FOMA M1000
Power Management API
Specification

DOCUMENT CONTROL NUMBER : IL93-MISC-05-0604

Version :01.00

Date : 03-04-2005

Revision History

Template Information :

Document Control Number	:	IL93-STD-01-0224
Version	:	01.00
Date	:	03-Nov-2001

Revision History:

Revision #	Date	Description
01.00	03-04-2005	Creation of document for outside FOMA M1000 developers

Contents

1. Introduction

1.1 Purpose

This document describes the Power Management API's provided in the FOMA M1000 products. Please refer to the reference documentation for details related to the architecture and design.

1.2 Target Audience

This document is intended for FOMA M1000 software developers that require the use of the Power Management interfaces.

1.3 Abbreviations, Acronyms and Definitions

See reference **Error! Reference source not found.**.

AP	application processor
API	application program interface
BP	baseband processor
BT	Bluetooth
DLC	dynamic link channel
DSR	virtual signal used to indicate a MUX channel should be closed.
IPC	interprocessor communications
MUX	27.010 multiplexor between AP and BP
PCAP	Platform Control Audio Power IC
PM	power management
PS	Power Server
RTC	Real-Time Clock
SW	software
UID	Baseband Unique ID
USB	Universal Serial Bus
UUUD	User Up, User Down (a Motorola server)
WLAN	Wireless, Local Area Network

1.4 Problem Reporting Instructions

Please send problems to http://www.motocoders.com.

2. Overview

2.1 Goals and Objectives

This document describes the Power Management APIs provided by the Power Server.

3. API Information

This section will detail API's that Power Server provides.

3.1 Starting the Power Server

3.1.1 powerserverexe.exe

The Power Server executable name that is required to start the server.

3.1.2 StartPowerServer()

This global function starts the Power Server process.

Input : void
Output: void
Usage : StartPowerServer();

3.2 Connection to the Power Server

3.2.1 RMPowerServer::Connect()

Connect to the Power Server. This will start the server if it isn't already.

Input : void
Output: TInt
Usage : TInt ret = iPowerServer.Connect();

3.2.2 RMPowerServer::Close()

Disconnect from the Power Server. The Power Server is not a transient server, so it will remain active even with no clients connected.

Input : void
Output: void
Usage : iPowerServer.Close();

3.3 System Activity/Condition Information

Clients to Power Management need to call these API's to inform it of the various system activities taking place. The Power Server uses this information as input to make state transition decisions.

3.3.1 Condition Types

The following lists the system conditions (TMConditionType) defined in mpowerserver_clientapi.h that the Power Management system needs to be aware of.

```
enum TMConditionType
{
      EMConditionTypeCallVoice,
      EMConditionTypeCallCsDataInternal,
      EMConditionTypeCallPacketDataInternal,
      EMConditionTypeCallCsDataExternal,
      EMConditionTypeCallPacketDataExternal,
      EMConditionTypeCallVideo,
      EMConditionTypeAudioPlayback,
      EMConditionTypeVideoPlayback,
      EMConditionTypeAccessorySerial,
      EMConditionTypeAccessoryUsb,
      EMConditionTypeAccessoryIrEnabled,
      EMConditionTypeAccessoryIrConnection,
      EMConditionTypeAccessoryBluetooth,
      EMConditionTypeBluetoothHeadsetProfile,
      EMConditionTypeBluetoothSerialProfile,
      EMConditionTypeBatteryPower,
      EMConditionTypeExternalPower,
      EMConditionTypeLowBatteryWarning,
      EMConditionTypeLowBatteryThreshold,
      EMConditionTypeConvenienceTimerExpired,
      EMConditionTypeChargeTimerExpired,
      EMConditionTypeSync,
      EMConditionTypeDataLogging,
      EMConditionTypeIPC,
      EMConditionTypeACM1,
      EMConditionTypeACM2,
      EMConditionTypeDLOGDSP,
      EMConditionTypeDLOGMCU,
      EMConditionTypeBT,
      EMConditionTypeCarKitPresent,
      EMConditionTypeIgnition,
      EMConditionTypeAudioPlayerPlayback,
      EMConditionTypeMidrateCharge,
      EMConditionTypeWebMediaPlayback,
      EMConditionTypeRingVoice,
      EMConditionTypeRingVideo,
      EMConditionTypeSMSTerminate,
      EMConditionTypeCBTerminate,
      EMConditionTypePIN1Request,
      EMConditionTypeAccessoryDumb,
      EMConditionTypeAccessoryHeadset,
      EMConditionTypeAccessoryUnsupported,
      EMConditionTypeAccessoryMmc,
      EMConditionTypeNetworkService,
      EMConditionTypeBacklightOn,
      EMConditionTypeEmail,
      EMConditionTypeWLAN
};
```

3.3.2 RMPowerServer::InformConditionStart()

Inform the Power Server that a defined condition has started. The caller is required to inform the Power Server when the condition has stopped/ended.

Input : const TMConditionType aCondition
Output: void
Usage : iPowerServer.InformConditionStart (EMConditionTypeCallVoice);

3.3.3 RMPowerServer::InformConditionStop()

Inform the Power Server that a defined condition has stopped/ended. This also calls InformConditionPause() on the specific condition to clear the client's conditions completely.

Input : const TMConditionType aCondition
Output: void
Usage : iPowerServer.InformConditionStop (EMConditionTypeCallVoice);

3.3.4 RMPowerServer::InformConditionPause()

Inform the Power Server that a condition has paused. The caller may use this instead of stop because certain resources may still be in use even though the condition is not actively running. An example of this might be the MP3 player. Pausing may require that certain resources are still loaded on the DSP.

Input : const TMConditionType aCondition
Output: void
Usage : iPowerServer.InformConditionPause (EMConditionTypeAudioPlayback);

3.3.5 RMPowerServer::InformConditionResume()

Inform the Power Server that a condition that was paused has now resumed.

Input : const TMConditionType aCondition
Output: void
Usage : iPowerServer.InformConditionResume (EMConditionTypeAudioPlayback);

3.3.6 User Activity/Inactivity

The following lists the activity/inactivity types (TMInactivity) defined in mpowerserver_clientapi.h by Power Management. These are used to inform the Power Server of the various levels of user activity and inactivity.

```
enum TMInactivityType
{
EMActivity,
EMInactivityDisplayFrontlight,
EMInactivityFrontlight,
EMInactivityDisplay,
EMResetInactivity,
```



```
EMActivityTouchScreen,
EMActivityGameA,
EMActivityGameB,
EMActivityShortcut,
EMActivityHutch,
EMActivityEnd,
EMActivitySend,
EMActivityUp,
EMActivityDown,
EMActivityLeft,
EMActivityRight,
EMActivityCenter,
EMActivitySpeaker,
EMActivityVolUp,
EMActivityVolDn,
EMActivityVR,
EMActivityDevLock,
};
```

3.3.6.1 EMActivity

Power Server is informed of this type anytime there is user interaction with the device via the keypad or the touchscreen.

3.3.6.2 EMInactivityDisplayFrontlight

Power Server is informed of this type when there hasn't been user interaction with the device via the keypad or the touchscreen for the Frontlight and Display Inactivity time set in the Control Panel. Note that this is the equivalent of EMInactivityDisplay type since the Display timeout is always equal to or greater than the Frontlight timeout.

3.3.6.3 EMInactivityFrontlight

Power Server is informed of this type when there hasn't been user interaction with the device via the keypad or the touchscreen for the Frontlight Inactivity time set in the Control Panel.

3.3.6.4 EMInactivityDisplay

Power Server is informed of this type when there hasn't been user interaction with the device via the keypad or the touchscreen for the Display Inactivity time set in the Control Panel.

3.3.6.5 EMResetInactivity

Power Server is informed of this type when the current inactivity time should be reset. This prolongs the time before a standby transition by a minimum of the current display inactivity timeout.

3.3.6.6 EMActivityTouchScreen

Power Server is informed of this type when a touch screen event occurs. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.7 EMActivityGameA

Power Server is informed of this type when the Game A key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.8 EMActivityGameB

Power Server is informed of this type when the Game B key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.9 EMActivityShortcut

Power Server is informed of this type when the Shortcut key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.10 EMActivityHutch

Power Server is informed of this type when the Hutch key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.11 EMActivityEnd

Power Server is informed of this type when the End key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.12 EMActivitySend

Power Server is informed of this type when the Send key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.13 EMActivityUp

Power Server is informed of this type when the Up key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.14 EMActivityDown

Power Server is informed of this type when the Down key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.15 EMActivityLeft

Power Server is informed of this type when the Left key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.16 EMActivityRight

Power Server is informed of this type when the Right key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.17 EMActivityCenter

Power Server is informed of this type when the Center key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.18 EMActivitySpeaker

Power Server is informed of this type when the Speaker key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.19 EMActivityVolUp

Power Server is informed of this type when the VolUp key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.20 EMActivityVolDn

Power Server is informed of this type when the VolDn key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.21 EMActivityVR

Power Server is informed of this type when the VR key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.6.22 EMActivityDevLock

Power Server is informed of this type when the Device Lock key is pressed. This prolongs the time before a standby transition by a minimum of the current display and front light inactivity timeout.

3.3.7 RMPowerServer::InformInactivity()

Interface to inform Power Server of changes to the user activity level. The return type of ETrue indicates that the EMActivity type should be disgarded.

Input : const TMInactivityType& aInactivityType
Output: TBool
Usage : iPowerServer.InformInactivity (EMActivity);

3.4 Power Events

The following is used to allow clients to be informed of various power events in the system by requesting notification. Note that when clients register for notification of events, they will receive ALL events. It is up to the client to determine which events they wish to take action on.

3.4.1 Power Events

The following lists the Power Events (TMPowerEvent) defined in mpowerserver_clientapi.h.

```
enum TMPowerEvent
{
EMPowerEventPowerOn,
EMPowerEventPowerOff,
EMPowerEventRfOn,
EMPowerEventRfOff,
EMPowerEventBpReset,
EMPowerEventPowerCut,
EMPowerEventLinkSuspend,
EMPowerEventLinkAvailable,
EMPowerEventPowerOnComplete,
```

```
EMPowerEventUuudPowerOff,
EMPowerEventWakeUp,
EMPowerEventLinkResume,
EMPowerEventBpSuspended,
EMPowerEventRtcOn,
EMPowerEventUuudPowerOffSilent,
EMPowerEventChargeOnly,
EMPowerEventSleep,
EMPowerEventApReset
};

typedef TPckgBuf<TMPowerEvent> TMPowerEventPckgBuf;
```

3.4.1.1 EMPowerEventPowerOn

Power Server sends this event when the phone has completed loading required servers and apps on cold boot. At this point, power management has established a connection and set the state of the Baseband processor. The system may either be in Active or Airplane mode at this point. This may be determined by calling the GetRfState API (refer to 3.5.3). User interaction will begin at the control of the UUUD server and app, which manages things such as the splash screen and password prompts.

3.4.1.2 EMPowerEventPowerOff

Power Server sends this event when the phone is going to transition to User Off Mode. This event implies that the BP is being shut off and the USB Link between the boards will be disconnected (27.010 MUX not available).

3.4.1.3 EMPowerEventRfOn

Power Server sends this event when the RF state of the phone is changed to RF On.

3.4.1.4 EMPowerEventRfOff

Power Server sends this event when the RF state of the phone is changed to RF Off. This implies that RF services are not available.

3.4.1.5 EMPowerEventBpReset

Power Server sends this event when the BP has experienced a reset, due to a panic, a failure or from the AP resetting it.

3.4.1.6 EMPowerEventPowerCut

Power Server sends this event when the phone has experienced a power cut. This event will be sent after the phone has recovered from the power cut (the BP has been restored).

3.4.1.7 EMPowerEventLinkSuspend

Power Server sends this event when it has decided to suspend the USB link to the Baseband. If a client requires the link to write data to the Baseband, the client may call the RequestLinkResume() method on the PowerServer.

3.4.1.8 EMPowerEventLinkAvailable

Power Server sends this event when the 27.010 MUX has been successfully started and is available for clients to open a channel. Clients must not open a channel until this event has been sent. Clients must query the state of the link via the GetLinkState() method after subscribing for the event the first time to ensure that event was not sent prior to subscription for the event notification.

3.4.1.9 EMPowerEventPowerOnComplete

Power Server sends this event when clients (specifically the UUUD server) notify Power Server that the EMPowerEventPowerOn event has been completely processed.

3.4.1.10 EMPowerEventUuudPowerOff

Power Server sends this event when it first detects a power off condition. This event is meant only for the UUUD Server to handle some initial Application level shutdown.

3.4.1.11 EMPowerEventWakeUp

Power Server sends this event when the Power Mode of the system transitions out of a standby mode. It can be used by entities like the status bar to know when to refresh UI level display items, for example.

3.4.1.12 EMPowerEventLinkResume

Power Server sends this event when it has successfully resumed the USB link to the Baseband. This event may be the result of a client calling the RequestLinkResume() method on the PowerServer or by the Baseband requesting a remote wakeup because it has a message to send to the AP.

3.4.1.13 EMPowerEventBpSuspended

Power Server sends this event when it receives notification from the Baseband that it has entered SUSPEND mode. This event indicates that services that were available may no longer be available due to the mode change.

3.4.1.14 EMPowerEventRtcOn

Power Server sends this event when it wakes from User Off mode due to an RTC alarm. Additionally, this event may be used to indicate a silent startup, the matching startup event to EMPowerEventUuudPowerOffSilent, silent shutdown event.

3.4.1.15 EMPowerEventUuudPowerOffSilent

Power Server sends this event when it first detects a power off condition. This event is meant only for the UUUD Server to handle some initial Application level shutdown. This event will be sent instead of EMPowerEventUuudPowerOff if the shutdown animation and audio playback should be skipped to silently shut off the device. This event would be used in cases of a panic shutdown/restart or other cases where the obvious user feedback of the shutdown would like to be avoided.

3.4.1.16 EMPowerEventChargeOnly

Power Server sends this event when it transitions the system to Charge Only Mode. In this mode, the user is unable to fully interact with the device other than to observe the current charging status.

3.4.1.17 EMPowerEventSleep

Power Server sends this event when it is transitioning to Data/MP3 Standby mode. This indicates that the display is off and any periodic updates to the display should be avoided (e.g., signal strength). If possible, the Baseband should be told to prevent sending asynchronous messages that would simply result in updates to the display.

3.4.1.18 EMPowerEventApReset

Power Server sends this event when the AP has experienced a reset, due to a panic. This is sent in place of the EMPowerEventPowerOn event.

3.4.2 RMPowerServer::NotifyOnPowerEvent()

This utility allows clients to register for notification of system power events. The various power events are detailed above.

Input : TRequestStatus& aReqStat, TMPowerEventPckgBuf& aPowerEventPckg
Output: void
Usage : iServer.NotifyOnPowerEvent(iStatus, iPowerEvent);

A wrapper API exists for this function called CMPowerEvent. Please refer to the following section for more details.

3.4.3 RMPowerServer::CancelNotifyOnPowerEvent()

This utility allows clients to cancel registration for notification of system power events.

Input : TRequestStatus& aReqStat, TMPowerEventPckgBuf& aPowerEventPckg
Output: void
Usage : iServer.CancelNotifyOnPowerEvent();

A wrapper API exists for this function called CMPowerEvent. Please refer to the following section for more details.

3.4.4 Power Event Wrapper API

The following classes are part of a “convenience” wrapper for the Power Event API’s. It implements an Active Object and provides a simple mix-in observer class (MMPowerEventObserver) that gets called on power events. It also handles making a connection to the PowerServer (creating a session) and registering for subsequent power events (required as part of the NotifyOnPowerEvent API).

3.4.4.1 MMPowerEventObserver::HandleNotificationEvent

CMPowerEvent makes this callback whenever a Power Event occurs. The parameter contains the event that occurred. The client does NOT need to call the NotifyOnPowerEvent method to get the next event; this is done automatically.

Input : TMPowerEvent aPowerEvent
Output: void
Usage : HandlerNotificationEvent(thePowerEvent);

3.4.4.2 Power Event Processing Modes

The following lists the Power Event Processing Modes (TMPowerEventProcessMode) defined in mpowerevent.h.

```
enum TMPowerEventProcessMode
{
EProcessModeNone,
EProcessModeAuto,
EProcessModeManual
};
```

3.4.4.2.1 EProcessModeNone

Power Server will not wait for the client to process the event. This is the default behavior of the class.

3.4.4.2.2 EProcessModeAuto

Power Server will wait for the client to process the event. The server is notified of event process completion automatically after the HandleNotificationEvent callback is done.

3.4.4.2.3 EProcessModeManual

Power Server will wait for the client to process the event. The server is notified of event process completion as a result of the client calling PowerEventProcessingComplete. Clients are required to acknowledge every event, even those they are not interested in.

3.4.4.3 CMPowerEvent::NewL()

The CMPowerEvent class is a convenience class that provides some additional functionality on top of the RMPowerServer API's. This utility allows clients to register for notification of system power events without needing to explicitly connect to the Power Server. The utility also automatically re-registers for power event notification.

Input : void
Output: CMPowerEvent*
Usage : CMPowerEvent* myPowerEvent = CMPowerEvent::NewL();

Note that this utility uses an observer callback to handle the power event received from the Power Server. The observer is the mix-in class MMPowerEventObserver. A pure virtual function HandleNotificationEvent is called on the observer. Please refer to cmpowerevent.h and mpowerserver_clientapi.h file or the API summary documentation for full details.

3.4.4.4 CMPowerEvent::NotifyOnPowerEvent()

The CMPowerEvent class is a convenience class that provides some additional functionality on top of the RMPowerServer API's. This utility allows clients to register for notification of system power events without needing to explicitly connect to the Power Server. The utility also automatically re-registers for power event notification.

Input : MMPowerEventObserver& aObserver
Output: void
Usage : iPowerEvent.NotifyOnPowerEvent(this);

Note that this utility uses an observer callback to handle the power event received from the Power Server. The observer is the mix-in class MMPowerEventObserver. A pure virtual function HandleNotificationEvent is called on the observer. Please refer to cmpowerevent.h and mpowerserver_clientapi.h file or the API summary documentation for full details.

3.4.4.5 CMPowerEvent::CancelNotifyOnPowerEvent()

The CMPowerEvent class is a convenience class that provides some additional functionality on top of the RMPowerServer API's. This method allows clients to cancel registration for notification of system power events without needing to explicitly connect to the Power Server.

Input : MMPowerEventObserver& aObserver
Output: void
Usage : iPowerEvent.CancelNotifyOnPowerEvent(this);

The observer is the mix-in class MMPowerEventObserver. A pure virtual function HandleNotificationEvent is called on the observer. Please refer to cmpowerevent.h and mpowerserver_clientapi.h file or the API summary documentation for full details.

3.4.4.6 CMPowerEvent::SetPowerEventProcessingMode()

The CMPowerEvent class is a convenience class that provides some additional functionality on top of the RMPowerServer API's. This method allows clients to set the system power event processing mode.

Input : MMPowerEventObserver& aObserver
Output: void
Usage : iPowerEvent.CancelNotifyOnPowerEvent(this);

The observer is the mix-in class MMPowerEventObserver. A pure virtual function HandleNotificationEvent is called on the observer. Please refer to cmpowerevent.h and mpowerserver_clientapi.h file or the API summary documentation for full details.

3.4.4.7 CMPowerEvent::PowerEventProcessingComplete()

The CMPowerEvent class is a convenience class that provides some additional functionality on top of the RMPowerServer API's. This method allows clients to notify the Power Server that the last power event received has been processed. This allows the Power Server to continue on to the next state. Note that this method shall only be called when the ProcessingMode has been set to EMProcessModeManual.

Input : MMPowerEventObserver& aObserver
Output: void
Usage : iPowerEvent.CancelNotifyOnPowerEvent(this);

The observer is the mix-in class MMPowerEventObserver. A pure virtual function HandleNotificationEvent is called on the observer. Please refer to cmpowerevent.h and mpowerserver_clientapi.h file or the API summary documentation for full details.

Note that it is the client's responsibility to call this method for EVERY power event received when the mode is EMProcessModeManual. Failure to do so could corrupt the Power Server's state.

3.5 RF State

The following is used to allow clients to query and change the RF state of the machine.

3.5.1 RF States

The following lists the RF States (TMRfState) as defined in mpowerserver_clientapi.h for use with RF APIs.

```
enum TMRfState
{
EMRfStateRfOff,
EMRfStateRfOn,
};
```


3.5.2 RMPowerServer::RequestSetRFState()

Request that the Power Server set the RF state of system. State can be EMRfStateRfOff or EMRfStateRfOn.

Input : const TMRfState aRfState
Output: void
Usage : iPowerServer.RequestSetRFState (EMRfStateRfOn);

3.5.3 RMPowerServer::GetRFState()

Get the RF State of the system. State can be EMRfStateRfOff or EMRfStateRfOn.

Input : void
Output: TMRfState& aRfState
Usage : iPowerServer.GetRfState (theRfState);

3.6 Link State

The following is used to allow clients to query and possibly change the state of the IPC Link to the Baseband.

3.6.1 Link States

The following lists the Link States (TMLinkState) defined in mpowerserver_clientapi.h by Power Management.

```
enum TMLinkState
{
EMLinkStateDisconnected,
EMLinkStateConnected,
EMLinkStateSuspended
};
```

3.6.1.1 EMLinkStateDisconnected

Power Server returns this state when the link is not connected. Therefore, the mux is unavailable for use by the clients. Power Server has either not yet started the link or has had to restart the link and it is not yet available.

3.6.1.2 EMLinkStateConnected

Power Server returns this state when the link has been successfully started. MUX clients may open their channels on the link.

3.6.1.3 EMLinkStateSuspended

Power Server reports this state if it has decided to suspend the USB link to the Baseband. If a client requires the link to write data to the Baseband, the client may call the RequestLinkResume() method on the PowerServer.

3.6.2 Link Failure Cause Codes

The following lists the Link Failure/Problem codes (TMLinkProblem) defined in mpowerserver_clientapi.h by Power Management. The code helps identify the problem that was detected. This may be used to determine a course of action.

```
enum TMLinkProblem
{
EMLinkProblemNone,
EMLinkProblemUnknown,
EMLinkProblemNoResponse,
EMLinkProblemErrors
};
```

3.6.3 RMPowerServer::GetLinkState()

This utility allows clients to query the current state of the link. Clients should call this method prior to attempting to open a channel on the MUX. After first registering for Power Events from the Power Server, clients should also call this method in case the link became available prior to the clients having been started (this will almost certainly be the case during cold boot).

Input : TMLinkState& aLinkState
Output: void
Usage : iServer.GetLinkState(iLinkState);

3.6.4 RMPowerServer::RequestLinkResume()

This utility allows clients to request Power Management to resume a suspended link. Clients should call this method prior to attempting to use the mux after being told that the link is suspended via a Power Event or after getting a LinkSuspended state back from GetLinkState(). *Note: The suspend feature will not be implemented until a much later release of Power Management.*

Input : void
Output: void
Usage : iServer.RequestLinkResume();

3.6.5 RMPowerServer::InformLinkFailure()

This method is an input to the Power Manager to indicate that there is a potential problem on the link. Users that have a DLC to the BP would call this if they detect a potential problem on their respective link. Clients should attempt to identify a Link Failure Cause and supply the API with the name of the channel being used. Power Server will ping its own peer on the Baseband to attempt to reproduce the problem. If PM detects a problem, the link may be restarted.

If restarting the link does not solve the problem, a Baseband reset will be initiated. Clients can be notified of the Baseband reset via other Power Server API’s.

Input : const TMLinkProblem& aCause, const TDesC& aCsyMuxPort
Output: void
Usage : iServer.InformLinkFailure();

3.7 Resets

The following provides a mechanism to allow clients to request various system resets.

3.7.1 RMPowerServer::RequestSystemReset()

Request Power Management to reset the system. Following this call, the machine will undergo a full cold boot by transitioning from Off Mode to the last User Mode (Active or Airplane).

Input : void
Output: void
Usage : iServer.RequestSystemReset();

3.7.2 RMPowerServer::RequestSystemShutdown()

Request Power Management to shutdown the system. This causes a transition to the User Off Mode. Subsequent power up will be considered a warm boot and the system will transition to the last User Mode (Active or Airplane).

Input : void
Output: void
Usage : iServer.RequestSystemShutdown();

3.7.3 RMPowerServer::RequestMasterClear()

Request Power Management to initiate a master clear operation. This results in a system reset. Subsequent power up will be considered a cold boot.

Input : void
Output: void
Usage : iServer.RequestMasterClear();

3.7.4 RMPowerServer::RequestMasterReset()

Request Power Management to initiate a master reset operation. This results in a system reset. Subsequent power up will be considered a cold boot.

Input : void
Output: void
Usage : iServer.RequestMasterReset();

3.8 Device Management

The following provides a mechanism to allow clients to control power to various system devices.

3.8.1 Bluetooth

Power Management interfaces to control Bluetooth hardware.

3.8.1.1 RMPowerServer::BluetoothEnable()

Enable Bluetooth hardware power regulator.

Input : void
Output: void
Usage : iServer.BluetoothEnable ();

3.8.1.2 RMPowerServer:: BluetoothDisable()

Disable Bluetooth hardware power regulator.

Input : void
Output: void
Usage : iServer.BluetoothDisable ();

3.8.1.3 RMPowerServer:: BluetoothReset()

Reset the BT hardware.

Input : void
Output: void
Usage : iServer.BluetoothReset ();

3.8.1.4 RMPowerServer:: IsBluetoothEnabled()

Query the current state of the Bluetooth hardware power regulator. ETrue = BT regulator enabled, EFalse = BT regulator disabled.

Input : void
Output: TBool
Usage : btEnabled = iServer.IsBluetoothEnabled();

3.8.1.5 RMPowerServer::GetBluetoothDeviceAddress()

This method is used to request the unique Bluetooth device address from the Baseband processor. The Bluetooth Device Address consists of 6 bytes of data. The call may return an error if the Power Server is unable to talk to the Baseband in order to retrieve the address or KErrNone if successful.

Input : void
Output: TInt err, TDes8& aUid
Usage : err = iServer.GetBluetoothDeviceAddress (theAddress);

3.8.2 Baseband

Power Management interfaces to query information about the baseband.

3.8.2.1 RMPowerServer::GetBasebandUid()

A mechanism to request the unique ID from the Baseband processor. The Baseband ID consists of 8 bytes of data. The call may return an error if the Power Server is unable to talk to the Baseband in order to retrieve the ID or KErrNone if successful.

Input : void
Output: TInt err, TDes8& aUid
Usage : err = iServer.GetBasebandUid (theUid);

3.8.3 AP DSP

Power Management interfaces to aid in the management of the AP's DSP.

3.8.3.1 RMPowerServer::RequestDspClock()

Request that Power Server ensure that the DSP can support the requested speed in order to successfully complete operations. Power Server will attempt to move to a state that would satisfy the requirement passed in. This should be called when the DSP requirements change in both directions (require more speed or less speed). Power Server may decide to move the system to a lower power mode and run the DSP at a slower clock speed if the requirements allow it and the current ongoing activities allow. If the DSP is not required and could be idled, the speed should be set to 0.

Input : void
Output: TBool err, TInt aDspClockSpeedInMhz
Usage : err = iServer.RequestDspClock(120);

3.8.4 GPS

Power Management does not expose any interface for managing GPS. However, certain actions such as switching modes to Airplane (via RequestSetRfState(EMRfStateRfOff) will automatically turn off GPS.

3.8.5 Camera

Power Management interfaces to control Camera hardware.

3.8.5.1 RMPowerServer::CameraEnable()

Enable Camera hardware power regulators.

Input : void
Output: void
Usage : iServer.CameraEnable ();

3.8.5.2 RMPowerServer:: CameraDisable()

Disable Camera hardware power regulators.

Input : void
Output: void
Usage : iServer.CameraDisable ();

3.8.5.3 RMPowerServer:: CameraReset()

Reset the Camera hardware.

Input : void
Output: void
Usage : iServer.CameraReset ();

3.8.5.4 RMPowerServer:: IsCameraEnabled()

Query the current state of the Camera hardware power regulators. ETrue = Camera regulators enabled, EFalse = Camera regulators disabled.

Input : void
Output: TBool
Usage : CameraEnabled = iServer.IsCameraEnabled();

3.8.6 Keypad Backlight

Power Management interfaces to control the Keypad Backlight.

3.8.6.1 RMPowerServer::KeypadBacklightEnable()

Enable the Keypad Backlight Feature

Input : void
Output: void
Usage : iServer.KeypadBacklightEnable ();

3.8.6.2 RMPowerServer::KeypadBacklightDisable()

Disable the Keypad Backlight Feature

Input : void
Output: void
Usage : iServer.KeypadBacklightDisable ();

3.8.6.3 RMPowerServer::SetKeypadBacklightTime()

Set the time (in microseconds) for how long the Keypad Backlight is on.

Input : Tint
Output: void
Usage : iServer.SetKeypadBacklightTime (20000000)

This sets the Keypad Backlight time to 20 seconds. The keypad backlight is turned off when the display turns off.

3.8.7 USB Client/Function

Power Management does not expose any interface for managing the USB client interface. However, the entity controlling managing USB client will need to be aware of Power Modes via Power Events because the supplies powering the interface may be switched off as a result of power cuts or User Off transitions.

3.8.8 RTC

Power Management does not expose any interface for managing the RTC. However, Power Management does ensure that the RTC maintains the proper value and abstracts the use of the PCAP RTC from the normal use of the AP RTC. The same applies for system alarms.

3.8.9 IR

Power Management does not expose any interface for managing the IR. However, Power Management does require that it be notified when IR is in use.

3.9 Power Management Sample Code

```
#include <mpowerserver_clientapi.h>
#include <cmpowerevent.h>
#include <getmuxport.h>

class CExampleClass : public MMPowerEventObserver
{
public:
   void CExampleClass();
   //Derived from MMPowerEventObserver Mixin class (cmpowerevent.h)
   void HandleNotificationEvent(TMPowerEvent aPowerEvent);

private:
   //See cmpowerevent.h
   CMPowerEvent* iPowerEvent;
};

void CExampleClass::CExampleClass()
{
   //Create the Power Event wrapper object
   //This connects to the server automatically
   iPowerEvent = CMPowerEvent::NewL();
   //Add this object as a power event observer
   iPowerEvent->NotifyOnPowerEvent(*this);

   //Connect to the Power Server to query the initial state
   RMPowerServer iServer;
   iServer.Connect();
   TMLinkState theState;
   //Get the current state of the link in case we missed it becoming available
   iServer.GetLinkState(theState);

   //States defined in mpowerserver_clientapi.h
   if (theState != EMLinkStateDisconnected)
   {
      //Link is available, check to see if there is a channel for me
      TInt ret = CMGetMuxPort::GetMuxPortConfigL(KMuxPowerMan, csyPortName);
      if (ret == KErrNone)
      {
         //Open the channel name supplied in csyPortName
      }
      else
      {
         //Either MUX isn't supported or my channel has been selectively disabled
      }
   }
   else
   {
      //Link isn't ready yet, wait for EMPowerEventLinkAvailable
   }
   //Can close connection right now, unless other API's will be used...
   iServer.Close();
}

//Implementation of the pure virtual method defined in MMPowerEventObserver
void CExampleClass::HandleNotificationEvent(TMPowerEvent aPowerEvent)
{
   //Events defined in mpowerserver_clientapi.h
   if (aPowerEvent == EMPowerEventLinkAvailable)
   {
      //Link is available, check to see if there is a channel for me
      TInt ret = CMGetMuxPort::GetMuxPortConfigL(KMuxPowerMan, csyPortName);
      if (ret == KErrNone)
      {
         //Open the channel name supplied in csyPortName
      }
      else
      {
         //Either MUX isn't supported or my channel has been selectively disabled
      }
   }
   //No need to re-register, done automatically by CMPowerEvent
}
```

