Engineering Report

Report Title:	***Sierra Wireless Linux QMI SDK Application Programmer's Guide***
Project Name:	***SLQS***
Document #:	***4110914***
Legacy #:	***N/A***
Revision:	***1.04***
Customer Name:	
Author:	***Calvin Fong***
Location:	***FileHold***
Date:	***November 7, 2012***

Revision History

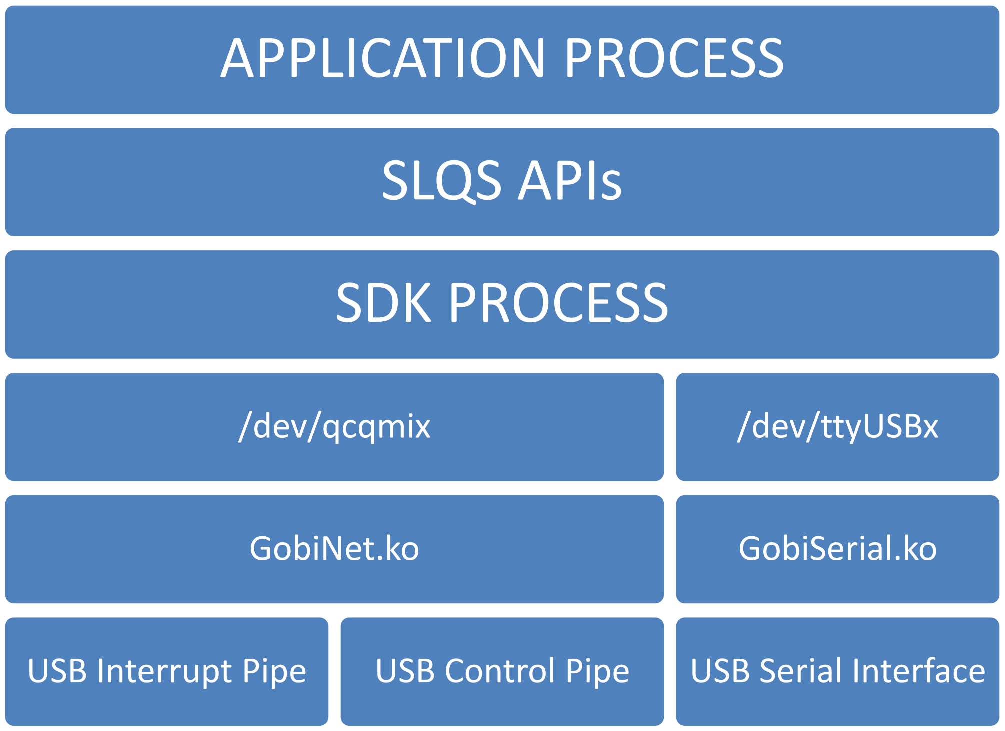
Doc Rev	CR#	Date	Name	Comment
1.00		Sept. 09/11	O.Hadary	Initial Release
1.01		Oct. 05/11	O.Hadary	Added section 5.4: Connection Manager Sample Application; Updated sections 3.1, 4.2, 4.3, 5.2, 5.3
1.02		Nov. 04/11	Boon Lee	Added Section 6.2 RAM dump tool, corrected header level for 6.1
1.03		Mar. 15/12	O. Hadary	Added section 4.2; Updated/revised sections 2 and 3.
1.04		Nov. 07/12	Calvin Fong	Added SL9090 as supported device in Section 2.3; Added PDS & SWIOMA Sample App in Section 5; Update titling on Section 4.4; Update code routine at Section 3.3.1

Table of Contents

1 SLQS System Architecture

1.1 Overview

- The Application process communicates with the device by executing SLQS APIs.
- The API calls are translated in QMI request SDUs that are sent to the SDK process over a local IPC datagram socket.
- The SDK writes the QMI PDUs to a device file named /dev/qcqmix, where x is an integer, associated with the QMI interface.
- The QMI PDUs are sent to the device over the USB control channel via the GobiNet.ko driver module
- Notifications are received over the interrupt channel, which prompts the driver to read the responses coming over the USB control channel.
- The SDK reads the QMI response from /dev/qcqmix and sends a response to the application process over a local IPC datagram socket.



2 SLQS Prerequisites

2.1 Supported Processors

The following processors are supported:

- x86
- ARM
- PPC
- MIPS

2.2 Supported Devices

The following devices are supported:

- MC7700/10/50
- MC8305/55.
- SL9090

Note: The MC77xx devices must operate in "QMI Mode" and not in "Direct-IP" mode.

The tables below list the hexadecimal values of the Vendor ID (VID) and Product ID (PID) pairs supported by the SLQS.

Supported Application-mode VID/PIDs

VID	1199	1199	1199	1199	1199	3F0
PID	68A2	9011	9013	9015	9019	371D

Supported Boot-mode VID/PIDs

VID	1199	1199	1199	1199	1199	3F0
PID	68A2	9010	9012	9014	9018	361D

To check your device's VID/PID, issue the **lsusb** command. The output will present a list of USB devices with a column showing each device's manufacturer. The device VID/PID can be read from the row containing the correct device manufacturer. Additionaly, on MC77xx devices, you can use the **AT!UDINFO?** AT command to check VID/PID information. If your VID/PID does not match the any of the entries in the tables above, contact your FAE for support.

2.3 Device Drivers

2.3.1 Acquiring the drivers

Get in touch with your FAE for acquiring drivers for your device if you are operating in QMI mode – the mode of operation required for using the SLQS.

2.3.2 Supported Linux kernels

We support open source kernel version 2.6.32 or newer. It is the customer's responsibility to modify the SDK and drivers for kernels outside the scope of what is supported.

2.3.3 System dependencies

Make sure you have a network connection and issue the following commands:

```
sudo apt-get install build-essential make gcc
sudo apt-get install linux-headers-`uname -r`
```

2.3.4 Building and installing the drivers

```
cd GobiSerial; make; sudo make install
cd GobiNet; make; sudo make install
sudo modprobe GobiSerial [debug=Y]
sudo modprobe GobiNet [debug=Y]
```

2.3.5 Querying driver versions and supported devices

```
modinfo GobiSerial
modinfo GobiNet
```

2.3.6 Unloading the drivers

```
sudo rmmod GobiSerial
sudo rmmod GobiNet
```

2.3.7 Enabling and disabling the drivers' diagnostic messages

```
Note: enabling and disabling the driver's diagnostic messages
requires root privileges.

Enable diagnostic messages:
echo 1 > /sys/modules/GobiSerial/parameters/debug
echo 1 > /sys/modules/GobiNet/parameters/debug

Disable diagnostic messages:
echo 0 > /sys/modules/GobiSerial/parameters/debug
echo 0 > /sys/modules/GobiNet/parameters/debug
```

2.3.8 Verifying proper driver operation

1. Open terminal and type tailf /var/log/syslog
2. Plug in Sierra Wireless device
3. Check /dev/ for existence of the following devices (check syslog in case the device nodes are static i.e. built into the kernel image and not dynamically mounted)
    - /dev/ttyUSB0
    - /dev/ttyUSB1
    - /dev/ttyUSB2
    - /dev/qcqmix ; where x is an integer starting at 0.

3 User Application Development

3.1 SDK process

3.1.1 Building the SDK executable

```
navigate to pkgs:   cd pkgs
clean then build:   make –f pkgs.mak complete
clean:              make –f pkgs.mak clean
build:              make –f pkgs.mak
```

3.1.2 Verifying SDK and target platform interoperability

The SDK periodically checks to see if a supported device is connected to the target platform. If you do not see the following message[1] in your logs, then the device has not been detected and the SDK will not be able to communicate with the device. In this case, it is most likely that you are either using an unsupported device or that your drivers need to be updated to support the device.

```
usb 2-1.5: new high speed USB device using ehci_hcd and address 10
usb 2-1.5: config 1 has an invalid interface number: 8 but max is 3 usb 2-1.5: config 1 has no interface number 1
usb 2-1.5: configuration #1 chosen from 1 choice GobiSerial 2-1.5:1.0: GobiSerial converter detected
usb 2-1.5: GobiSerial converter now attached to ttyUSB0 GobiSerial 2-1.5:1.2: GobiSerial converter detected
usb 2-1.5: GobiSerial converter now attached to ttyUSB1 GobiSerial 2-1.5:1.3: GobiSerial converter detected
usb2-1.5: GobiSerial converter now attached to ttyUSB2
usb0: register 'GobiNet' at usb-0000:00:1d.0-1.5, QmiNet Ethernet Device, 3e:a6:1f:b3:66:62
SWI SDK Process: USDT:Device State Change 0 -> 1
creating qcqmi0
USDT:Device State Change 1 -> 1
```

If you see the message above but do not see the following message in your logs, then the device's interfaces have not been successfully mapped to their respective /dev/ttyUSBx and/or /dev/qcqmix device special files and the SDK will not be able to communicate with the device.

```
USDT:Device State Change 1 -> 2
USDT:Device ready: VID 1199, PID 68a2, 4 interfaces
QM:qm_ds_handle_app_dev_ready: devstate 1
QM:SDK<-Mdm: ch/QMImsgid/QMImsglen/IPCmsglen: 1/0000/0/25
QM:DS Device Event Notification received 1
```

In this case, it is possible that: (1) your drivers don't support the inserted device (2) you have not added a device node for /dev/qcqmix (usually 0 or 1) with the proper major and minor numbers (3) the interface configuration of your device is not supported by the SDK (4) the SDK's device scanning routine requires custom modifications specific to your platform's sysfs /sys/bus/usb/devices entry for the device in question. The major and minor numbers of the device can be determine by issuing **ls -l /dev/qcqmix** on the command line. The fourth and fifth columns contain the major and minor numbers, respectively (see **man ls** for details).

[1] SDK messages will be displayed in both /var/log/user.log and /var/log/syslog

3.2 User Application process

3.2.1 Building the Application Executable

Refer to any one of the Sample Applications’ make files as a starting point for writing a script for building your application. Remember to add the “strip” command to your script in order to remove all symbol information from your libraries and application image if your system is memory constrained.

3.2.2 Communicating with the device

The application must adhere to the SDK’s stop and wait (synchronous execution) protocol; there can be only one outstanding transaction between the Application and SDK, at any time. All API function calls are blocking and execute within the context of the application process. When the Application executes API function, the corresponding request is constructed and sent to the SDK process over a local IPC. The request(response) is sent to(received from) the device from within the execution context of the SDK Process. The response from the device is validated and sent back to the Application Process over a local IPC socket. Following, the message contents are unpacked and used for populating the user supplied agruments.

Notifications, on the other hand, are asynchronous and therefore, may arrive at any time. The application receives notifications within the execution context of a dedicated notification thread that is created and used by the SDK within the Application’s process. Thus, it is important that minimal processing be done inside the registered callback functions.

3.3 User Application Development

3.3.1 Where do I start?

The Connection Manager Sample Application is a good place to start. The source code is located at SampleApps/Sample_Connection_Manager/src/SLQSsampleCM.c.

The following outlines the recommended method for integrating SLQS initialization code into your application. Note that all variables below are assumed to have been defined.

```
/* Set the SDK executable path for your target platform */

SetSDKImagePath("<path to executable>/slqssdk");

/* Launch the SDK process and create IPC sockets over which the APP and SDK will exchange
 * messages.
 */
rc = SLQSStart();

/* Check the return code to verify that the SDK process was successfully created and that the
 * IPC sockets are available.
 */

if( eQCWWAN_ERR_SWICM_SOCKET_IN_USE == rc )
{

    /* If the SDK is already being used by another application, exit */

    rcprint("Another APP is using the SDK", rc);
```

```
        cleanup( __func__, rc );
    }
    else if( eQCWWAN_ERR_NONE != rc )
    {

        /* Otherwise, kill the current SDK process and restart */

        SLQSKillSDKProcess();

        if( eQCWWAN_ERR_NONE != (rc = SLQSStart()) )
        {
            /* If all else fails, exit */
            cleanup( __func__, rc );
        }
        else
            rcprint("SDK process restarted", rc );
    }
    else
    {
        syslog(LOG_USER, "%s: APP<->SDK IPC init successful\n", __func__);
    }

    /* Register for device state change notification */

    rc = SetDeviceStateChangeCbk( &devstatechgcb );

    if( eQCWWAN_ERR_NONE != rc )
    {
        cleanup( "APP failed to register for Device State Change notification", rc );
    }
    else
    {
        syslog( LOG_USER,
                "%s: APP registered for Device State Change notification\n",
                __func__ );
    }

    /* Enumerate the device */
    while( eQCWWAN_ERR_NONE != QCWWAN2kEnumerateDevices( &DevicesSize, (BYTE *)(pdevices) ))
    {
        syslog( LOG_USER,
                "%s: Unable to find device..\n",
                __func__ );
        sleep(1);
    }

    /* if it has, register your application with the SDK using the information obtained
     * for the detected device; if it hasn't, there is no need to take further action as you
     * will be notified via the registered callback - devstatechgcb
     */
    rc = QCWWANConnect( pdev->deviceNode, pdev->deviceKey );

    if( eQCWWAN_ERR_NONE == rc )
    {
        /* no error - optional application code */
    }
    else
    {
        /* error encountered - optional application code */
        rcprint(__func__, rc );
    }

/* Graceful SLQS teardown */
static void cleanup(const char *msg, ULONG ercode)
{
    syslog( LOG_USER, "%s: %s (0x%lX)", __func__, msg, ercode );

    /* If the application is connected to the SDK, then disconnect to (1) terminate threads and
     * free resources that have been created and allocated, respectively, for communicating with
     * the device, and (2) allow other applications to communicate with the device via the SDK.
     */
    if( sdkbound )
        QCWWANDisconnect();

    /* Insert application specific code */
}

/* Device State Change Callback Function
 *
 * Note that this function executes in the context of the SDK's notification thread.
 * Therefore, minimize the amount of processing done in this function.
 */
static void devstatechgcb(eDevState device_state)
{

    /* Your application code should cache the device state and take appropriate action such as
     * enumerating the device and connecting to the SDK as shown above.
     */
}

/* macro used in code segments above */
#define rcprint(s, u) syslog(LOG_USER, "%s: rc = 0x%lX, %s", s, u, slqserrstr(u))

/* You can add error code to error string mapping to the table below in order to aid your
 * application debugging.
 */
typedef struct{
    enum eQCWWANError e;
    const char *es;
}slqserr_s;

static slqserr_s errstr[] =
{
    { eQCWWAN_ERR_INTERNAL,           "eQCWWAN_ERR_INTERNAL" },
    { eQCWWAN_ERR_MEMORY,             "eQCWWAN_ERR_MEMORY" },
    { eQCWWAN_ERR_INVALID_ARG,        "eQCWWAN_ERR_INVALID_ARG" },
    { eQCWWAN_ERR_BUFFER_SZ,          "eQCWWAN_ERR_BUFFER_SZ" },
    { eQCWWAN_ERR_NO_DEVICE,          "eQCWWAN_ERR_NO_DEVICE" },
    { eQCWWAN_ERR_SWIDCS_IOCTL_ERR,   "eQCWWAN_ERR_SWIDCS_IOCTL_ERR" },
    { eQCWWAN_ERR_QMI_MISSING_ARG,    "eQCWWAN_ERR_QMI_MISSING_ARG" },
    { eQCWWAN_ERR_SWICM_SOCKET_IN_USE,          "eQCWWAN_ERR_SWICM_SOCKET_IN_USE" },
    { eQCWWAN_ERR_SWIDCS_DEVNODE_NOT_FOUND,     "eQCWWAN_ERR_SWIDCS_DEVNODE_NOT_FOUND" },
    { eQCWWAN_ERR_SWIDCS_IOCTL_ERR,             "eQCWWAN_ERR_SWIDCS_IOCTL_ERR" },
    { eQCWWAN_ERR_SWIDCS_APP_DISCONNECTED,      "eQCWWAN_ERR_SWIDCS_APP_DISCONNECTED" },
    { eQCWWAN_ERR_SWICM_QMI_SVC_NOT_SUPPORTED,  "eQCWWAN_ERR_SWICM_QMI_SVC_NOT_SUPPORTED" },
    { 0, "" }
};

static const char *slqserrstr(ULONG er)
{
    int count = 0;

    while( errstr[count].e ){
        if( errstr[count].e == er )
        {
            return errstr[count].es;
        }
        count++;
    }

    return "";
}
```

3.3.2 What is the QCWWANDisconnect API for?

When your application no longer needs to communicate with the device it should execute the QCWWANDisconnect API in order to (1) free the resources allocated by the SDK for communicating with the device (2) to deregister from all but the device state change callback and (3) to allow other applications to use the services of the SDK. As long as the device is connected to the target and the SDK process is alive, an application can always reconnect at a later time.

3.3.3 What if I want to terminate the SDK process?

To kill the SDK process, execute the SLQSKillSDKProcess API. Note that this API requires that the SDK image be named slqssdk, as is the case for the images located in the build/bin sub-directories of the SDK release.

3.3.4 How will I know if the device resets and what should my application do?

Assuming the application has registered for the device state change callback, it will be notified whenever a device is disconnected or detected. Following a device reset, once the device has been detected by the SDK, all of the callback functions that the application had registered for will be re-registered by the SDK on the application's behalf. Thus, the application need not take any action on a device reset aside from managing itself.

3.4 UMTS, LTE, and CDMA Data Sessions

This section describes the APIs for configuring profiles for use in a data session call a start and stopping data session calls. For details of the API parameters, refer to the doxygen documentation of the APIs.

3.4.1 Profile Configuration

Profiles must be set before a data call can be made. Some carriers fix the profiles that can be used on their network. Without the use of SDK APIs, profiles can be created or modified using AT commands. The SDK provides the following APIs for profile configuration:

- GetDefaultProfile
- SetDefaultProfile
- GetDefaultProfileLTE
- SetDefaultProfileLTE

These APIs write and get the default profile to and from the device, respectively. The default profile will be the one used to establish a data session. The LTE version supports IPV6 in addition to IPV4.

- SLQSGetProfile
- SLQSSetProfile

These APIs perform the same functionality as the above APIs, but allow a profile id to be specified. Valid profile ID values are 1 to 16.

- SLQSDeleteProfile

This API deletes a configured profile stored on the device. The deletion of a profile does not affect profile index assignments.

- SLQSCreateProfile

This API is used to create a new profile with the specified parameters. Note that some firmware versions do not support the optional Profile ID parameter. In this case an error will be returned and the caller can subsequently create a profile by specifying a NULL pointer for the Profile ID parameter. The Profile ID pertaining to the newly created profile is returned in the response structure parameter.

- SLQSModifyProfile

This API is used to create a new profile with the specified parameters.

3.4.2 IP Family Preference

LTE devices start up in IPV4 mode by default. To select the IP class to use in your application use the following API before initiating a session.

- SLQSSetIPFamilyPreference

This API is used to set the control point IP preference. IPV4, IPV6, and IPV4V6 (as of LSQS01.05.00) are supported.

3.4.3 Session Initiation

- StartDataSession2

This API will use the default profile set up as described above to make the data connection. Some networks may require authentication fields. To start a data session after a device has been enumerated, the following API may be used. Technology should be changed for the appropriate network – UMTS or CDMA. Note that the optional parameters below are left as NULL for simplicity. Some of the optional parameters are supplied by the user as preferred information. The network may not be able to assign the preferred values and assign other values instead. In that case, the SLQSGetRuntimeSettings API may be used to retrieve some of this information once a data session has been established.

```
ULONG technology = 1;
ULONG sessionid = 0xFFFFFFFF;
ULONG failurereason = 0cFFFFFFFF;
rc = StartDataSession2( &technology, /* UMTS */
                        NULL, /* Primary DNS */
                        NULL, /* Secondary DNS */
                        NULL, /* Primary NBNS */
                        NULL, /* Secondary NBNS */
                        NULL, /* APN */
                        NULL, /* IP address */
                        NULL, /* authentication */
                        NULL, /* username */
                        NULL, /* password */
                        &sessionid,
                    &failurereason );
```

- StartDataSessionLTE

This API supports IPV4, IPV6, and IPV4V6 data sessions. Note that the device defaults to IPV4. Unless an IPV4 only data session is desired, your program will have to call SLQSSetIPFamilyPreference in order to configure the call for IPV6 or IPV4V6. Your program should not call SLQSSetIPFamilyPreference while a session is active.

Note: Do not use the IPFamilyPreference provided by this API – it has been deprecated.

- SLQSStartStopDataSession

This API can be used to start or stop a data session on a specified 3GPP or 3GPP2 profile or, using the default profile if a specific profile index is not provided. This is the only data session API that allows an application to start a data session using any valid profile ID, whereas the other data session APIs mentioned above all use the default profile.

3.4.4 Session Termination

- StopDataSession

This API is used to terminate any currently active data session given the session ID.

3.5 Unsupported Features

3.5.1 Multiple Device Support

The SDK will only communicate with the first device that its scanning algorithm detects. It is not recommended that the SDK be used on a host with multiple connected devices; the behaviour under such circumstance is not deterministic. For example, if two supported devices are connected to the host system, then upon a device reset it cannot be guaranteed that the same device will be chosen by the scanning algorithm.

3.5.2 Multiple Application Support

The SDK does not currently support multiple applications running concurently. The behaviour under such circumstances is undefined. The SDK does, however, support the running of sequential applications as long as every application gracefully disconnects from the SDK prior to running the next application.

3.5.3 Multiple PDN Support

Currently, one can only execute a data session for one PDP context at a time i.e. only one network interface is supported at this time.

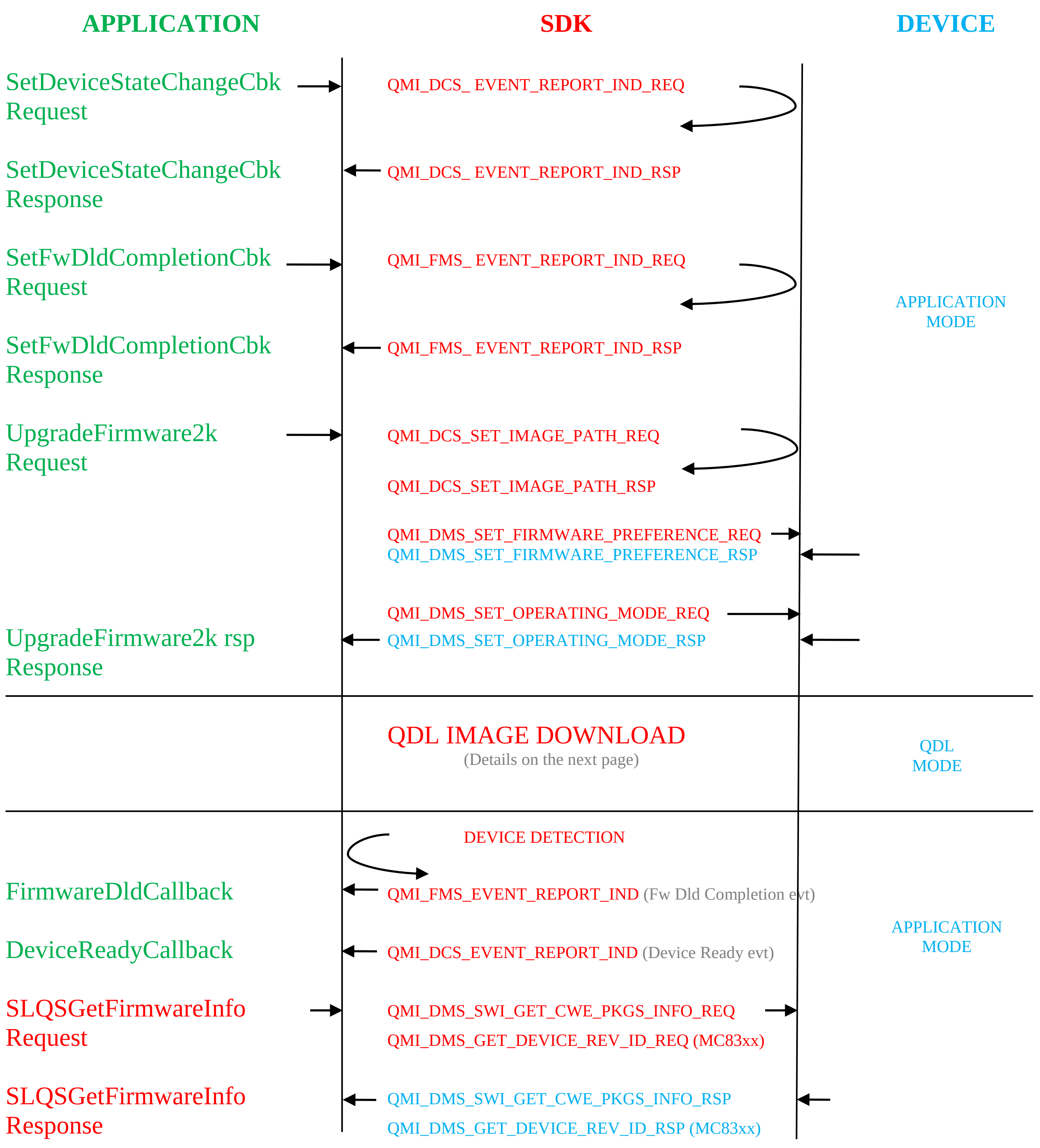
4 SLQS Image Management

4.1 Overview

The Gobi Image Management and MC77xx Image Management sample applications can be used to:

1. Query information about the firmware stored on the device
2. Query information about firmware images stored on the host.
3. Download firmware to the device

4.2 MC77xx / MC83xx Firmware Upgrade Process


APPLICATION
SDK
DEVICE
SetDeviceStateChangeCbk Request
QMI_DCS_ EVENT_REPORT_IND_REQ
SetDeviceStateChangeCbk Response
QMI_DCS_ EVENT_REPORT_IND_RSP
SetFwDldCompletionCbk Request
QMI_FMS_ EVENT_REPORT_IND_REQ
APPLICATION MODE
SetFwDldCompletionCbk Response
QMI_FMS_ EVENT_REPORT_IND_RSP
UpgradeFirmware2k Request
QMI_DCS_SET_IMAGE_PATH_REQ
QMI_DCS_SET_IMAGE_PATH_RSP
QMI_DMS_SET_FIRMWARE_PREFERENCE_REQ
QMI_DMS_SET_FIRMWARE_PREFERENCE_RSP
QMI_DMS_SET_OPERATING_MODE_REQ
UpgradeFirmware2k rsp Response
QMI_DMS_SET_OPERATING_MODE_RSP
QDL IMAGE DOWNLOAD
(Details on the next page)
QDL MODE
DEVICE DETECTION
FirmwareDldCallback
QMI_FMS_EVENT_REPORT_IND (Fw Dld Completion evt)
APPLICATION MODE
DeviceReadyCallback
QMI_DCS_EVENT_REPORT_IND (Device Ready evt)
SLQSGetFirmwareInfo Request
QMI_DMS_SWI_GET_CWE_PKGS_INFO_REQ
QMI_DMS_GET_DEVICE_REV_ID_REQ (MC83xx)
SLQSGetFirmwareInfo Response
QMI_DMS_SWI_GET_CWE_PKGS_INFO_RSP
QMI_DMS_GET_DEVICE_REV_ID_RSP (MC83xx)

SLQS image management sequence diagram

Referring to the figure above:

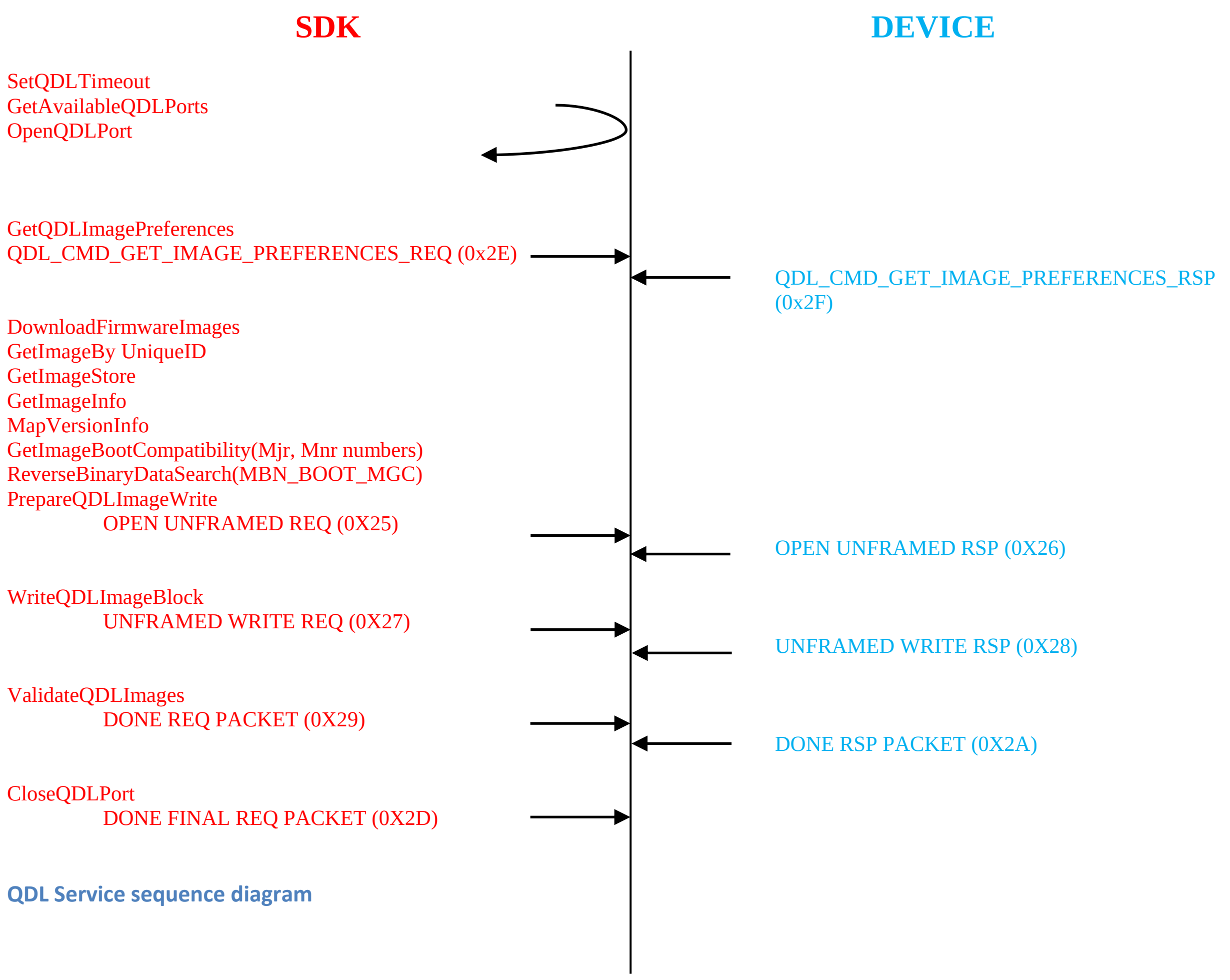
1. The Application may choose to register for a firmware download completion callback in order to be notified when the image download process has completed. Additionally, the application should register for the device state change callback in order to be notified of when the device has entered application mode subsequent to the image download, and is ready to communicate with the host.

2. To upgrade the firmware on the device, the application must issue the UpgradeFirmware2k API.

3. The Application should not issue any further API requests until the firmware download has completed and the device is ready.

4. Reception of the firmware download completion callback does not guarantee that the download process was successful. Once the device is ready, the application should issue the GetFirmwareRevisions API (MC83xx) or the SLQSGetFirmwareInfo API (MC77xx) to determine if the upgrade was successful.

QDL IMAGE DOWNLOAD



QDL Service sequence diagram

4.3 MC700/10/50 Image Management

4.3.1 MC77xx Image Management Sample Application

Location: SampleApps/MC77xx_Image_Management/

Purpose: Query image information for a MC7xx image located on the host
Query image information for the image running on a MC77xx device
Download firmware to a MC77xx device

Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel

Execute:	i386:	sudo ./mc7xximgmgmti386
	ARM:	sudo ./mc7xximgmgmtarm9
	PPC:	sudo ./mc7xximgmgmtppc
	MIPS BE:	sudo ./ mc7xximgmgmtmips
	MIPS LE:	sudo ./ mc7xximgmgmtmipsel

Reference: SampleApps/MC77xx_Image_Management/readme.txt

- The only supported file is a *_SPKG.cwe file.
- The program must be executed from the SampleApps/MC77xx_Image_Management/bin directory
- There must only be one *.cwe file in the path specified for any option which requires the user to specify a path.
- If you encounter errors when specifying a relative path, specify the fully qualified path instead.
- for more details refer to the readme.txt file

4.4 MC83xx & SL9090 Image Management

4.4.1 Gobi Image Management Sample Application

Location: SLQSab.cd.ef /SampleApps/Gobi_Image_Management/

Purpose: Query carrier image information for Gobi images located on the host
Query carrier image information for the images stored on a device
Download firmware to a device

Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel

Execute:	i386:	sudo ./gobiimgmgmti386
	ARM:	sudo ./gobiimgmgmtarm9
	PPC:	sudo ./gobiimgmgmtppc
	MIPS BE:	sudo ./gobiimgmgmtmips
	MIPS LE:	sudo ./gobiimgmgmtmipsel

Reference: SLQSab.cd.ef /SampleApps/Gobi_Image_Management/readme.txt

- The only supported file types are *.mbn files.
- The program must be executed from the SampleApps/Gobi_Image_Management/bin directory
- If you encounter errors when specifying a relative path, specify the fully qualified path instead.
- for more details refer to the readme.txt file

5 Sample Application

5.1 Overview

Information regarding the MC7xx and MC83xx Image Management sample applications can be found in Section 4. Information for the Connection Manager, SMS and Developer Tutorial sample applications is provided below.

5.2 SMS Sample Application

Location:	SampleApps/Gobi_Image_Management/	
Purpose:	Send, read, and delete SMS messages	
Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel
Execute:	i386:	sudo ./SMSSampleAppi386
	ARM:	sudo ./SMSSampleApparm9
	PPC:	sudo ./SMSSampleAppppc
	MIPS BE:	sudo ./SMSSampleAppmips
	MIPS LE:	sudo ./SMSSampleAppmipsel
Reference:	SampleApps/SMSSampleApp/readme.txt	

5.3 SLQS Tutorial Sample Application

Location:	SampleApps/SLQS_Tutorial/	
Purpose:	Familiarize Application Developers with the SDK and provide a starting point for writing an application.	
Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel
Execute:	i386:	sudo ./slqstutoriali386
	ARM:	sudo ./slqstutorialarm9
	PPC:	sudo ./slqstutorialppc
	MIPS BE:	sudo ./slqstutorialmips

MIPS LE: sudo ./slqstutorialmipsel

Reference: n/a

5.3.1 Using the SLQS Tutorial to learn about the SDK

Open two terminals, one for running the application, the other for viewing the message log.
In the message log terminal execute: `tailf /var/log/syslog | grep slqstuotrial`
In the application terminal execute: `sudo ./slqstutorial`

Two example sessions are shown below with interleaved explanations. Messages in green were echoed to /var/log/syslog from a third terminal to explain what is being done.

5.3.1.1 Execution with Root priveleges

```
slqstutorial: Run the Application ( sudo ./slqstutorial )
slqstutorial: cigetnumappclients: count: 1
slqstutorial: wSLQSStart: APP<->SDK IPC init successful
slqstutorial: wSLQSStart: APP registered for Device State Change notification
The application has set the SDK image path, registered for the device state change callback, and
started the SDK i.e. called SLQSStart which creates the SDK process and local IPC sockets.
slqstutorial: Physically Remove the Device
slqstutorial: Device State Change Callback Invoked: rc = 0x0,
slqstutorial: appstatechange: device disconnected, APP disconnected from SDK
slqstutorial: appstatechange: device ready, APP disconnected from SDK
slqstutorial: appstatechange: device ready, APP connected to SDK
The two messages above illustrate that the application will continue to receive device state
change notifications even after calling the QCWWANDisconnect API.
slqstutorial: Attempt to Enumerate the device while it is absent ( Option 1 )
slqstutorial: wQCWWANEnumerateDevices: rc = 0x6, eQCWWAN_ERR_NO_DEVICE
slqstutorial: #devices: 1 deviceNode:  deviceKey:
slqstutorial: wQCWWANConnect: rc = 0x6, eQCWWAN_ERR_NO_DEVICE
Device enumeration has failed as the SDK did not detect a device
slqstutorial: Physically plug in the device
slqstutorial: Device State Change Callback Invoked: rc = 0x1,
slqstutorial: appstatechange: device ready, APP disconnected from SDK
The application is notified of the device state change
slqstutorial: Attempt to Enumerate the device ( Option 1 )
slqstutorial: Enumerate, Connect, Connect/Disconnect device
slqstutorial: wQCWWANEnumerateDevices: rc = 0x0,
slqstutorial: #devices: 1 deviceNode: /dev/qcqmi0 deviceKey: 00000000000000
slqstutorial: appstatechange: device ready, APP disconnected from SDK
Device enumeration is successful but note that the application is still not bound to the SDK (APP
disconnected from SDK.
slqstutorial: Attempt to Connect to the enumerated device  ( Option 2 )
slqstutorial: wQCWWANConnect: rc – 0x0
slqstutorial: appstatechange: device ready, APP connected to SDK
The application is now bound to the SDK (APP connected to SDK)and may therefore issue any API
function hereon.
slqstutorial: Physically remove the device while the application is bound to the SDK
slqstutorial: Device State Change Callback Invoked: rc = 0x0,
slqstutorial: appstatechange: device disconnected, APP connected to SDK
The application is notified of the device state change
slqstutorial: Plug in the device while the application is still bound to the SDK
slqstutorial: Device State Change Callback Invoked: rc = 0x1,
slqstutorial: appstatechange: device ready, APP connected to SDK
The application is notified of the device state change
slqstutorial: Execute some APIs to confirm that the application is still bound to the SDK
slqstutorial: wGetSessionState: rc = 0x0,    ( Option 5 )
slqstutorial: wStartDataSession: rc = 0x0,   ( Option 6 )
slqstutorial: wStopDataSession: rc = 0x0,    ( Option 7 )
Successful execution of APIs as indicated by a return code of 0x0
```

```
slqstutorial: Kill the SDK Process ( option 10 )
slqstutorial: wSLQSKillSDKProcess: rx = 0x0,
SDK process has been terminated (issue ps -eT | grep slqs to confirm the process is no longer
running)
slqstutorial: Restart the SDK process ( option 0 )
slqstutorial: Device State Change Callback Invoked: rc = 0x1,
slqstutorial: appstatechange: device ready, APP diconnected from SDK
The application has set the SDK image path, registered for the device state change callback, and
started the SDK i.e. called SLQSStart which creates the SDK process and local IPC sockets.
slqstutorial: Exit the application  ( option 11 )
slqstutorial: cleanup: Good bye! (0x0)
```

5.3.1.2 *Execution without root priveleges*

```
Note that there must not be an SDK daemon running with root priveleges or you will not see the
same behaviour as described for below. Issue sudo killall slqssdk to make sure this is the case.
slqstutorial: Run the application w/o root priveleges
slqstutorial: cigetnumappclients: count: 1
slqstutorial: wSLQSStart: APP<->SDK IPC init successful
slqstutorial: wSLQSStart: APP registered for Device State Change notification

slqstutorial: wSLQSStart: APP<->SDK IPC init successful
slqstutorial: wSLQSStart: APP registered for Device State Change notification
The application has set the SDK image path, registered for the device state change callback, and
started the SDK i.e. called SLQSStart which creates the SDK process and local IPC sockets.
slqstutorial: Attempt to Enumerate the device ( Option 1 )
slqstutorial: wQCWWANEnumerateDevices: rc = 0xE901, eQCWWAN_ERR_SWIDCS_IOCTL_ERR
Notice that an error is returned because anyone trying to access the /dev/qcqmix device special
file must have root priveleges.
slqstutorial: #devices: 1 deviceNode:  deviceKey:
Since the IOCTL issued by the SDK to the driver fails, the device key is not returned and the
returned values are blank.
slqstutorial: Attempt to Connect to the non-enumerated device  ( Option 2 )
slqstutorial: wQCWWANConnect: rc = 0x6, eQCWWAN_ERR_NO_DEVICE
No device has been enumerated as indicated by the error above
slqstutorial: Attempt to execute other APIs
slqstutorial: wQCWWANGetConnectedDevice: rc = 0x6, eQCWWAN_ERR_NO_DEVICE          ( Option 4 )
slqstutorial: wGetSessionState: rc = 0xE903, eQCWWAN_ERR_SWIDCS_APP_DISCONNECTED  ( Option 5 )
slqstutorial: wStartDataSession: rc = 0xE903, eQCWWAN_ERR_SWIDCS_APP_DISCONNECTED ( Option 6 )
slqstutorial: wStopDataSession: rc = 0xE903, eQCWWAN_ERR_SWIDCS_APP_DISCONNECTED  ( Option 7 )
The application is not bound to the SDK and errors are received as shown above
```

5.4 Connection Manager Sample Application

Location:	SampleApps/Connection_Manager/	
Purpose:	Create, delete, view, and modify profiles. Start/stop data sessions.	
Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel
Execute:	i386:	sudo ./connectionmgri386
	ARM:	sudo ./connectionmgrarm9
	PPC:	sudo ./connectionmgrppc
	MIPS:	sudo ./connectionmgrmips
	MIPS BE:	sudo ./connectionmgrmipsel

5.5 Position Determination Service Sample Application

Location:	SampleApps/PDS_Service/	
Purpose:	Set and Get GPS Service State. Start/stop tracking session.	
Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel
Execute:	i386:	sudo ./pdsservicehosti386
	ARM:	sudo ./pdsservicearm9
	PPC:	sudo ./pdsserviceppc
	MIPS:	sudo ./pdsservicemips
	MIPS BE:	sudo ./pdsservicemipsel

5.6 SWIOMA Sample Application

Location:	SampleApps/SWIOMA_Application/	
Purpose:	Set and Get SWIOMADM setting. Start/cancel SWIOMADM session.	
Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel
Execute:	i386:	sudo ./SWIOMASampleApphosti386
	ARM:	sudo ./SWIOMASampleApparm9
	PPC:	sudo ./SWIOMASampleAppppc
	MIPS:	sudo ./SWIOMASampleAppmips
	MIPS BE:	sudo ./SWIOMASampleAppmipsel

6 Tools

6.1 DM Logging Tool

Location:	/tools/logging/dm
Purpose:	This tool can be used to send DM filters to the device and log raw DM packets for real-time analysis with QPST (remote logging option) or post-hoc analysis (local or remote logging).

Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel

Usage: navigate to **tools/logging/dm** and execute the following in a shell:
./dmcapture.sh

6.2 RAM dump Tool

Location:	/tools/logging/ramdump
Purpose:	This tool supports the capturing of the device RAM contents when the device is in boot and hold mode. RAM contents are saved in files written to the current working directory.

Build:	i386:	make
	ARM:	make CPU=arm9
	Power PC:	make CPU=ppc
	MIPS BE:	make CPU=mips
	MIPS LE:	make CPU=mipsel

Usage:

Prior to Execution

1. Enter the following AT commands:
   at!entercnd="A710"
   at!eroption=0

2. Either reproduce a crash you are investigating, or reset the device

Execution

3. Within a shell, execute the following (for i386):
./ramdumptooli386 /dev/ttyUSB<digit>

/dev/ttyUSB<digit> = DM interface ttyUSB device file in boot and hold mode (usually /dev/ttyUSB0)

NB: This tool works independent of the SDK.

7 SLQS Documentation

To view the SLQS's API documentation:

1. Navigate to **docs/SwiApiReference** and open **index.html**
3. Select on the **modules** tab
4. Select the module of interest e.g. **"Short Message Service (SMS)"** module
5. Select the header file e.g. **"qaGobiApiSms.h"**

NB: An API function header's **"Device Supported"** section contains a list of devices that have been successfully tested against that API.

8 Reference Documents

R-0	SLQS Release Notes
R-1	80-VF459-1 Supplement to Streaming Download Protocol