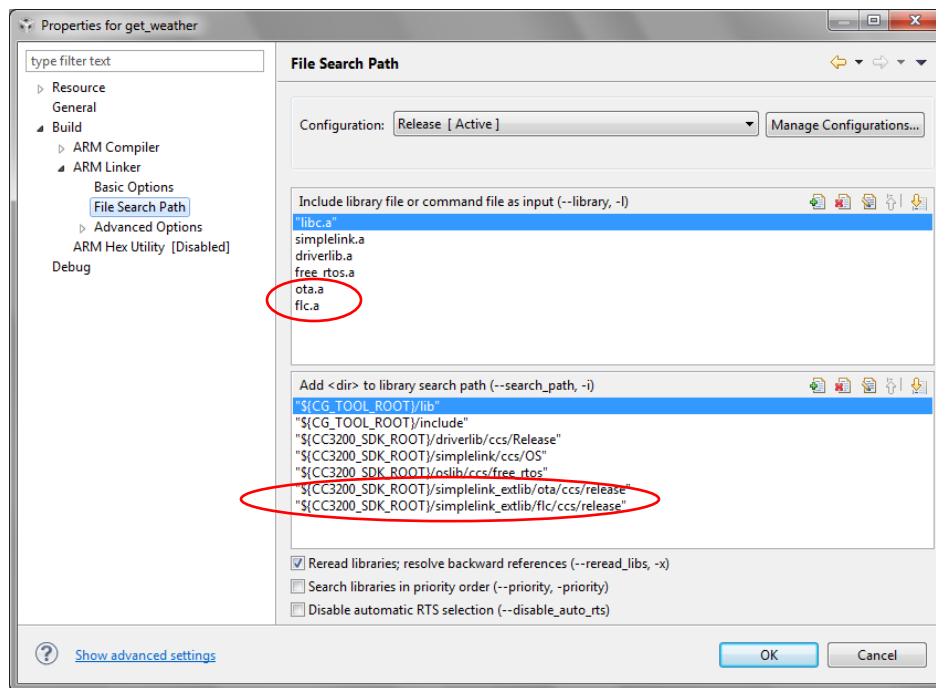


Figure 7: CCS Snapshot – Adding OTA / FLC libraries to existing MCU App

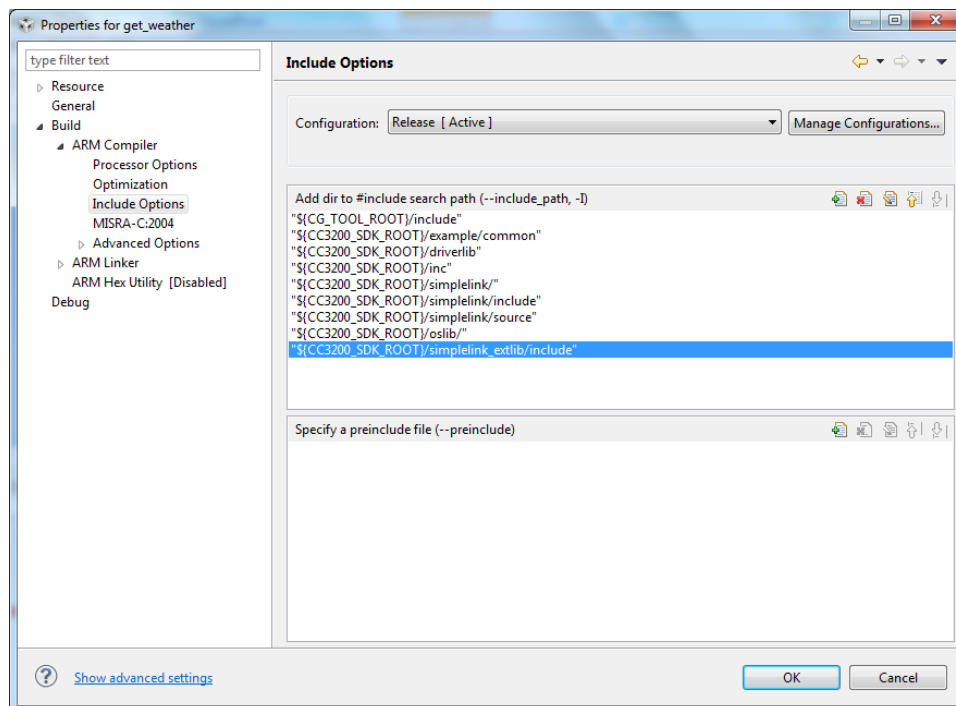


Figure 8: CCS Snapshot – Adding the "Include" Folder

5.3 Adding / Linking OTA Components with GET_WEATHER Application

- In CCS, open main.c (of “get_weather”) and make the updates based on the following table (a zip file is embedded in the document as well enable copy and paste of the following statements - The zip file extracts 8 files with segments to be added into the code)



Edits_for_OTA.zip

Include Section	<code>#include "flc_api.h"</code> <code>#include "ota_api.h"</code>
Global / Define section	<code>#define OTA_SERVER_NAME "api.dropbox.com"</code> <code>#define OTA_SERVER_IP_ADDRESS 0x00000000</code> <code>#define OTA_SERVER_SECURED 1</code> <code>#define OTA_SERVER_APP_TOKEN "<dropbox access token>"</code> <code>#define OTA_VENDOR_STRING "Vid00_Pid00_Ver00"</code>
Function Prototypes	<code>int OTAServerInfoSet(void **pvOtaApp, char *vendorStr);</code> <code>static void RebootMCU();</code>
Local Variables	To be added to <code>GetWeatherTask()</code> function <code>long OptionLen1</code> <code>unsigned char OptionVal;</code> <code>int SetCommitInt;</code> <code>unsigned char ucVendorStr[30];</code>
OTA Initialization	To be added to <code>GetWeatherTask()</code> function after <code>GPIO_IF_LedOn(MCU_GREEN_LED_GPIO)</code> function call <code>// Initialize OTA</code> <code>pvOtaApp =</code> <code>sl_extLib_OtaInit(RUN_MODE_NONE_OS RUN_MODE_BLOCKING,0);</code> <code>strcpy((char *)ucVendorStr,OTA_VENDOR_STRING);</code> <code>OTAServerInfoSet(&pvOtaApp,(char *)ucVendorStr);</code>
OTA main construct /logic	To be added to <code>GetWeatherTask()</code> function after <code>lRetVal = Network_IF_GetHostIP((char*)g_ServerAddress,</code> <code>&ulDestinationIP);</code>

	<pre> // // Check if this image is booted in test mode // sl_extLib_OtaGet(pvOtaApp, EXTLIB_OTA_GET_OPT_IS_PENDING_COMMIT, &OptionLen, (_u8 *)&OptionVal); UART_PRINT("EXTLIB_OTA_GET_OPT_IS_PENDING_COMMIT? %d \n\r", OptionVal); if (OptionVal == true) { UART_PRINT("OTA: PENDING COMMIT & WLAN OK ==> PERFORM COMMIT \n\r"); SetCommitInt = OTA_ACTION_IMAGE_COMMITTED; sl_extLib_OtaSet(pvOtaApp, EXTLIB_OTA_SET_OPT_IMAGE_COMMIT, sizeof(int), (_u8 *)&SetCommitInt); } else { UART_PRINT("Starting OTA\n\r"); lRetVal = 0; while (!lRetVal) { lRetVal = sl_extLib_OtaRun(pvOtaApp); } UART_PRINT("OTA run = %d\n\r", lRetVal); if (lRetVal < 0) { UART_PRINT("OTA: Error with OTA server\n\r"); } else if (lRetVal == RUN_STAT_NO_UPDATES) { UART_PRINT("OTA: RUN_STAT_NO_UPDATES\n\r"); } else if ((lRetVal & RUN_STAT_DOWNLOAD_DONE)) { // Set OTA File for testing lRetVal = sl_extLib_OtaSet(pvOtaApp, EXTLIB_OTA_SET_OPT_IMAGE_TEST, sizeof(int), (_u8 *)&SetCommitInt); UART_PRINT("OTA: NEW IMAGE DOWNLOAD COMPLETE\n\r"); UART_PRINT("Rebooting...\n\r"); RebootMCU(); } } } </pre>
Banner Display	<p>Replace "DisplayBanner(APP_NAME);" with the new code</p> <pre> DisplayBanner(OTA_VENDOR_STRING); </pre>
New Functions	<p>Added at the end of "main.c"</p> <pre> static void RebootMCU() { // // Configure hibernate RTC wakeup // PRCMHibernateWakeupSourceEnable(PRCM_HIB_SLOW_CLK_CTR); } </pre>


```
//
// Delay loop
//
MAP_UtilsDelay(8000000);

//
// Set wake up time
//
PRCMHibernateIntervalSet(330);

//
// Request hibernate
//
PRCMHibernateEnter();

//
// Control should never reach here
//
while(1)
{
}
}

int OTAServerInfoSet(void **pvOtaApp, char *vendorStr)
{
    //
    // Set OTA server info
    //
    g_otaOptServerInfo.ip_address = OTA_SERVER_IP_ADDRESS;
    g_otaOptServerInfo.secured_connection = OTA_SERVER_SECURED;
    strcpy((char *)g_otaOptServerInfo.server_domain, OTA_SERVER_NAME);
    strcpy((char *)g_otaOptServerInfo.rest_hdr_val,
OTA_SERVER_APP_TOKEN);

    //
    // Set OTA server Info
    //
    sl_extLib_OtaSet(*pvOtaApp, EXTLIB_OTA_SET_OPT_SERVER_INFO,
                    sizeof(g_otaOptServerInfo),
                    (_u8 *) &g_otaOptServerInfo);

    //
    // Set vendor ID.
    //
    sl_extLib_OtaSet(*pvOtaApp, EXTLIB_OTA_SET_OPT_VENDOR_ID,
                    strlen(vendorStr),
                    (_u8 *)vendorStr);

    //
    // Return ok status
    //
    return RUN_STAT_OK;
}
```

- Set **OTA_VENDOR_STRING** with the name of your Dropbox library/folder
E.g: "TI_CC3200_WTHR01"
- Set **OTA_SERVER_REST_HDR_VAL** with the dropbox token you have received when creating the dropbox API
- Rebuild the "get_weather" project

5.4 Configuring the default SFLASH contents

MCU Application (Secondary) Boot Loader

- Make sure SOP2 jumper is mounted (Set to '1')
- Open CC3200 uniflash
- Press "new target configuration" and then "ok"
- Set the CC3200 port number
- Choose /sys/mcuimage.bin
- Set the url field to: C:\ti\CC3200SDK_x.x.x\cc3200-sdk\example\application_bootloader\ccs\Release\application_bootloader.bin
- Check the Erase, Update, Verify boxes

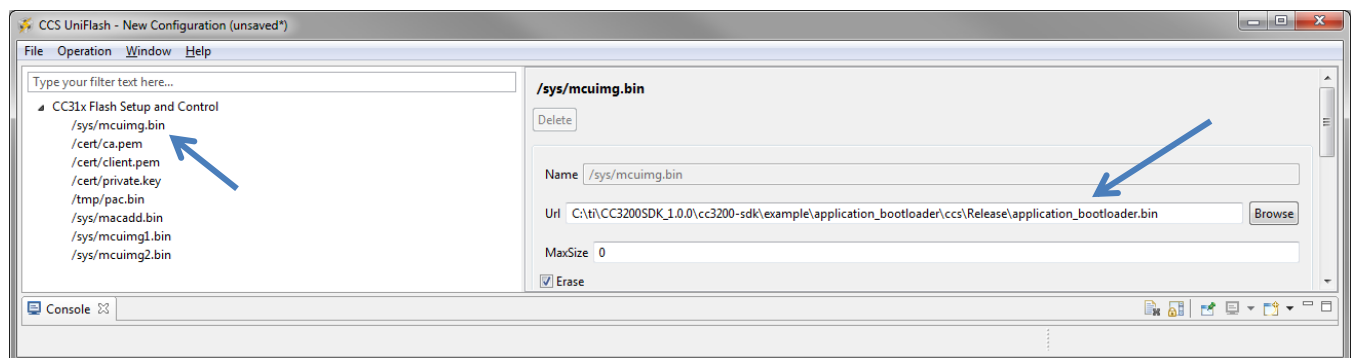


Figure 9: Uniflash Snapshot – Programming Application Bootloader

MCU Application Image(s)

- Choose Add file (on Uniflash session)

- Change the new file name to “/sys/mcuimg1.bin”
- Set the file url to “get_weather.bin”
(out come of our last build, located under the workspace folder and not under the SDK, For example: C:\Users\youname\workspace\get_weather\Release)
Check the Erase, Update, Verify boxes
- Set the MaxSize to the maximum size you wish to reserve for your application
- Choose Add file
- Change the new file name to “sys/mcuimg2.bin”
- Repeat on the setting of the previous file

So now we have the Factory Default programmed (/sys/mcuimg1.bin) and we have also flashed the same as the back-up image (/sys/mcuimg2.bin)

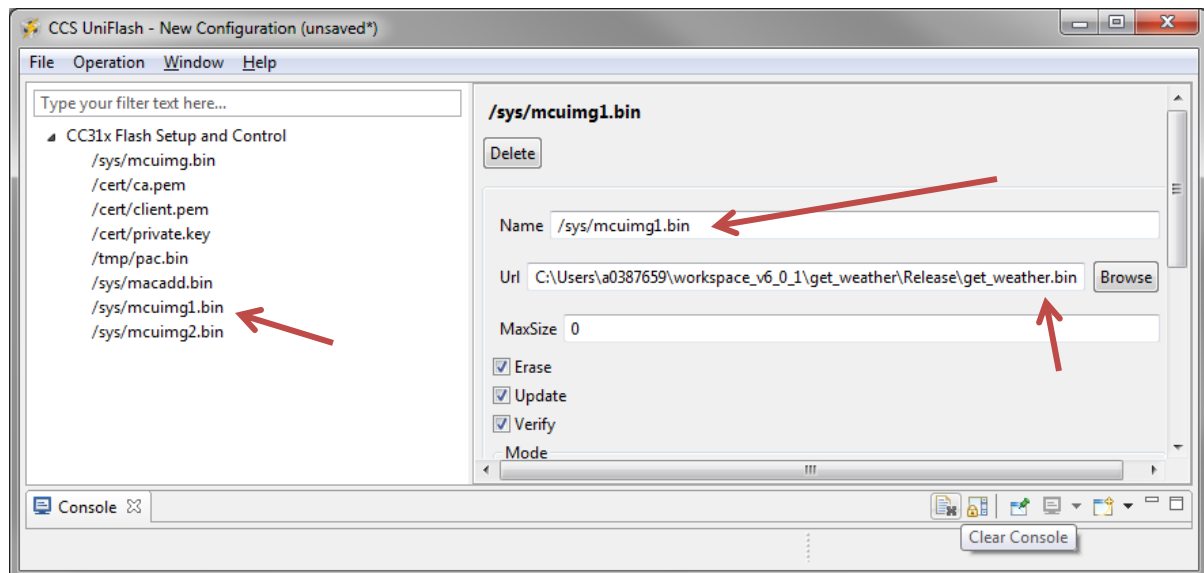


Figure 10: Uniflash Snapshot – Programming Factory Default

Programming

- Go back to the main Uniflash screen
- Press Format
- Press Choose Service pack update and browse for the binary (.bin) of the latest servicepack
- Press “Program”

- Note: Please follow the instructions in the console menu, directing you to reset the CC3200 LP from time to time

- Note:
 1. Once finished with UniFlash programming, Remove SOP2 jumper
 2. Connect a UART terminal to the LP COM port, set baud rate to 115200
 3. Restart the LP, by pressing the SW1 switch
 4. Connect to a WLAN network and see that it is trying to check for updates

5.5 Uploading the new Image to Cloud

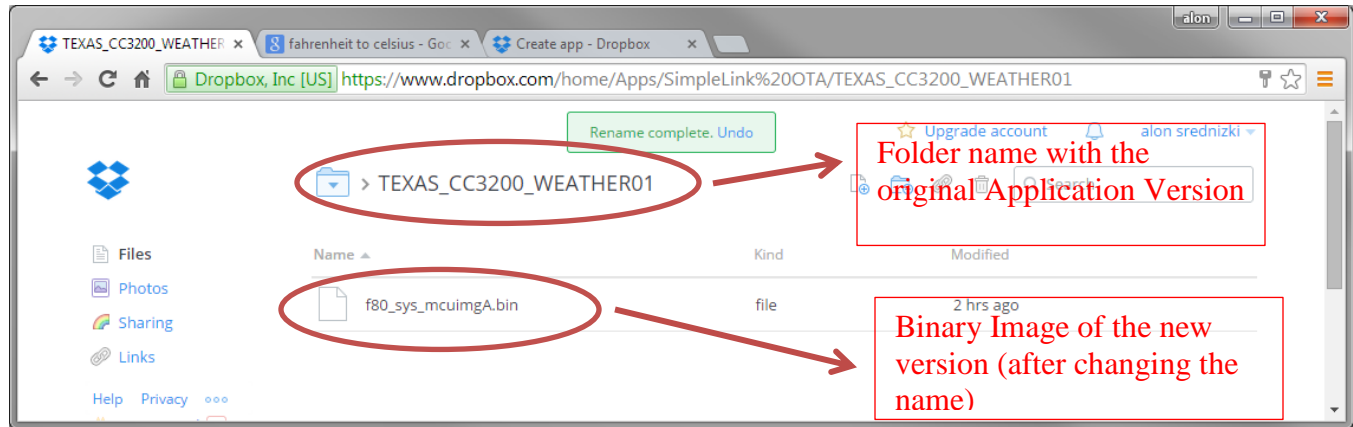


Figure 11: DropBox Snapshot – Getting the content uploaded

- Change OTA_VENDOR_STRING to a new version then used before
 - E.G: Original name: "TI_CC3200_WTHR01"
New name: "TI_CC3200_WTHR02"
- Build the project
- Drag and drop the new binary to the Dropbox app folder
- Rename the file name to "f80_sys_mcuimgA.bin"
- Restart the LP, and connect to a WLAN network

6 Moving to other File Hosting Services

6.1 Porting OTA Library to other servers

Updating the TI's CC3x00 OTA library to work with different hosting services would require adapting transactions to the protocol requirements of the specific hosting service for accessing content.

The library focuses on providing implementation for OTA using Dropbox server. Users requiring to port this library to a different server will have to, but not limited to, re-implement following low level APIs.

1. OtaClient_UpdateCheck:

This function is expected to get list and number of updates from the OTA server and store it in the internal data structure *pOtaClient->pOtaCheckUpdateResponse->rsrcList*. This array holds the absolute path name and size of each file on the server.

This function is responsible for generating HTTP request packet, sending it over to the server and parsing out the required details

For e.g. to get the list of file from the Dropbox server under *Vid01_Pid00_Ver01* folder the required HTTP request will be:

```
GET https://api.dropbox.com/1/metadata/auto/Vid01_Pid00_Ver01
Authorization: Bearer <access_key>
```

And the response

```
{"hash": "6f01f3a9e21da382bd3f9b6eee57fa2c", "rev": "96826f56f0b",
"thumb_exists": false, "path": "/Vid01_Pid00_Ver01", "is_dir": true,
"contents": [{"rev": "97d26f56f0b", "thumb_exists": false, "path":
"/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin", "is_dir": false,
"client_mtime": "Tue, 09 Sep 2014 12:29:44 +0000", "icon": "page_white",
"bytes": 51812, "modified": "Tue, 09 Sep 2014 12:29:44 +0000", "size":
"50.6 KB", "root": "app_folder", "mime_type": "application/octet-stream",
"revision": 2429}], "icon": "folder", "bytes": 0, "modified": "Tue, 09 Sep
2014 12:00:05 +0000", "size": "0 bytes", "root": "app_folder", "revision":
2408}
```

2. OtaClient_ResourceMetadata

This function is responsible for following:

- Get the path of the resource on the CDN server
- Extracting the file name of the resource removing the prefixes and flags. This would be the name with which the file will be stored on local storage (sFlash)
- Extract the flags from the resource name

This function is responsible for generating the HTTP request packet, sending and parsing the response received

For e.g. to get the absolute path of a resource (*/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin*) from Dropbox the HTTP request will be

GET https://api.dropbox.com/1/media/auto/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin
Authorization: Bearer <access_key>

And the response:

```
{"url":  
"https://dl.dropboxusercontent.com/1/view/19y6u9ma456k9p5/Apps/HELLO\_WORLD\_OTA/Vid01\_Pid00\_Ver01/f80\_sys\_mcuimgA.bin", "expires": "Fri, 09 Jan 2015 14:10:01  
+0000"}
```

Where

1. https://dl.dropboxusercontent.com/1/view/19y6u9ma456k9p5/Apps/HELLO_WORLD_OTA/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin is the path of the resource on the CDN,
2. _sys_mcuimg.bin converted to /sys/mcuimg.bin is the resource file name for storage
3. f80 if the flag. F if prefix and should be ignored.

For extracting file name and flags refer to OtaClient_ResourceNameConvert()
implementation for Dropbox.

3. CdnClient_ConnectByUrl

This function is responsible for connecting to the CDN serve, initiating the download of the resource file and skipping any header. No file data is read by this API.

For e.g. for the resource

https://dl.dropboxusercontent.com/1/view/19y6u9ma456k9p5/Apps/HELLO_WORLD_OTA/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin. Here <https://dl.dropboxusercontent.com> is the CDN domain name to which the connection is to be made and [/1/view/19y6u9ma456k9p5/Apps/HELLO_WORLD_OTA/Vid01_Pid00_Ver01/f80_sys_mcuimgA.bin](#) is the resource on this server.

6.1.1 Server Info Structure

This structure holds the server related parameter like the domain name, authorization key, REST APIs, log server and vendor string. Following member variables are required to be initialized and passed to OTA Library as part of initialization.

Depending on the end server to which this library is being ported, some of the parameters might be obsolete or user might have to add more parameters/elements.

server_domain: This holds the server name for the OTA server.

Eg: api.dropbox.com for Dropbox REST APIs

secured_connection: This holds if the connection to the OTA server and CDN server is secure or non-secure.

rest_update_chk: This defines the REST API for getting the list of resources from the server
Eg: /1/metadata/auto/

rest_rsrc_metadata: This defines the API for getting the details of each resource on the server
Eg: /1/media/auto

rest_hdr: This holds the additional HTTP headers (like authorization) for the server
Eg: Authorization: Bearer

rest_hdr_val: Holds the header value, like the access key.
Eg: BwPuaYu9AoAAABBBAAAAA-uhCfuTU_Jw54oBVgBCtZaMAsDfhTZcV8ILK7ruzD51r

log_server_name: Server name for logging the OTA logs
Eg: api-content.dropbox.com

rest_files_put: This holds the REST API for writing to the server
Eg: /1/files_put/auto/

log_mac_address: MAC address of the current device using the OTA library. This is used for logging

6.2 Porting OTA Library and example to exosite server

The below description assumes that you have familiarized with file hosting capabilities and format of the exosite server. Contact exosite for any queries or further details. By default the OTA library and example which is part of SDK works with the dropbox server. In order to use OTA library and example with exosite below changes need to be done.

1. File : example\ota_update*\otaconfig.h. Below OTA server specific parameters need to be changed. The parameter values mentioned below are specifically for one of the example file that was hosted on the **exosite** server. Values for parameters "OTA_SERVER_NAME" and "OTA_VENDOR_STRING" should match your **exosite** configuration.

```
//#define OTA_SERVER_NAME      "api.dropbox.com"
#define OTA_SERVER_NAME      "texasinstruments.m2.exosite.com.dropbox.com"

//#define OTA_SERVER_APP_TOKEN "<dropbox access token>"
#define OTA_VENDOR_STRING     "TBD X-Exosite-CIK: value"

//#define OTA_VENDOR_STRING     "Vid00_Pid00_Ver00"
#define OTA_VENDOR_STRING     "vendor=texasinstruments&model=cc3200lp_v1"
```


- File : simplelink_extlib\ota\otacommon.h. Below OTA server specific parameters need to be changed. The parameter values mentioned below are specifically for one of the example file that was hosted on the **exosite** server.
Note that OTA_DROPBOX_V2 should not be defined as exosite is similar to Dropbox V1 API. This requires that OTA library will be rebuilt (make sure that the corresponding lib binary is linked to your application).

```
//#define OTA_DROPBOX_V2

//#define OTA_SERVER_REST_UPDATE_CHK    "/1/metadata/auto/"
#define OTA_SERVER_REST_UPDATE_CHK    "/provision/download?"

//#define OTA_SERVER_REST_RSRC_METADATA    "/1/media/auto"
#define OTA_SERVER_REST_RSRC_METADATA    "/provision/download?vendor=texasinstruments&model=cc3200lp_v1&info=true&id="

//#define OTA_SERVER_REST_HDR            "Authorization: Bearer"
#define OTA_SERVER_REST_UPDATE_CHK      "X-Exosite-CIK:"
```

- File: simplelink_extlib\include\ota_api.h. Below defines highlighted in red need to be changed so as to accommodate info from exosite server.

```
#define MAX_SERVER_NAME    32+50
#define MAX_PATH_PREFIX    48+50
#define MAX_REST_HDRS_SIZE 96+50
#define MAX_USER_NAME_STR  8
#define MAX_USER_PASS_STR  8
#define MAX_CERT_STR        32+50
#define MAX_KEY_STR         32+50
```

- File: simplelink_extlib\include\ota_api.h. To the structure OtaFileMetadata_t add the parameters cdn_hdr and cdn_hdr_val as below. For every access to exosite server “access key” needs to be provided. The below additional parameters are for that.

```
86 /*****
87 * File metadata Structure
88 *****/
89 typedef struct
90 {
91     /* files server name */
92     _u8 cdn_url[256];
93     _u8 cdn_hdr[100];
94     _u8 cdn_hdr_val[100];
95
96     /* file flags */
97     _i32 flags;
98
99     /* file name */
```

- File: simplelink_extlib\ota\CdnClient.c. Change the _ReadFileHeaders declaration as below. For every access to exosite server “access key” needs to be provided. The below additional parameters are for that

```
i32 OpenStorageFile(CdnClient_t *pCdnClient, u8 *file_name, i32 file_size, u32 *ulToken, i32 *lFileHandle);
i32 _ReadFileHeaders(_i16 fileSockId, _u8 *domian_name, _u8 *file_name, _u8 *hdr, _u8 *hdr_val);
i32 _RecvFileChunk(_i16 fileSockId, void *pBuf, i32 len, i32 flags);
```

6. File: simplelink_extlib\ota\CdnClient.c. Change the size of req_uri as below. This increase of size is to accommodate longer path on the server

```
i32 CdnClient_ConnectByUrl(void *pvCdnClient, OtaFileMetadata_t *pResourceMetadata, i32 secured_connection)
{
    CdnClient_t *pCdnClient = (CdnClient_t *)pvCdnClient;
    u8 domain_name[64];
    u8 req_uri[512];
    u8 *cdn_url = pResourceMetadata->cdn_url;
```

7. File: simplelink_extlib\ota\CdnClient.c. Invocation of _ReadFileHeaders inside CdnClient_ConnectByUrl needs to be changed as below. For every access to exosite server “access key” needs to be provided. The below additional parameters are for that.

```
/* read file headers, skip all not relevant headers */
status = _ReadFileHeaders(pCdnClient->fileSockId, pCdnClient->cdn_server_name, req_uri, pResourceMetadata->cdn_hdr, pResourceMetadata->cdn_hdr_val);
if (status < 0)
{
    Report("CdnClient_ConnectByUrl: ERROR, _ReadFileHeaders, status=%ld\r\n", status);
    CdnClient_CloseServer(pvCdnClient);
    return CDN_STATUS_ERROR_READ_HDRS;
}
```

8. File : simplelink_extlib\ota\CdnClient.c. Change _ReadFileHeaders as below. For every access to exosite server “access key” needs to be provided. The below additional parameters are for that.

```
i32 _ReadFileHeaders(_i16 fileSockId, _u8 *domian_name, _u8 *file_name, _u8 *hdr, _u8 *hdr_val)
{
    i32 status=0;
    i32 len;
    u8 *send_buf = http_send_buf();

    Report("_ReadFileHeaders: domain=%s, file=%s\r\n", domian_name, file_name);

    http_build_request (send_buf, "GET ", domian_name, NULL, file_name, hdr, hdr_val);
```

9. File : simplelink_extlib\ota\OtaApp.c. Remove invocation of LogClient_ConnectAndPrint inside sl_extLib_OtaRun. To enable logging on the exosite server the LogClient_ConnectAndPrint API needs to be changed. As currently this not updated this invocation should be commented out.

```
Report("sl_extLib_OtaRun: ----- end of updates\r\n");

WriteStatFile(( u8 *)pOtaApp->pStatistics, sizeof(g OtaApp statistics));
//LogClient_ConnectAndPrint(pOtaApp->pvLogClient, pOtaApp->pOtaServerInfo, pOtaApp->vendorStr, pOtaApp->pStatistics);

status = RUN_STAT_DOWNLOAD_DONE;
return status;
```

10. File : simplelink_extlib\ota\OtaApp.c. Add below lines to the state OTA_STATE_RESOURCE_LIST inside sl_extLib_OtaRun. For every access to exosite server “access key” needs to be provided. The below additional parameters are for that.

```
pOtaApp->pResourceMetadata->sflash_file_size = pOtaApp->file_size; /* !!! the only parameter
strcpy(pOtaApp->pResourceMetadata->cdn_hdr, pOtaApp->pOtaServerInfo->rest_hdr);
strcpy(pOtaApp->pResourceMetadata->cdn_hdr_val, pOtaApp->pOtaServerInfo->rest_hdr_val);
break;
```

11. File : simplelink_extlib\ota\OtaApp.h. Change size of parameter vendorStr inside the structures OtaApp_t and OtaApp_statistics_t as below. This increase in size is to match the vendor string modified for the exosite application.

```
typedef struct
{
    _u8 vendorStr[100];
    _i32 startCount;
    _i32 continuousAccessErrorCount;

typedef struct
{
    OtaState_e state;
    _i32 runMode;
    _u8 vendorStr[100];
```

12. File simplelink_extlib\ota\OtaClient.c. Add below lines to OtaClient_ResourceMetadata. Ensure that EXOSITE_OTA_SERVER is defined. As http_build_request is not compatible with exosite server these parameters are explicitly set.

```
#ifdef TI_OTA_SERVER
    http_build_request (send_buf, "GET ", pOtaServerInfo->server_domain, pOtaServerInfo->rest_rsrc_metadata, NU
#elif EXOSITE_OTA_SERVER
    strcpy((char *)pMetadata->cdn_url, (char const *)"http://");
    strcat((char *)pMetadata->cdn_url, (char const *)pOtaServerInfo->server_domain);
    strcat((char *)pMetadata->cdn_url, (char const *)"/provision/download?");
    strcat((char *)pMetadata->cdn_url, (char const *)pOtaClient->pVendorStr);
    strcat((char *)pMetadata->cdn_url, "&id=");
    strcat((char *)pMetadata->cdn_url, (char const *)resource_file_name);
    return OTA_STATUS_OK;
#else
    http_build_request (send_buf, "POST ", pOtaServerInfo->server_domain, pOtaServerInfo->rest_rsrc_metadata, re
#endif
```

13. File `simplelink_extlib\ota\OtaClient.c`. Add below lines to `OtaClient_UpdateCheck`. Ensure that `EXOSITE_OTA_SERVER` is defined. This is to invoke specific parsing needed for exosite.

```
#ifdef TI_OTA_SERVER
    numUpdates = json_parse_update_check_resp(pOtaClient->serverSockId, pOtaClient->pOtaCheckUpdateResponse->rsrclist, response_buf, len);
#elif EXOSITE_OTA_SERVER
    numUpdates = http_parse_exosite_update_check_resp(pOtaClient, pOtaClient->pOtaCheckUpdateResponse->rsrclist, response_buf, len);
#else
    numUpdates = json_parse_dropbox_metadata(pOtaClient->serverSockId, pOtaClient->pOtaCheckUpdateResponse->rsrclist, response_buf, len);
#endif
```

14. File `simplelink_extlib\ota\OtaClient.c` : Add below lines inside `OtaClient_ResourceNameConvert`. Ensure that `EXOSITE_OTA_SERVER` is defined. This is to support specific parsing needed for exosite.

```
1  i32 file_flags;
2  _u8 temp_str[100];

3
4  pMetadata->p_file_name = &pMetadata->rsrc_file_name[10]; /* space for prefix */
5  strcpy((char *)pMetadata->p_file_name, (const char *)resource_file_name);
6
7  /* Add metadata info (remove it after server complete) */
8  pMetadata->p_cert_filename = NULL;
9  pMetadata->p_signature = NULL;
10 pMetadata->flags = 0;

11 #ifndef EXOSITE_OTA_SERVER
12     /* skip vendor string - as directory in the file name */
13     if (pOtaClient->pVendorStr)
14     {
15         pMetadata->p_file_name = &pMetadata->p_file_name[strlen((const char *)pOtaClient->pVendorStr)+1]; /* skip "/" */
16     }

17     if (pMetadata->p_file_name[1] != 'f') /* must start with 'f' flags */
18     {
19         Report("OtaClient_ResourceMetadata: ignore file name: %s, without f prefix\r\n", pMetadata->p_file_name);
20         return OTA_STATUS_ERROR;
21     }
22 #else
23     strcpy((char *)temp_str, (char *)pMetadata->p_file_name);
24     strcpy((char *)&pMetadata->p_file_name[1], (char *)temp_str);
25     pMetadata->p_file_name[0] = '/';
26     if (pMetadata->p_file_name[1] != 'f') /* must start with 'f' flags */
27     {
28         Report("OtaClient_ResourceMetadata: ignore file name: %s, without f prefix\r\n", pMetadata->p_file_name);
29         return OTA_STATUS_ERROR;
30     }
31 #endif
```

15. File `simplelink_extlib\ota\OtaHttp.c` : Add the below routine `http_parse_exosite_update_check_resp`. This is to support specific parsing needed for exosite.

```

~/
_i32 http_parse_exosite_update_check_resp(OtaClient_t *pOtaClient, RsrcData_t *pRsrcData, _u8 *read_buf, _i32 size)
{
    _u8 fileNumber=0;
    _u8 Ndx;
    _u8 *pBuf;
    _u8 *pBuf1;
    _i32 len;
    RsrcData_t currentFile;
    OtaOptServerInfo_t *pOtaServerInfo = pOtaClient->pOtaServerInfo;

    //
    // Skip headers
    //
    pBuf = (_u8 *)strstr((char *)read_buf, "\r\n\r\n") + 4;
    pBuf1 = (_u8 *)strstr((char *)pBuf, "\r\n");

    while(NULL != pBuf1)
    {
        memset(&currentFile, 0, sizeof(RsrcData_t));
        strncpy((char *)currentFile.filename, (char *)pBuf, pBuf1-pBuf);

        pBuf = pBuf1 + 2;
        pBuf1 = (_u8 *)strstr((char *)pBuf, "\r\n");

        pRsrcData[fileNumber++] = currentFile;
    }

    for(Ndx=0; Ndx < fileNumber; Ndx++)
    {
        memset(send_buf, 0, HTTP_SEND_BUF_LEN);
        memset(read_buf, 0, HTTP_RECV_BUF_LEN);
        http_build_request (send_buf, "GET ",
                           pOtaServerInfo->server_domain,
                           pOtaServerInfo->rest_rsrc_metadata ,
                           pRsrcData[Ndx].filename,
                           pOtaServerInfo->rest_hdr,
                           pOtaServerInfo->rest_hdr_val);

        len = sl_Send(pOtaClient->serverSockId, send_buf, (_i16)strlen((const char *)send_buf), 0);

        len = sl_Recv_eagain(pOtaClient->serverSockId,
                             read_buf, HTTP_RECV_BUF_LEN,
                             0, MAX_EAGAIN_RETRIES);

        if(len <= 0 )
            return OTA_STATUS_ERROR;

        pBuf = read_buf;

        pBuf = (_u8 *)strstr((const char *)pBuf, "\r\n\r\n"); /* skip HTTP response header */
        if (pBuf == NULL)
            return OTA_STATUS_ERROR;

        pBuf = (_u8 *)strstr((const char *)pBuf, ",");
        if (pBuf == NULL)
            return OTA_STATUS_ERROR;

        pBuf += 1;

        len = strstr((const char *)pBuf, ",") - (char *)pBuf;
        if (len <= 0)
            return OTA_STATUS_ERROR;

        pRsrcData[Ndx].size = strtoul((char const *)pBuf, NULL, 10);
    }

    return fileNumber;
}

```

16. File simplelink_extlib\ota\OtaHttp.h : Add the declaration for

http_parse_exosite_update_check_resp

```
_i32 json_parse_update_check_resp(_i16 sockId, RsrcData_t *pRsrcData, _u8 *read_buf, _i32 size);
```

```
_i32 http_parse_exosite_update_check_resp(OtaClient_t *pOtaClient, RsrcData_t *pRsrcData, _u8 *read_buf, _i32 size);
```

```
_i32 json_parse_exosite_metadata_url(_u8 *media_response_buf, OtaFileMetadata_t *pMetadata);
```

7 Limitations/Known Issues

1. OTA cannot update an image to a newer image of same name if the size of the newer image is larger than the max size allocated to that image.
2. File revert is only supported for MCU image file.

8 Appendix

8.1 Detailed Representation of the OTA flow

Followed diagrams could be referred to in conjunction with Section-3

8.1.1 OTA Application State Machine

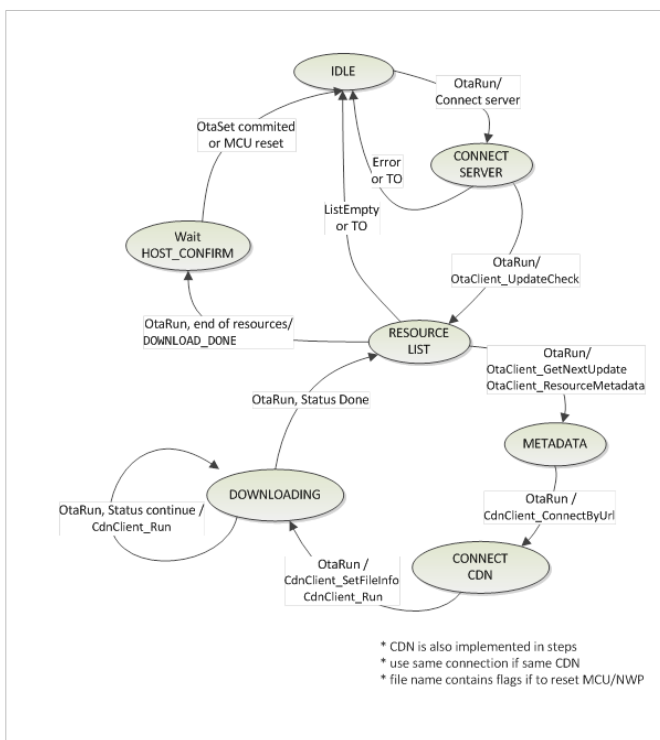


Figure 13 : OTA Application State Machine

8.1.2 Sequence Diagrams

OTA client/server sequence

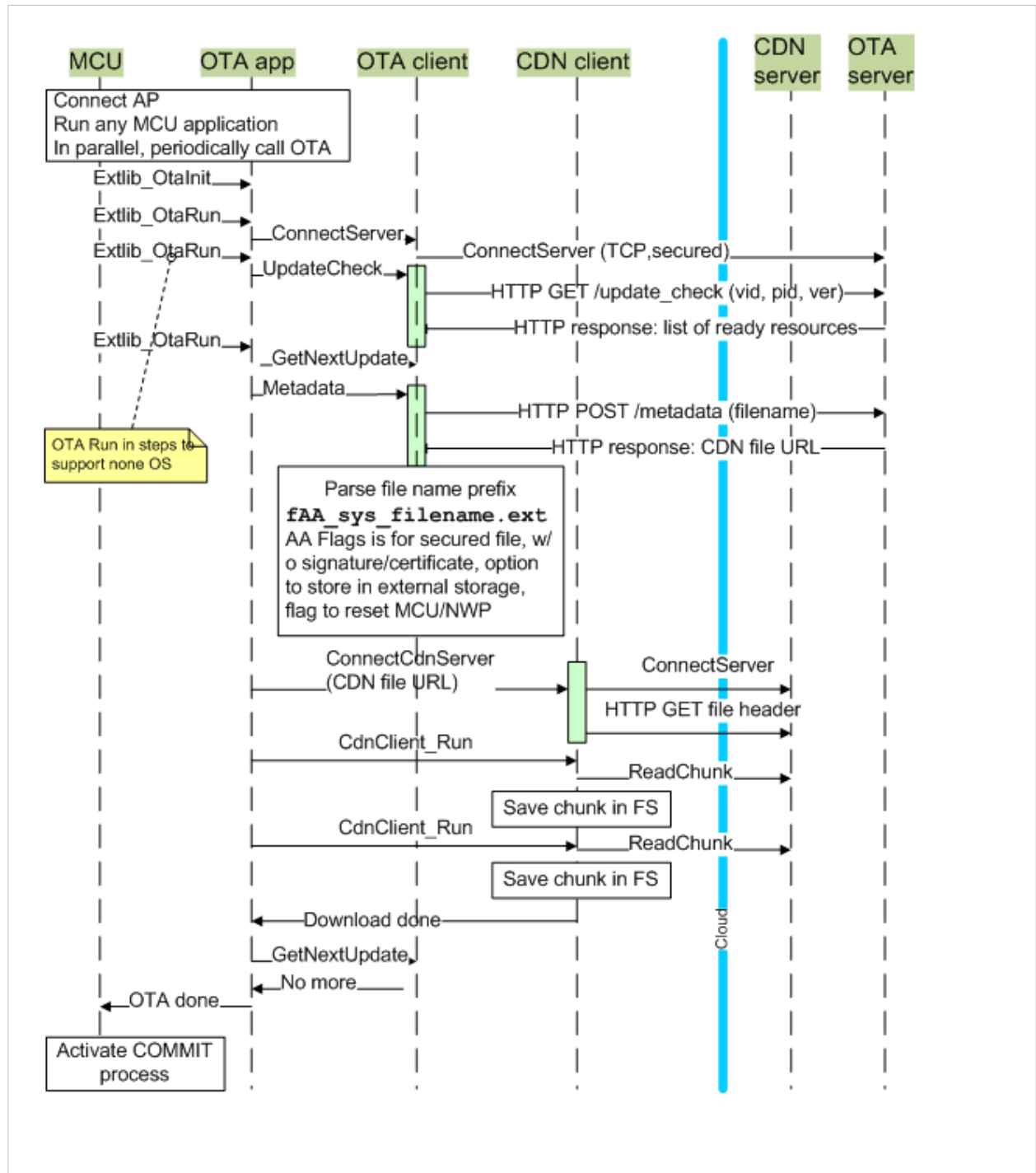


Figure 13 : OTA client/server sequence

OTA and MCU Commit sequence – Success

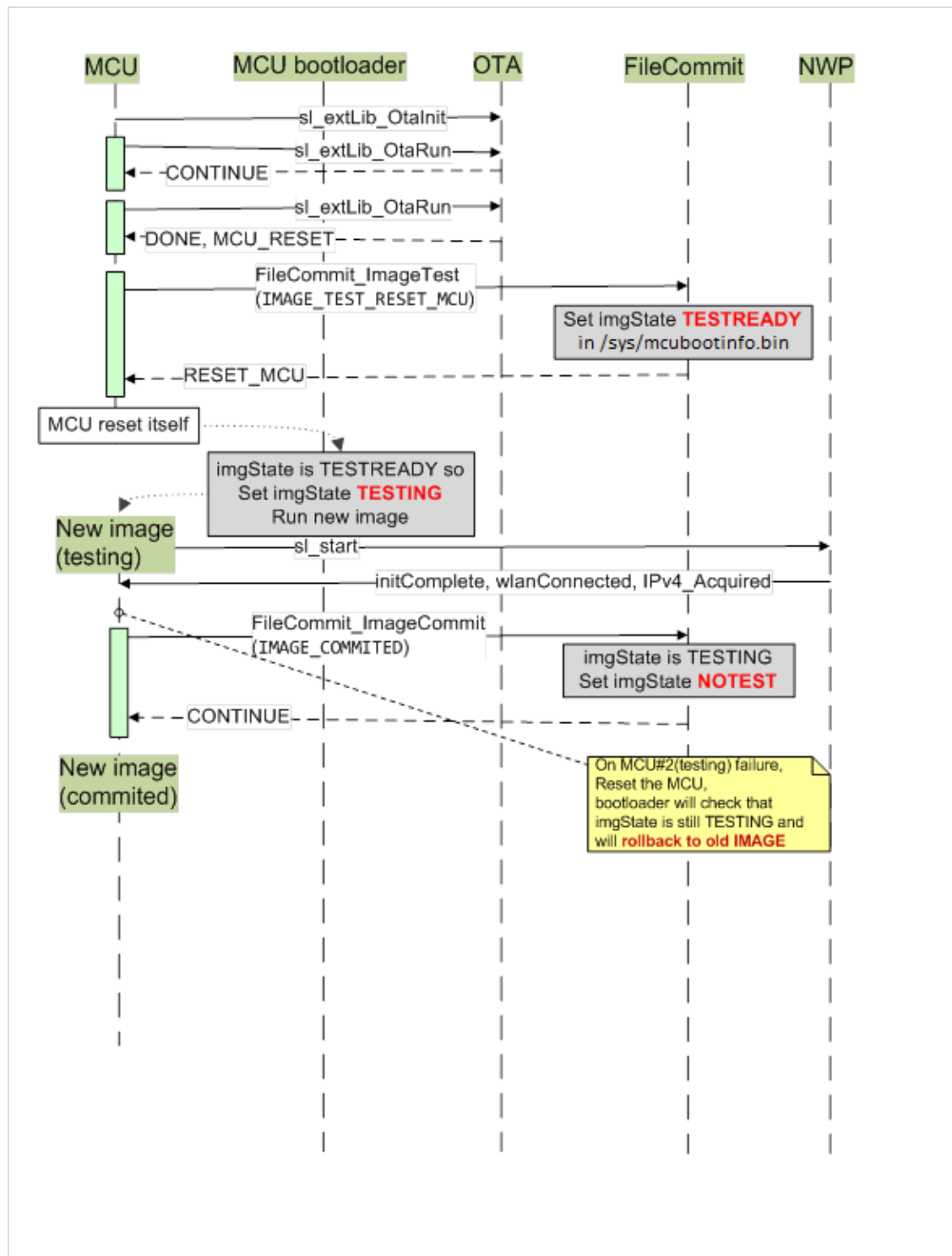


Figure 14 : OTA and MCU Commit sequence - success

OTA and MCU Commit sequence – Rollback

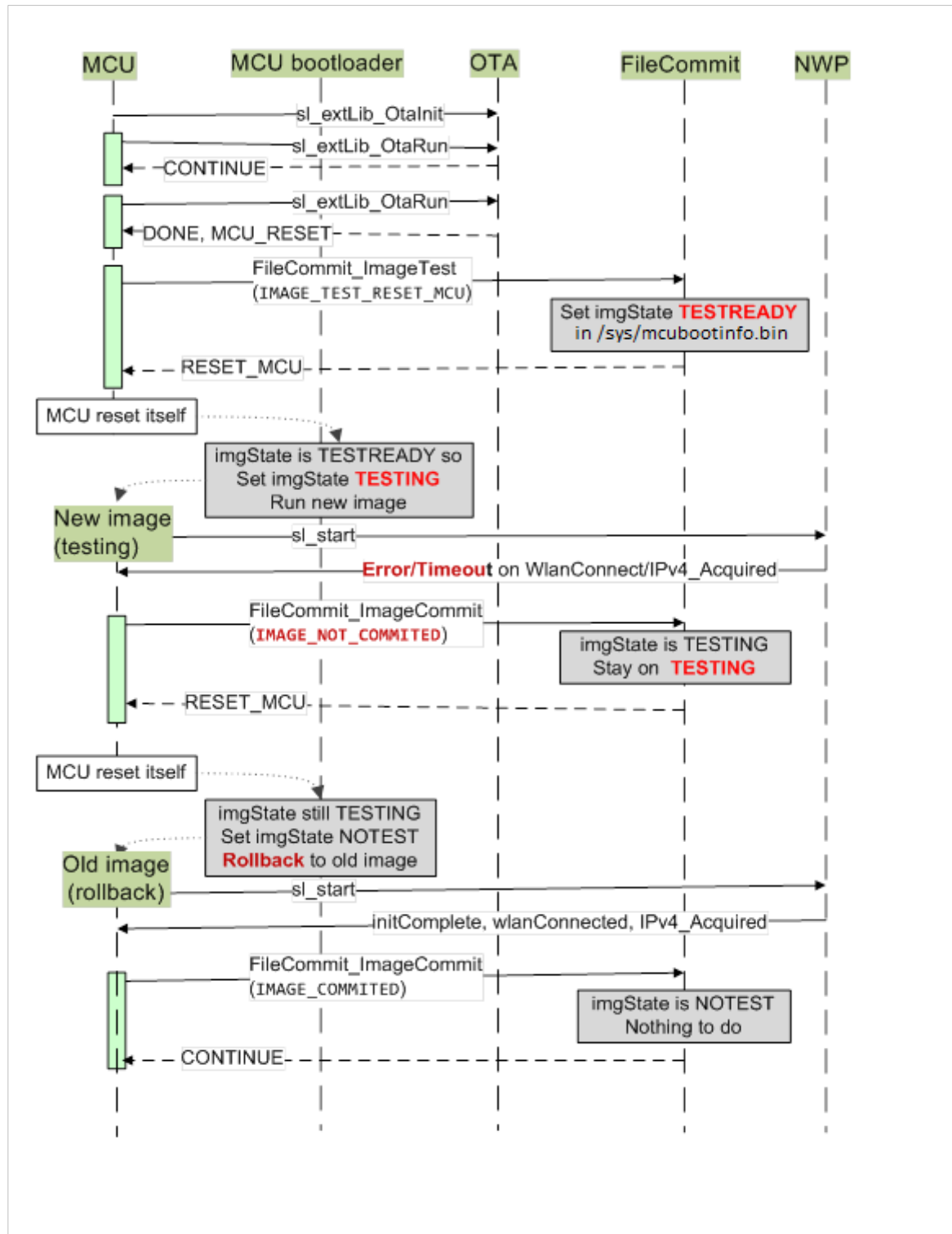
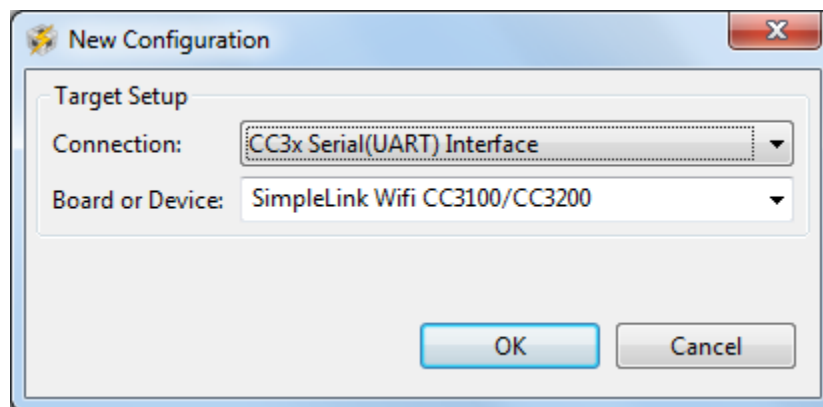


Figure 14 : OTA and MCU Commit sequence – rollback

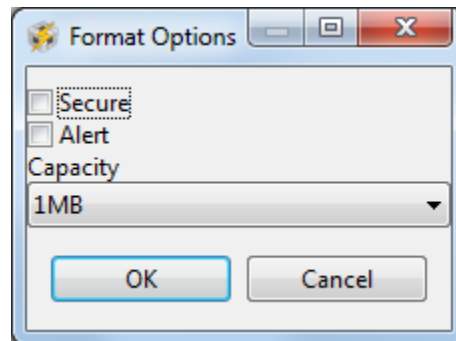
8.2 Generic Guidelines for Enabling OTA on CC3100/CC3200

This section list down the recommended order of flashing during production, using UniFlash tool, to enable OTA on CC3100/CC3200 device(s):

1. Open UniFlash tool for CC3200/CC3100.
2. Go to File->New Configuration and press ok with following option



3. Format the storage with "Secure" and "Alert" unchecked.



4. Update the service pack.
5. Create a new file in the sessions file list using "Add File" option and rename it to "/sys/mcuimg1.bin".
6. Create a new file in the sessions file list using "Add File" option and rename it to "/sys/mcubootinfo.bin".

7. Set the following options for different files in the session

Table 3 : UniFlash File Options

File	URL	Options
/sys/mcuimg.bin	Application bootloader	Erase, Update, Verify
/sys/mcuimg1.bin	1 st OTA Image	Erase, Update, Verify
/sys/mcubootinfo.bin	-	Erase

Note: For “/sys/mcuimg.bin”, “/sys/mcuimg1.bin” or any other user file set the “MaxSize” to maximum size that will ever be required. OTA will fail to download any image with the same name and size greater than its max size set during first time creation via UniFlash.

8. Save the configuration file
9. Press “Program” to program the file onto the device storage.

Refer to

[http://processors.wiki.ti.com/index.php/CC31xx %26 CC32xx UniFlash Quick Start Guide](http://processors.wiki.ti.com/index.php/CC31xx_%26_CC32xx_UniFlash_Quick_Start_Guide) for complete UniFlash quick start guide.

8.3 Enabling 'FAST BOOT' option in Application Bootloader and OTA library

The OTA scheme described in this document uses a two-stage booting. At the first stage the ROM bootloader loads the 'application bootloader'. The second stage starts with application bootloader identifying the user application image to boot using boot info file and finally loading and executing the image.

The 'FAST BOOT' option reduces the overall loading time of the application image marked as 'ACTIVE' (*refer to section "3.2.4 Application Image Selection and Rollback Logic"*).

The trade-off here is the application bootloader run time size which increases from 16KB to 32KB when 'FAST BOOT' is enabled. To work with this application bootloader the user (or OTA) application needs to be re-compiled and linked to run from **0x2000_8000**.

Note:

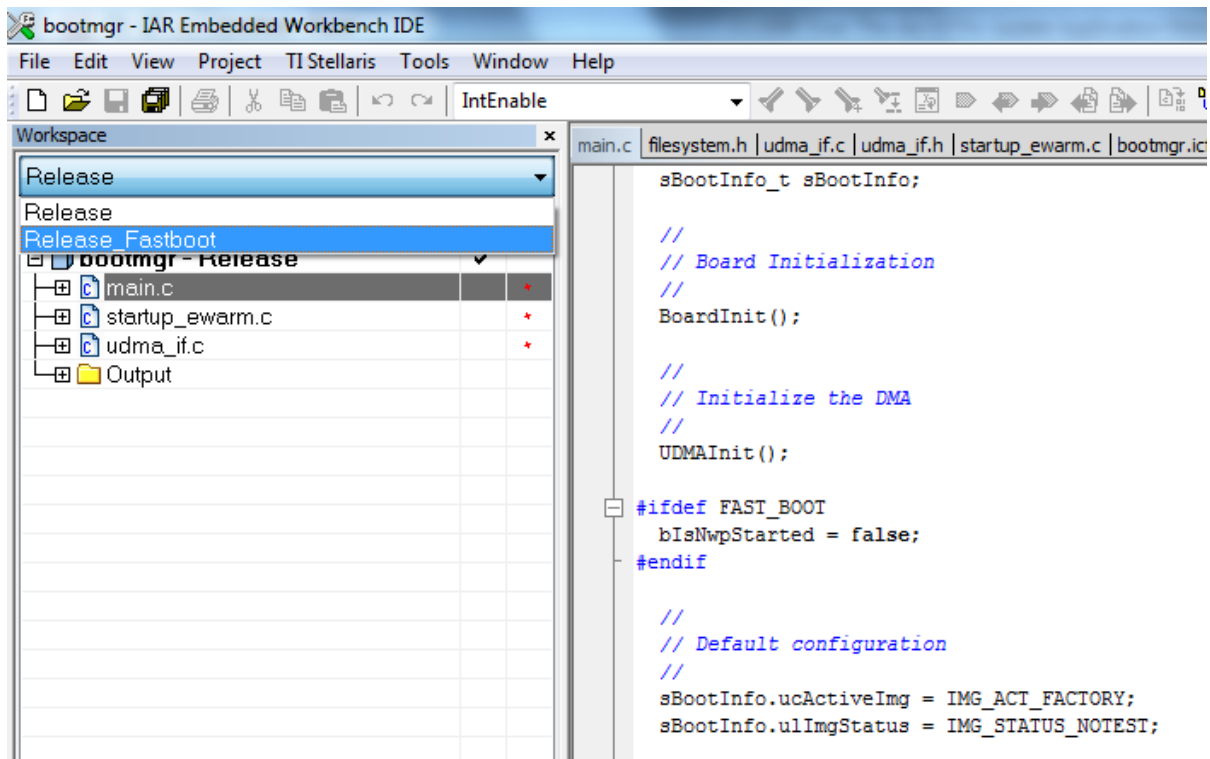
1. With this configuration application does not support the **Factor default image**.
2. The 32-KB of memory used by application bootloader can still be used by the user application for read-write data section.
3. This option only improves the loading time of image when **OTAU_MCU_IMAGE_STATE** equals **IMG_STATUS_NOTEST**. Otherwise, the loading time remains unchanged.

For enabling FAST BOOT:

1. Generate 'fast boot' application bootloader.
2. Compile the 'flc.a' library for 'FAST BOOT'
3. Compile and link the application to the new flc.a and to run from **0x2000_8000**

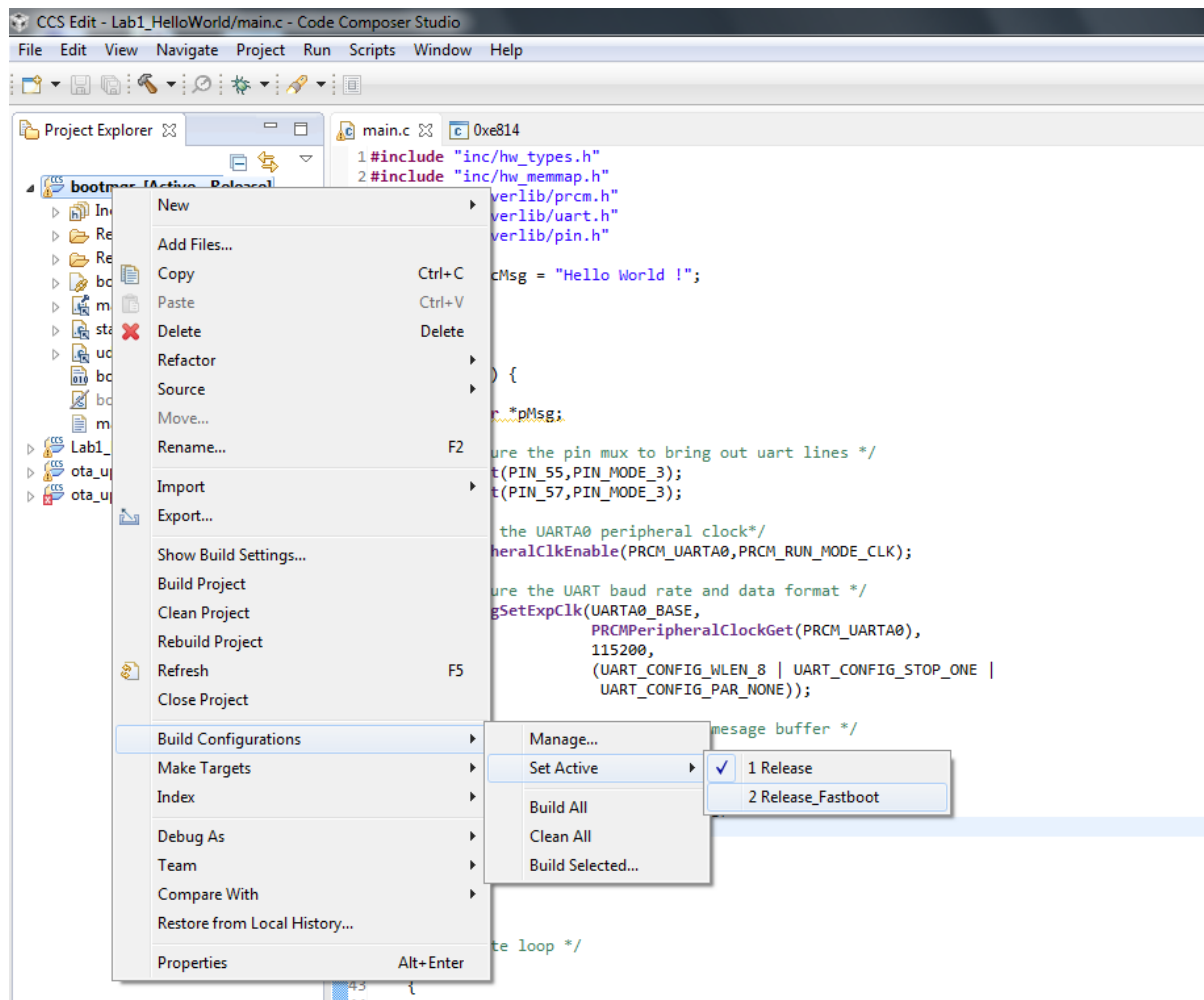
8.3.1 Generating 'FAST BOOT' Application Bootloader in IAR

- Open the application bootloader project in IAR
- From the workspace window, choose 'Release_Fastboot' project configuration as shown below
- Run a 'rebuild'
- Output : application_bootloader.bin "cc3200-sdk\example\application_bootloader\ewarm\Release_Fastboot\Exe"



8.3.2 Generating 'FAST BOOT' Application Bootloader in CCS

- Import the application bootloader project in CCS workspace
- Right click the project in Project Explorer to get the menu as shown below
- Set 'Release_Fastboot' as active build configuration.
- Run a 'rebuild'
- Output : application_bootloader.bin is generated under "cc3200-sdk\example\application_bootloader\ccs\Release_Fastboot"



8.3.3 Generating 'FAST BOOT' Application Bootloader in GCC

- In Cygwin, switch to 'cc3200-sdk\example\application_bootloader\gcc' directory
- Run "make -f Makefile_fastboot"
- Output : application_bootloader.bin is generated under "cc3200-sdk\example\application_bootloader\gcc\exe_fastboot"

8.3.4 Generating 'FAST BOOT' flc.a library

The steps for generating flc.a library is same as the for application bootloader. The flc .a IARorCCS project file/ GCC Makefile can be found under the directoy "cc3200-sdk\simplelink_extlib\flc" in the SDK installation.

8.3.5 Re-Compiling the application in CCS

- Import the 'ota_update_nonos' project into CCS workspace
- Expand the project to see the list of files
- Open ti_rtos\ti_rtos_config\app.cfg and change as below:
 Hwi.vectorTableAddress = **0x20008000**;
 Hwi.resetVectorAddress = **0x20008000**;
- Locate and open 'cc3200v1p32.cmd' file
- Do the following modification.
- Do a rebuild

```
#define RAM_BASE 0x20008000

/* System memory map */

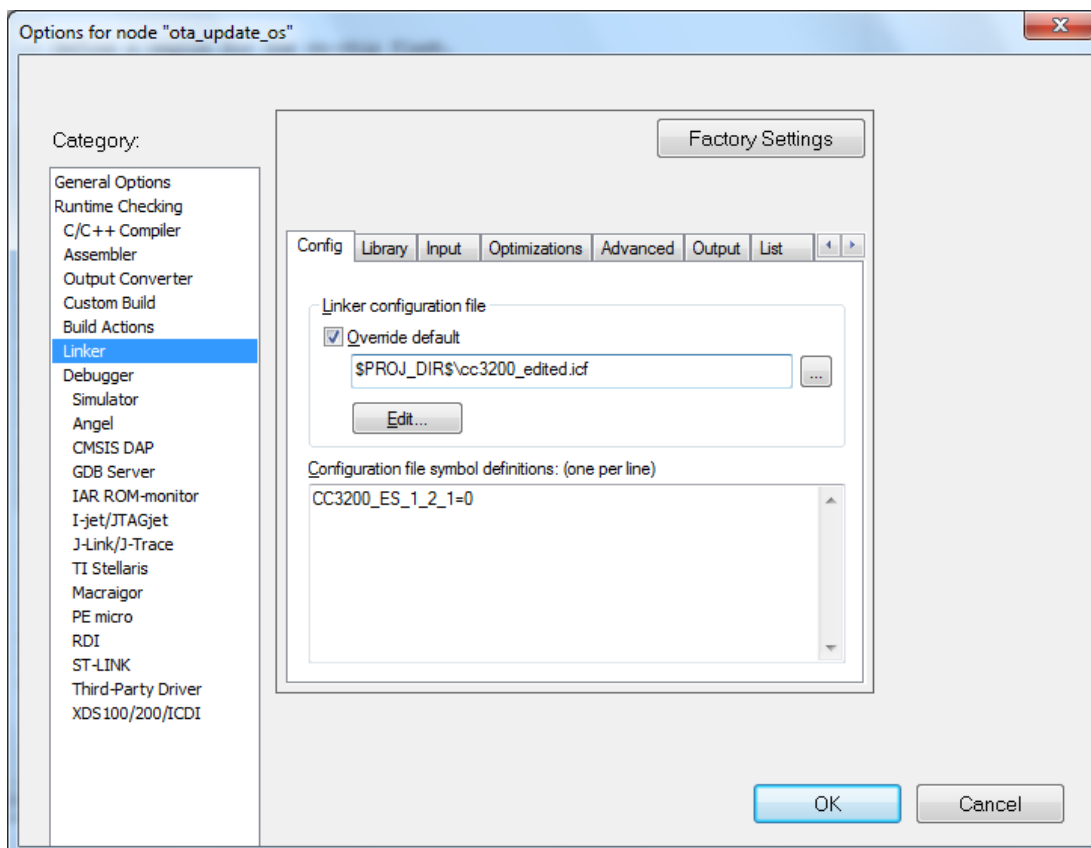
MEMORY
{
    /* Application uses internal RAM for program and data */
    SRAM_CODE (RWX) : origin = 0x20008000, length = 0x13000
    SRAM_DATA (RWX) : origin = 0x2001B000, length = 0x11000
}
```


8.3.6 Re-Compiling the application in IAR

- Open 'ota_update_nonos' project into IAR workspace
- Create a copy of the default linker file. Typical path "<IAR installation root>\arm\config\linker\TexasInstruments\cc3200.icf"
- Edit the linker script as shown below

```
define region SRAM = mem:[from 0x20000000 to 0x2002FFFF];
}
else
{
define region SRAM = mem:[from 0x20008000 to 0x2002FFFF];
}
```

- Open project option window, under linker, override the default 'linker configuration file' to point to the newly edited linker script.(shown below)
- Do a 'Rebuild All'



8.3.7 Re-Compiling the application in GCC

- Open linker script ota_update_nonos.ld from ota_update_nonos/gcc
- Modify as shown below.
- Rebuild the application.

```
MEMORY
{
    SRAM (rwx) : ORIGIN = 0x20008000, LENGTH = 0x00030000
}
```

8.3.8 Flashing Images

With ‘FAST BOOT’ option enabled, the application bootloader uses a different set of file name. The following table lists the changes

File (w/o FAST BOOT)	File (w/ FAST BOOT)	URL	Options
/sys/mcuimg.bin	/sys/mcutst.bin	Application bootloader	Erase, Update, Verify
/sys/mcuimg1.bin	/sys/mcuimg.bin	1 st OTA Image	Erase, Update, Verify
/sys/mcubootinfo.bin	/sys/mcureserved.bin	-	Erase