Software Design Specification

for

Connection Manager

Version 0.7



25-February-2012

# Table of Contents

Table of Contents ii

Revision History iv

Approved By iv

1. Introduction 1

1.1. Purpose 1

1.2. Scope 1

2. References 1

3. Definitions and Acronyms 1

5. Design Considerations 2

5.1. Assumptions 2

5.2. Constraints 2

5.3. System Environment 2

5.4. Risks and Volatile Areas 2

6. Architecture 2

6.1. Overview 2

6.2. CM Client 3

6.3. CM Server 3

7. High Level Design 3

7.1. Client 5

7.2. Server 5

7.3. Client Server Interaction (Socket communication) 5

7.4. CM Configuration 5

7.5. Device Abstraction Layer 6

8. Low Level Design 6

8.1. Relevant API and AT Commands 6

8.2. Relevant Events that the CM tracks 8

8.3. Network Search 10

8.4. Connect command 10

8.5. Disconnect command 12

8.6. CM Client Server communication 13

8.6.1. Client functions 13

8.6.2. Server functions 14

8.7. Statistics Update 16

8.8. Configuration File 16

8.9. Network Configuration 17

8.9.1. IP Stack Configuration 18

8.9.2. IP Address Configuration 18

8.9.3. IPv6 Address Configuration 19

8.10. Connection Status Indication 19

8.11. CM State Machine 19

8.11.1. Connecting State 21

9. User Interface Design 22

Appendix A: Configuration file 24

Appendix B: Client Server Interface Definitions 24

Appendix C: Abstraction Layers 25

Device Abstraction Layer 25

OS Abstraction Layer 26

Appendix D: IP Address Assignment 27

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Name** | **Reason For Changes** | **Date** |
| *0.1* | *Sathyanarayanan S*  *Vasuki Setlur* | *Initial Revision* | *20/12/2010* |
| *0.2* | *Sathyanarayanan S* | *Review comments from Broadcom incorporated* | *13/01/2011* |
| *0.3* | *Sathyanarayanan S* | *Review comments from Soma incorporated* | *28/01/2011* |
| *0.4* | *Vasuki Setlur* | *IPv6 configuration section added* | *12/02/2011* |
| *0.5* | *Vasuki Setlur* | *IP Configuration flowchart updated* | *01/03/2011* |
| *0.6* | *Sathyanarayanan S* | *Updated as per the latest Design, Device discovery section added* | *01/07/2011* |

# Approved By

Approvals should be obtained for project manager, and all developers working on the project.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Signature** | **Department** | **Date** |
|  |  |  |  |
|  |  |  |  |

# Introduction

* 1. Purpose

This document specifies the Design for the Connection Manager (CM) module for the BCS5xx chipset based products.

* 1. Scope

Connection Manager (CM) Connects to, Disconnects from and provides the status of the LTE Network Connections available to a device. The scope of this project is limited to building a Connection Manager for the Windows platform (XP, Vista and 7).

# References

1. BC4-ENG-SW-TL-BLRxyz-BCS500-CM-spec-v0\_1 and BC4-ENG-SW-TL-XYZ-BCS500-CM-spec-v1p1\_review\_111010 – Requirement Specification Document from Beceem Communications Inc.
2. Software Requirement Specification Document from UMA – SRS\_Connection\_Manager v0.2
3. UConnect-LTE Proposal for Beceem v0.10 – Proposal document from UMA for Connection Manager
4. Common\_API\_Reference.doc - Common API Functions for BCS500

# Definitions and Acronyms

APN Access Point Name

CHAP Challenge Handshake Authentication Protocol

IMEI International Mobile Equipment Identity

IMSI International Mobile Subscriber Identity

LCS Location Services

LTE Long Term Evolution

PAP Password Authentication Protocol

PDP Packet Data Protocol

PLMN Public Land Mobile Network

PIN Personal Identity Number

PUK Personal Unblocking Key

RAT Radio Access Technology

SMS Short Messaging Service

UICC Universal Integrated Circuit Card

USIM Universal Subscriber Identity Module

# Design Considerations

The CM would perform the following operations:

1. Connect to a Network
2. Disconnect from a Network
3. Display the Status of the Connection(s)
4. Detect a USB LTE modem
5. Network Interface Configuration (IP Address, Subnet mask, Gateway Information etc.)
6. Configuration setting (Auto, Manual modes)
   1. Assumptions
7. The CM would depend on modem drivers and APIs which would be provided by Beceem.
8. The coding for the CM would be done in the standard C/C++ language.
9. No open source modules would be used during the development of the CM.
   1. Constraints

None identified so far.

* 1. System Environment

CM would be built for the Windows platform. The development environment would be Microsoft Visual Studio (Version 8.0). The CM code would be made GNU compatible, so that it could be used in a Unix/Linux environment in the future. Since CM would interact with the hardware of a device to extract Network related information, OS specific calls would be a part of the implementation. To abstract these OS calls, an OS Abstraction Layer would be designed, which would make sure the CM is portable to other platforms without much overhead.

* 1. Risks and Volatile Areas

1. Availability of the Modem specific APIs to be called within the CM.
2. The CM design is explained assuming the OS Hosted mode. However, the CM should work in the Host-less mode, with minimum changes.

# Architecture

* 1. Overview

On a Desktop/Laptop, we would encounter a situation where we would have to connect to available Wireless networks in the vicinity of our presence. Many applications today need to connect to the web, to be able to download/upload data. CM provides the user and these applications with a fast and transparent way of making connection choices. The CM we intend to design and develop would cater to the LTE class of networks. At a very top-level, the CM would have:

1. A Client, with a user interface to issue commands to the CM
2. A Server, which would perform the network related tasks of the CM
   1. CM Client

When a user or an application requires connection to a network, there would be a need to search the available networks and then connect to the preferred one. This necessitates the presentation of a few options to the user. The client provides the user with an interface, through which a command could be issued. Some of those commands would be:

1. Network Search (Auto/Manual modes)
2. Connect
3. Disconnect
4. Connection Status Information

The Client presents these options to the user via a Command Line (CLI) or a Graphical User Interface (GUI). The client, on receiving the command from the user contacts the CM Server to perform the specified task through the socket communication protocol.

* 1. CM Server

The Client only receives the operation to be performed. But it has no interaction with the Modem. A CM Server is therefore required, which would interact with the underlying modem. It responds to the commands from the Client. It also notifies the Client whenever a connection is established, broken or presents a list of networks available. In order for the server to be able do the specified tasks, it needs to be exposed to the Driver level APIs of the modem.

The interaction between the client and server as mentioned in the section above is through the socket communication protocol.

# High Level Design

As described in section 5 above, the CM would have a Client and a Server, which would perform a set of activities to connect to/disconnect from a network. There are other blocks which make up the CM, as shown in the picture below and described in the sections below:



* 1. Client

The CM Client abstracts all the underlying interactions with the modem and presents the user with a simple, intuitive interface. The interface helps him key in the commands that are required to connect to/disconnect/get status from the network. If the interface is a GUI, buttons would be presented to the user, which would help him make a choice of the operation to be performed. In the case of a CLI, the user is presented a list of commands on keying a command like “help”. Keying in the suitable command helps him perform the operation that he intends from the CM. The initial release of the CM would be CLI and the further versions would be GUI based. Adobe AIR or QT would be the choices for the GUI development.

* 1. Server

As has been mentioned in section 5.3 above, the CM Server interacts with the underlying modem to perform the task (command) chosen by the user. To do this, the APIs of the underlying modem are exposed to the CM Server. If say a Connect operation to a particular network is intended, the server uses the “net\_attach” API from the Beceem common APIs to perform the operation. Other common APIs such as “net\_detach”, “net\_search” etc., also are used based on the type of command received from the client. The server also monitors various events of the modem and notifies the client of the same. A more detailed explanation of the common APIs and the events is given in the Low Level Design section of this document.

* 1. Client Server Interaction (Socket communication)

The CM Client and Server interact with each other via the Socket mechanism. The interaction between the server and the client is bi-directional, i.e., the client informs the server of the operation to be performed. The server performs the client requested operation and reports the result of the same back to the client. The change in status of any event too is communicated back to the client, so that it can display the same to the user. The server waits for a bind from the client, before performing any operation. In addition to this, the Server would receive Unsolicited events from the LTE Stack (Net attach, Net detach events), which would be sent back to the Client. The GUI Client would use these events to notify the user of the Events from the LTE Stack.

* 1. CM Configuration

The CM can connect to a PLMN (Public Land Mobile Network) automatically on startup or it can also wait for the user to select from a list of available PLMN ids. Based on this, the CM can have two Configurations, the **Auto** mode (automatic connection to a PLMN) and the **Manual** mode (PLMN selection by the user). When the CM is started, it would be ideal if the information regarding the previous configuration is available. Thus, the configuration options are stored in a file or in non-volatile memory. This information is then read by the CM on startup and any changes to the configuration are stored back on closing the CM session. Details about the Configuration file are given in 8.7.

* 1. Device Abstraction Layer

This software layer abstracts the Device driver API calls, so that the CM is operable with modems other than the BCS5xx series. A more detailed explanation is available in Appendix C.

# Low Level Design

This section presents a more detailed explanation of the various tasks that the CM handles. The API calls used in this section are Beceem specific. For ease of reference, the APIs and events that the CM commands would use are given below. A more detailed explanation of the same can be had from the *Common\_API\_Reference.doc - Common API Functions for BCS500* document from Beceem.

* 1. Relevant API and AT Commands

|  |  |  |  |
| --- | --- | --- | --- |
| API | Description | Associated AT commands | CM Action |
| equip\_config\_get | Retrieve the current functionality of the device (ON/OFF status) | CFUN | Called from CM Server to find if the modem is switched off or fully functional |
| equip\_config\_set | Configure various equipment parameters | CFUN | Called from CM Server to switch ON/OFF the modem |
| equip\_info\_get | Retrieve permanent equipment information and equipment status (battery status, manufacturer ID, IMEI, Power Mode) | CBC, CGSN, ATI, CGMI, CGMM, CFUN | CM Server calls this API to update its structure with various equipment related parameters |
| net\_search | Retrieve list of available PLMNs | AT, COPS | Called when user initiates a Network search. A time out value is given as the input parameter. Returns the status of posting the command to the modem. Results sent in the callback function. |
| net\_attach | Attempt to attach to a PLMN and register for EPS and non-EPS services | COPS, CGATT | Called when user issues a Connect command in the Manual mode. Happens implicitly in the Auto mode. PLMN Select Mode, PLMN ID (Manual mode), RAT (Manual mode) and Primary context ID are the input parameters. EVT\_LTE\_REG\_STATUS indicates the status of this call. |
| net\_cfg\_get | Retrieve network related parameters (Activated RATs and mobile operating class) | CGCLASS, CEMODE, WS46 | CM Server calls this API to retrieve network configuration information in the form of parameters activated RAT, Operation class (2G/3G/4G), PDP Type (IPv4/v6) |
| net\_cfg\_set | Configure network related parameters | CGCLASS, CEMODE, WS46 | CM Server calls this API to configure network information with parameters Operation class and PDP type. This API has to be called before the call to net\_attach. |
| net\_detach | Detach from connected RAT(s) | COPS, CGATT | Called when the user issues a Disconnect command. This API also relieves the primary and secondary EPS bearer contexts. |
| net\_status\_get | Obtain registration information from specified RAT | CEREG, CGREG, CREG | Called when the CM needs to know of the registration status for a particular RAT, which is given as a parameter to this API. |
| net\_sig\_quality\_get | RSSI and BER of specified RAT | CSQ | User would request the RSSI and BER for a specified RAT and this would require a call to this API. |
| ps\_dfn\_info\_get | Retrieve status information of all defined PS contexts | CGDCONT, CGSDCONT | Server calls this API to retrieve status of all defined PS contexts. A T\_PS\_INFO structure with the contexts is returned in the callback function. |
| ps\_ctx\_define\_primary | Create a primary PDP or EPS default bearer context | CGDCONT | Taking the context id, PDP type, APN, IP Address and compression parameters, the CM Server calls this API to define a primary EPS bearer context for a PLMN. This API is called before the net\_attach call. |
| ps\_ctx\_define\_secondary | Create a secondary PDP or EPS dedicated EPS bearer context | CGSDCONT | Taking the context id, PDP type, APN, IP Address and compression parameters, the CM Server calls this API to define a secondary EPS bearer context for a PLMN. |
| ps\_ctx\_undefine\_primary | Delete a previously defined primary context | CGDCONT | Takes the context id to destroy a Primary EPS bearer context. This is one of the key steps to perform net\_detach. |
| ps\_ctx\_undefine\_secondary | Delete a previously defined secondary context | CGSDCONT | Takes the context id to destroy a Secondary EPS bearer context. |
| ps\_ctx\_cfg\_get | Retrieve parameter settings specific to a PS context | CGQREQ, CGQMIN, CGERQREQ, CGEQMIN, CGEQOS, CGTFT | Called when the CM Server needs to know the settings specific to a PS context. Parameters for this API are the context ID and the PS context parameter that the CM server is looking for. |
| ps\_ctx\_cfg\_set | Set parameter setting specific to a PS context | CGQREQ, CGQMIN, CGERQREQ, CGEQMIN, CGEQOS, CGTFT | Called by the CM Server to set parameters specific to a PS context. Context ID and the parameter to be set are the inputs to this API. |
| ps\_ctx\_info\_get | Retrieve context specific status information | CGDCONTRDP, CGEQNEG, CGEQOSRDP, CGPADDR, CGTFRDP | CM Server calls this API to retrieve status on context specified information like the QoS of the EPS bearer contexts for 2G/3G/4G, number of bytes sent and received since the creation of the context etc. Context ID and the parameter whose status is required are the inputs to this API. |
| ps\_ctx\_activate | Activate a previously defined PS context | CGACT | Takes a context ID and presents it for activation. CM Server calls this in the manual mode before a net\_attach. The status of this call is received by a call to ps\_ctx\_info\_get. |
| ps\_ctx\_modify | Modify a previously activated PS context | CGCMOD | CM Server calls this when a PS context already defined has to be modified. Ps\_ctx\_info\_get retrieves the status of this call. |
| ps\_ctx\_deactivate | Deactivate a PS context | CGACT | CM Server presents a context ID to this API to deactivate a PS context. |
| prop\_APN\_table\_set | Configure APN table (for the Beceem common driver) | None | CM Server calls this API to set the APN table. This information for this API is available from the Configuration file on startup. |
| prop\_APN\_table\_get | Retrieve current APN table configuration | None | On CM exit, the server calls this API and stores the information in the configuration file for reading on the next startup. |

* 1. Relevant Events that the CM tracks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Event | Description | Associated AT commands | Type | CM Action |
| EVT\_RSSI\_2G3G | Event signaling change in RSSI from 2G/3G RAT | CSQ | unicast | Intimates the user about the change in RSSI from 2G/3G RAT |
| EVT\_RSSI\_4G | Event signaling change in RSSI from LTE RAT | CSQ | unicast | Intimates the user about the change in RSSI from LTE RAT |
| EVT\_CS\_REG\_STATUS | Event signaling change in registration status for the CS domain | CREG | broadcast | Indicates to the user that the registration status has changed in the CS domain (T\_REG\_INFO) |
| EVT\_UMTS\_REG\_STATUS | Event signaling change in registration status for the UMTS domain | CGREG | broadcast | Indicates to the user that the registration status has changed in the UMTS domain (T\_REG\_INFO) |
| EVT\_LTE\_REG\_STATUS | Event signaling change in registration status for the LTE domain | CEREG | broadcast | Indicates to the user about a change in the LTE Registration status (T\_REG\_INFO) |
| EVT\_ROAMING\_IND | Event signaling change in roaming status | CREG, CGREG, CEREG | broadcast | Updates the roaming status in the CM Configuration  (T\_ROAM\_INFO) |
| EVT\_PRIMARY\_ACT | Event signaling activation of a primary PDP or default EPS bearer context | CGEREP, CGEV | broadcast | Moves ahead to perform net\_attach  (T\_EVT\_PRIMARY) |
| EVT\_SECONDARY\_ACT | Event signaling activation of a secondary PDP or dedicated EPS bearer context | CGEREP, CGEV | broadcast | Indicates the secondary context activation to the user (T\_EVT\_SECONDARY) |
| EVT\_PRIMARY\_DEACT | Event signaling deactivation of a primary PDP or default EPS bearer context | CGEREP, CGEV | broadcast | Indicates the success of Disconnect to the user (T\_EVT\_PRIMARY\_DEACT) |
| EVT\_SECONDARY\_DEACT | Event signaling deactivation of a secondary PDP or dedicated EPS bearer context | CGEREP, CGEV | broadcast | Indicates the  secondary context deactivation to the user (T\_EVT\_SECONDARY\_DEACT) |

* 1. Network Search

The CM operates in two modes: Auto and Manual. In Manual Mode, the modem returns a list of available PLMNs which are displayed by CM. The user then selects any one of the displayed PLMNs. In auto mode, the modem picks up the order of preference for a PLMN from the SIM Card and connects to the one that is found to be the most suitable in the list. The configuration mode chosen by the user (Auto/Manual) is updated in the CM Configuration file.

* 1. Connect command

The Connect command is the user’s input to the CM to indicate connection to a network. A flow of the steps that take place during a Connect command is as follows:

* CM initializes and configures the IP stack capability version using “net\_cfg\_set” CAPI and waits for commands from the user
* CM receives “Connect” command
* If the mode is “Manual”, CM returns a list of the available PLMN ids (results obtained from the modem) and the user selects one on these PLMNs
* It the mode is “Auto”, the LTE stack or the RAT stack selects the appropriate PLMN id from the USIM and connects to it
* The Primary context is defined by calling “ps\_ctx\_define\_primary”
* PCO is configured (if needed) by calling “ps\_ctx\_cfg\_set”
* “net\_attach” API is called by the CM Server (described in section 8.1)
* The PDP context/EPS bearer is activated for Data transmission (PDP context for 2G/3G and EPS bearer for LTE)
* The User Equipment (UE) gets the IP address(es), via NAS on the device and other Network Configuration Information, which are allocated by the network
* “ps\_ctx\_info\_get” is called from the CM to obtain the Context specific information
* CM configures the Host Network Interface and ensures the required information for traffic classification is provided

On a “net\_attach”, there can be a few failures, which have to be returned to the user as described by the 3GPP NAS document. These error messages are reported by the LTE stack and the CM has to present these to the user. A few of those errors are:

* IMSI Invalid (Internation Mobile Subscribe Id)
* IMEI invalid
* PLMN is “forbidden” in USIM
* LAI is “forbidden” in USIM

A flow of the Connect command would look as below:



* 1. Disconnect command

The user is given the option to disconnect from a network that he is connected to. Unlike in the case of the Connect command, the network can detach itself from the connection. The flow of the steps during a Disconnect command as described below:

* When the user issues the Disconnect command, the CM Client receives it and passes it on to the CM Server, which in turn calls the “net\_detach” Beceem API
* When the Network initiates the disconnect, the CM is informed through a event about the status change (EVT-\*\*\*-REG-STATUS)
* On receiving the Disconnect command from the user, the PDP context/ EPS bearer for that particular connection is de-activated implicitly by the modem. De-activating the secondary context only breaks that context. However, de-activating a primary context breaks all the secondary contexts that have been activated. The events EVT\_PRIMARY\_DEACT and EVT\_SECONDARY\_DEACT inform the deactivation of the contexts.

The flowchart of the Disconnect command from the UE perspective appears as below:



* 1. CM Client Server communication

The Client interacts with the User and the Server communicates with the Beceem device. These two modules in-turn have to interact with each other to execute the commands from the User. This happens via the Socket mechanism. A few salient points about this communication are as below:

* The Server creates a socket with a known IP (or loopback IP) and Port number during initialization and waits for the Client to associate (connect). The communication process begins with the Client connecting to the Server
* The Server and the Client have dedicated threads to read data on their ports. The Client/ Server will encapsulate user command parameters/ responses in a common structure. The Client will be light and will communicate with the Server to retrieve any information
* Data and messages exchanged between the two are as per the format mentioned in Appendix B (Client Server Interface Definitions)
* All requests from the Client have a unique TokenID associated with them. Responses from the server would have the same TokenID, for the corresponding Requests.
* The Client interacts with Server on a request-response basis. Requests that are sent with the requestResponse parameter set to ‘0’, receive no responses from the server, i.e., request only messages. Requests which have requestResponse set to ‘1’, would receive a response with the matching TokenID
* The Server interacts with the Client by means of Notification messages, which are a result of the Event changes in the modem associated with the CM. These are Unsolicited/Asynchronous messages and would not have a TokenID

These event changes are polled by the client at regular intervals, and the end user will be intimated of any changes.

As part of the Client Server Communication protocol, the following methods would be implemented:

* + 1. Client functions

1. **Client Request Message:**

void cmClientRequest(int requestId, void \*requestData, int dataLen, int TokenId, int returnResponse);

1. **Client Response Message:**

void cmClientResponse(int TokenId, void \*responseData, int dataLen, int error);

1. **Client Unsolicited Response Message:**

void cmUnsolResponse(int responseID, void responseData, int dataLen);

* + 1. Server functions

1. **Server Session Creation:** A working session is created after the socket connection is established with a client. All memories and resources should be allocated in this function and used throughout the life of the socket connection. After the session is created the server thread is ready to accept client’s request commands, which includes AT Commands and private commands.

Int CreateSession(int sessionIndex);

1. **Server Session Deletion:** Remove the existing CreateSession. It should delete all the memories and resources that have been allocated for this session. It should also terminate the socket session.

Int DeleteSession(int sessionIndex);

1. **Server Parse AT Commands:** AT command requests are sent by the client and get parsed in ParseATCommandFromClient function. The output of the parse is interpreted and appropriate Common APIs are called and the result is stored in a known memory location so that it can be returned to the client in client response request.

Int ParseATCommandFromClient(uint commandID, void \*pParam)

This function should call the AT APIs defined in Common API reference documentation. The function should open the configuration port, COM port, and send the request to the API Router and extract results from the same port.

1. **Server Process Client Command:**  after the socket read thread receives a data buffer this function should be called to parse the data and determine if this is AT commands then calls ParseATCommandFromClient or if it is other request from the client then it should call appropriate function.

Int ProcessClientCommand(int genericCommand, void \*pParam);

The Client should send a response command following a client request command. Out of order commands should be treated as error in ErrorHanler function.

1. **Server Get Network Interface Information:** this function obtain IP addresses, APN Ids, EP identifiers, and interface ID through the configuration port.

Int GetNetworkInterfaceInformation(networkInfo \*pNetInfo);

IP address information is then used to program the interface based on USB EP identifier.

1. **Server Program IP Address To Interface:** program an IP address to a network interface. The interface needs to be reset afterward for the IP to go into effect by calling DisableAdapter and then EnableAdapter.

Int SetIPAddress2Interface(int interfaceID);

1. **Server Enable Adapter:** enable an adapter. After the IP address is programmed the network interface the network stack needs to be reset, by disable and then re-enable the adapter.

Int EnableAdapter(char \*pAdapterName);

1. **Server Disable Adapter:** disable an adapter.

Int DisableAdapter(char \*pAdapterName);

1. **Server Download Firmware (optional):** to download firmware to the device and check the download result.

Int DownloadFirmware(char \*filename);

1. **Open Device:** open a device driver to communicate and control the hardware. This includes COM port and network interface.

Int OpenDevice(char \*driverDosName);

1. **Close Device:** Close the device driver.

Int CloseDevice(int driverHandle);

1. **Server Get System Software Information:** return firmware version, driver version and configuration versions.

Int GetSystemSoftwareInformation();

1. **Enable Logging:** selectively turn on logging using filter index.

Int EnableLogging(int filterIndex);

1. **Disable Logging:** turn off certain logging using filter index.

Int DisableLogging(int filterIndex);

1. **Server Parse Argument:** parse the input command line and interpret its argument for the server executable.

Int ParseArgument(int argc, char \*\*argv);

1. **Server Error Handle:** returns an error code that is shared by both client and server program

Int ErrorHandler(int errorCode);

This function should copy the error code to the known location and return it with the next client response request.

* 1. Statistics Update

The user would want to enquire the CM Server for various statistics and display the same. In the CLI client, the user would use a command to fetch the statistic of interest and in the GUI client, a window would display all the statistics. In order for the client to have the latest Statistics values, the CM Server fetches the same at a fixed interval and stores them in a data structure. On enquiry from the client, the latest available values in this data structure are sent.

* 1. Configuration File

On startup, the CM has to have the details of the previous connection it made before exiting. These details would include the configuration mode (Auto/Manual), Logging level, RATs to search for, Roaming Status etc. Appendix A has the details of the configuration file. When the CM restarts, the information from this configuration file would be read, to restore the configuration. When a crash occurs in the CM or the modem, and the CM is restarted after this crash, the last saved configuration before the crash, is restored. The same configuration information can also be stored into Non-volatile memory (depending on the whether the environment is CPE or a USB dongle).

The CM would call an API “Retrieve\_CM\_Info”, which would recover the information from the previous CM session. To save the current configuration information into Non-volatile (NV) memory or a file, an API “Save\_CM\_Info” would be called. The Save\_CM\_Info in turn calls the “prop\_APN\_table\_get” common API during the exit, to get the current APN table configuration and this information is stored onto the NV memory or the configuration file. The saving of the configuration may not take place in the case of a CM/Modem crash. The Retrieve\_CM\_info calls “prop\_APN\_table\_set” to set the APN table as per the information that was stored prior to an exit. This activity is done only after bearer activation is completed on a Connect command.

The CM records the information about the following in the Configuration file:

1. Configuration mode (Auto/Manual) – stored when the user makes a choice of the configuration mode
2. Logging Enable and Level – the user’s choice of whether he wants Debug, Information or Error logs and if at all he requires the Client to log information
3. RATs to be searched – specifies the RATS that would be searched for (e.g. search only for LTE RATs)
4. Roaming Status – indicates if the device is in the roaming mode, updated by the CM and not by the user
5. VID/PID Information – The Connection Manager for LTE is expected to detect the insertion/presence of an USB LTE modem and use the same for IP Configuration. The detection happens by means of a unique VID/PID that is assigned to the device. The CM scans the Device list in the OS using appropriate APIs and tries to find a match for the VID/PID given in the configuration file. If found, it shall extract the Adapter Index/Adpater name. In Windows, the IP Helper and SetupDi class of calls could be used for this operation.
   1. Network Configuration

On receiving commands from the client, the server has to interact with the modem to get Network related information (IP Address, Subnet mask) and configure the interface. Once the modem is detected by the Host, the IP configuration would be done with the help of the Adapter Index/Name using an OS utility/application. In Windows systems, the “netsh.exe” utility would be used to do this.

There are up to 4 Access Point Names (APN) that the device can connect to. These are shown up as 4 different Networking Interfaces on the PC. LTE/3G/2G systems allow mechanism for the network to assign IP address and routing information through NAS (air interface signaling mechanisms). So, our CM implementation shall support both the DHCP option as well as NAS option. It is most likely that networks would NOT be using DHCP option so it is important to implement NAS based IP address acquisition from the very first implementation, which is configured in the PCO, prior to the bearer activation.

The CM server communicates with the Beceem Common Driver for interface configuration in the following scenarios:

* A generic signaling mechanism on APN additions, IPv6 interface identifier, IPv6 router prefix, IPv4 address, Subnet information, Network Adapter name, USB Endpoints.
* Linkup and Disconnect notifications for each APN
* IPv4 allocation through NAS message
* IPv4 allocation through DHCP
* IPv6 allocation through DHCPv6
* IPv6 allocation using the router prefix via RS/RA for the corresponding APN and OS generated interface id
* IPv6 allocation wherein interface identifier is through the NAS message and Prefix through RS/RA
  + 1. IP Stack Configuration

The TCP/IP Stack has different layers, which have to be configured when the connection to a PDN is to be achieved.  The following steps would be followed while doing so:

* Device exposes up to 4 network interfaces, can be configured for DHCP or NAS
* CM interacts with the NAS layer to configure the IP Stack capability (net\_cfg\_set), define the primary context (ps\_ctx\_define\_primary) and to define the PCO for the context (ps\_ctx\_cfg\_set).  In addition to this,
  + CM interacts with the IP stack to ensure that the link-local addresses are formed using the interface id provided in NAS messages (if applicable)
  + In the case of DHCP, the CM provides the client ID (possibly based on IMSI, or based on the user input)
    - If the operator requires, the CM will be required to configure the IP stack on the host to ensure that DHCP requests are not issued (use static IP address instead)
  + CM will configure the IP stack for selectively turning off the various sources of neighbor solicitation messages (as per the requirement of the operator and/or LTE)
    - In the case where the IP stack of the host does not provide such configuration, the CM will interact with one of the Beceem components to achieve the end-effect of suppressing the neighbor solicitation messages
  + CM will ensure that a default interface is realized, it should be set to DHCP and configured with either IPv4 or IPv6 depending on DHCP server addressing scheme, that is unless specially required, then the CM should use NAS’s IPv4 or IPv6 addressing scheme.
    1. IP Address Configuration
* After the EPS bearer activation, the “prop\_APN\_table\_set” API call would set the APN mapping table.  The IP Address configuration is done only after the default bearer activation.
  + Calling the api “**prop\_APN\_table\_get**” returns a table of the mappings
  + CM shall read 4 MACs through 4 network interfaces
  + CM shall read a table with pairing MAC1-IP1, MAC2-IP2, MAC3-IP3, MAC4-IP4 through any network interface for NAS case
  + CM shall record pairing MAC1-IP1, MAC2-IP2, MAC3-IP3, MAC4-IP4 to OS’s  registry for APN aware applications to later use
  + Pair MAC1-IP1 is always default for internet access.  Other pairs will need to be connected and provisioning by APN aware applications.  Each pair refers to a functional use according to its order.
* Or CM can configure network interfaces to use dynamic DHCP
* APN aware applications are expected to use the different network interfaces using their inbuilt logic for mapping the adapters. The CM is not expected to bridge the communication between APN aware application and the network adapter.
* During software installation each network interface is assigned a specific interface metric.  Internet access interface should have higher priority over the other interfaces.
* In the case where one USB-end point/network interface is shared by applications connection to different APNs, the CM will configure the inter-APN mapping logic in the device (this is not expected to be based on the netfilter hooks) and assign NAS IP addresses as static IP addresses to the exposed network interfaces, each IP is mapped to an APN in the device (“**prop\_APN\_table\_set**” is the API used here).  This scheme will not work with dynamic DHCP.
  + 1. IPv6 Address Configuration

All IPv6 address configuration on the host would be done with the help of the “netsh.exe” utility. The following steps would be performed before setting the interface to UP.

* After the establishment of the default EPS bearer context, the event “EVT\_PRIMARY\_ACT” is accompanied with the IP Stack parameters for this bearer. This information is received as the structure “T\_PS\_CTX\_INFO” mentioned in the Common API Reference document.
* The IP Address field in the received information would have the interface ID, which would be used to configure the link-local address of the interface.
* The Stateless auto-configuration option is enabled on the interface, to get a globally routable IP address.
* We would use Beceem APIs to turn off the DAD messages. If these APIs are not available, we would set the “maxdadattempts” value in the IPv6 privacy settings to 0 using netsh.
* The DNS would be configured based on the information extracted from the “EVT\_PRIMARY\_ACT” event.

**NOTE**: We would need Beceem APIs to turn off the DAD messages as not all Operating Systems provide APIs to disable the same.

For the non-Verizon version of the CM, if the PDP/IP address is NULL, then we would enable the DHCP option on the interface.

* 1. Connection Status Indication

Section 8.2 summarizes the different events the CM tracks. The EVT\_LTE\_REG\_STATUS event notifies the state of the registration of an LTE connection, after a net attach is issued. The EVT\_PRIMARY\_ACT and EVT\_PRIMARY\_DEACT indicate the activation and deactivation of a Primary PDP context for a connection. For the secondary context, the EVT\_SECONDARY\_ACT and EVT\_SECONDARY\_DEACT events inform the status. When a connection is established or disconnected, the change in the status of an event is communicated to the client. This serves as information to the user, regarding the connection status. Since these events are asynchronous, callbacks are registered for these events, so that the client is notified of the changes very quickly.

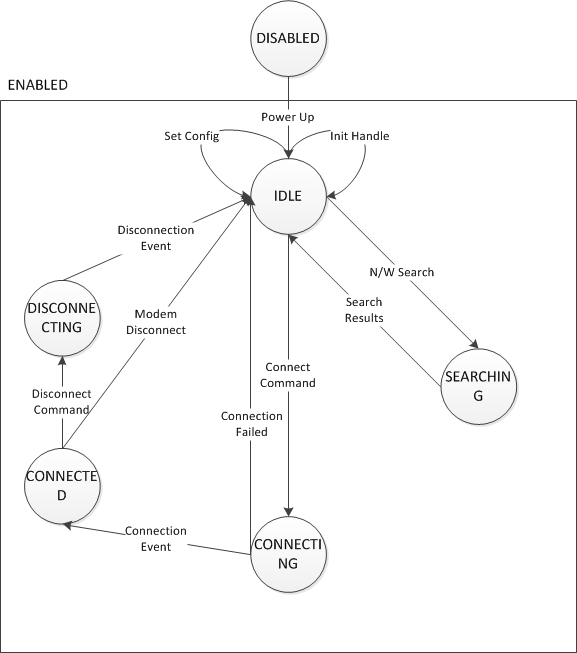
* 1. CM State Machine

The underlying modem that the CM interacts with would maintain a Finite State Machine. These states would provide an indication about the operations taking place within the modem. The CM would maintain a state machine within itself, that would be based upon the States of the modem and we would refer to this as the CM State Machine. The basic States of the CM would be a Disabled and an Enabled State. The Disabled state is when the CM is switched off and the Enabled state is when the CM is powered up. These two states form the parents of the State Machine. Within the Enabled State, we would have the following states:

1. IDLE – The Handle (Init Handle) for the CM Session is initialized in this state. The Configuration file is read in this state and the user also sets the mode to Auto/Manual in this mode.
2. SEARCHING – In the Idle state, the network search command is issued, if the mode is set to “Manual”, after initialization and the configuration setting. A list of networks available is returned; in the case of an error, the suitable error code is returned. The state machine reverts to the IDLE state after the network search. The “net\_cfg\_set” is used to set the IP stack version capability, the Primary context is defined (“ps\_ctx\_define\_primary”) and then the “net\_attach” on the chosen PLMN is carried out. This takes the machine to the “CONNECTING” state.

In the “Auto” mode, the State machine remains at SEARCHING, till the “EVT\_LTE\_REG\_STATUS” event is received from the modem. If the registration is successful, the machine moves to the “CONNECTING” state.

1. CONNECTING – On obtaining a list of the networks available, the CM (depending on the auto/manual mode selection), tries to connect to one among the list returned (Manual mode). This would move the state machine to a Connecting state. On successful completion of the states within the Connecting state, we move to the Connected State. In the case of an error, the state machine returns to the Idle state.
2. CONNECTED – When the connecting state succeeds in performing the steps needed for a successful connection, the machine moves to the Connected state. The decision to move to this state is based on the events notified from the Connecting state.
3. DISCONNECTING – After being connected, the user may decide to disconnect from a network. This is where the Disconnecting state makes an appearance. The modem may disconnect from a network without the user intervention. This leads directly to the Idle state. The user is informed via an event about this disconnection.

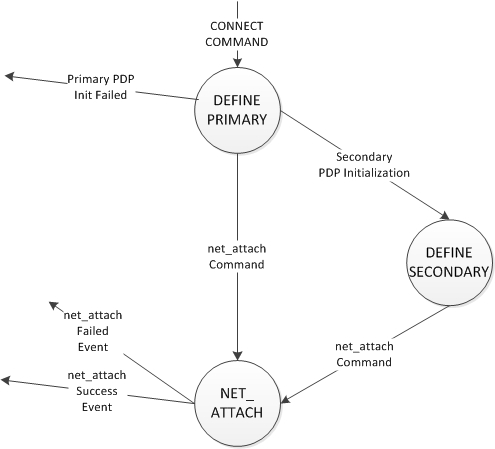


* + 1. Connecting State

A state machine exists within the Connecting State. Decisions are taken here based on the Connect command and on Events that the CM is registered to be notified of. On receiving the Connect command, the Primary EPS bearer or PDP context is defined and PCO configuration is done. This moves “Connecting” to the DEFINE PRIMARY state. A failure in this State notifies the CM of the Primary PDP Init Failed event. A success in the Primary EPS bearer of PDP context creation, can lead to

1. Creation of a Secondary PDP or EPS Bearer context, which leads to the DEFINE SECONDARY state
2. A “net\_attach” call to connect to the network. This leads to the NET\_ATTACH state. A success in this operation is the “net\_attach Success” event and a failure is the “net\_access Failed” event.

The “Primary PDP Init Failed” and the “net\_attach Failed” events are the “net\_attach Failed   
Events”. A success on net\_attach (i.e. “net\_attach Success Event”) moves the State machine to the CONNECTED State.



1. User Interface Design

The Client in the CM would present the user with an interface, which would help him key in the commands and view the status of the connections available to him. There would be two variations to the Client:

1. **Command Line Interface (CLI)** : On startup, the Client binds to the Socket created by the Server. This establishes the first-level of communication in the Connection Manager. Once the connection is established, the user on the CLI is presented with a prompt to enter the commands to interact with the CM. To see the various commands available, the user can type the “help”command, which would list the other commands available to him. A summary of the commands available are (The APIs referred below are from section 8.1):
   1. Search (calls “net\_search”)
   2. Connect (calls “net\_attach”)
   3. Disconnect (calls “net\_detach”)
   4. Connection Status (calls “net\_status\_get”)
   5. View Network related parameters (calls “net\_cfg\_get”)
   6. Check signal strength (calls “net\_sig\_quality\_get”)
2. **Graphical User Interface (GUI)** : The options presented to the user in the case of a CLI client, are presented as clickable buttons in the GUI interface of the client. This feature would be a part of the later releases of the CM. QT would be used for the design of the GUI. The GUI client would consist of atleast 3 different tabs for information display and CM configuration.
   1. **Network Tab :** This tab would present the user with the PLMN ids after a successful Network search command (PLMN selection mode set to Manual). The user selects the PLMN id of his choice and presses a Connect button to perform Netattach. To disconnect from the network, he would use a Disconnect button. The APN ids, the Mobile Country Code, Mobile Network Code and the duration of the connection would be displayed on this tab too.
   2. **Configure Tab :** The CM can be configured to run in the Auto or Manual PLMN selection modes. This would be available as an option under the Configure Tab. In addition to this, the Log Enabling and the Log level selection would also be available under this tab.
   3. **Statistics Tab** : The Statistics tab displays the statistics received from the LTE Stack (via the CM Server). RSSI/CINR, Modem State would also be displayed here. The user would be allowed to change the Statistics Refresh interval under this tab.
   4. **CM State and Device Status Indication :** The GUI would display the CM states and the status of the Device on the GUI. This would inform the user of the availability of the device. Signal strength would be indicated on one of the corners of the GUI.
3. Unified Connection Manager Features

UMA’s Connection Manager version 1.x, 2.x and 3.x are meant in particular to manage LTE/4G connections. We intend to design and implement a Unified Connection Manager (UCM) that supports managing connections for Wi Fi, 3G/2G/EDGE/GPRS, VPN networks. With this in mind, our CM design needs to be changed to handle various Network and Technology scenarios. This section describes the UCM design in detail.

* 1. UCM High Level Description

The current CM design has a client and server communicating though Sockets to perform user operations and inform the user of changes in the Modem/CM. This communication would still hold good. However, we would add more such events as the bouquet of technologies that we would now encounter would increase. This would mean the Client (GUI) would have new tabs/forms/options for the user to choose the following:

1. **The Connection Technology that the user wants the CM to manage:** The user with the help of the CM GUI would be able to choose between LTE/4G, 3G, Wi Fi networks. He would also be able to enter details for a VPN connection. Each option would be presented to the user via a Form/Dialog box.
2. **Priority of the Connections to be managed:** The general tendency for a user is to prefer a Wi Fi connection over say a 3G or a 4G connection to save costs. But when a Wi Fi network is unavailable, the user would need to switch automatically to a Wireless technology like 3G/4G. All these and many such scenarios would need to be a part of the the CM Server.
3. **Wi Fi Offloading:** Wireless networks offer the feature of Wi Fi offloading, wherein the 3G/4G network is available as a Wi Fi Access Point. The CM GUI would allow the user to harness this feature wherever available.
4. **Tethering Profiles:** The user would be given a choice of connecting to the Data connection on a mobile phone or he could create a Tethering profile of any of the available connections, so that another user could connect through that.

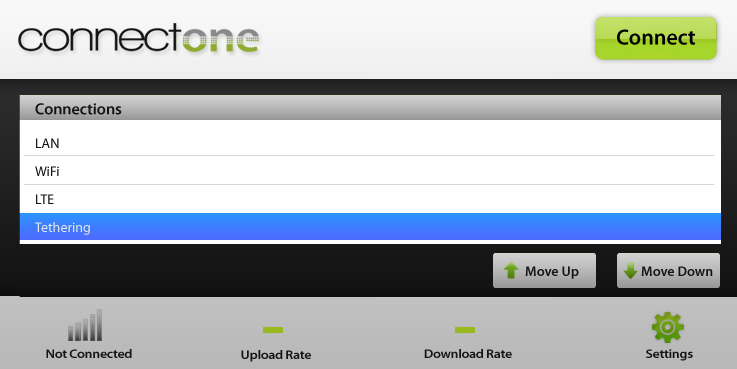
The CM Server would need to undergo a major design change to support the UCM features. A High-level block diagram of the UCM is as follows:



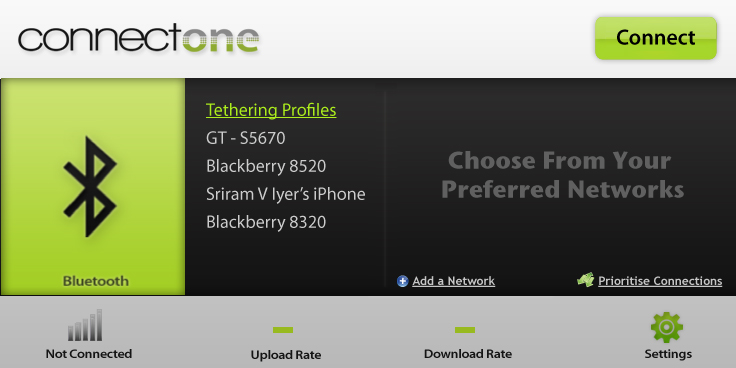
As seen above, the CM Server Block would undergo a significant change in order to handle more Connection Standards. The following sections describe the blocks in the diagram above.

* + 1. CM Client

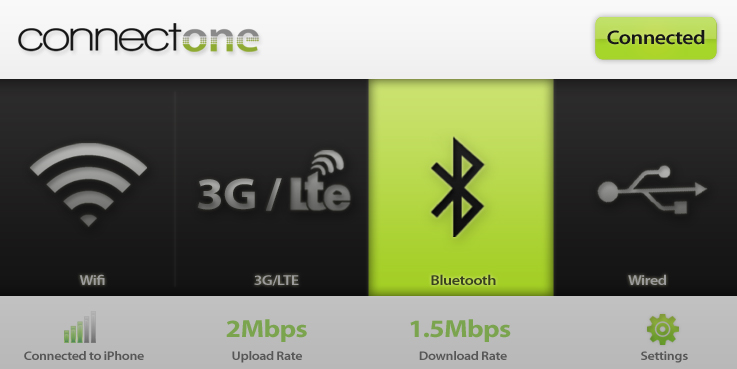
As in the earlier versions of the CM (1.x, 2.x and 3.x), the Client could either be a Command Line (CLI) or a Graphical User Interface (GUI) version. There would be many more options presented to the user, so that the various Network Technologies are chosen and the priorities set accordingly. The communication between the Client and Server remains through sockets (or it could be a monolithic product, wherein the CM Server is built as a library and linked into the CM Client). A further set of events would need to be handled by the Client, which would be described in a later section. The following snapshots depict the various options that the user would have in controlling the operations of the CM.



The image above is an indication of the various network connection options available to the user.



This figure represents the options that would be presented to the user when he chooses a type of connection.



Seen above is a snapshot of the GUI, when a connection with Bluetooth is established.

The above screens are only indicative of what the GUI should look like and are not exact visuals of how they would look when used with actual networks.

* + 1. CM Server

The CM Server would have a new block added after the socket layer. This block would resolve the calls that would need to be made to perform a user command received from the client.

* + - 1. Socket Interface

User options like selection of the Network Technology, Priority setting, Enabling Statistics etc would be presented to the user as Graphical or command line options through the client. Socket communication between the Client and the Server would be used to convey these options in order that the server behaves appropriately. As mentioned earlier, the server and client could be built into one executable as well. For this to happen, the APIs to the various network technologies would be available as APIs to the client.

* + - 1. SAPI & Bearer Manager

Server API (SAPI) is the layer in the CM server, which would expose the various operations of the server in the form of APIs. 1.x, 2.x and 3.x versions of the CM supported APIs only for the LTE connections. With the UCM, there is a need to have many more APIs pertaining to Wi Fi, 3G, VPN connections. In addition to this, on receiving a client request there needs to be intelligence built into the server to call the right set of APIs for each type of network. For e.g., a call to Connect to an LTE network would need to be routed to the right hardware or a call to initiate background scanning for Wi FI networks would need routing to the right OS calls. The Bearer Manager (BM) takes care of routing the clients’ requests to the right set of SAPI calls. SAPI exists as part of the earlier versions of the CM and the Bearer Manager is required as part of the UCM. These two layers would be fused into one in the UCM design.

The BM also takes care of establishing and breaking connections to different networks based on the priorities set by the user through the client. A lesser priority value indicates a higher priority and vice-versa. If two connections have the same priority, the BM manages both connections in parallel.

The BM needs information from the Client about the Network technologies and priorities chosen. This would require a Configuration file. This file could be a simple text file with options and their values or an XML file, parsing which would let the server know of what technologies to enable.

* + - 1. Network Manager

Each network technology that the UCM supports would have a Network Manager (NM) associated with it. The NM brings out the various operations and options that the technology supports in the form of APIs. These APIs would be called NMAPIs from here onwards in this document. The BM calls the NMAPIs based on the user request or in the event of a change in the priorities of the connections.



An NM is a combination of the following blocks as seen in the diagram above:

1. **State Machine/Tracker** – The LTE NM would require a state machine, which would keep a track of the various states that the modem transits. In a few other cases like the Wi Fi and the 3G connections, a state tracker would suffice to inform the user of the events taking place in the device as the state machine runs as part of the hardware concerned. Section 8.11 has a detailed description of the state machine for the LTE network connection. The diagram above is only indicative that a state machine would be a part of the NM. However, the same could be viewed even as a state tracker.
2. **NMAPI Layer** – As described earlier, this layer is a set of API calls that the BM needs to call to perform the action that a user requests. This layer would have a few basic APIs, but depending on the use cases in any given network, there could be many more.

The following NMAPIs would be supported in the UCM:

1. **NM\_SetPriority** – Sets the priority of a network connection. ‘0’ indicates highest priority.
2. **NM\_GetPriority** – Retrieves the current priority of a network connection.
3. **NM\_Enable –** Enables a network connection
4. **NM\_Disable –** Disables a network connection
5. **NM\_Init –** Initializes variables, opens virtual communication (COM) ports, Enables/Disables interface
6. **NM\_Cleanup** – Equivalent to the destructor in an object. This API takes care of freeing up memory, resetting variables
7. **NM\_Connect –** Connects to the network
8. **NM\_Disconnect –** Disconnects from an established connection
9. **NM\_Search –** Manually searches for networks available to the selected technology
10. **NM\_ModemStatistics –** Retrieves the statistics from the device
11. **NM\_GetVersionInfo –** Retrieves version information from the device
12. **NM\_EventCB –** Callback function to report events back to the client
13. **NM\_SetParam –** Set values of configuration parameters like Enable/Disable logging, Logging level, Auto/Manual search mode etc.
14. **NM\_GetParam –** Retrieves values of configuration parameters described in the API above
15. **DAL** – Device Abstraction Layer abstracts all the device functionality in order that a change in the device would only need a change in this layer. For the LTE CM (1.x, 2.x and 3.x), the DAL has a set of APIs which would call the Device APIs or the relevant AT commands to perform the action requested by the client. In addition to this, the DAL also is linked to an AT Command parser, which creates events based on the responses and posts them to the State Machine. These events trigger the transitions to different states within the NM State machine.

Wi Fi and the 3G NMs do not really require a state machine as the device (usually a USB dongle in the case of 3G and a Wi Fi adapter in the case of Wi Fi) has the state machine running within. However, to inform the client of the states that the device transits through, a State Tracker (ST) would be required.

The communication of the DAL with the Modem/Network interface has been depicted as “REQUEST” and “RESPONSE” in the NM block diagram above. The Requests would be the APIs to the device or AT commands. The Responses would be the responses to the APIs, AT command responses or unsolicited events. The parser in the DAL forms events from these responses and events and helps the transitions in the State Machine.

1. **NMInfo Structure –** Versions 1.x, 2.x and 3.x of the CM have a structure called the CMInfo, which stores all the configuration information and the network connection related values. This structure would now be called the NMInfo. To make the operations multi-entrant and to keep information related to a Network connection unique, an NMInfo would exist in each of the NM objects. The NMAPI through its interaction with the socket layer would store and retrieve information from NMInfo. This structure would also be a source of information for the State machine/tracker in its various states and transitions. The DAL and its attached parser would access this structure in order to store vital information obtained through parsing of modem responses and events. All of the three blocks have a Read/Write access to the NMInfo structure.

# 3G Network Manager:

3G Network Manager has following different sections –

1. State Machine
2. Module for RAS dial
3. AT-DAL, which contains the DAL part of 3G and
4. Buffer Parser, which parses all the AT command responses.

# State Machine –

3G Network manager State machine has following states –

* Disable
* Enable



Figure 1 – State Diagram for 3G NM.

3G NM is in Disable State when

* Either the Dongle is not present
* Or the User has not enabled the 3G technology

The Events handled in disable state are –

|  |  |
| --- | --- |
| Event | Action |
| UCONN\_BM\_NM\_ENABLE - Enable Pressed by User | If Dongle already present, then exit to Enable state |
| UCONN\_DEVICE\_STATE\_CHANGE(NOTI\_DEVICE\_OPEN) - Dongle Plugged in | If Enable already pressed by user, then exit to Enable state |
| UCONN\_DEVICE\_STATE\_CHANGE(NOTI\_DEVICE\_NOT\_OPEN) - Dongle plugged out | No Action. |

The 3G NM transits to Enable state as soon as the user enables the 3G technology as well as the 3G dongle is present.

All the further transitions of handling AT command responses, connecting and disconnecting from RAS etc, are all handled in Enable state itself.

On Entry into Enable state 3G NM sends the AT command “AT+CFUN?”, To the modem to find out its state.

If the response is “CFUN:1” then 3G NM continues with sending the following AT commands

* Enable network registration unsolicited result codes with “AT+CGREG=1”. This command will report events proactively whenever the state of network registration changes.
* Set the mode as Auto, so that the modem camps on to the best available network, “AT+COPS=0,0”.

The following unsolicited events from network are handled –

^RSSI:x – Update the RSSI value in the 3G NM variable to be sent to GUI on stats timer expiry.

+CGREG:x – ‘x’ tells the network status and can have the following values –

0 not registered, MT is not currently searching an operator to register to

1 registered, home network

2 not registered, but MT is currently trying to attach or searching an operator to register to

3 registration denied

4 unknown

5 registered, roaming

The valid values are only 1 and 5, and “NM\_READY\_STATE” indication is sent to BM in this case.

If the value is 0, 2, 3 or 4, NM\_NOT\_READY state indication is sent to BM, informing inability to camp onto a network.

Any change in the network status, initiates an AT command to find the network name (AT+COPS?) to be sent to GUI for display.

The Events handled in enable state are –

|  |  |
| --- | --- |
| Event | Action |
| UCONN\_BM\_NM\_DISABLE - Disable Pressed by User | If an ongoing connection, then disconnect and exit to disable state. |
| UCONN\_DEVICE\_STATE\_CHANGE(NOTI\_DEVICE\_NOT\_OPEN) - Dongle plugged out | If an ongoing connection, then disconnect and exit to disable state. |
| UCONN\_BM\_NM\_CONNECT - Connect received from BM | Initiate the RAS connection. |
| UCONN\_BM\_NM\_DISCONNECT - Disconnect received from BM | Hangup the RAS connection and send “Ready” or “Not Ready” to BM depending on the current status of Network (received in CGREG or CREG) |
| UCONN\_EVT\_TIMEOUT(RASDialTimer) | On Expiry of timer check if RAS connection is established, if not exit to NM\_STATE\_NOT\_READY and send a |
| UCONN\_EVT\_TIMEOUT(ThreeGStatsTimer) | On Expiry of this timer send the 3G stats to GUI and also send AT command to Modem to get the RSSI and CINR values.  If a connection is in progress then get RAS connection stats are also sent to the GUI. |
| UCONN\_POWERUP\_RSP | This event occurs when the response of CFUN is “1” which means the modem is in PowerUp state. |
| UCONN\_EVT\_PARAM\_UPDATE | Update the RSSI, CINR and modem state depending upon what has been received from the modem. |

Two timers are handled in Enable state -

RAS Timer - Timer is started as soon as a RAS connection establishment is initiated. Timer is stopped as soon as the RAS connection is established. For Expiry handling see the table above.

3G Stats timer – Started on entry into enable state and stopped on exit from Enable state. For Expiry handling see the table above.

On Expiry of this timer also the RAS connection status is checked. This is done to keep track of the scenario when the RAS gets disconnected on its own due to network issues. If the RAS status says RASC\_Disconnected, then RasHangup Is called and all the RAS and 3G NM variables are initialized so that another RAS connection can be established.

BM is also informed so that if required, another connection can be initiated.



Figure 2 - Call flow diagram for 3G NM

# RAS Connection –

3G NM uses RAS API’s to establish/maintain/hangup a connection with the network.

For establishing a RAS connection the different steps involved are –

1. Initialize - Initialization of RAS connection implies, finding the right RAS entry from a list of RAS entries. “RasEnumEnteries function lists all the entry names in the remote access phone book. The function “InitRas”, does a search of all RAS entries and finds the entry corresponding to the Huawei dongle. The RAS entry name is stored in a global variable “RasEntriesNames”.

“RasEnumEnteries function parameters -

|  |  |
| --- | --- |
| Parameters | Information / Values. |
| Reserved | Reserved – Must be NULL. |
| lpszPhonebook | Pointer specifying full path and file name of phone book file. In case of NULL the entries are enumerated from all the remote access phone book files.  Set to NULL in 3G NM. |
| lpRasEntryName | Pointer to a buffer that on return from function contains the array of RASENTRYNAME structure.  Set to NULL in 3G NM. |
| lpcb | Pointer to a buffer that on return from the function contains the size of array of RASENTRYNAME structure.  Set to 0 or minimum size depending upon the Windows OS version. See comment below. |
| lpcEnteries | On return contains, Number of phone book entries. |

In case of **Windows Vista or later:**To determine the required buffer size, call **RasEnumEntries** with lprasentryname set to **NULL** and lpcb should be set to zero. The function will return the required buffer size in lpcb and an error code of**ERROR\_BUFFER\_TOO\_SMALL**.

In case of WindowsXP or before: To determine the required buffer size, call **RasEnumEntries** with lprasentryname set to **NULL and lpcb set to some size( size of RASENTRYNAME in code)**. The function will return the required buffer size in lpcb and an error code of**ERROR\_BUFFER\_TOO\_SMALL or ERROR\_SUCCESS**.

On getting the correct buffer size, the function is called again with the correct size to get the details in the lpRasEntryName structure.

Browse through the lpRasEntryName and find the RAS entry specific to the 3G dongle plugged in and update the global variable, RasEnteriesNames.

1. Connect – The “RasGetCredentials” is called to get the User name and password for the RAS entry “RasEntriesNames” and the information is used to initialize the structure for parameter, lpRasDialParams, In the RasDial function below.

RasDial function

|  |  |
| --- | --- |
| Parameters | Information / Values. |
| lpRasDialExtensions | Set to NULL, since none of the features of RASDIALEXTENSIONS are used. |
| lpszPhoneBook | Pointer specifying full path and file name of phone book file. In case of NULL the entries are enumerated from all the remote access phone book files.  Set to NULL in 3G NM case. |
| lpRasDialParams | Contains the calling parameters for the RAS connection.  Set to the details of RasEntriesNames. |
| dwNotifierType | Specifies the nature of lpvNotifier param.  Set to ‘0L’ in 3G NM since lpvNotifier points to a RasDialFunc call back function. |
| lpvNotifier | Specifies a RasDialFunc call back function.  Set to RasDialEventNotifier function. |
| lphRasConn | Should be set to NULL before calling the function and if RasDial succeeds then this contains the handle to the RAS connection. |

1. Disconnect –

To Disconnect the RAS, the API called in RasHangUp.

|  |  |
| --- | --- |
| **Parameters** | **Information / Values.** |
| lphRasConn | The RAS connection handle which was initialized during RasDial function call. |

1. Connection status and statistics –

To get RAS connection status, the AI used is RasGetConnectStatus-

|  |  |
| --- | --- |
| **Parameters** | **Information / Values.** |
| Hrasconn | The RAS connection handle which was initialized during RasDial function call. |
| Lprasconnstatus | Pointer to the RASCONNSTATUS structure, which on output has RAS connection status. |

To get RAS connection statistics, the API used is RasGetConnectionStatistics –

|  |  |
| --- | --- |
| **Parameters** | **Information / Values.** |
| hRasConn | The RAS connection handle which was initialized during RasDial function call. |
| lpStatistics | Pointer to RAS\_STATS structure, which on output contains the statistics of RAS connection. |

* 1. ThreeG AT-DAL –

The sending of AT commands to the Huawei device is done using a class “UThreeGDal”, which is derived from “UconnAtDal” .

The different AT commands sent are –

* AT+CFUN=1 – To set the functionality of modem to full.
* AT+CFUN=? – To get the modem functionality.
* AT+CGREG=1 – To register for all the unsolicited events.
* AT+CGREG=? – To get the network status.
* AT+CSQ – To get the RSSi and CINR values.
* AT+COPS=0,0 – To set the network mode as auto and also to set the format as “Long format alphanumeric”. This format gets helps in getting the full network name.
* AT+COPS? – To initiate a search for all available networks.

UThreeGDal is the Device abstraction Layer for getting responses of AT commands from the Huawei device.

* 1. ThreeG Buffer Parser -

ThreeGBufferParser parses and stores the responses from the Modem. This could be unsolicited events sent from the network or the responses to AT commands sent to the modem.

The following AT Command Responses are handled as of now –

* +COPS: - To get the name of the network.
* +CSQ: - To get the RSSI and CSQ values.
* +CGREG: or CREG: - To get the network status.
* +CFUN: - To get the modem functionality.
* +CUSD: - To get the USSD strings.
* +ATI: - To get the information about the modem device.

The following unsolicited events are handled as of now –

* +CGREG: or CREG: - To get the network status.
* ^RSSI: – To get the RSSI value.
* +CUSD: - To get the USSD strings from the network.

# Appendix A: Configuration file

The CM configuration file would contain the following information:

PLMNSelectMode: CM Configuration mode - Auto/Manual

Enable4G: Switch ON/OFF 4G

LogtoFile: Informs the Server to write logs into a file

LogFile: The filename which would contain the logs

LoggingLevel: Error, Debug or Information Log enabling

AttachRAT: RAT on which network attach is to be performed (Manual Mode)

VidPidName: Device Id that would help the CM recognize the device when its plugged in

# Appendix B: Client Server Interface Definitions

cmCommon.h

1) #define CM\_REQUEST\_QUERY\_NETWORK\_SELECTION\_MODE 0

 //Query current network selectin mode

Request::

 "requestData" is NULL

“returnResponse” is 1

Response::

 "responseData" is int \*

     0 for automatic selection

     1 for manual selection

 errors:

  SUCCESS

  FAILURE

2) #define REQUEST\_SET\_NETWORK\_SELECTION\_AUTOMATIC 1

// Specify that the network should be selected automatically

Request::

"requestData" is NULL

"returnResponse" is 1

Response::

“responseData” is NULL (errors field conveys the success/failure result in this case)

// This request must not respond until the new operator is selected and registered

errors:

 SUCCESS

 FAILURE

3) #define REQUEST\_SET\_NETWORK\_SELECTION\_MANUAL 2

// Manually select a specified network.

Request::

 "requestData" is const char \* specifying MCCMNC of network to select (eg "310170")

 "returnResponse" is 1

Response::

“responseData” is NULL (errors field conveys the success/failure result in this case)

// This request must not respond until the new operator is selected and registered

errors:

SUCCESS

FAILURE

# Appendix C: Abstraction Layers

Device Abstraction Layer

The CM could be used with a modem other than BCS5xx family. In such a case, the Common API calls for that particular modem would be different. It thus makes sense to abstract these API calls by having a Device Abstraction Layer. The APIs from the Device Abstraction layer would be directly called from the CM Server, and this layer in turn would call the APIs such as “net\_attach”, “net\_detach”, “net\_search” etc. Each of the abstracted APIs assembles the parameters obtained from the client (through sockets) and calls the modem device driver APIs with those parameters.

A summary of these APIs is as follows:

|  |  |
| --- | --- |
| **Device Abstraction Layer APIs** | **Common APIs Abstracted** |
| uconnBcmNetSearch | net\_search |
| uconnBcmNetAttach | net\_attach |
| uconnBcmNetDetach | net\_detach |
| uconnBcmEquipInfoGet | equip\_info\_get |
| uconnBcmEquipConfigGet | equip\_config\_get |
| uconnBcmEquipConfigSet | equip\_config\_set |
| uconnBcmNetCfgSet | net\_cfg\_set |
| uconnBcmNetCfgGet | net\_cfg\_get |
| uconnBcmNetStatusGet | net\_status\_get |
| uconnBcmNetSigQualGet | net\_sig\_quality\_get |
| uconnBcmDfnInfoGet | ps\_dfn\_info\_get |
| uconnBcmPsCtxDefPrimary | ps\_ctx\_define\_primary |
| uconnBcmPsCtxDefSecondary | ps\_ctx\_define\_secondary |
| uconnBcmPsCtxUndefPrimary | ps\_ctx\_undefine\_primary |
| uconnBcmPsCtxUndefSecondary | ps\_ctx\_undefine\_secondary |
| uconnBcmPsCtxCfgGet | ps\_ctx\_cfg\_get |
| uconnBcmPsCtxCfgSet | ps\_ctx\_cfg\_set |
| uconnBcmPsCtxInfoGet | ps\_ctx\_info\_get |
| uconnBcmPsCtxActivate | ps\_ctx\_activate |
| uconnBcmPsCtxModify | ps\_ctx\_modify |
| uconnBcmPsCtxDeactivate | ps\_ctx\_deactivate |
| uconnBcmpropAPNTableGet | prop\_APN\_table\_get |
| uconnBcmpropAPNTableSet | prop\_APN\_table\_set |

OS Abstraction Layer

Across the CM implementation, we would have a number of OS related calls, examples being Socket creation, binding, thread creation, message queue creation. Each OS has its own APIs for these system calls and therefore, abstracting those calls would make portability of the CM on other Operating Systems an easier task. The layer which abstracts these system calls would be called the Operating System Abstraction Layer (OSAL).

# Appendix D: IP Address Assignment

To perform IP switching on Windows, the “netsh” utility is used. This utility configures the interface with the IP parameters like EnableDHCP, IPAddress, SubNetMask etc., for IPv4. For IPv6, netsh.exe is used to setup and tear down the DHCP and setup static IP address. Though this section describes the IP setting in Windows, equivalent calls shall be used on Linux/MAC OS to achieve a similar result. A flowchart for the IP assignment is as below:

