



CDMA Linux SDK

Developer's Guide



SIERRA
WIRELESS™

2131113
Rev 2

Important Notice

Due to the nature of wireless communications, transmission and reception of data can never be guaranteed. Data may be delayed, corrupted (i.e., have errors) or be totally lost. Although significant delays or losses of data are rare when wireless devices such as the Sierra Wireless modem are used in a normal manner with a well-constructed network, the Sierra Wireless modem should not be used in situations where failure to transmit or receive data could result in damage of any kind to the user or any other party, including but not limited to personal injury, death, or loss of property. Sierra Wireless accepts no responsibility for damages of any kind resulting from delays or errors in data transmitted or received using the Sierra Wireless modem, or for failure of the Sierra Wireless modem to transmit or receive such data.

Safety and Hazards

Do not operate the Sierra Wireless modem in areas where blasting is in progress, where explosive atmospheres may be present, near medical equipment, near life support equipment, or any equipment which may be susceptible to any form of radio interference. In such areas, the Sierra Wireless modem **MUST BE POWERED OFF**. The Sierra Wireless modem can transmit signals that could interfere with this equipment.

Do not operate the Sierra Wireless modem in any aircraft, whether the aircraft is on the ground or in flight. In aircraft, the Sierra Wireless modem **MUST BE POWERED OFF**. When operating, the Sierra Wireless modem can transmit signals that could interfere with various onboard systems.

Note: Some airlines may permit the use of cellular phones while the aircraft is on the ground and the door is open. Sierra Wireless modems may be used at this time.

The driver or operator of any vehicle should not operate the Sierra Wireless modem while in control of a vehicle. Doing so will detract from the driver or operator's control and operation of that vehicle. In some states and provinces, operating such communications devices while in control of a vehicle is an offence.

Limitation of Liability

The information in this manual is subject to change without notice and does not represent a commitment on the part of Sierra Wireless. SIERRA WIRELESS AND ITS AFFILIATES SPECIFICALLY DISCLAIM LIABILITY FOR ANY AND ALL DIRECT, INDIRECT, SPECIAL, GENERAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES INCLUDING, BUT NOT LIMITED TO, LOSS OF PROFITS OR REVENUE OR ANTICIPATED PROFITS OR REVENUE ARISING OUT OF THE USE OR INABILITY TO USE ANY SIERRA WIRELESS PRODUCT, EVEN IF SIERRA WIRELESS AND/OR ITS AFFILIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR THEY ARE FORESEEABLE OR FOR CLAIMS BY ANY THIRD PARTY.

Notwithstanding the foregoing, in no event shall Sierra Wireless and/or its affiliates aggregate liability arising under or in connection with the Sierra Wireless product, regardless of the number of events, occurrences, or claims giving rise to liability, be in excess of the price paid by the purchaser for the Sierra Wireless product.

Patents

This product may contain technology developed by or for Sierra Wireless Inc.

This product includes technology licensed from QUALCOMM® 3G.

This product is manufactured or sold by Sierra Wireless Inc. or its affiliates under one or more patents licensed from InterDigital Group.

Copyright

©2011 Sierra Wireless. All rights reserved.

Trademarks

AirCard® and Watcher® are registered trademarks of Sierra Wireless. Sierra Wireless™, AirPrime™, AirLink™, AirVantage™ and the Sierra Wireless logo are trademarks of Sierra Wireless.

Windows® and Windows Vista® are registered trademarks of Microsoft Corporation.

Macintosh® and Mac OS X® are registered trademarks of Apple Inc., registered in the U.S. and other countries.

QUALCOMM® is a registered trademark of QUALCOMM Incorporated. Used under license.

Other trademarks are the property of their respective owners.

Contact Information

Sales Desk:	Phone:	1-604-232-1488
	Hours:	8:00 AM to 5:00 PM Pacific Time
	E-mail:	sales@sierrawireless.com
Post:	Sierra Wireless 13811 Wireless Way Richmond, BC Canada V6V 3A4	
Fax:	1-604-231-1109	
Web:	www.sierrawireless.com	

Consult our website for up-to-date product descriptions, documentation, application notes, firmware upgrades, troubleshooting tips, and press releases:

www.sierrawireless.com

Revision History

Revision number	Release date	Changes
1.0	Nov 2008	Initial release.
1.1	Dec 2008	<p>Added:</p> <ul style="list-style-type: none"> • List of Figures • Location-based Services on page 63 • PDSM-related notifications (page 33) • Utilities on page 80 • “TECHNOLOGY=CDMA” to the “make -f pkgs.mak” command • Command to remove all objects for all packages (Rebuilding SDK packages on page 82) <p>Changed:</p> <ul style="list-style-type: none"> • Figure 16-1 on page 77 • List of permanently-enabled notifications • build/bin on page 78.
1.2	Apr 2009	<p>Changed:</p> <ul style="list-style-type: none"> • Table 2-1 on page 17. • Opening the host application on page 31 • Checking that the modem is available on page 32 • Startup, normal operation, and shutdown on page 70 • Figure 16-1 on page 77 • Device detection on page 92 (removed sentence that device detection does not work on Linux 2.4 kernels). <p>Added:</p> <ul style="list-style-type: none"> • AirCard 402, MC5728V • Utilities/Common on page 80 • SYMBOLS=ON option (page 81) <p>Removed</p> <ul style="list-style-type: none"> • References to “Version 2” of the SDK. • USB 101.
1.3	Oct 2009	<ul style="list-style-type: none"> • Added support for multiple applications. • Executable is now swisdk. • Replaced Rebuilding SDK packages on page 82. • Added pkgs/cl description for pkgs/ci on page 85. • Miscellaneous edits.
2.0	June 2011	<ul style="list-style-type: none"> • New branding



Contents

About This Guide	15
SDK introduction	15
Scope of this guide	15
System requirements	15
Other reference material	16
Using the API documentation	17
SDK Installation and Setup	19
Installing the SDK	19
Setting up your environment	19
Installing the device driver	19
Distributing files	19
Software architecture	21
Software architecture	21
SDK multiple application support	21
Host application layer	23
Modem driver layer	23
Firmware layer	23
API layer	23
Interaction between components	26
API Initialization, Device Management, and Notifications	27
Using these API functions	27
Examples	28
Initializing the API	28
Handling a modem reset	30
Shutting down the API	30

Host application API usage	31
Opening the host application	31
Checking that the modem is available	32
Handling notifications	33
Closing the host application	34
Activation	35
Activation overview	35
Activation process	35
OMA-DM	36
OTASP	36
IOTA	37
Modem resets	37
Activation information	37
SMS Messaging	39
SMS message overview	39
Receiving SMS messages	39
Sending SMS messages	40
Connection Management	43
Connection management	43
Terminating (answering) a call	43
Call Waiting	43
Originating a call	44
3-Way calling	44
Ending a call	45

Audio Services	47
Voice and audio functions	47
Headset	47
Speaker volume	47
Microphone mute	48
DTMF tone volume	48
Dialing	49
Alerts	49
Security Features	51
Security overview	51
Controlling security	51
Changing the passcode	51
Data Connections	53
Data connection overview	53
Types of data connections	53
Determining modem status	54
Originating a data connection	54
Answering a data call	55
Ending a data connection	55
Characteristics and Status	57
Unit characteristics	57
Power state	58
Power save status	58
Temperature	58
Activation information	58

Service characteristics	58
Signal strength (RSSI)	59
Channel number	59
In service	60
Protocol revision (PREV)	60
Roaming	60
CDMA network time	61
Base station GPS location	61
Statistics	61
Location-based Services	63
PDSM	63
Types of PDSM clients	63
Notifications	63
Getting the location of the terminal	63
Requesting the last known location	63
Requesting a single location fix	64
Downloading ephemeris and almanac data.	64
Clearing ephemeris and almanac data	64
Tracking the location of the terminal	64
Frequent tracking of the terminal host	64
Single location fix	65
Terminating a position determination session	66
Terminating a location fix session	66
Terminating a download session	66
Accessing the default position determination parameters	66
Retrieving the default position determination parameters	66
Modifying the default position determination parameters	67
Getting satellite information	67
Demultiplexing APIs	69
Using these APIs	69

Process model	70
Supported services	70
Startup, normal operation, and shutdown	70
Error handling	73
Error codes.	73
Handling errors.	73
Building an Application	75
Building an application – SDK libraries	75
Package breakdown.	76
Package naming convention	76
Package services	76
SDK directory tree contents	77
Packages	80
SDK Portability	89
Operating System wrapper layer	89
Porting to other versions of Linux.	90
Porting to other embedded operating systems.	90
SDK process model	90
SDK thread model	91
API call handling	91
Device detection	92
USB access	92
Sierra Wireless driver considerations	92
Index	93

List of Figures

Figure 4-1: Software layers	21
Figure 4-2: Request—response patterns	24
Figure 4-3: API/software/hardware interaction	26
Figure 5-1: API initialization when air servers are available	28
Figure 5-2: API initialization when no air servers are available	29
Figure 5-3: Handling modem reset	30
Figure 5-4: Shutting down the API.	31
Figure 5-5: Application startup.	32
Figure 13-1: Initiating a single location fix	65
Figure 14-1: Process model.	70
Figure 14-2: Data flow: Demux application	72
Figure 16-1: SDK file folder structure	77

>> 1: About This Guide

1

SDK introduction

The Linux Software Development Kit (SDK) allows software developers to create Linux-based applications for Sierra Wireless' CDMA products.

The SDK includes:

- This document (the SDK Developer's Guide)
- Application Programming Interface (API) source code and objects, with a supporting online API reference guide describing API functions, data structures, constants, and files
- Sample programs and utilities showing how to use several API calls
- The Linux SDK Integration Guide
- Licence for Software Tools & Software Development Kits—You are required to accept the terms of this license before downloading the SDK and distributing the SDK with any products developed with it. The license can be found in \$INSTALL_FOLDER/docs (for information on \$INSTALL_FOLDER, see [page 19](#)).

Note: The SDK does not include the Linux driver. The driver is open source and is distributed independently of the SDK. For information on obtaining and installing the driver, refer to the Linux SDK Integration Guide included in the SDK.

Scope of this guide

This guide describes system requirements for installing and using the SDK, and how to perform typical tasks from the modem's feature set. It is up-to-date with the SDK it is bundled with.

It does not describe the use of the AT, USB, or other interfaces. For more information, refer to the appropriate product-specific references available at www.sierrawireless.com.

System requirements

Sierra Wireless technical support is available for x86 PCs using Ubuntu Linux 8.04 and ARM9 using Debian Sarge, as detailed in [Development systems](#) on page 16. For information on other combinations of architecture and Linux kernel, contact Sierra Wireless Professional Services.

Development systems

- PC: Ubuntu 8.04 on x86/32 architecture — minimum recommended configuration¹:
 - 700 MHz x86 processor
 - 384 MB of system memory (RAM)
 - 8 GB of disk space
 - Graphics card capable of 1024x768 resolution
 - A network or Internet connection
- ARM9: Technologic Systems TS-7800 with Debian Sarge kernel. (For more information and where to buy, see [Other reference material](#) on page 16.)
- Any language that can call C-language functions. (The SDK is coded in the C-language and exports its APIs using standard C-language prototypes.)

Other reference material

Additional references that may help you use the SDK to develop your applications include:

- Sierra Wireless (www.sierrawireless.com)—Product specifications for your products, glossary of terms and acronyms, etc.
- Technologic Systems (www.embeddedarm.com/product/board-detail.php?products=TS-7800)—Information on the TS-7800 SBC and how to order it.

For some background on the CDMA 1X and 1xEV-DO data networks, we suggest you consult web sites including (but not limited to):

- CDMA Developers Group (www.cdg.org)
- QUALCOMM (www.qualcomm.com/cdma/)
- Carrier infrastructure manufacturers:
 - Ericsson (www.ericsson.com)
 - Lucent (www.lucent.com)
 - Motorola (www.mot.com/home/)
 - Nortel Networks (www.nortelnetworks.com)
- Spread Spectrum Scene magazine (www.sss-mag.com)

1. From the Ubuntu website www.ubuntu.com

>> 2: Using the API documentation

When you install the Linux SDK, it unpacks an online API reference guide into its own directory. The online API reference contains function prototypes, structure descriptions, and other useful information. You can access this directory using a standard browser such as Firefox. To access this reference:

1. Start the browser.
2. Select **File > Open** and navigate to `$INSTALL_FOLDER/docs/SwiApiReference`, where `$INSTALL_FOLDER` is the directory the SDK was placed in during installation. (See [Installing the SDK](#) on page 19.)
3. Open the `Index.html` file.

This is the main page of the API reference. Use the tabs at the top of the page to navigate to information on structures, APIs, files, and more.

[Table 2-1](#) describes the hyperlinks found on the `Index.html` page.

Table 2-1: Locations of information in API documentation

To find this ...	Look in this area of the documentation
Data structures	Data Structures tab
Defines	Files > Globals > Defines
Enumerations	Files > Globals > Enumerations
Functions	Files > Globals > Functions
Header files	Files > File List
Typedefs	Files > Globals > Typedefs
Variables	Files > Globals > Variables

>> 3: SDK Installation and Setup

Sierra Wireless Technical Support delivers the SDK as a .tar file. (See [Contact Information](#) on page 4.)

Installing the SDK

Note: These instructions are for the installation of the SDK on a Linux system. A different SDK is available from Sierra Wireless for use with Windows systems.

Note: Installing the SDK over an older version is not recommended.

If you are upgrading from a previous version of the SDK, uninstall and delete all the files in the SDK folder (\$INSTALL_FOLDER from Step 1), then install the later version.

To install the SDK on a Linux system:

1. Extract the contents to a destination folder (any directory where you have read/write access serves as a suitable destination: /home/<user>/Sierra/sdk, where: <user> is an existing user account directory on the file system). This is referred to as \$INSTALL_FOLDER throughout this document.

All SDK components (header files, documentation, libraries, etc.) install under the \$INSTALL_FOLDER in meaningfully-named folders.

Setting up your environment

Set your development environment to locate the files required for your host application's platform and operating system. These files are found in folders below the \$INSTALL_FOLDER. For details of the directory contents, see [SDK directory tree contents](#) on page 77.

Installing the device driver

Information and instructions for installing the device driver are available in the Linux SDK Integration Guide.

Distributing files

When you distribute your host application to users, you must consider where to put a copy of the swisdk executable. When you start up your application(s), they must specify the path where this executable is located.

>> 4: Software architecture

This chapter describes how the API interfaces with the host application and modem and how the modem driver interfaces with Linux.

Software architecture

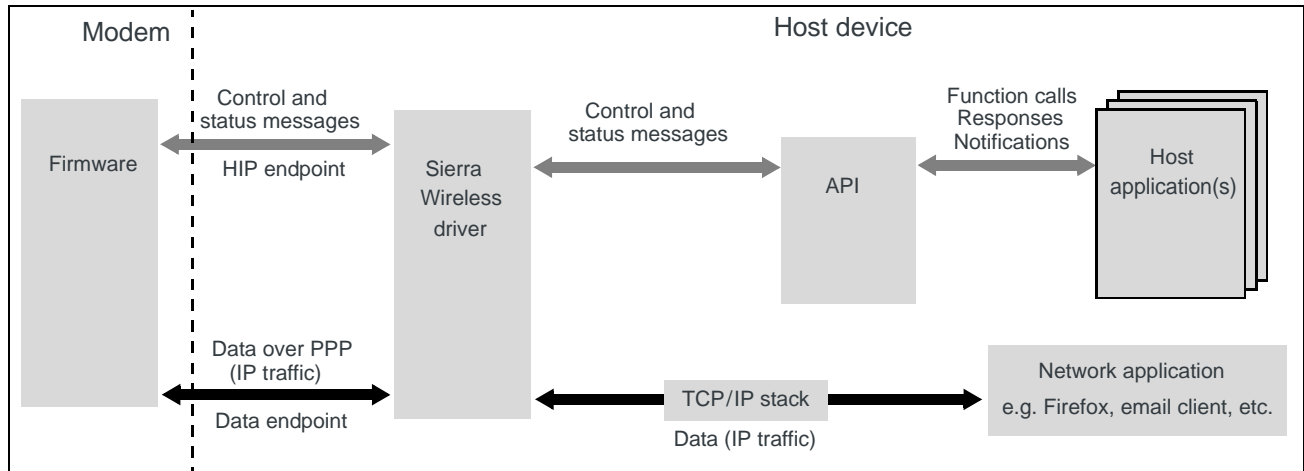


Figure 4-1: Software layers

The API facilitates the exchange of control and status messages between your host application and the modem. These messages pass through the four software layers described in the following sections.

SDK multiple application support

The SDK (release 1.3.0.0 and higher) supports simultaneous, transparent API access by multiple applications. The SDK manages the multiple connections to the modem, and each application uses the APIs as though it is the only active application.

When the SDK daemon process starts, it opens pairs of IPCs (inter-process communication channels) for the maximum number of possible concurrent applications. For example, release 1.3.0.0 (and higher) supports five simultaneous applications, so five IPC pairs are opened. IPC management (processing of modem notifications and API requests/responses) is encapsulated within API functions, so applications do not need to manage the channels directly.

Host application requirements/considerations

To manage simultaneous API access by multiple applications, applications must be compiled using static linking of SDK libraries—dynamic linking is not supported.

Each application must follow the required startup sequence (as described in [API Initialization, Device Management, and Notifications](#) on page 27), and interact with APIs in the same way. For example:

- Applications must call **SwiApiStartup()** to be assigned an available IPC channel pair. If no pairs are available (the maximum number of applications are already accessing the SDK), the call returns **SWI_RCODE_BUSY**.
- Applications must maintain separate threads for handling API calls and notification traffic, as described in [API layer](#) on page 23.
- Applications must not make API calls until they receive a **SWI_NOTIFY_AirServerChange** notification, or **SwiGetAvailAirServers()** indicates the modem is up and running.

Note: Developers should read the remainder of this document before implementing designs, to ensure API call sequences and restrictions are understood.

Applications must include a timeout value in each API call. This timeout period takes effect when the SDK daemon starts processing the request. However, API requests are handled by the SDK daemon on a first-come-first-served basis. Therefore, the maximum time an API request will take to complete (by timing out) may be greater than the requested timeout value: <time to process prior requests> + <requested timeout>.

Most API calls are 'stop and wait' so the calling application is blocked until a response is received or the call times out. If multiple applications are accessing the modem, the overall CPU load remains basically unchanged, but RAM is consumed for each application, which may be an issue for some embedded platforms.

Concurrent application limits

The SDK source code limits the number of concurrent applications (five, in release 1.3.0.0 and higher). Modifying this number and recompiling is allowed, but SWI makes no performance guarantees.

Note that if you will be running any of the SDK utilities (for example, Relay Agent) or Sample Code applications, these count against the maximum number of concurrent applications.

When performing firmware upgrades, only the upgrade utility should be running. At this time the modem is in the boot loader, and attempting to run other applications concurrently could result in excessive errors being logged, or unexpected behavior occurring.

Host application layer

Note: As described in [SDK multiple application support](#) on page 21, multiple applications can access APIs simultaneously. The SDK daemon handles this transparently—each application deals with APIs as though it is the only application running.

This is the client application that you develop to control the modem and present a user interface. Your application communicates with the modem using the functions and data structures of the API.

Note: The Linux driver creates a separate file handle in the `/dev` directory for each interface enumerated by the modem. These handles are similar to COM ports on a Windows machine and external programs can open them directly and exchange information with the Sierra Wireless modem. However, this SDK also provides a mechanism for accessing the services multiplexed over these ports through a set of APIs (see [Demultiplexing APIs](#) on page 69). This method saves programs from the overhead of having to figure out which file handle is associated with a particular service and also manages the driver, releasing handles whenever the modem resets.

Modem driver layer

Modem driver software is installed on the host machine to provide the interface between the modem and the Linux operating system. For information on installing the driver, refer to the *Linux SDK Integration Guide* included in the SDK.

Firmware layer

The firmware manages data traffic between the modem and the cellular network. It can be upgraded as new releases become available. (Firmware releases are posted on the Sierra Wireless web site, www.sierrawireless.com as part of new software downloads.)

API layer

Note: Data traffic is not handled by the API. All data traffic is handled by an application (like Firefox) communicating directly with the TCP stack. It is also the external application's responsibility to take the actions necessary to establish the data connection.

The API layer includes several types of files on the host system. These files provide functions and data structures that your host application uses to communicate with the modem. This communication takes the form of:

- Host-initiated requests/responses (symmetric notifications)
- Modem-initiated notifications (asymmetric notifications)

[Figure 4-2](#) on page 24 illustrates the data flow patterns of a function that receives only a return code, and a function that receives a return code and a symmetric notification. Both patterns are used depending on the operation being performed on the modem.

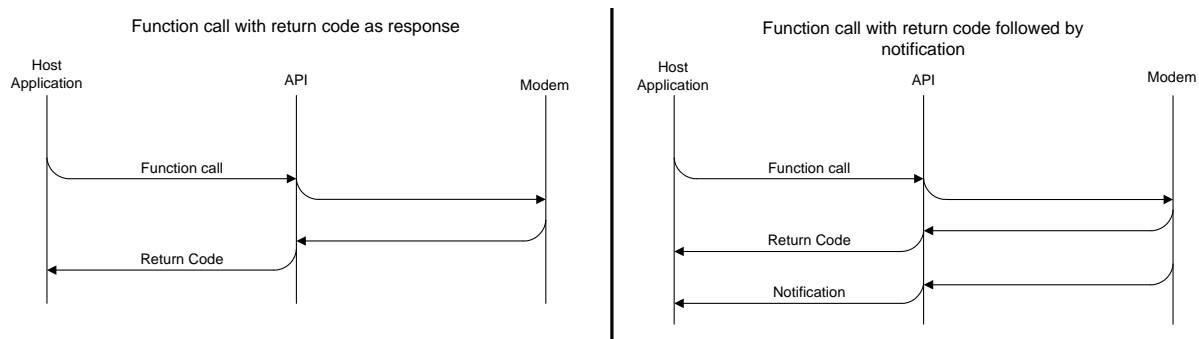


Figure 4-2: Request—response patterns

Host-initiated requests / responses

*Note: Where possible, use available notifications (asymmetric) to monitor modem status, rather than polling with function calls. For example, register to receive the notification **SWI_NOTIFY_Rssi_EcIo** rather than frequently calling the function **SwiGetRssiEcIo**.*

API function calls have the following characteristics:

- **Requests**—Host-to-modem function calls passed to the modem via the modem driver. Each function call returns a standard return code (see **SwiRcodes.h** in the API online reference guide).
- **Responses (symmetric notifications)**—Some function calls receive a response (modem-to-host message) containing additional information after the return code is sent. The API online reference guide indicates, for each function, if such notifications will be received. The API indicates if you have to wait for the notification (or for it to time out) before calling related functions. For example, the host may have to wait for call notifications before calling additional call-related functions, but can call other functions (such as RSSI) while it waits.
- **Timeouts**—Each call requires a non-zero timeout value. If the modem doesn't respond to the request in time, the API returns a timeout error code to your application; if the modem responds after the function times out, the API ignores the response. (Make sure that you set your timeout values appropriately—3000 ms is recommended as an initial setting, which you can adjust during development.)

Modem-initiated notifications

Asymmetric notifications have the following characteristics:

- The host application must explicitly register to receive most desired notifications, otherwise the messages are ignored. The only ones that do not have to be registered are API-driven notifications (such as **SWI_NOTIFY_AirServerChange**). Refer to the API online reference guide for details of all available notifications.

Note: Notifications are the preferred method for managing the modem. This lets the modem operate at optimum power savings by not continually polling the modem for responses.

Note: When the host registers for a particular notification, the modem often issues a notification immediately in response. This allows applications to synchronize with the current status of the modem. Then, during runtime, the modem generates additional notifications in response to significant changes to the item being measured.

- These notifications occur when specific system state changes occur (for example, available networks, modem temperature, etc.).

Managing notifications

To manage notifications from the API, your application must:

- Create a separate thread on startup especially for handling notifications. Call the special API function **SwiApiWaitNotification**—note this function never returns, so your new thread should not be given any other jobs to do beyond handling notifications via this function call.
- Register a callback function: Use **SwiRegisterCallback** to register a function that takes action on all event types that your application enables for notification. The callback function is executed in the context of the notification thread (discussed in the previous bullet).
- Use appropriate notification API calls to enable the notifications required by your application. (You can also disable notifications if necessary.) See [Handling notifications](#) on page 33 for details.

Note: If the modem resets unexpectedly, notifications are disabled. The host application must detect this condition and explicitly re-enable the notifications.

Interaction between components

Figure 4-3 on page 26 shows the interaction between the API and other software and hardware components.

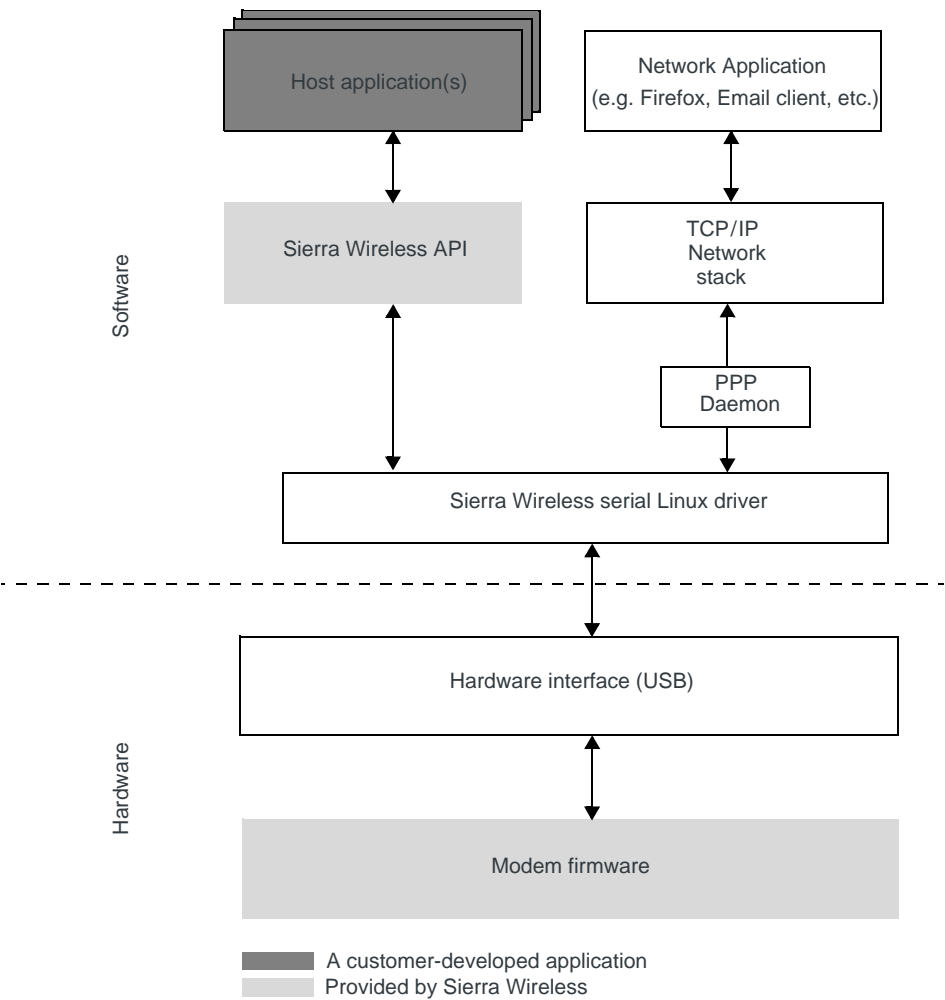


Figure 4-3: API/software/hardware interaction

>> 5: API Initialization, Device Management, and Notifications

Note: You must perform the steps (described in this and the following chapters) for each application that uses the associated service.

This chapter describes how to use API functions to perform the following tasks:

- Initialize or shut down the API sub-system
- Identify the available air server, get information on it and connect to it
- Prepare the host application to work with the API, including handling notifications

The Linux API provides support for a single application interacting with a single modem attached to a host computer.

Using these API functions

The following are key usage notes for functions described in this chapter (see the API online reference guide for specific details about these functions and structures):

- **SwiApiStartup:** Call this function first to initialize the API sub-system before using any other API calls.
 - As part of this function call you must provide a path to the SDK executable image . For details, see **SWI_STRUCT_ApiStartup** (in **SwiApiCmBasic.h**).
 - Once initialized, and before proceeding to use the main APIs, your application should call **SwiGetAvailAirServers** and possibly **SwiGetBootAndHoldMode**.
 - Do not call it again. (You do not have to call it again when a device is removed.)
- **SwiRegisterCallback:** Always call this function immediately after you register with an air server.
- **SwiGetAvailAirServers:** This returns an empty list when there are no available air servers.
- **SwiNotify:** Enable each notification that the callback should handle (see **SwiApiNotify.h** in the API online reference guide for the list of available notifications). If the modem resets, you must re-enable the notifications after you re-register the callback function.

Examples

The following flow diagrams illustrate common situations addressed using these modules.

Initializing the API

When air servers are available Initialize the API and bind the modem to a specific air server.

1. Call **SwiApiStartup** to initialize the API (enabling device detection).
2. Call **SwiRegisterCallback** to register a callback function to receive API notifications.
3. Call **SwiGetAvailAirServers** to get a list of all available air servers.

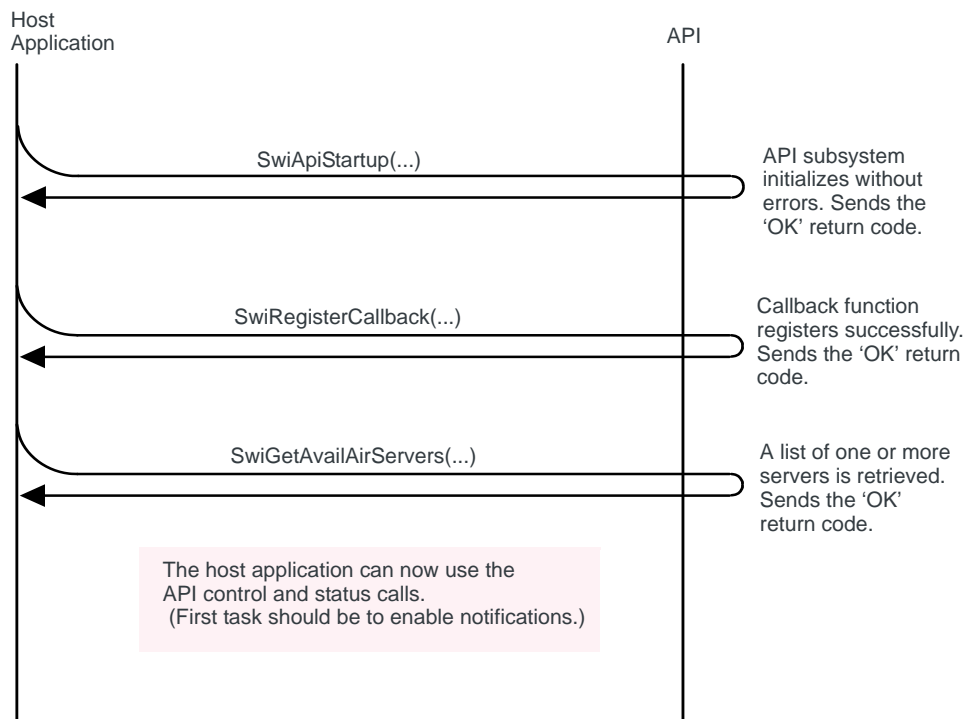


Figure 5-1: API initialization when air servers are available

When no air servers are available After initializing the API, if there is no air server available, wait for the **SWI_NOTIFY_AirServerChange** notification. After receiving that notification it is OK to use the APIs.

1. Call **SwiApiStartup** to initialize the API (enabling device detection).
2. Call **SwiRegisterCallback** to register a callback function to receive API notifications.
3. Call **SwiGetAvailAirServers** to determine if there is an air server connected to the computer. If there is not, then wait for the notification **SWI_NOTIFY_AirServerChange**. When that notification arrives, proceed to the next step.

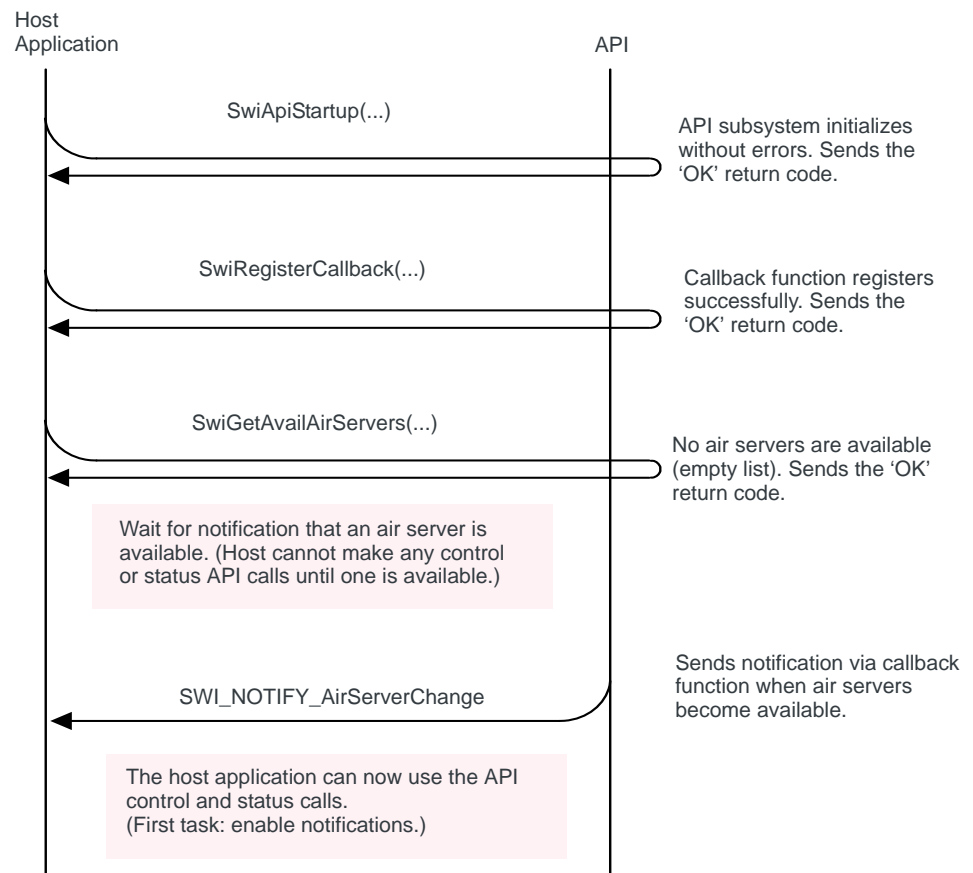


Figure 5-2: API initialization when no air servers are available

Handling a modem reset

When the modem is reset the modem drivers are unloaded and reloaded.

When the drivers unload and reload:

1. The host receives a **SWI_NOTIFY_AirServerChange** notification indicating the drivers have unloaded.
2. The host must wait for another **SWI_NOTIFY_AirServerChange** notification indicating the drivers have reloaded.
3. Re-enable the API notifications that are handled by the callback function.

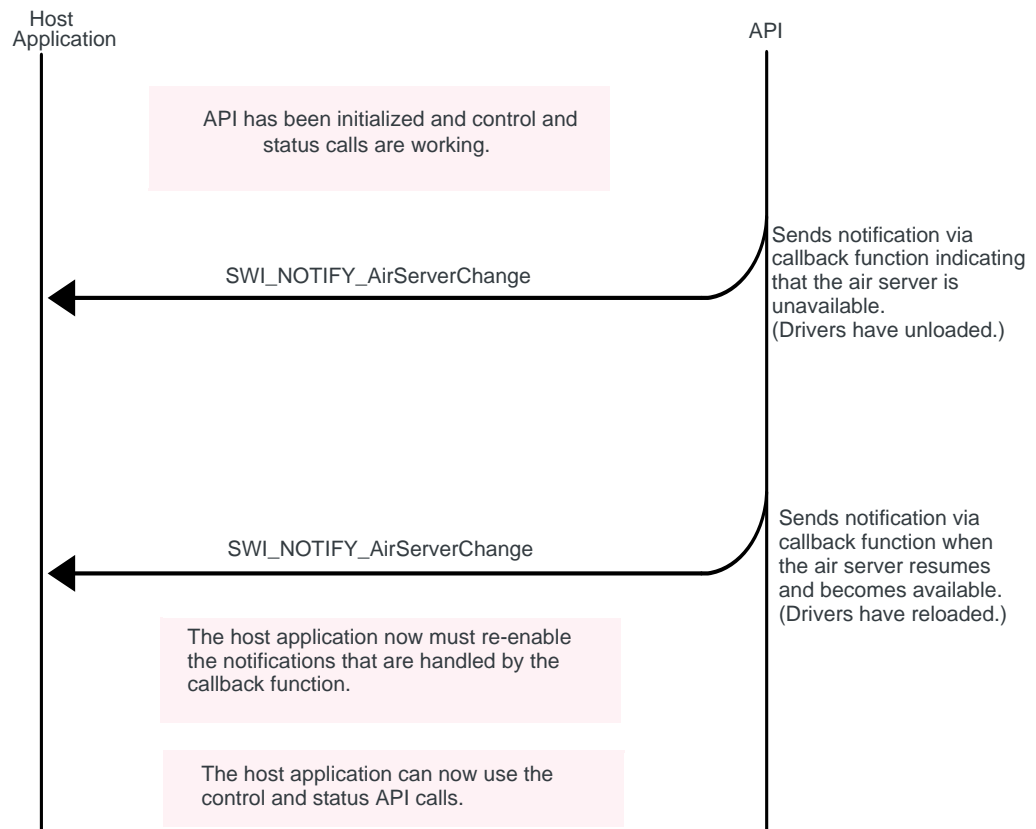


Figure 5-3: Handling modem reset

Shutting down the API

When ready to shut down the API, disable notifications, deregister the callback function, and then shutdown the API.

1. Call **SwiStopNotify**. Multiple calls may be required—one call to disable each individually-enabled or each group -enabled notifications. (For information on group-enabled notifications, see [Enabling notifications](#) on page 33.)
2. Call **SwiDeRegisterCallback** so the API will stop using the callback function.

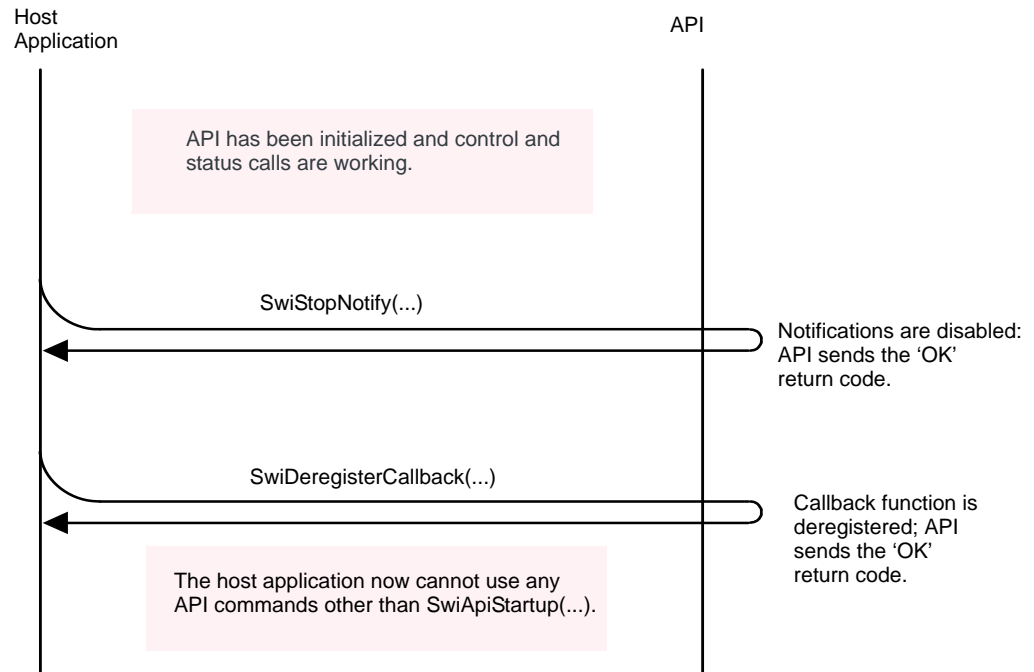


Figure 5-4: Shutting down the API

Host application API usage

Opening the host application

Typically, on start-up, the host application should:

1. Open the API (**SwiApiStartup**).

If **SwiApiStartup** fails, you must close the API (see [Shutting down the API](#) on page 30) before you call **SwiApiStartup** again.

Note: Up to five applications can interact with the modem at the same time using these APIs.

2. Immediately register a callback function (**SwiRegisterCallback**) to handle notifications, as shown in [Figure 5-2](#) on page 29.
3. Retrieve the list of available modems (**SwiGetAvailAirServers**).
4. Check that the modem is available (**SwiIsModemReady**).
5. Enable desired notifications (**SwiNotify**).
6. Continue with normal application operations.

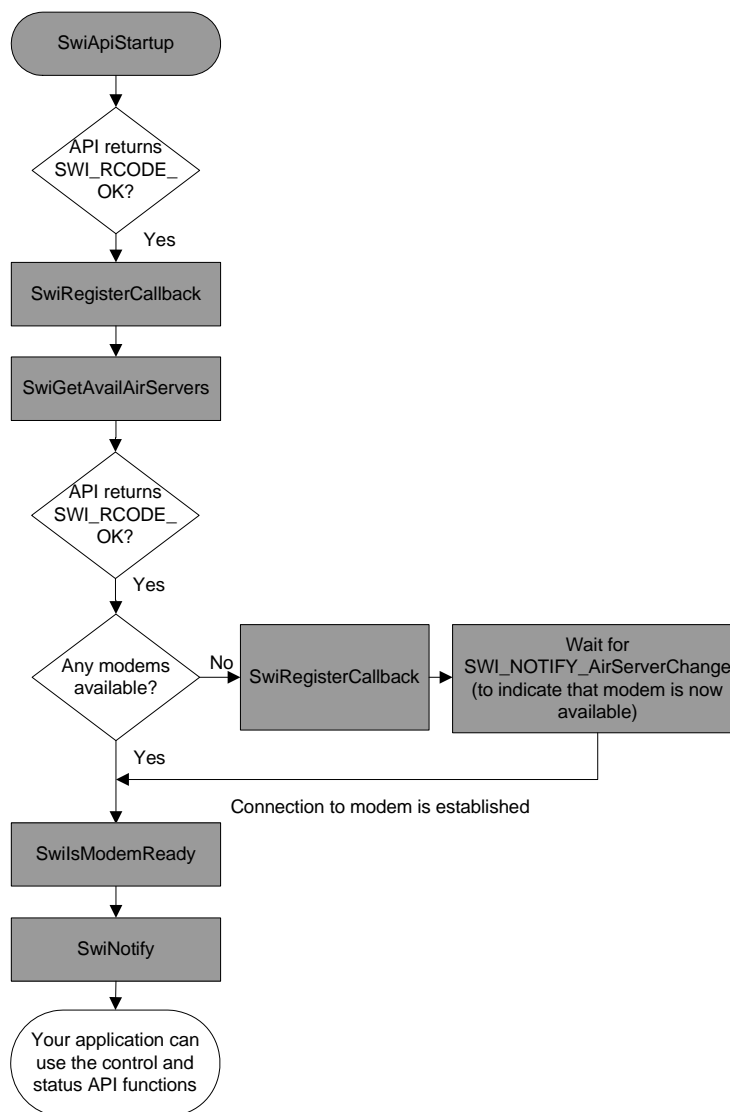


Figure 5-5: Application startup

Checking that the modem is available

If **GetAvailAirServers** indicates that no modems are available, then, instead of repeatedly calling **SwiGetAvailAirServers**, it's recommended that your application monitor for **SWI_NOTIFY_AirServerChange** (a callback function must be registered with **SwiRegisterCallback**).

(Callback registration is with the API and does not directly access the modem.)

The host application can perform this check by calling **SwiIsModemReady**.

Handling notifications

Enabling notifications

Call **SwiNotify**, specifying the notifications required for your application.

You must have a registered callback function before you can receive any of these notifications.

You must have created a separate thread in your application for receiving and handling notifications. Create the thread and from inside it, call **SwiApiWaitNotification**. Note that this function never returns, so the thread should not be assigned any other responsibilities.

*Note: The notification **SWI_NOTIFY_AirServerChange** is always enabled. Most others are disabled when the modem powers up or is reset.*

Generally, notifications are explicitly enabled (**SwiNotify**) and disabled (**SwiStopNotify**) by your application. By default, following a reset or modem startup, the notifications are disabled. A small group of notifications are always enabled.

The permanently enabled notifications are:

- **SWI_NOTIFY_AirServerChange**

The following call management notifications are enabled and disabled as a group, using the single notification type **SWI_NOTIFY_CallNotificationStatus**:

- **SWI_NOTIFY_CallNotificationStatus**
- **SWI_NOTIFY_CallConnected**
- **SWI_NOTIFY_CallIncoming**
- **SWI_NOTIFY_CallDisconnected**
- **SWI_NOTIFY_CallError**
- **SWI_NOTIFY_CallDormant**
- **SWI_NOTIFY_OTASPState**

The following notifications (related to PDSM location session data) are enabled and disabled as a group, using the function **SwiSetLbsLocNotifyStatus**:

- **SWI_NOTIFY_LbsPdBegin**
- **SWI_NOTIFY_LbsPdData**
- **SWI_NOTIFY_LbsPdEnd**
- **SWI_NOTIFY_LbsPddone**
- **SWI_NOTIFY_LbsPdUpdateFailure**

To determine which of the above notifications are enabled, call the function **SwiGetLbsLocNotifyStatus**.

The following notifications (related to PDSM Data download) are enabled and disabled as a group, using the function **SwiSetLbsDloadNotifyStatus**:

- **SWI_NOTIFY_LbsPdDloadBegin**
- **SWI_NOTIFY_LbsPdDloadData**
- **SWI_NOTIFY_LbsPdDloadEnd**
- **SWI_NOTIFY_LbsPdDloadDone**

To determine which of the above notifications are enabled, call the function **SwiGetLbsDloadNotifyStatus**.

The following notifications (related to changes made to LBS [Location Based Services] NV items) are enabled and disabled as a group, using the function **SwiSetLbsParamNotifyStatus**:

- **SWI_NOTIFY_LbsPaIpAddr**
- **SWI_NOTIFY_LbsPaGpsLock**
- **SWI_NOTIFY_LbsPaPtmMode**
- **SWI_NOTIFY_LbsPaPortID**
- **SWI_NOTIFY_LbsPaPrivacy**
- **SWI_NOTIFY_LbsPaNetAccess**

To determine which of the above notifications are enabled, call the function **SwiGetLbsParamNotifyStatus**.

All other notifications require individual enabling and disabling. Your application must selectively enable the notifications it wants to receive by calling **SwiNotify** with the event type(s) to enable.

If notifications are enabled when there is no registered callback function, the API silently discards them as they arrive.

To determine which HDR (1xEV-DO) notifications are enabled, call the function **SwiGetHdrNotificationStatus**.

To determine which Tech HDR (1xEV-DO) notifications are enabled, call the function **SwiGetTechHdrNotificationStatus**.

The notification equivalent **SWI_NOTIFY_TechHdrNotificationStatus** indicates which Tech HDR (1xEV-DO) notifications are enabled. This is preferred over the polling command.

Disabling notifications

Should your application (or user) no longer need notifications of an event, they can be disabled by calling **SwiStopNotify**.

Processing notifications

When an enabled event notification occurs:

1. API calls the registered callback function, passing the event data in the proper structure to the host application.

Note: When you receive a notification, cache the information passed by the notification and return from the callback function as quickly as possible to allow the thread to resume monitoring for new notifications.

Closing the host application

When the host application closes, disable notifications from the modem and possibly shut it down as well—see [Disabling notifications](#) (above).

Activation overview

The CDMA modems support one user account, called NAM (Number Assignment Module). A NAM account profile is composed of several elements:

- MDN (Mobile Directory Number)—the phone number of the mobile device
- MIN (Mobile Identification Number) that identifies the unit within the service provider's network; part of a larger CDMA specified identity called the IMSI (International Mobile Station Identity). The IMSI is also known as MSID by some providers.
- SID (System Identifier)
- NID (Network ID)

To log onto the service provider's data services, you may need a user ID and password. (Check the activation and data connection requirements for the target carrier.)

The SDK includes the SwiActivate sample application (in Sample_Code/cdma).

Activation process

Activation can be made through a combination of [OMA-DM](#), [OTASP](#), [IOTA](#), and/or "manually" through your application. The data and activation process required for activation varies between service providers. Any data not required by the service provider will have been entered at the factory.

To manually activate an account:

1. Report the ESN or MEID to the user via **SwiGetESN** or **SwiGetMEIDESN**. The user provides this number to the service provider; the service provider then provides a number (MSL, OTSL, or SPC) to enable the activation functions.
2. Unlock the activation access by passing the number reported by the service provider to **SwiPstUnlock**.
3. Enter the phone number (MDN) via the function **SwiSetMobileDirNum**.

Note: Simple IP User ID and password can be set or changed at any time. The modem does not require unlocking or resetting to make these changes.

4. Store the Simple IP user ID and password (needed for carrier networks that support Simple IP for packet data connections) using **SwiPstSetSipUserId** and **SwiPstSetSipPassword**, respectively.
5. Enter the IMSI (also known as MSID by some providers), if required by the service provider, using **SwiSetIMSI**. This function is particularly important for implementation of Wireless Local Number Portability (WLNP), separating the MDN (step 3) from the IMSI.
6. Enter a table of home SID and NID numbers (if required by the service provider) using **SwiPstSetHomeSidNid**.
7. Commit (or cancel) the changes by calling **SwiPstLock**.
8. Reset the modem using **SwiResetModem** to have the changes take effect.

Note: After the reset, you should reenable desired notifications. (See [Enabling notifications](#) on page 33.)

The provisioning date (**SwiGetProvisioningDate**) is set whenever the NAM profile is written. The date comes from the CDMA network at the time the account profile is written, or after CDMA coverage becomes available.

OMA-DM

If the modems are used on networks of service providers that use OMA-DM, the modem can be requested to initiate an OMA-DM session with the network by calling **SwiStartOMADMSession** when a good RSSI signal is available on the home network of the service provider. The OMA-DM session can be cancelled at any time by calling **SwiCancelOMADMSession**. Notification and status of an OMA-DM session (initiated by the client via **SwiStartOMADMSession** or initiated by receiving a **SWI_NOTIFY_OMADM_NI_Alert** notification) is available by registering for OMA-DM status notifications **SWI_NOTIFY_OMADM_State** and **SWI_NOTIFY_OMADM_NI_Alert**.

OTASP

OTASP (Over The Air Service Provisioning) is a method to remotely configure and update the modem. The user calls a special OTASP number (typically *228xx) and the modem then automatically enables OTASP mode, which means that it is ready to receive configuration and PRL updates. During such a call, the modem receives **SWI_NOTIFY_OTASPState** notification, which provides additional information while the call progresses. A successful OTASP session always consists of several new configuration items (for example, PRL and NAM) followed by a successful “commit”. Without the “commit”, the new configuration is not permanently saved in modem, and the OTASP session has to be considered as having failed.

When initiating an OTASP call, set the **CallType** to Voice (**SWI_CALL_TYPE_Voice**). The connection process can be tracked using API notifications **SWI_NOTIFY_CallConnecting** and **SWI_NOTIFY_CallConnected**.

IOTA

If the modems are used on networks of service providers that use IOTA, the modem can be requested to initiate an IOTA session with the network by calling **SwiStartIOTASession** when a good RSSI signal is available on the home network of the service provider (not currently roaming). The IOTA session can be cancelled at any time by calling **SwiStopIOTASession**.

Notification and status of an IOTA session initiated by the client via **SwiStartIOTASession** or initiated by a WAP trigger received by the network (and processed internally by the modem) is available by registering for IOTA status notifications (**SWI_NOTIFY_IOTA_Status**). In addition, log files for the most recent IOTA session attempt can be read out of the modem via **SwiGetIOTALog**.

The SDK includes the Swilota sample application (in Sample_Code/cdma).

Modem resets

SwiResetModem triggers a modem reset. The API remains open, however notifications are disabled and need to be re-enabled. (See [Enabling notifications](#) on page 33.)

Your application should allow 15 seconds for the modem to complete the reset, self-test, and start-up.

A reset is required to have account profile changes take effect.

Activation information

Account

To read the NAM account profile, use **SwiPstGetCurrentNam**.

Activation state

Upon application startup, you may want to query **SwiGetActivationStatus** to determine the activation status of the current account in the modem. This function can be used to determine if the modem has been activated with a NAM account profile. If not, your application should offer the user a mechanism to activate the modem with a service provider, through a combination of OMA-DM, OTASP, IOTA, and/or direct data entry (depending upon the activation requirements of the target carrier).

Phone number

SwiGetPhoneNumber returns the phone number from the NAM account profile. The phone number is returned as a null-terminated ASCII string of 10–15 numerals.

Simple IP username (user ID)

SwiPstGetSipUserId returns the Simple IP username (user ID) used for packet data connections that use Simple IP.

SwiPstSetSipUserId allows you to modify the Simple IP username (user ID).

NAI (1xEV-DO)

SwiPstGetHdrUserId returns the HDR (1xEV-DO) User ID—also known as the NAI (Network Access Identifier).

SwiPstSetHdrUserId allows you to modify the HDR User ID.

Simple IP password

SwiPstGetSipPassword returns the Simple IP password used for packet data connections that use Simple IP.

SwiPstSetSipPassword allows you to modify the Simple IP password.

HDR (1xEV-DO) password

SwiPstGetHdrPassword returns the HDR (1xEV-DO) password.

SwiPstSetHdrPassword allows you to modify the HDR (1xEV-DO) password.

SMS message overview

The modems support both sending and receiving SMS messages—known as mobile-originated SMS (MO-SMS) and mobile-terminated SMS (MT-SMS) respectively. However, not all service providers or accounts permit SMS services.

The modem reads a setting in the Product Release Instructions (PRI) for indication of services that are blocked. If a function call is made to a blocked service, such as MO-SMS, the caller receives the general FAILED return code. No method is available to confirm the reason for the failure; you have to know the limitations of the account or device.

The SDK includes the SwiSMS sample application (in `Sample_Code/cdma`).

Receiving SMS messages

SMS messages arrive from the network and are stored on the modem. The modem's storage is limited, and it is up to your application to clear the message queues in a timely fashion.

The modem can be configured to notify your application when a message arrives. To enable this feature, use **SwiNotify** with the event type **SWI_NOTIFY_SmsStatus**. Alternatively, your application can poll the modem using **SwiGetCdmaSMSMessageStatus**, although this mechanism is strongly discouraged in favor of the notification.

The recommended process is to use **SwiGetCdmaSMSMessageStatus** when your application starts, then rely on notifications from that point on.

To manage incoming SMS messages, the following procedure is recommended:

1. On application startup (after registering the callback function and verifying the modem is responding), use **SwiGetCdmaSMSMessageStatus** to determine if there are any messages on the modem. If there are none, then skip to step 3. **SwiGetCdmaSMSMessageStatus** reports the status of the modem's storage for incoming SMS messages. The modem stores messages in three queues:
 - Urgent (`cntUrgentMsg`) as determined by the SMS message header
 - Regular (`cntRegularMsg`) as set by the message header
 - Voice (`cntVoiceMsg`), which is either 0 or 1, indicating the absence or presence of one or more voice messages in the users normal voice mail.

2. Retrieve all messages using **SwiRetrieveCdmaSMSMessage**. Call the function repeatedly until the field **cntRemainingSMS** in **SWI_STRUCT_CDMA_SMS_RetrieveSms** reports zero.

The modem reports messages from each of the three queues in the sequence: urgent, regular, voice. Within each queue, the modem reports the most recently arrived message first and works its way to the oldest message in the queue. When a queue is empty, the next queue is accessed.

The normal process for reading all messages from the modem is implemented in the sample code. This procedure is designed to ensure all messages get read from the modem. The intention is that your application maintain a database of all messages as each message is automatically deleted from the modem after retrieval.

3. Enable notifications for additional incoming messages via **SwiNotify**, using the event type **SWI_NOTIFY_SmsStatus**.

When the modem notifies your application that there are messages in the modem, your application should read through all messages, as described in step 2 above.

For details on how this algorithm is implemented, consult the sample code **SwiSmsCDMA**.

Sending SMS messages

The process of sending an SMS involves several steps:

1. Compose a message in the host application and create the SMS header (**SWI_STRUCT_CDMA_SMS_StoreSms** and **SWI_STRUCT_SMS_CdmaHeader**).
2. Store the message on the modem, similar to placing it in an outbox, via **SwiStoreCdmaSMSMessage**. Retain a copy in your application in case a retry is needed.

The modem can store several SMS messages in the outbox before you need to send them to the network. This storage step can be repeated until an error code is returned (indicating the outbox is full).

The multi-message outbox feature can save call setup and tear-down time when several messages will be sent at once. This is particularly useful for sending one message to a group of recipients.

3. Send the outbox message(s) to the network via **SwiSendCdmaSMS**.

When sent, the message is removed from the modem's outbox.

The notification **SWI_NOTIFY_SmsSendStatus** is provided to report the message transmission confirmation from the network. Only after confirmation by the network should your application delete its retained copy of the sent message.

*Note: Timeouts of **SwiStoreCdmaSMSMessage** can result in the message being stored but not acknowledged in the function return code. If a timeout occurs, call **SwiSendCdmaSMS** to send stored messages, and monitor the notification **SWI_NOTIFY_SmsSendStatus** for acknowledgement of the message. If the message is acknowledged, then the store was successful. If the message is not acknowledged in a reasonable time, then repeat the store and send process.*

Your application can call **SwiStoreCdmaSMSMessage** while not in coverage, but this is not recommended. The stored message remains in the modem until the earliest of these events:

- The message is sent via **SwiSendCdmaSMS**
- The modem resets
- The modem is power-cycled

Your application must control the store-and-send process to prevent message loss.

The network confirmation of sent messages is handled through the notification **SWI_NOTIFY_SmsSendStatus**. The full store-and-send process should be redone for any failed messages.

Sending a stored message while the modem is not in coverage generates the **SWI_NOTIFY_SmsSendStatus** notification, reporting a fault.

>> 8: Connection Management

8

Connection management

Note: The API supports data calls only through the PPP daemon. Voice call functionality is supported only in MC5725V/MC5727V/MC5728V modems that are provisioned to support voice. The AirCard 402/595/595U/597E/ Compass597/ MC5725/MC5727/ USB 598 products do not support voice calling.

This group of functions controls the connection status of the modem. This includes originating, answering (terminating), and disconnecting calls, as well as using the flash feature.

The SDK includes the SwiConnect sample application (in Sample_Code/cdma).

Terminating (answering) a call

The CDMA network requires you to set the modem for the type of call to answer (voice, data, or fax) *before* you receive notification of the incoming call. You set the call answer mode using **SwiSetAnswerState**.

The modem will advise you when a call is incoming (ringing) by sending the **SWI_NOTIFY_CallIncoming** notification.

If the network provides caller ID services, the standard caller ID string is included with the incoming call notification. Both standard and extended caller IDs can be provided in explicit notifications. The standard caller ID can also be queried using **SwiGetCallerID**.

To answer (terminate) the call, use **SwiSetCallAnswerCmd**. Don't call this function unless you have detected an incoming call request (ring) via **SWI_NOTIFY_CallIncoming**. The mode the modem will use to answer the call is set by **SwiSetAnswerState** that must have been set *prior* to receiving the incoming call notification.

The notification **SWI_NOTIFY_CallConnecting** reports the call setup.

The success or failure of the connection is reported by notification (**SWI_NOTIFY_CallConnected**) or by query to **SwiGetCallConnectionState**.

Call Waiting

For accounts that support call waiting, the modem may alert the application through normal incoming call notification (**SWI_NOTIFY_CallIncoming**) that a new call is coming in. The application must know that a voice call is active. (See [Terminating \(answering\) a call](#) on page 43).

When this happens, the application can use the normal caller ID functions to identify the waiting call and present this information to the user.

Typically, the user can then ignore the waiting call or use a “SEND” or “TALK” button to place the current call on hold and answer the second call. To make the modem do this, the application calls **SwiSetCallFlashCmd** with the service option value **SOValue = SWI_TYPE_FlashContext_DontCare**.

Subsequent use of the SEND or TALK button should switch between the two calls. Repeated application calls to **SwiSetCallFlashCmd** perform this call switch.

Normal call disconnection is used as described in [Ending a call](#) on page 45. This disconnects the active call and switches to the alternate call. Another disconnection command is needed to end both calls.

Originating a call

To dial a call, use **SwiSetCallOriginateCmd**. This command specifies the type of call being made.

CNS_CALL_TYPE_VOICE Used for voice and OTASP calls by all CDMA products.

CNS_CALL_TYPE_ASYNC_DATA Used for creating a QNC connection.

CNS_CALL_TYPE_PACKET_DATA Used for creating a 1X/1xEV-DO packet data connection.

The notification **SWI_NOTIFY_CallConnecting** reports the call setup.

The success or failure of the connection is reported by notification (**SWI_NOTIFY_CallConnected**) or by query to **SwiGetCallConnectionState**.

3-Way calling

Some accounts support three-way, or conference calling. To do this, the application originates a call to one party as described above.

To connect to the second party, the application uses the flash command. In all cases the flash service option is set to:

SOValue = SWI_TYPE_FlashContext_DontCare.

1. Your application calls **SwiSetCallFlashCmd** to place the first call on hold (usually triggered by the user pressing a SEND or TALK button).
2. Your application accepts and stores the user input of the second number to call.
3. To start the second connection, use **SwiSetCallFlashCmd** with the parameter **pszCallNumber** set to the number to dial.
4. Call **SwiSetCallFlashCmd** again to connect the two calls; typically triggered by the user pressing a SEND or TALK button.

To end the conference call, disconnecting both parties, use the normal call disconnect, described below.

Ending a call

To end a call, use **SwiSetCallDisconnectCmd**. The usual “OK” return code does not mean that the call was successfully disconnected. The modem will report the change in connection state using the **SWI_NOTIFY_CallConnected** notification. Alternatively you can query **SwiGetCallConnectionState**.

Other connections, such as SMS or dormant data connections, may remain active after disconnecting a call.

Voice and audio functions

Note: Voice call functionality is supported only in MC5725V/MC5727V/ MC5728V modems that are provisioned to support voice. The AirCard 402/ 595/595U/597E/ Compass597/ MC5725/MC5727/ USB 598 products do not support voice calling.

The following group of functions are used to control audio features, generally for voice calls. Also covered are functions related to DTMF tones.

The SDK includes the SwiVoice sample application (in Sample_Code/cdma).

Headset

Voice calls should be permitted only when a headset (or TTY device) is detected by the modem. Your application can be notified of headset connection and removal using **SWI_NOTIFY_HeadsetState**.

Alternatively, your application can query for the state of connection using **SwiGetHeadsetState**. This command should be used after application startup and enabling the notification to provide your application with the initial state of headset connection.

Speaker volume

The volume of the headset speaker is adjustable and mutable.

SwiGetCdmaSpeakerVolume reports the volume level of the speaker on a scale of 0 (silent) to 6 (loudest). The level can be set using **SwiSetCdmaSpeakerVolume**. A normal practice is to step the volume up or down, one unit at a time.

The level setting is retained across resets and power cycles.

Muting

The volume level setting is retained while the speaker is muted using the separate mute command, **SwiSetSpeakerMuteStatus**. Use the mute state to switch the speaker on or off without altering the normal listening level. The mute status can be checked using **SwiGetSpeakerMuteStatus**.

If the volume level reported by **SwiGetCdmaSpeakerVolume** is 0, the speaker is muted regardless of the setting of the mute state. If the mute is on, the speaker level reported is the normal setting that is used when mute is switched off.

The mute setting is automatically disabled (speaker on) when a call is disconnected, and when the modem resets.

Microphone mute

The loudness level (sensitivity) of the microphone cannot be set by the user; however the microphone can be muted (switched off or on) if needed.

SwiGetMICMuteStatus reports the status of the microphone (headset input) mute setting. The mute state is controlled with **SwiSetMICMuteStatus**.

The mute setting is automatically disabled (MIC enabled) when a call is disconnected or the modem is reset.

DTMF tone volume

DTMF “comfort” tones are played in the headset speaker as keys are pressed. Tones don’t actually go to the network, except during over dialing (entering key tones while in an active voice call to control automated answering systems).

The volume at which these tones are played is controlled independently of the general speaker volume described on [page 47](#).

SwiGetToneLevel reports the volume level of DTMF tones sent to the speaker using the same 0 to 6 scale. The level can be set using **SwiSetToneLevel**. The level setting is retained across resets and power cycles.

Muting DTMF tones

Like the normal speaker mute, there is a separate tone mute (**SwiGetDTMFMuteStatus**) that does not alter the tone volume setting.

SwiGetDTMFMuteStatus reports the status of the DTMF tone mute setting. This mute status is distinct from both the tone level of zero (muted), and the state of the speaker mute. The volume level set with **SwiSetToneLevel** is retained regardless of the setting of the mute status.

If the speaker mute (**SwiGetSpeakerMuteStatus**) is on, DTMF tones are not heard regardless of the tone level or DTMF mute status.

If the tone level reported is 0, no DTMF tones are heard on the speaker regardless of the settings of the mute states.

The mute setting is *not* retained across resets and power cycles. The modem *does* retain the setting of DTMF mute during the session (across multiple calls).

Dialing

Although DTMF tones are not actually used over the network for dialing a call, a user needs “comfort” tones to be assured that keys have been entered. These can be simple beeps or actual DTMF tones.

The modem is capable of generating the tones in the headset speaker by using any of the following commands:

- **SwiSetDTMFKey** generates a single DTMF key tone both to the headset and over the air (for over dialing). The duration of tone playback is set using **SwiSetDTMFDuration**, and can be read using **SwiGetDTMFDuration**. The settings are for the duration of DTMF tone generation for both single key events and multi-key strings.
- **SwiSetKeyPressed** causes the modem to play audible DTMF key tones to the speaker (headset output). Playback continues until **SwiSetKeyReleased** is called. This tone is sent to the headset but not over the air. This is a human feedback for button pressing on a software keypad.

Alerts

The modem can generate alerts, in the form of beeps in the headset, for the following events:

- Minute marker—**SwiPstGetMinuteCallBeep** reports the state of the setting for generating an audible beep shortly before the one minute intervals of a voice call. The alert is enabled or disabled using **SwiPstSetMinuteCallBeep**. The beep sounds at 10 seconds before each minute elapses, to allow the caller to disconnect before starting into the next minute.
- Service alert—**SwiPstGetServiceAreaAlert** reports whether the alert tone for changes in service (IS-95 vs. 1X, and entering and leaving roaming) is enabled or disabled. The setting can be changed using **SwiPstSetServiceAreaAlert**. A call to **SwiGetCurrentBaseStation** will confirm the nature of the change in service.

Security overview

This family of services relate to the modem's user passcode security feature. This is similar to a PIN intended to prevent unauthorized use of the device.

The lock status can be queried (**SwiGetLockStatus**) at any time (whether the modem is locked or not). If the modem is locked, other functions may not be available.

By default, from the factory, the lock is off.

The default passcode is typically "0000", however it may be set, during factory provisioning, to a value determined by the service provider. The passcode is automatically set to the last four digits of the phone number whenever an account is activated (written). This happens regardless of the previous passcode value.

Controlling security

The lock can be set to come on whenever the modem is reset or power cycled; typically at modem startup. The lock can also be toggled on demand, at any time.

To lock the modem on startup: Call **SwiSetAutoLock** with a parameter other than zero. The lock is enforced the next time the modem starts up.

To disable the autolock: Call **SwiSetAutoLock** with a parameter of zero. To use this function, the modem must be unlocked.

To toggle the lock state at any time: Call **SwiLockModem** with the current passcode. If a timeout of this service occurs, the lock state may or may not toggle based on the acceptance or rejection of the passcode provided. Your application should pause and then use **SwiGetLockStatus** to determine if the modem lock changed state or not. That will determine if a retry of this function is needed.

Changing the passcode

You change the passcode anytime the modem is unlocked by calling **SwiChangeLockPIN**.

If the change function failed to complete within the timeout, the password change still may take place in the modem. This situation requires special care. If the OldPassword was accepted, the new setting is applied. A subsequent retry will report an invalid password (OldPassword) because the new password is in effect. On the other

hand, if the OldPassword was rejected, the settings are unchanged. A retry will report an invalid password, because the user failed to match. Both cases give exactly the same result on a retry!

The recommended course of action is to have your application silently retry the command using the NewPassword value in *both* the old and new parameters. If the modem accepts this, then the change is complete. If the modem rejects it, then OldPassword was likely rejected in the original call and the user has to start the process over.

Data connection overview

To establish a data connection, the modem must be registered on a CDMA network. Registration requires that:

- A valid account NAM is activated and selected.
- The modem is within the coverage area of a CDMA network and the signal strength is adequate.
- There are no account restrictions preventing registration.

Types of data connections

Three types of data connections are supported:

- **1xEV-DO/1X Packet Data**—In areas where there is 1xEV-DO or 1X coverage, the modem is capable of making high-speed packet-switched connections. The data rate of this connection is dependent on the network, but the maximum theoretical data rate is:
For 1xEV-DO Rev. A: 3.1 Mbps on the downlink (network to modem) and 1.8 Mbps on the uplink;
For 1xEV-DO Rev. 0: 2.4 Mbps on the downlink (network to modem) and 153 Kbps on the uplink;
For 1X: 153 Kbps on the downlink and 153 Kbps on the uplink.

Note: Packet data connections can seamlessly handoff between 1xEV-DO and 1X, depending upon the available service. 1xEV-DO has preference.

- **QNC**—Where there is CDMA coverage (either IS-95 or 1X) the modem is capable of making lower-speed packet-switched connection using Quick Net Connect (QNC). This places a circuit-switched call to a special number at the network. The network then provides a packet-switched Internet connection. The data rate of this connection is dependent on the network, but the maximum theoretical data rate is 14.4 Kbps.

Note: If only 1xEV-DO service is available (no IS-95 or 1X service), QNC connections cannot be made.

- **Circuit-switched**—In a circuit-switched connection, the modem behaves as an asynchronous dial-up modem with a maximum data rate of 14.4 Kbps. Circuit-switched connections are possible in any CDMA coverage area (IS-95 or 1X) as it is handled by the network as a voice call. The API does not provide any functions with which to make a circuit-switched connection.

Note: If only 1xEV-DO service is available (no IS-95 or 1X service), circuit-switched connections cannot be made.

Determining modem status

These notifications provide information about the modem's status and ability to establish a data connection:

- **SWI_NOTIFY_ServiceIndication** indicates the availability of a CDMA network. This is issued anytime the service status changes.
- **SWI_NOTIFY_Prev** indicates the type of CDMA service available (IS-95 or 1X).
- **SWI_NOTIFY_Hdr_Srv_State** indicates whether or not 1xEV-DO service is available, and the type of service (Rev. 0 or Rev. A).
- **SWI_NOTIFY_CallNotificationStatus** is needed to enable the family of notifications that monitor call connection.

Note that the type of service that your application should check for depends on the call type that is to be made. For packet data calls, your application should check for 1xEV-DO service (**SWI_NOTIFY_Hdr_Srv_State**) or 1X service (**SWI_NOTIFY_ServiceIndication** and **SWI_NOTIFY_Prev**; **PRev** must be ≥ 6). For QNC and circuit-switched calls, your application should check for CDMA (IS-95 or 1X) service (**SWI_NOTIFY_Prev**; **PRev** must be < 6).

Originating a data connection

To make a data connection of any kind on any device, you must first:

1. Open the API ([page 28](#)).
2. Get available servers.
3. Register a callback function.
4. Verify the modem is responding.
5. Enable notifications ([page 33](#)).
6. Confirm you are in service, either by notification (see [“Determining modem status”](#), above) or query to **SwiGetServiceIndication**.
7. Verify the type of data connection supported (PREV) either by notification or query to **SwiGetCurrentBaseStation**.
8. Determine if 1xEV-DO is available by query to **SwiGetHdrSrvState**.

At this point your application knows what types of data connections are possible.

On Linux, data connections are handled through PPP.

To dial a packet data connection:

1. Use **SwiSetCallOriginateCmd**, with parameters set for the type of call to make; for example:

```
SwiSetCallOriginateCmd (SWI_CALL_TYPE_PacketData, (swi_char
*)"#777", API_TIMEOUT);
```

Type	Dial string
Packet data	#777
QNC	Carrier-dependent
Circuit-switched	[phone number of server]

2. Launch the PPP daemon on data port(ttyUSB0).

The connection process can be tracked using API notifications:

SWI_NOTIFY_CallConnecting and **SWI_NOTIFY_CallConnected**. At that point the modem has opened the physical layer (the radio link to the network). PPP then handles authentication and notifies you when the connection is open and ready to use.

Disconnection is handled through PPP as well. Notifications advise your application of the process, ending with the API notification that the physical layer is disconnected (**SWI_NOTIFY_CallDisconnected**).

Answering a data call

The modem must be set for the type of call to answer (disabled, voice, data, or fax) *before* the modem receives notification of the incoming call. The call answer mode can be set using **SwiSetAnswerState**.

All incoming data calls are treated as circuit-switched and must be handled by an application using the modem driver.

The modem advises you when a call is incoming (ringing) by sending the **SWI_NOTIFY_CallIncoming** notification.

The application on the API does not take any action. The application on the modem driver should receive the RING indicator and answer the call. The API forwards the notification of the connection state but does not take any part in answering or disconnecting the call.

The success or failure of the connection is reported by notification (**SWI_NOTIFY_CallConnected**) or by query to **SwiGetCallConnectionState**.

Ending a data connection

To end a data call:

1. Use **SwiSetCallDisconnectCmd**.

The usual "OK" return code does not mean that the call was successfully disconnected. The modem reports the change in connection state using the **SWI_NOTIFY_CallDisconnected** notification. Alternatively **SwiGetCallConnectionState** can be used.

If a packet data connection was dormant when the disconnect function is called, you may receive a sequence of notifications as the connection is rebuilt so that it can be gracefully disconnected.

2. Stop the PPP daemon.

>> 12: Characteristics and Status

12

Unit characteristics

These functions report information about the modem (hardware and software).

The SDK includes the `SwiQueryNotify` sample application (in `Sample_Code/cdma`).

ESN

SwiGetESN and **SwiGetMEIDESN** return the device's unique identity as a string.

SwiGetESN returns the device's ESN (Electronic Serial Number).

SwiGetMEIDESN is recommended over **SwiGetESN**. Depending on whether MEID (Mobile Equipment Identifier) support is enabled on the device, **SwiGetMEIDESN** returns either the device's ESN only, or the pseudo ESN (pESN) and MEID.

Managing the display of the ESN/pESN as a string is up to you. In the CDMA marketplace, the ESN/pESN is usually presented as the single 8-character string, in the form of a hexadecimal number; for example: CE0271C8.

MEID

SwiGetMEIDESN returns the device's unique identity as a string.

Depending on whether MEID (Mobile Equipment Identifier) support is enabled on the device, **SwiGetMEIDESN** returns either the device's ESN only, or the pseudo ESN (pESN) and MEID.

Note: Depending on the device's factory configuration, MEID support may or may not be enabled. You cannot change this setting.

Managing the display of the MEID as a string is up to you. In the CDMA marketplace, the MEID is usually presented as a 14-character string, in the form of a hexadecimal number.

Version numbers

SwiGetFirmwareVersion returns the device's firmware version in a string. The firmware version string varies in length. It is recommended to allocate a character string of 80 bytes to store this return value.

The string has additional data appended. Your application should truncate the string at the first <space> character.

There is a bootstrap loader that is controlled separately from the operational firmware of the modem. The version of this component can be reported using **SwiGetBootVersion**.

Also available is a function to report the hardware version, **SwiGetHardwareVersion**. This has value for support and maintenance issues.

SwiGetERIVer returns the version of the ERI (Enhanced Roaming Indicator) file stored in the modem.

Power state

SwiHWGetPower reports the modem/API power state. The notification equivalent **SWI_NOTIFY_Power** is reported whenever the modem/API power state changes.

Power save status

SwiGetTechPowersaveMode reports the power save status (applies only to IS-95 or 1X; does not apply to 1xEV-DO). **SwiPowerSaveExit** forces the modem out of power save mode (CDMA deep sleep).

For IS-95 or 1X, the notification **SWI_NOTIFY_PowerSaveEnter** is reported whenever the modem enters power save mode; **SWI_NOTIFY_PowerSaveExit** is reported whenever the modem exits power save mode.

For 1xEV-DO, the notification **SWI_NOTIFY_Hdr_PowerSave_Enter** is reported whenever the modem enters power save mode; **SWI_NOTIFY_Hdr_PowerSave_Exit** is reported whenever the modem exits power save mode.

Temperature

SwiGetTemperature reports the temperature of the radio within the modem.

The modem firmware monitors the radio temperature to protect electronics and prevent drift from specification. To determine the way the modem handles high temperature, consult the product specification.

Monitoring the temperature is *not* provided in a notification.

Activation information

For activation/account information, see [Activation information](#) on page 37.

Service characteristics

These functions provide information about the modem's connection to the host, and the radio connection to the service provider.

Signal strength (RSSI)

CDMA does not use an RSSI (Received Signal Strength Indicator) per se. To produce the RSSI numbers, the modem uses an algorithm that incorporates signal quality elements.

The interpretation is described in the following table.

Table 12-1: RSSI interpretation

Value	Meaning
-125	No service
-124 to -108	Very poor
-107 to -99	Poor
-98 to -89	Fair
-88 to -77	Good
> -76	Excellent

To determine the RSSI for IS-95 or 1X, use the functions **SwiGetRssi** or **SwiGetRssiEcio**. To determine the RSSI for 1xEV-DO, use the function **SwiGetRssiEcio**.

The notification equivalent **SWI_NOTIFY_Rssi** is reported whenever the value changes. This is preferred over the polling command.

The SDK includes the **SwiSignalStrength** sample application (in **Sample_Code/cdma**).

Channel number

SwiGetChannelNumber returns the current channel number. The valid range of channel numbers depends on the underlying radio band (to determine the current band, use the notification **SWI_NOTIFY_TechBandClass**):

- Cellular: 0–799, and 991–1023
- PCS: 0–1200

The modem's PRL limits the actual channels the modem uses to a much smaller subset.

If the radio is scanning (or not in coverage), this function returns the last channel tuned by the radio.

There is a notification equivalent for this function (**SWI_NOTIFY_ChannelNumber**), but it may generate large numbers of notifications during scanning. The notification **SWI_NOTIFY_ChannelState** reports when the radio is scanning and when it has acquired a channel. You can minimize notification traffic by polling for the channel number only when a channel is acquired.

In service

SwiGetServiceIndication reports whether or not there is CDMA (IS-95 or 1X) service in the current location. The presence of CDMA service does not guarantee that connection is possible. The modem may fail to authenticate with the local service provider.

The notification equivalent (**SWI_NOTIFY_ServiceIndication**) advises your application when entering or leaving CDMA (IS-95 or 1X) coverage. This is preferred over the polling command.

SwiGetHdrSrvState reports whether or not 1xEV-DO service is available, and the type of service (Rev. 0 or Rev. A).

The notification equivalent (**SWI_NOTIFY_Hdr_Srv_State**) advises your application when 1xEV-DO is available, or when a change in 1xEV-DO service occurs (Rev. 0 to Rev. A, or vice versa). This is preferred over the polling command.

Protocol revision (PREV)

SwiGetCurrentBaseStation retrieves the protocol revision (service type) of the base station currently serving the modem. This command is used to determine if making 1X calls is possible.

A protocol revision of 6 or higher means 1X service is available. Protocol revision 1 is not supported.

Roaming

SwiGetRoamingStatus reports whether the device has acquired service from a cellular service provider other than that to which the user has subscribed. Results when called while not in coverage are undefined. The modem cannot verify that you are roaming until it can detect IS-95 or 1X coverage (**SwiGetServiceIndication**) and identify the service provider(s).

You should limit the use of this function. The notification

SWI_NOTIFY_RoamingStatus reports the same data, and is preferred over direct polling.

Roaming (1xEV-DO)

SwiGetHdrRoamStatus reports whether the device has acquired 1xEV-DO service from a cellular service provider other than that to which the user has subscribed. Results when called while not in 1xEV-DO coverage are undefined. The modem cannot verify that you are roaming until it can detect 1xEV-DO coverage (**SwiGetHdrSrvState**) and identify the service provider(s).

You should limit the use of this function. The notification

SWI_NOTIFY_Hdr_Roam_Status reports the same data, and is preferred over direct polling.

CDMA network time

SwiGetSystemTime returns the date and time from the CDMA network. Coverage is required to return a valid time.

Base station GPS location

The AirCard 402/595U/597E/ Compass597/ MC5727/MC5727V/MC5728V/ USB 598 wireless products have an internal GPS receiver. The AirCard 595 PC Card supports limited GPS functionality, but can report the location of the currently serving base station. (GPS services may be enabled at the PRI and network/carrier level).

To access the data, use **SwiGetTechBSInfo**. The mechanism to decode the results is included in the function header.

For information on location-based services, see [Location-based Services](#) on page 63.

Statistics

A byte counter of transmitted and received data during a connection session is available using **SwiGetByteCounter**. Counters contain the total bytes sent and received over the local (host/modem) interface. The counter resets when the call is disconnected.

The preferred mechanism for tracking this data is to use the notification equivalent **SWI_NOTIFY_CallByteCounter**.

>> 13: Location-based Services

This chapter describes how to use API functions to perform the following tasks on GPS-capable Sierra Wireless modems that have been configured to support GPS functionality:

- Get and set modem parameters and statuses
- Perform single location fixes
- Manage a tracking session
- Keep almanac / ephemeris data up to date

PDSM

Position Determination Session Management (PDSM) is a GPS feature supported by some carriers/networks. The CDMA network is used to assist the modem to acquire a GPS location fix.

The SDK includes the SwiLbs sample application (in Sample_Code/cdma).

Types of PDSM clients

Registered clients are clients that are interested in receiving notification about PDSM events.

A requesting (or active) client is a client that has initiated the current PDSM session or event. A requesting client is typically also a registered client.

Notifications

Several PDSM-related notifications are enabled and disabled as a group. For more information, see [page 33](#) ([SwiSetLbsLocNotifyStatus](#), [SwiSetLbsDloadNotifyStatus](#), and [SwiSetLbsParamNotifyStatus](#)).

Getting the location of the terminal

Requesting the last known location

Your application can be notified of the results of the position fix using [SWI_NOTIFY_LbsPdData](#).

Alternatively, your application can query the last known location, by calling [SwiGetLbsPdData](#).

Requesting a single location fix

Your application can be notified of the results of a single location fix:

- Session has begun, using **SWI_NOTIFY_LbsPdBegin**
- Results of the location fix, using **SWI_NOTIFY_LbsPdData**
- Session has ended, using **SWI_NOTIFY_LbsPdDone** and **SWI_NOTIFY_LbsPdEnd**.

Alternatively, your application can request a single location fix, by calling **SwiSetLbsPdGetPos**.

Downloading ephemeris and almanac data

Your application can be notified that a download session of ephemeris and almanac data has:

- Begun, using **SWI_NOTIFY_LbsPdDloadBegin**.
- Completed, using **SWI_NOTIFY_LbsPdDloadData**.

Your application can be notified that a position determination download session has completed using **SWI_NOTIFY_LbsPdDloadEnd** and **SWI_NOTIFY_LbsPdDloadDone**.

Alternatively, your application can request data download of ephemeris and almanac data, by calling **SwiSetLbsPdDownload**.

Clearing ephemeris and almanac data

Your application can request the modem to clear various location parameters to simulate a cold start, by calling **SwiSetLbsClearAssistance**.

Tracking the location of the terminal

Frequent tracking of the terminal host

Your application can be notified, throughout the location tracking, of the following events:

- Download session of ephemeris and almanac data has begun, using **SWI_NOTIFY_LbsPdDloadBegin**
- Download session of ephemeris and almanac data has completed, using **SWI_NOTIFY_LbsPdDloadData**
- Download session of ephemeris and almanac data has completed, using **SWI_NOTIFY_LbsPdDloadEnd** and **SWI_NOTIFY_LbsPdDloadDone**
- Tracking session has begun, using **SWI_NOTIFY_LbsPdBegin**
- Results of the location fix, using **SWI_NOTIFY_LbsPdData**

- Tracking session has ended, using **SWI_NOTIFY_LbsPdDone** and **SWI_NOTIFY_LbsPdEnd**

Alternatively, your application can initiate tracking, by calling **SwiSetLbsPdTrack**.

Single location fix

Your application can be notified, throughout the location tracking, of the following events:

- Tracking session has begun, using **SWI_NOTIFY_LbsPdBegin**
- Results of the location fix, using **SWI_NOTIFY_LbsPdData**
- Tracking session has ended, using **SWI_NOTIFY_LbsPdDone** and **SWI_NOTIFY_LbsPdEnd**

Alternatively, your application can initiate tracking, by calling **SwiSetLbsPdTrack**.

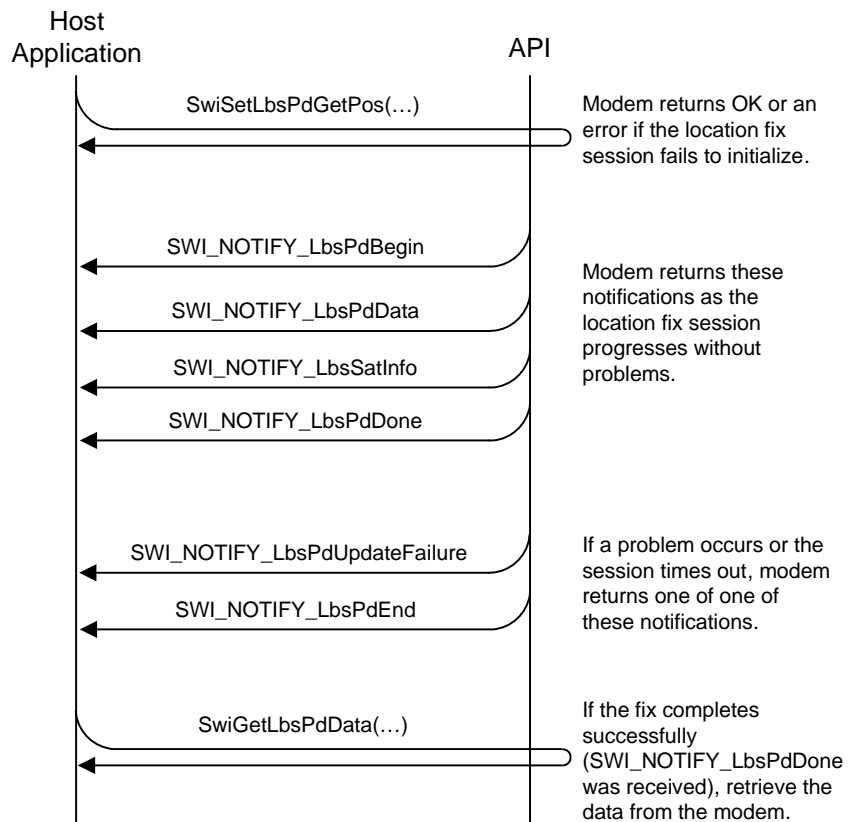


Figure 13-1: Initiating a single location fix

Terminating a position determination session

Terminating a location fix session

Your application can be notified of the termination of a location fix session using `SWI_NOTIFY_LbsPdDone` and `SWI_NOTIFY_LbsPdEnd`.

Alternatively, your application can terminate a location fix session, by calling `SwiSetLbsPdEndSession`.

Terminating a download session

Your application can be notified of the termination of a download session using using `SWI_NOTIFY_LbsPdDloadEnd` and `SWI_NOTIFY_LbsPdDloadDone`.

Alternatively, your application can terminate a download session, by calling `SwiSetLbsPdEndSession`.

Accessing the default position determination parameters

Retrieving the default position determination parameters

To retrieve the default position determination parameters, call `SwiGetLbsPaParam`.

Alternatively, you can retrieve each parameter individually:

- GPS IP Address: `SwiGetLbsPaIpAddr`
- GPS Lock: `SwiGetLbsPaGpsLock`
- GPS PTLM Mode: `SwiGetLbsPaPtlmMode`
- GPS PortID: `SwiGetLbsPaPortId`
- GPS Privacy: `SwiGetLbsPaPrivacy`
- GPS NetAccess: `SwiGetLbsPaNetAccess`

Modifying the default position determination parameters

Note: If you are modifying more than one parameter, your application must wait for the notification from the previous update, before requesting another update. Otherwise, an error will occur.

To change this setting:	Call this function:	Upon completion, modem sends this notification:
GPS IP Address	SwiSetLbsPaIpAddr	SWI_NOTIFY_LbsPaIpAddr
GPS Lock	SwiSetLbsPaGpsLock	SWI_NOTIFY_LbsPaGpsLock
GPS PTLM Mode	SwiSetLbsPaPtlmMode	SWI_NOTIFY_LbsPaPtlmMode
GPS PortID	SwiSetLbsPaPortId	SWI_NOTIFY_LbsPaPortId
GPS Privacy	SwiSetLbsPaPrivacy	SWI_NOTIFY_LbsPaPrivacy
GPS NetAccess	SwiSetLbsPaNetAccess	SWI_NOTIFY_LbsPaNetAccess

Getting satellite information

Your application can be notified of the satellite information using **SWI_NOTIFY_LbsSatInfo** notification.

Alternatively, your application can request the modem to get various GPS satellite information, by calling **SwiGetLbsSatInfo**.

14: Demultiplexing APIs

Note: Two utilities that use the APDX API are available: Diagnostic, and Relay Agent, located in \$INSTALL_FOLDER/Utilities/Common.

Sierra Wireless modems expose multiple ports over the serial interface they use to communicate with external devices (USB). The number of these ports depends on the specific model of Sierra Wireless modem, and the type of data that can run over the port depends upon the modem's configuration. Examples of these data services include:

- Packet Data Protocol (for IP data session traffic)
- AT commands and data
- Location Based Services data (NMEA protocol)
- Sierra Wireless-proprietary control information
- Third-party proprietary diagnostics data

The main APIs of this SDK pertain to interactions with the modem via the Sierra Wireless-proprietary control channel. The SDK provides a set of APIs that an external application can use to obtain diagnostic information about the modem. This chapter describes the APIs available for sending/receiving diagnostic data (multiplexed over the HIP service) between the modem and an external application.

Using these APIs

Note the following when using the demultiplexing APIs:

- The external application using these APIs must be separate from the main application.
- The external application must provide a separate thread to handle callbacks containing data and status from the SDK. Calls to send data to the modem and to initialize the demux API must be made from a separate thread.
- Using these APIs causes the SDK process to start and initialize if there isn't one already running. If one is already running, then the demux application interacts with that one in parallel with the main application.
- Certain Sierra Wireless modems offer services on their own USB interfaces that external applications can access directly using standard Linux file open/close and read/write operations. The SDK provides an entry point, **SwiGetUsbPortName()**, that applications can use to determine which /dev/ttyUSBn handle to use to access a specific modem service.

Note: Applications that interact with the modem directly using one of these file handles must be sure to detect when the modem resets for any reason and close all file handles they may have open at the time.

- External applications are free to access the tty's directly for Sierra Wireless modems, which support these services over separate ttys. But there are some advantages to external applications if they use these APIs instead:
 - The external application does not need to detect when the modem resets – so it does not need to include logic to release the file handle when the modem is no longer available.
 - The external application does not require additional logic to determine exactly which tty is required to access a particular service on the modem; the SDK takes care of this.
 - The external application can interact with as many modem data ports as it desires by registering itself with the Demux API for each service of interest.
 - The external application is expected to know how to interpret the information it receives from a particular port on the modem and how to originate packets that are meaningful to the service.

Process model

The diagram below illustrates how the processes interact when the main application, the Demux application and the SDK process are all running.

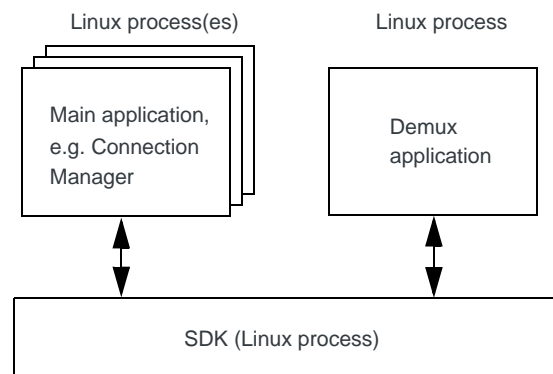


Figure 14-1: Process model

The application processes shown communicate with the SDK process over their own dedicated IPC channels. The Demux application can access as many data ports in the modem as this facility supports.

Supported services

Currently, the Demux API offers the Demux application the ability to access the diagnostics service. Other services will be available in subsequent releases of the SDK.

Startup, normal operation, and shutdown

The external application—referred to as the Demux application—provides the processing context that uses the Demux API entry points. The application must ensure that a separate thread is available for processing notifications of incoming data and modem status changes from the SDK.

The Demux application must ensure the SDK process is running and initialize various components of the Demux API by calling:

- **SwiApiDxStartup()** – Initialize the Demux API and start the SDK

The Demux application then registers itself to receive data and notifications from the modem/SDK by calling:

- **SwiApiDxRegister()** – Register callbacks for the desired service

One last step is required before the interface is ready to exchange data and status: the Demux application must inform the SDK that it can begin forwarding traffic to it by calling:

- **SwiApiDxBegin()** – Tell the SDK to initiate sending modem data

When done, the Demux application will receive status information and data packets from the modem whenever the modem sends it.

The Demux API provides a pair of entry points that allow the external application to send data to the selected service on the modem. To send a data packet to the modem the Demux application calls:

- **SwiGetDataPldOffset()** – Initialize the Demux application's supplied data header

followed some time later by:

- **SwiApiDxSend()** – Send a data packet to the modem.

These two APIs must always be called one after the other; the **SwiGetDataPldOffset()** call accepts a pointer to a flat buffer into which data will be written prior to sending. It returns a pointer to an offset within the buffer where the caller should start writing their data. Once that data is copied into the buffer, the caller should then submit the buffer for sending using the original pointer—not the offset returned to it by **SwiGetDataPldOffset()**.

Not all modem services are bi-directional on their ports and it is up to the Demux application to contain this level of knowledge in its implementation. For instance, the NMEA traffic is one-way outbound from the modem so there is no valid reason why the Demux application would need to send a data packet to the modem's NMEA service over the NMEA port.

If the Demux application subsequently wishes to shut down, it should first inform the modem, if applicable, by sending a packet to it instructing it to stop sending data on that port by calling:

- **SwiApiDxEnd()** – Halt the SDK Demux agent sending modem data

The following data flow diagram illustrates which components of the SDK are affected by calling these APIs and what actions are taken by the components.

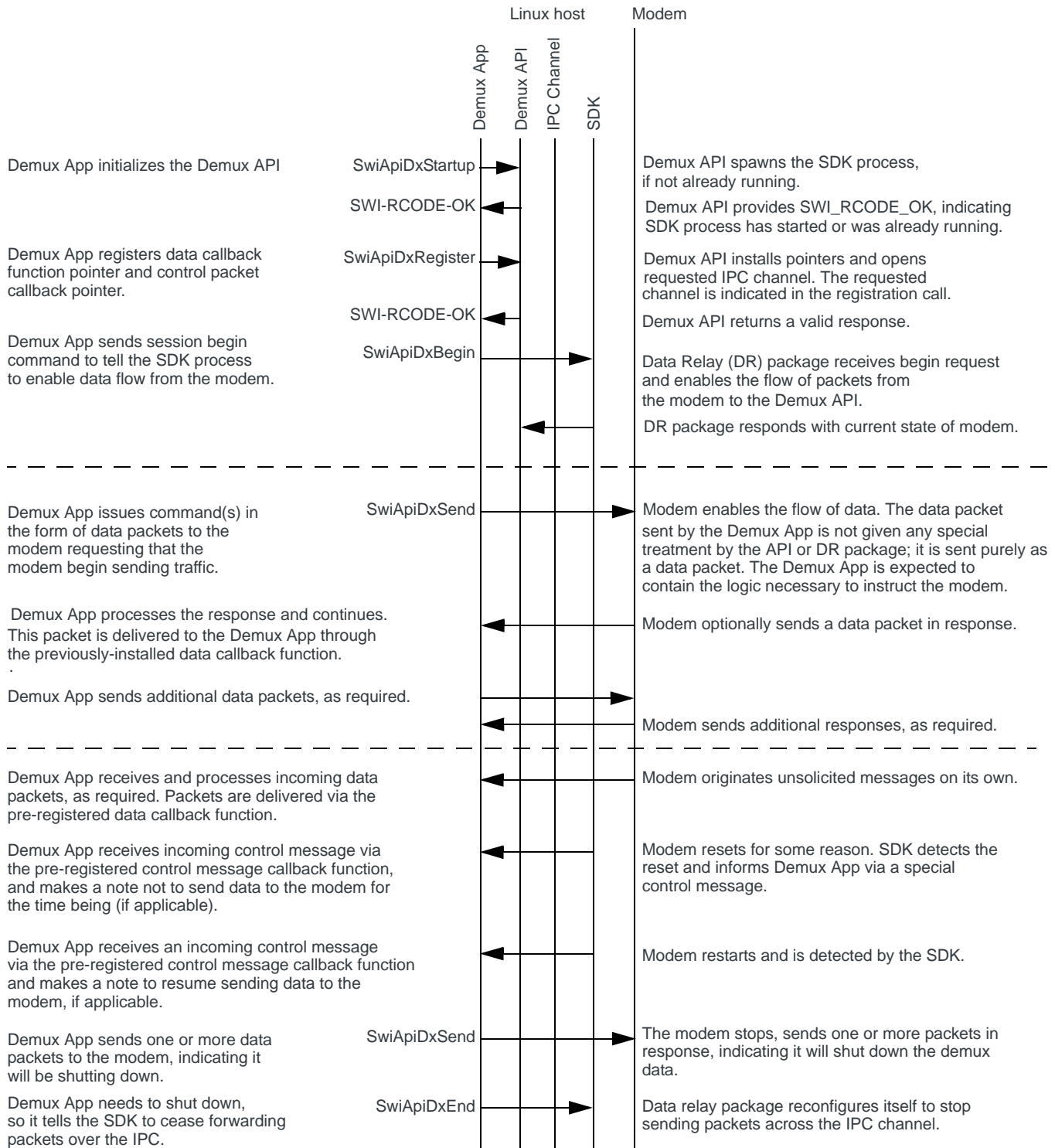


Figure 14-2: Data flow: Demux application

>> 15: Error handling

15

This chapter describes how to handle error codes returned from API function calls.

Error codes

Refer to the API online reference guide (**SWI_RCODE** enumeration) for a complete list of possible error codes returned from API function calls.

The reference guide is in the docs/SwiApiReference directory.

Handling errors

To retrieve information about the last error returned from a control and status API function, call **SwiGetLastError**, passing a buffer to hold the transaction error.

If your application encounters errors that cannot be resolved, you can contact Sierra Wireless Technical Support for assistance. (See [Contact Information](#) on page 4.)

>> 16: Building an Application

16

Building an application – SDK libraries

Your SDK release comes pre-compiled for Ubuntu 8.04 desktop-based 80x86/32 machines. If your development machine is the same configuration then there is no need to build the SDK from scratch to create an executable image of your application. Instead, you can build an executable image directly, using the as-delivered libraries. However, building the SDK images and libraries is straightforward and the procedure is documented in the Sierra Wireless Linux SDK Integration Guide, located in the <\$INSTALL_FOLDER>/docs directory. Please refer to that document for details on how to build the SDK.

To build your application, you need to set up your make files to point to the directory containing the Linux SDK libraries. These are architecture-dependent – currently 80x86/32 and ARM9 targets are supported. These libraries are located in:

`<$INSTALL_FOLDER>/build/lib/<architecture>`

where <\$INSTALL_FOLDER> is the path to the location where you uncompressed your release of the SDK and <architecture> is either

i386

or

arm9

For example, if you were using an Intel-based development machine and your SDK resides in its own directory under the usr/local path, then the complete path to the libraries would be:

`/usr/local/LinuxSDK_V1_0_0_0/build/lib/i386`

and this is the location you need to set in your make file for the path to the SDK APIs.

Note: You can limit the size of your linked application by selecting only the libraries your applications requires to do its job. Only statically-linked libraries are supported.

Package breakdown

The SDK source files are organized into logical groups referred to as *packages*. There are several packages (consisting of C-language source code files) that make up the SDK; they are located in subdirectories under the directory:

`<$INSTALL_FOLDER>/pkgs`

Package naming convention

Each package has its own directory, which is named using a 2-to-4-character prefix. All files, functions, data structures (including their members), global storage locations and constants (macros) within a package must begin with that package's prefix. This provides an easy at-a-glance way to know where to find information about a specific package element while reading source code.

Package services

In general, the content of a package can be thought of as being equivalent to a class in an object-oriented language, such as C++ or Java. If a package offers services to other packages, these services are provided as function calls defined with the Sierra Wireless keyword *global*. Functions provided for in-package use only are defined with the Sierra Wireless keywords *package* or *local*, and it is forbidden for these functions to be called from anywhere other than source files within the package where they are defined.

Enforcement of this rule is by convention, meaning that there is nothing stopping developers from actually calling a package level function from outside that package except convention. In support of this concept, a set of header files has been created whose use also must follow a particular convention.

xxdefs.h xxiproto.h

Virtually every package has a pair of header files with these names, where *xx* is replaced by that package's actual prefix. The *xxdefs.h* file contains internal package definitions, meaning that any source file within the *xx* package is permitted to include this header file and use those definitions.

The *xxiproto.h* file contains prototypes for functions defined using the *package* keyword. This header file is always included by *xxdefs.h*, therefore files in the *xx* package have to include only *xxdefs.h* to automatically receive package level function prototypes.

xxudefs.h/xxuproto.h

Virtually every package contains a pair of header files with these names, where the *xx* is replaced by that package's actual prefix. The *xxudefs.h* file contains definitions other packages would require when using the services of the *xx* package. Included in such definitions are structures, constants and other items as needed. Note, if a structure definition is included in an *xxudefs.h* file, callers are generally forbidden from accessing its members directly, even though the structure is clearly published within the header file. Instead, callers must usually

pass pointers to these structures into XX package functions defined using the *global* keyword, where those functions operate on the structure instead. The reason for including structure definitions in some xxdefs.h files is so that external packages can allocate an appropriate amount of memory using the *sizeof (struct xxstruct)* construct.

The xxdefs.h file includes the xxuproto.h file directly so that other packages only need to include the xxdefs.h file in their sources. The xxuproto.h contains function prototypes for all the global functions contained in the xx package.

SDK directory tree contents

The following diagram shows a partial view of the directory structure of the Linux SDK.

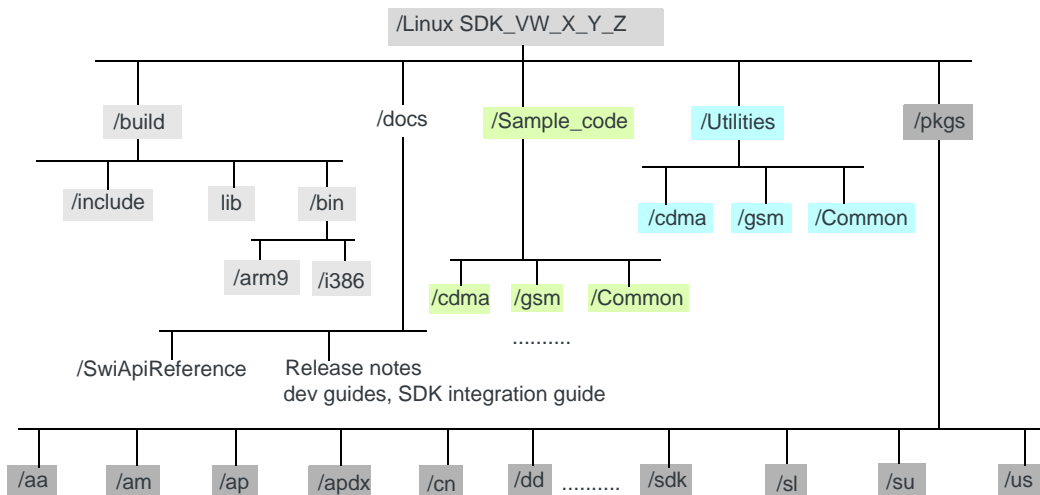


Figure 16-1: SDK file folder structure

build

This branch of the directory tree contains all the header files and library files that developers should need to be able to build applications. The build/bin branch contains the executables for the Firmware Updater utility and for the sample applications.

There are no source files in this tree. If you do not need to alter the source code within the pkgs tree, then the contents of the build directory and its subdirectories should be completely sufficient for you to develop and link your applications. (You should link to the libraries in build/lib when building your applications.)

If you need to recompile the source code in the pkgs tree, see [Rebuilding SDK packages](#) on page 82.

If you need to recompile the Firmware Updater utility, see [Utilities/cdma](#) on page 80.

build/bin

The bin subdirectory contains the executables for the Firmware Updater utility and for the sample applications, along with the corresponding SDK executable.

There are subdirectories for each supported target architecture, ix86/32 and ARM9. Each target subdirectory contains executables for:

- Sample applications
- CDMA (FWUpdaterCDMA...) and GSM/UMTS (FirmwareDownloadUMTS...) Firmware Updater utility
- The SDK (swisdk), which is used as the default by all sample code executables if no path is specified.

When running a sample application, you must specify the path to the subdirectory for your architecture; the sample application executes the image stored in the specified subdirectory. For help, run the application with the “-h” command line argument.

When running the Firmware Updater utility, you must specify the firmware file (CWE image) to download and its location (absolute path); optionally, you can specify the path and filename of the SDK executable (if not specified, then the default ./swisdk is used). For help, run the application with the “-h” command line argument.

build/include

The include subdirectory contains the header files your applications should include when linking to the APIs. At a minimum, you need to include:

- SwiDataTypes.h
- SwiRcodes.h
- SwiApiCmBasic.h

In addition, your applications need to include the header file(s) containing prototypes for the APIs you are using. For instance, if the application is calling CDMA Network-related APIs it would need to include either or both of:

- SwiApiCdmaNetwork.h
- SwiApiCdmaBasic.h

Source code located in the pkgs/ap directory can provide an example of what header files need to be included.

build/lib

There are subdirectories for each supported target architecture, ix86/32 and ARM9. When linking your application, the make procedures should include the library files in the suitable target architecture subdirectory beneath the build/lib subdirectory. Note that the SDK does not support Shared Object libraries at this time.

docs

The docs directory contains:

- This document
- UMTS Developer's Guide
- Linux SDK Integration Guide
- Licensing agreement
- Release notes for the particular version of the SDK
- A subdirectory structure SwiApiReference (described below).

docs/SwiApiReference

This tree contains the API reference guide in HTML format. To view this documentation, follow the instructions in [Using the API documentation](#) on page 17.

Sample_Code

This subdirectory contains all the sample applications created to demonstrate how to accomplish certain operations using the SDK. Sample applications are provided in source code and executable format and can be started right out of the box on supported Ubuntu systems.

Sample_Code/cdma

This directory contains a collection of subdirectories—one for each different sample application. All code samples are provided in C-language and are fully operational. Review the sample applications to gain insight into how the applications are written using the SDK and its APIs and to decrease the amount of ramp-up time required to get started. Examples of sample applications contained in this tree are:

- SwiActivate
- SwiConnect
- SwiLota
- SwiLbs
- SwiMip
- SwiPowerMgmt
- SwiQueryNotify
- SwiSignalStrength
- SwiSms
- SwiVoice

Sample_Code/Common

This directory contains a common piece of code in which the “main” routine common to all sample apps is stored. The sample apps must provide an entry point called by main() called “Run_App()”.

Sample_Code/gsm

This directory contains a collection of subdirectories—one for each different sample application for GSM/UMTS products. For more information, see the UMTS Developer's Guide (in \$INSTALL_FOLDER/docs).

Utilities

This subdirectory contains utilities that demonstrate how to accomplish certain operations using the SDK. Utilities are provided in C source code. The executable image is in build/bin (see [page 78](#)) and can be started right out of the box on supported Ubuntu systems.

Utilities/cdma

This directory contains a collection of subdirectories—one for each different utility. All utilities are fully operational. Review the source code of the utilities to gain insight into how the applications are written using the SDK and its APIs and to decrease the amount of ramp-up time required to get started.

This tree contains the utility:

- Firmware updater

If you need to recompile the Firmware updater utility, enter the SwiFWUpdater subdirectory and use the following command for the appropriate platform:

- 80x86/32 Linux platforms:
 - make
- ARM9 platforms:
 - make CPU=arm9

Utilities/gsm

This directory contains a collection of subdirectories, one for each different utility for GSM/UMTS products. For more information, see the UMTS Developer's Guide (in \$INSTALL_FOLDER/docs).

Utilities/Common

This directory contains utilities that can be used by both CDMA and UMTS modems. The Diagnostic and Relay Agent utilities are in this directory.

Packages

This section provides a quick tour of the contents of the source code directories. If you are not planning to make changes to your copy of the SDK source, you may skip this section.

The Linux SDK software source code is composed of several software components referred to as “packages”. Each package is stored in its own subdirectory within the pkgs folder (e.g. the “am” package in the directory tree diagram [[Figure 16-1](#) on page 77]). Packages are usually named with a 2-to-4-character identifier and by convention, all functions, global memory

locations, structure names, their members and even constant definitions (macros) are defined so that the first two to four characters of their name match the package identifier.

Note: The directory tree diagram, [Figure 16-1](#) on page 77, shows only a sample of the total package directories.

Packages are coded to perform a specific function within the Linux SDK executable. The identifier describes the service that package provides within the image. For instance, the String Library package has the identifier prefix “SL”, the HIP protocol package has “HP”, the Startup manager package has “SU”, and so on. A more detailed description of each package and its contents starts on [page 83](#) in this document.

Note: This document refers to API-side and SDK-side. These terms refer to the process model used when an application is running and using the SDK. API-side refers to an application process calling APIs, while SDK-side refers to the SDK daemon process. For more details about the process model, see [SDK process model](#) on page 90.

Chosen names of SDK source files are somewhat arbitrary, but here are some of the conventions used:

- If a filename ends in “_sdk.c” or “_api.c”, etc, its contents are available only on the identified side.
- The absence of the suffix noted above does not necessarily mean the associated code is meant to run on both sides. For instance, the “su” package contains “su.c” and the code in this file runs only on the SDK side.
- Each package may contain the following header files: xxdefs.h, xxiproto.h, xxudefs.h, xxuproto.h. For details, see [Package services](#) on page 76.
- **Note that external packages need to include only the xxudefs.h file.** This, in turn, brings in that package’s xxuproto.h file, so external packages automatically have access to the XX package’s function prototypes.
- Most packages contain an xx.txt file. This file provides a brief description of the package and how external packages may use it, if at all.
- Every package contains an xx.mak file. This is the makefile for the package. To build just the image for that package, use
make -f xx.mak [clean | CPU=[i386 | arm9]] TECHNOLOGY=CDMA [SYMBOLS=ON]
The clean option removes all objects for that package. If you use the clean option, remake again without it to get object files again.
The CPU= option allows builders to compile and link the target for a specific architecture—i386 (default if no CPU is specified) or ARM9.
If you intend to use the GDB debugger to debug your code, use the SYMBOLS=ON switch. However, using this switch significantly increases the size of the output files; if this is a concern, then do not use this switch when building production executables.
- Most packages contain an xctest.c file. This file contains a small program that, when compiled and linked with the appropriate libraries, generates a

small executable that tests the functionality of the package. Not all the packages' `xxtest.c` files contain useful tests.

Rebuilding SDK packages

Note: Please note that the default settings for `TECHNOLOGY`, `SYMBOLS` and `CPU`, shown below, can be overridden by specifying them directly on the command line.

To recompile some or all of the SDK source code in the `pkgs` tree, use the `pkgs.mak` make file located in `$INSTALL_FOLDER/pkgs`.

The make file contains several build targets, including:

- Incremental build—Compile files that have changed since the last build, and packages whose build targets are missing:

```
make -f pkgs.mak
```

This is equivalent to:

```
make -f pkgs.mak TECHNOLOGY=ALL SYMBOLS=OFF CPU=i386
```

- Complete build—Remove all object files, build the full source tree, update libraries in `build/bin` and `build/lib`, and then build each sample application and utility and place their executables in `$INSTALL_FOLDER/build/bin/i386`.

```
make -f pkgs.mak complete
```

This is equivalent to:

```
make -f pkgs.mak clean TECHNOLOGY=ALL
```

```
make -f pkgs.mak TECHNOLOGY=ALL CPU=i386 SYMBOLS=OFF
```

```
make -f pkgs.mak build TECHNOLOGY=ALL CPU=i386 SYMBOLS=OFF
```

Note: A complete build must be done at least once before building sample applications or utilities. Also, if the core SDK packages are out of date, rebuild them before building sample applications or utilities.

- Sample application build—Build all of the sample applications in their own directories:

```
make -f pkgs.mak samples
```

This is equivalent to:

```
make -f pkgs.mak samples TECHNOLOGY=ALL SYMBOLS=OFF  
CPU=i386
```

- Utilities build—Build all of the utilities in their own directories:

```
make -f pkgs.mak utilities
```

This is equivalent to:

```
make -f pkgs.mak utilities TECHNOLOGY=ALL SYMBOLS=OFF  
CPU=i386
```

pkgs/

This directory is the root directory within which all the source for the SDK resides in individual subdirectories that are described below. The pkgs folder itself contains three files of interest:

- **pkgs.trg**
Whenever a new package is added to the lineup, this file must be updated to include it. There are two sections to this file, an alphabetical list of package names followed by an ordered list of subdirectories all on one line. The order of packages in the subdirectories list is important and should not be changed without understanding the consequences; for instance, changing the order of the files in this list may cause the image to fail to link during building
- **pkgs.mak**
This is the main makefile for the project. If you want to rebuild the entire world, use this file, first with the clean option, then with the appropriate target (i.e. i386 – default – or ARM9). On most Linux machines building the target takes less than a minute.
- **gen.mak**
This file contains the rules for making packages and the project in general. This file is included by other make files, and it references the gcc compiler chain. If you're using different cross-compile tools and targets, you must edit this file to reference the correct tools and targets.

The remainder of this section provides a short overview of the contents of the subdirectories within the pkgs directory.

pkgs/aa

Contains a single header file, aaglobal.h. This file *must* be included by all other packages such that a package's definitions use the typedefs defined within aaglobal.h¹. This package exists to create a level of abstraction of SDK data types away from standard C types for portability.

pkgs/am

The AM package provides a layer upon which all messages flowing across any IPC channel created and used by the SDK must use. Global functions in this package generally provide other packages with a means to build/parse and send/receive packets across the IPC channels.

pkgs/ap

This is the main package containing all API function source code for the SDK. You can find the source code for all APIs you might call for any purpose. All of the header files in the build/include directory are located in the AP package as well and these should be considered the main source of information in the event of a discrepancy between those in the build/include directory and these ones.

1. Actually, aaglobal includes SwiDataTypes.h, which is where the typedefs reside. Your code and Sierra Wireless code alike should use these types for code associated with the API/SDK.

There are two naming conventions used by files within this package:

SwiApixxx.*
apxxx.*

Files whose names begin with SwiApi contain code and definitions you need to build your applications. The set of files whose names begin with *ap* are test programs useful for quickly testing changes to the SDK and API sources. If you build the SDK using the command

```
make -f pkgs.mak TECHNOLOGY=CDMA
```

from within the pkgs directory, you can then return to the AP package and execute a test version of an application that is used by Sierra Wireless staff during testing of SDK. To execute this test program and to obtain help for it, use the following command from a Linux Bash shell command line:

```
./aptesti386 -- help
```

The file aptest.c contains the *main()* entry point for the API side application used at Sierra Wireless for testing. It can also serve as a reference for you on how to implement the part of your application that interfaces with the APIs, similar to the way the sample code does. Starting the SDK using the aptesti386 command causes the SDK process and an API test program to start. Once the SDK process is started it never stops running unless terminated by a user as discussed earlier.

Additional information regarding the naming conventions followed in this directory are in order:

SwlIntxxx.c—For internal files required for the API to work properly

SwiApixxx.c—For APIs that your application calls.

For the “xxxx” portion of the names, above, you will see files with Cdma, Gsm or Cm followed by a *category* designation. There are several categories identified and they were created as a means of grouping APIs according to their overall function. For instance, APIs for voice features have *Voice* as their category name. Other categories are *Network* for Network-specific APIs, *Basic* for APIs that perform basic operations such as fetching firmware or hardware versions, *Sim*, for APIs related to SIM function on GSM systems, etc.

Please note that for all sample code in this AP directory Sierra Wireless does not perform any range checking on arguments needed by APIs. In general, it is the responsibility of your applications to perform and enforce any range checking required.

When you are executing the SDK process, since there is no available console for *printf()* statements to appear on, the SDK process generates logging messages during runtime. On Ubuntu systems, you can view these runtime logs by executing the following command from a bash command line:

```
tailf /var/log/user.log
```

and this prints the latest log information to the console window every time there's a change to that log file. Start the test application using:

```
./aptesti386 -p ../../build/bin/i386/swisdk -n t1
```

You should be able to see the SDK process running by typing:

```
ps -ef | grep sdk
```

pkgs/apdx

The APDX package contains code that implements the APIs and underlying management software for the Demux Messaging for the SDK.

pkgs/ci

The CI package contains code and data that allow the SDK to support several applications simultaneously. If more (or fewer) applications need to be supported, editing and recompiling is limited to this package.

pkgs/cn

The CN package contains code that runs within the API and SDK sides and the source files in this package are suitably named for this. The header files for this package are common to both sides, thus `cnidefs.h` contains definitions some of which may be used within an API process and others in the SDK process.

The CN package implements the Control and Status (CnS) message layer. CnS is the Sierra Wireless proprietary protocol upon which most of the APIs are built. In general, an API call accepts zero or more arguments, which are formatted into a CnS protocol packet and sent to the modem. Each modem-bound request generates a corresponding response from the modem. The response contains a header and zero or more parameters, which are unpacked within the API process and returned to the caller.

Complex input and output arguments are passed between applications and the API in structures; definitions for these structures can be found within the header file whose name corresponds to the source file containing the API of interest.

pkgs/dd

The DD package contains a small set of functions designed to provide callers with information about the modem device that is connected to the SDK.

pkgs/dl

The DL package provides a flexible logging capability to any package that needs it.

For packages to use the logging facility, they must first register with the DL package (`dlregister()`), then enable logging (`dlmasterenable()`). Callers must also provide a pointer to a control block in these calls and the control block is defined in `dludefs.h`. Therefore any package needing to use the logging facility also needs to include `dludefs.h` in its source. To actually generate a log message at runtime, the `dlLog()` call is used. There are plenty of examples available in the sources that can be `grep`'ed and examined.

Note: The logging facility is available only within the SDK daemon process—it is not available within the API-side.

Logging should be used sparingly and should generally be disabled (*dmlasterenable(..., FALSE)*) to minimize its load on the system. As noted previously, the log output can be viewed on Ubuntu systems in the file:

`/var/log/user.log`

pkgs/dr

The DR (Data Relay) package provides logic within the SDK process for exchanging packets of data between the modem and external applications such as the Diagnostic.c utility.

The package understands how to communicate with the modem to obtain NMEA and modem diagnostic traffic. It interacts with external applications via IPC channels and a specialized set of APIs contained within the APDX package.

Rather than interact directly with IPC channels, this package uses the services of the DS abstraction layer which, in turn, interacts directly with the IPC channel via sockets.

pkgs/ds

The DS package provides a layer of abstraction between serial-style interfaces, such as IPC channels and USB interfaces, and the rest of the threads within the SDK process. It is a low-level package that provides a transparent way for superior threads to exchange data traffic with external applications across IPC channels. The DS package is not permitted to interpret the data in any way; instead, it uses callbacks to send incoming traffic to its client superior thread; it also provides an entry point for sending packets of data to the serial interface, although this entry point runs within the context of the calling thread.

The rule for using the DS package is to create one thread per serial interface needed by the superior thread. The superior thread is responsible for creating and customizing the operation of its DS thread; startup of these threads must be done only from inside the thread context of the superior thread.

pkgs/er

Contains an *erAbort()* facility. By convention, whenever a situation arises in the code (which should theoretically never happen), coders are required to call the non-returning function

`erAbort()`

with an indication of the fault that caused the crash. The information is logged either to the console (API side) or the log file (SDK side) and then the process is terminated. Since this situation should never occur, when it does, there's a serious problem that needs to be addressed to improve the design.

pkgs/hd

This package contains code that frames and deframes packets exchanged between the modem and the SDK process over the HIP endpoint. This is the lowest protocol and above this package is the HIP layer (HP package).

pkgs/hp

This package implements the Sierra Wireless proprietary protocol known as HIP. This protocol provides a simple multiplexing layer so that a single serial interface may be shared by multiple different applications. The Control and Status (CnS) protocol sits atop the HIP layer. Therefore, the CN package builds CnS messages and sends them to the HIP package for transmission to the modem. Incoming CnS messages are routed to the CN package from the HIP layer.

pkgs/ic

This package provides a messaging service between different task-like entities. Processes and threads are both implementations of tasks with different properties. For sending packets between processes, the IC package provides an inter-process communication set of functions. For exchanging messages between threads, a shared memory inter-task message facility is provided but only for the SDK side. The API side offers no such facility as this would typically be up to the customer application to implement.

pkgs/mm

This package is for the SDK side only and provides a facility for allocating pools of fixed-size memory blocks. Threads within the SDK typically create pools for sending/receiving data messages with the modem and also to store data used to send messages from one thread to another.

pkgs/os

This package is a key component of the Linux SDK. It may be noticed when reading through the sources comprising the SDK that there are no Linux system calls in evidence. This is because the OS package provides an abstraction that sits atop the Linux system calls. By convention, SDK code is forbidden to call Linux calls directly and must call the corresponding OS package call instead. There are files in this package that run on both the SDK and API processes.

Together, the convention and the OS package guarantee that only a subset of Linux function calls are required, thus increasing the likelihood that it can be used on several distributions. The OS layer's abstraction also makes porting the SDK to other operating systems a simpler task. (See [SDK Portability](#) on page 89.)

pkgs/pi

The PI package provides facilities for packing and unpacking bytes from byte arrays in network byte order. By convention, multibyte fields can only be written/read into/from network packets using the facilities of the PI package.

pkgs/ql

This package contains a simple facility for managing doubly-linked queues.

pkgs/sdk

This package contains the *main()* function for the SDK-side process and also the SDK process's executable image. For example, from an application's point of view, there is no difference if the SDK process is started by manually typing

```
./sdi386
```

in this directory or by running the main application and having that application call **SwiApiStartup()**.

pkgs/sl

Contains functions for manipulating data in strings and byte arrays.

pkgs/su

Contains a small amount of code that coordinates the initialization and startup of threads within the SDK process.

pkgs/us

Contains code that provides read/write access to specified endpoints enumerated by the modem and provides device detection and scanning services.

>> 17: SDK Portability

17

The Sierra Wireless Linux SDK is designed to be easily portable to other versions of Linux and even to other embedded operating systems. This section summarizes the areas that will likely require attention to be successful in porting the SDK to other operating systems.

Note: Sierra Wireless provides technical support only to customers using the platforms described in [Development systems](#) on page 16.

Operating System wrapper layer

A glance through the SDK source code reveals that Linux system calls are only used in one of the several packages that comprise the SDK codebase. This is possible because all calls to the Linux OS are limited to a single SDK package, called the OS package. By convention, all other packages in the SDK are forbidden to make Linux system calls directly and must instead obtain the services offered by Linux by calling the corresponding function in the OS package. Anyone involved in making changes to the SDK source code is strongly advised¹ to maintain this convention. For example, if developers discover a call provided by the new target operating system is not provided by the SDK's OS wrapper package, instead of putting the call directly into the source code where it is required, a new OS wrapper package function should be provided and that new function call should be made instead.

When porting this SDK to another embedded operating system, a large part of the effort will be focused on converting the code within the OS package into calls to the new target operating system functions. If this conversion is approached with due care and attention, then the remainder of the SDK sources will likely not need to be touched.

To port the SDK to another operating system, the following services must be created using operating system calls from the target OS:

- Thread creation
- Semaphore creation and management calls
- Conditional wait and timed wait services
- USB Device Detection
- USB read/write calls
- Access to the file system

1. Please note, we are not suggesting that *your application* be limited to services provided by the SDK's OS wrapper layer. On the contrary, applications are free to interact with their computing environment any way they please.

- Access to the system logging mechanism
- Signal handling
- Inter-process communication services

Porting to other versions of Linux

Porting the SDK to other versions of Linux is usually as simple as recompiling the sources for the new target. Occasionally, even the recompilation step is not required because the as-delivered binaries are already compatible with other distributions of Linux. The SDK has been intentionally designed to use as small a set of Linux system calls as possible to reduce the likelihood of incompatibilities with other systems.

Porting to other embedded operating systems

Depending on the type and purpose of the new target system, you might need to review the design of some aspects of the SDK. A summary of these areas includes:

- Process and thread models ([page 90](#))
- Device detection ([page 92](#))
- Sending/receiving via USB ([page 92](#))
- The USB driver ([page 92](#))

A brief description of the issues follows in the next few sections.

SDK process model

In Linux a process and a thread are similar constructs. A process is created whenever a program is started on a Linux machine. Two separate processes are able to interact, but only through a variety of inter-process communication mechanisms supported by the operating system—Linux, in this case. The two processes have virtual address spaces, meaning that the two cannot simply select a common memory address and exchange information by reading/writing into this address without support for this from the kernel. Inter-process communication is possible, but only if the processes needing it both take some type of action that involves invoking calls to the operating system.

The Linux SDK consists of two processes—one supplied by the application developer (also referred to in this document as the API-side) in which the APIs are called and one built into the SDK by design. The former process causes the SDK process or *SDK daemon* (also referred to in this document as the SDK-side)—to start and run when it calls **SwiApiStartup()**. The two processes must interact to implement APIs, and the SDK facilitates this interaction by using the socket datagram protocol of the Unix address family. Details of starting this are hidden from the application in the underlying SDK codebase – however the OS wrapper layer (discussed in [Operating System wrapper layer](#) on page 89) and the IC package ([pkgs/ic](#) on page 87) play a significant role.

If porting the SDK to a different operating system, the need for processes may disappear particularly if the target product is an embedded processor and the entire product executes effectively in kernel mode (i.e. no user-mode available). If there is no requirement or ability to support processes in the target environment, then it would be necessary to convert the processes into tasks and replace the existing inter-process communication mechanism with something more efficient for the target environment. The IC package already provides a shared memory inter-task messaging facility that could be used instead of IPC channels.

SDK thread model

Just as a process in Linux (or collection of processes) represents a single application, a thread represents a single processing function within a process. In other words, a process may contain one or more threads. Since threads share a common parent process, they share a common memory space and can exchange messages by passing memory addresses between each other without requiring special setup through the operating system.

The OS wrapper layer supports thread creation within the SDK daemon component, but not the application component. Applications are free to create threads in any way they require, using whatever system calls they need to do so, although the application must provide at least 2 threads for the API-side, as mentioned earlier.

In the event the SDK is ported to an environment where processes are no longer supported then all the threads in the SDK daemon would become standalone entities, often referred to as tasks, in most embedded operating systems. The SDK threads would need to be converted into tasks but the relationship between them would not change.

API call handling

Virtually every API in the SDK is implemented as a blocking call, meaning that the thread that the API is called from is stopped until the function returns. When an API call is made, a data packet is assembled, using the information provided by the caller. The packet is sent to the SDK daemon over an inter-process communication channel and then passed on to the modem. The call remains blocked until the modem sends a response or the timeout period elapses, which can take between several milliseconds to several seconds, depending on the request.

The API supports notifications, which are unsolicited messages that the modem sends whenever there is a significant change to some operating parameter. The modem may send a notification at any time and there is no way to predict when a notification may arrive. Note that applications must first enable notifications by calling the appropriate API – **SwiNotify()**. A callback mechanism is used, meaning that the SDK calls a user-supplied function whenever a notification is received.

If only a single thread were provided, then notification traffic would suffer unnecessary delays. Instead applications must provide one thread for handling notifications and another one for handling the blocking API calls. With this

arrangement, notifications are never forced to wait until a pending API call is complete before they get some service and this translates into an overall better response to the application.

Device detection

The SDK detects the presence or absence of a Sierra Wireless modem. A separate thread within the SDK daemon is responsible for device detection—it resides in the US package.

If porting this SDK to other operating systems, you need to rework the device detection mechanism as well, depending upon how your system is configured. For embedded systems, device detection is generally much simpler than for Linux systems. For instance, in an embedded product, from the SDK daemon's point of view, the question might change from "is there a modem installed and if so, what type is it?" to "is the modem booted and running?"

USB access

All communication between the modem and the host computer takes place via USB; the SDK's OS wrapper package contains special functions for interacting with the modem via USB. Therefore any attempt to port the SDK to another operating system requires some special attention to the USB entry points in the OS layer. For instance, the SDK uses endpoint numbers to interact with the OS wrapper layer when sending or receiving packets. However, Linux and the Sierra Wireless device driver create file handles in `/dev/ttyUSBn`. USB reads and writes ultimately take place through these file handles and the OS wrapper layer performs the translation from endpoint to file handle internally.

Porting the SDK to another system would likely offer different APIs through the wrapper layer which use some proprietary method of accessing the USB endpoints, especially for smaller embedded systems that aren't based on Linux.

Sierra Wireless driver considerations

The driver's job is to detect the presence of a modem and create links to it in the form of file handles in the directory:

`/dev`

Depending upon the new target environment, porting the driver might be as simple as recompiling it or as complicated as having to write a new one from scratch. Porting it to other versions of Linux with the 2.6+ kernel, for example, is likely to require only a recompile and possibly other minor changes, while porting to a 2.4-based Linux distribution or a completely different operating system would require a complete rewrite.

There is no direct interaction between the Sierra Wireless Linux driver and the SDK. The SDK's job is to exchange packets with a modem over the USB interface by opening the handles in the `/dev` directory for reading or writing. Without the driver there would be no way to send or receive packets, but the SDK makes no calls to the driver either directly or indirectly.

Note: Sierra Wireless modems appear to host computers as USB devices.

>> Index

Symbols

*_api.c, 81
*_sdk.c, 81
*.mak, 81
*.txt, 81
*idefs.h, 76
*iproto.h, 76
*test.c, 81
*udefs.h, 76
*uproto.h, 76
\$INSTALL_FOLDER, 19

Numerics

1X data connection
 description, 53
 dial string, 55
 originate a call, 44
 password, 38
1xEV-DO data connection
 description, 53
 dial string, 55
 originate a call, 44
 password, 38
3-Way calling, 44

A

aaglobal.h, 83
account information, 37
account, manually activate, 35
activation
 overview, 35
 manual activation, 35
 state, determining, 37
active (PDSM) client, 63
air server
 available, 28
 not available, 29
alert, service, 49
alerts, 49
almanac data
 clear, 64
 download, 64
answer a call, 43, 55
ap*.*, 84
API layer, 23
API reference, online, 17
*_api.c, 81
application
 building, 75
 multiple application support, 21
 startup, 32
architecture, software, 21
asymmetric notifications, 23, 24

audio services, 47

B

band, radio, determining, 59
base station
 GPS location, 61
 protocol revision, 60
beeps, 49
"bin" folder, 78
blocking call, 91
bootstrap loader - version, 58
"build" folder, 77
"build/bin" folder, 78
"build/include" folder, 78
"build/lib" folder, 78
building
 application, 75
 packages, 82
byte counter, 61

C

call
 end (voice), 45
 originate, 44
 See also 1X data connection, and 1xEV-DO data connection.
call handling, 91
Call Waiting, 43
call, voice
 end, 45
callback function
 deregister, 30
 register, 25
Caller ID, 43
CDMA deep sleep, 58
CDMA network
 availability, 54
 time, 61
channel number, 59
characteristics of modem, 57
circuit-switched data connection
 description, 53
 answer call, 55
 dial string, 55
 end a call, 45
 originate, 54
clients, PDSM, types of, 63
closing the host application, 34
cnidefs.h, 85
CnS (Control and Status) message layer, 85, 87
CNS_CALL_TYPE_ASYNC_DATA, 44
CNS_CALL_TYPE_PACKET_DATA, 44
CNS_CALL_TYPE_VOICE, 44
cntRemainingSMS, 40

- cold start, simulate, [64](#)
- comfort tones, [49](#)
- communication with modem, [23](#)
- compiler chain, gcc, [83](#)
- concurrent applications, [21](#)
- conference calling, [44](#)
- connection
 - overview, [53](#)
 - end, [55](#)
 - originate, [54](#)
 - types, [53](#)
- contents of SDK, [15](#)
- Control and Status (CnS) message layer, [85](#), [87](#)
- conventions
 - naming of source files, [81](#)
- coverage, network
 - functions, [60](#)
- CSD data connection
 - description, [53](#)
 - answer call, [55](#)
 - dial string, [55](#)
 - end a call, [45](#)
 - originate, [54](#)

D

- data connection
 - overview, [53](#)
 - dial string, [55](#)
 - end, [55](#)
 - originate, [54](#)
 - password, [38](#)
 - types, [53](#)
- Data Relay (DR) package, [86](#)
- data structures, in online documentation, [17](#)
- data traffic, handling of, [23](#)
- date, returned from CDMA network, [61](#)
- default position determination parameters, [66](#)
- defines, in online documentation, [17](#)
- demultiplexing APIs, [69](#)
- Demux Messaging, [85](#)
- detection of modem, [92](#)
- development systems
 - system requirements, [15](#)
- device detection, [92](#)
- device driver
 - See driver.
- Diagnostic utility, [80](#)
- Diagnostic.c, [86](#)
- dial a call, [44](#)
- dial string, [55](#)
- dial tones, [49](#)
- directory tree contents, [77](#)
- disable the autolock, [51](#)
- display log information, [84](#)
- distributing files, [19](#)
- documentation, online, [17](#)
- download ephemeris and almanac data, [64](#)

- download session
 - terminate, [66](#)
- DR (Data Relay) package, [86](#)
- driver
 - considerations re porting, [92](#)
 - installing, [19](#)
 - layer, [23](#)
 - obtaining, [15](#)
 - unloading and reloading, [30](#)
- DTMF
 - description, [48](#)
 - mute tone, [48](#)
 - play tones, [49](#)
 - tone volume, [48](#)

E

- embedded operating systems, other, porting to, [90](#)
- end a data connection, [55](#)
- end a voice call, [45](#)
- endpoint, file handle for, [23](#)
- Enhanced Roaming Indicator (ERI), [58](#)
- enumerations, in online documentation, [17](#)
- environment, setting up, [19](#)
- ephemeris data, [64](#)
- erAbort(), [86](#)
- ERI (Enhanced Roaming Indicator), [58](#)
- error codes, [73](#)
- error handling, [73](#)
- ESN
 - during activation, [35](#)
 - SwiGetESN, [57](#)
 - SwiGetMEIDESN, [57](#)
- EVDO
 - See 1xEV-DO data connection.
- executable image of SDK process, [88](#)

F

- file folder structure, [77](#)
- file handle for endpoint, [23](#)
- firmware
 - layer, [23](#)
 - version (function SwiGetFirmwareVersion), [57](#)
- Firmware Updater utility
 - executables, [78](#)
 - source code, [80](#)
- fix. single location, [64](#), [65](#)
- flash command, [44](#)
- folder contents, [77](#)
- function call handling, [91](#)
- function prototypes, [17](#)
- functions - in online API documentation, [17](#)

G

- gcc compiler chain, [83](#)
- gen.mak, [83](#)

"global" keyword, 76
GPS IP Address, 66
GPS Lock, 66
GPS NetAccess, 66
GPS PortID, 66
GPS Privacy, 66
GPS PTLM Mode, 66
GPS. See location-based services, and PDSM.

H

hardware version, 58
HDR
 notifications that are enabled, 34
 password, 38
 user ID, 38
 See also 1xEV-DO data connection.
header files
 must be included - aaglobal.h, 83
 in online documentation, 17
headset, 47
HIP protocol, 87
host application
 building, 75
 closing, 34
 opening, 31
host application layer, 23
host-initiated requests/responses, 23, 24

I

IC (inter-process communication) functions, 87
*idefs.h, 76
IMSI
 function SwiSetIMSI, 36
in service, 60
include files, minimum set of, 78
"include" folder, 78
incoming call notification, 43
initializing the API, 28
\$INSTALL_FOLDER, 19
installation
 device driver, 19
 SDK, 19
inter-process communication (IC) functions, 87
IOTA, 37
IP Address, GPS, 66
IPC channels, 86
*iproto.h, 76

K

key tones, 48
keyword "global", 76
keyword "local", 76
keyword "package", 76

L

layer, OS wrapper, 89
layers, software, 21
LBS. See location-based services, and PDSM.
"lib" folder, 78
Linux system calls, 87
"local" keyword, 76
location fix session
 begun, 64
 end, 66
location fix, single, 64, 65
location of base station, 61
location of terminal
 request, 63
 track, 64
location-based services
 location fix session, 64, 66
 See also PDSM.
lock code (passcode)
 changing, 51
lock modem on startup, 51
lock state, 51
logging
 messages generated by SDK, 84
 use sparingly, 86

M

main() function for the SDK-side process, 88
"main" routine common to all sample apps, 79
*.mak, 81
make files
 building an image for a package, 81
 gen.mak, 83
 libraries to include, 75, 78
 pkgs.mak - main makefile, 83
manual activation, 35
MDN (Mobile Directory Number)
 description, 35
 entering during activation (function SwiSetMobileDir-
 Num), 35
MEID
 during activation, 35
 SwiGetMEIDESN, 57
messaging, short text
 See SMS.
microphone mute, 48
MIN (Mobile Identification Number), 35
minute marker, 49
Mobile Directory Number (MDN)
 description, 35
 entering during activation (function SwiSetMobileDir-
 Num), 35
Mobile Equipment IDentifier. See MEID.
Mobile Identification Number (MIN), 35
mobile-originated SMS, 40
mobile-terminated SMS, 39

modem

- availability, determining, [32](#)
- characteristics, [57](#)
- detection, [92](#)
- driver layer, [23](#)
- driver, obtaining and installing, [15](#)
- notifications, modem-initiated, [23](#), [24](#)
- reset, [30](#), [37](#)
- status, determining, [54](#)

modem-initiated notifications, [23](#)

MO-SMS, [40](#)

MSID, [36](#)

MT-SMS, [39](#)

multiple application support, [21](#)

mute

- DTMF tone, [48](#)
- microphone, [48](#)
- speaker, [47](#)

N

NAI (Network Access Identifier), [38](#)

NAM (Number Assignment Module)

- description, [35](#)
- components, [35](#)
- reading the account profile, [37](#)

NetAccess, [66](#)

Network Access Identifier (NAI), [38](#)

Network Access permissions used during PDSM sessions, [66](#)

network availability, [54](#)

Network ID, [35](#), [36](#)

network time, [61](#)

NID (Network ID), [35](#), [36](#)

NMEA traffic, [71](#)

notifications

- disabling, [34](#)
- enabling, [33](#)
- managing, [25](#)
- modem-initiated, [24](#)
- permanently enabled, [33](#)
- processing, [34](#)
- See also SWI_NOTIFY...

Number Assignment Module (NAM)

- description, [35](#)
- components, [35](#)
- reading the account profile, [37](#)

NV

- LBS items, [34](#)

O

OMA Device Management (OMA-DM)

- activation via, [36](#)

OMA DM (Device Management)

- activation via, [36](#)

online API reference, [17](#)

opening the host application, [31](#)

originate a call, [44](#)

OS wrapper layer, [89](#)

OTASP, [36](#)

P

"package" keyword, [76](#)

package services, [76](#)

packages

- adding - files that must be edited, [83](#)
- contents, [80](#)
- rebuild, [82](#)
- services, [76](#)

packet data

- description, [53](#)
- dial string, [55](#)
- See also 1X, and 1xEV-DO.

passcode (lock state), [51](#)

password

- 1X, [38](#)
- 1xEV-DO, [38](#)
- accessing the service provider's data services, [35](#)
- changing, [51](#)
- Simple IP, [36](#), [38](#)

PDSM

- clients, types of, [63](#)
- Port ID used in PDSM sessions, [66](#)
- position determination, [66](#)
- See also location-based services.

permanently enabled notifications, [33](#)

pESN

- during activation, [35](#)
- SwiGetESN, [57](#)
- SwiGetMEIDESN, [57](#)

phone number, [37](#)

PIN, change, [51](#)

pkgs folder, [83](#)

pkgs.mak, [83](#)

pkgs.trg, [83](#)

Port ID used in PDSM sessions, [66](#)

portability, SDK, [89](#)

porting to

- Linux, other versions of, [90](#)
- other embedded operating systems, [90](#)

position determination, [66](#)

- See also PDSM.

power save status, [58](#)

power state changes, [58](#)

PPP, [54](#)

PREV (protocol revision)

- retrieving, [60](#)

PRI (Product Release Instructions), [39](#)

print log information to the console, [84](#)

Privacy, GPS, [66](#)

process model, [70](#), [90](#)

Product Release Instructions (PRI), [39](#)

properties of modem, [57](#)

protocol revision (PREV)

- retrieving, [60](#)

protocols
 CnS (Control and Status), 85, 87
 HIP, 87
 provisioning date, 36
 PTLM Mode, 66

Q

QNC data connection
 description, 53
 dial string, 55
 originate a call, 44
 Quick Net Connect data connection
 description, 53
 dial string, 55
 originate a call, 44

R

radio band, determining, 59
 radio temperature, 58
 range checking
 not done in sample code, 84
 rebuild packages, 82
 receiving SMS, 39
 reference material, 16
 registered (PDSM) client, 63
 Relay Agent utility, 80
 release notes, 79
 requesting (PDSM) client, 63
 requests
 description, 24
 requests/responses
 host-initiated, 23, 24
 patterns, 24
 requirements, system, 15
 reset, modem
 description, 37
 handling, 30
 SWI_NOTIFY_Reset, 33
 responses, 24
 return code of functions, 24
 roaming, 60
 RSSI, 59

S

sample applications, 79
 sample code
 range checking not done in, 84
 root folder, 79
 Sample_Code folder, 79
 Sample_Code folder, 79
 satellite information, 67
 *.sdk.c, 81
 security, 51
 send SMS, 40

service
 alert, 49
 availability, 54
 characteristics, 58
 indication, 60
 setup of SDK, 19
 short message service
 See SMS.
 shutting down the API, 30
 SID (System Identifier), 35, 36
 signal strength (RSSI), 59
 Simple IP, 38
 simulate a cold start, 64
 single location fix, 64, 65
 SMS
 overview, 39
 receiving, 39
 sending, 40
 software architecture, 21
 source code
 main package, 83
 speaker, 47
 startup of application, 32
 statistics, 61
 status, modem
 determining, 54
 structure descriptions, 17
 SWI_CALL_TYPE_Voice, 36
 SWI_NOTIFY_AirServerChange, 29, 30, 33
 SWI_NOTIFY_CallByteCounter, 61
 SWI_NOTIFY_CallConnected, 33, 43, 55
 SWI_NOTIFY_CallConnecting, 43, 44, 55
 SWI_NOTIFY_CallDisconnected, 33, 55
 SWI_NOTIFY_CallDormant, 33
 SWI_NOTIFY_CallError, 33
 SWI_NOTIFY_CallIncoming, 33, 43, 55
 SWI_NOTIFY_CallNotificationStatus
 notifications it enables/disables, 33
 relation to other notifications, 54
 SWI_NOTIFY_ChannelNumber, 59
 SWI_NOTIFY_ChannelState, 59
 SWI_NOTIFY_Hdr_PowerSave_Enter, 58
 SWI_NOTIFY_Hdr_PowerSave_Exit, 58
 SWI_NOTIFY_Hdr_Roam_Status, 60
 SWI_NOTIFY_HeadsetState, 47
 SWI_NOTIFY_IOTA_Status, 37
 SWI_NOTIFY_LbsPaGpsLock, 34, 67
 SWI_NOTIFY_LbsPaPalpAddr, 34, 67
 SWI_NOTIFY_LbsPaNetAccess, 34, 67
 SWI_NOTIFY_LbsPaPortID, 34
 SWI_NOTIFY_LbsPaPortId, 67
 SWI_NOTIFY_LbsPaPrivacy, 34, 67
 SWI_NOTIFY_LbsPaPtlmMode, 67
 SWI_NOTIFY_LbsPaPtlmMode, 34
 SWI_NOTIFY_LbsPdBegin, 33, 64
 SWI_NOTIFY_LbsPdData, 33, 63
 SWI_NOTIFY_LbsPdDloadBegin, 33, 64
 SWI_NOTIFY_LbsPdDloadData, 33, 64
 SWI_NOTIFY_LbsPdDloadDone, 33, 64

SWI_NOTIFY_LbsPdDloadEnd, 33, 64
 SWI_NOTIFY_LbsPdDone, 33, 64
 SWI_NOTIFY_LbsPdEnd, 33, 64
 SWI_NOTIFY_LbsPdUpdateFailure, 33
 SWI_NOTIFY_LockAirServer, 33
 SWI_NOTIFY_OTASPSState, 33, 36
 SWI_NOTIFY_Power, 58
 SWI_NOTIFY_PowerSaveEnter, 58
 SWI_NOTIFY_PowerSaveExit, 58
 SWI_NOTIFY_Prev, 54
 SWI_NOTIFY_Reset, 33
 SWI_NOTIFY_RoamingStatus, 60
 SWI_NOTIFY_Rssi, 59
 SWI_NOTIFY_ServiceIndication
 relation to other notifications, 54
 SWI_NOTIFY_SmsSendStatus, 41
 SWI_NOTIFY_SmsStatus, 39
 SWI_NOTIFY_TechBandClass, 59
 SWI_NOTIFY_TechHdrNotificationStatus, 34
 SWI_RCODE, 73
 SWI_STRUCT_CDMA_SMS_RetrieveSms, 40
 SWI_STRUCT_CDMA_SMS_StoreSms, 40
 SWI_STRUCT_SMS_CdmaHeader, 40
 SWI_TYPE_FlashContext_DontCare, 44
 SwiApi*.*, 84
 SwiApi*.c, 84
 SwiApiCmBasic.h, 78
 SwiApiDxBegin, 71
 SwiApiDxEnd, 71
 SwiApiDxRegister, 71
 SwiApiDxSend, 71
 SwiApiDxStartup, 71
 SwiApiReference folder, 79
 SwiApiStartup, 27, 29, 31
 SwiApiWaitNotification, 25, 33
 SwiChangeLockPIN, 51
 SwiDataTypes.h, 78
 SwiDeRegisterCallback, 30
 SwiGetActivationStatus, 37
 SwiGetAvailAirServers, 27, 29, 31
 SwiGetBootVersion, 58
 SwiGetByteCounter, 61
 SwiGetCallConnectionState, 43, 44, 55
 SwiGetCallerID, 43
 SwiGetCdmaSMSMessageStatus, 39
 SwiGetCdmaSpeakerVolume, 47
 SwiGetChannelNumber, 59
 SwiGetCurrentBaseStation, 54, 60
 SwiGetDataPldOffset, 71
 SwiGetDTMFMTStatus, 48
 SwiGetERIVer, 58
 SwiGetESN
 description, 57
 during activation, 35
 SwiGetFirmwareVersion, 57
 SwiGetHardwareVersion, 58
 SwiGetHdrNotificationStatus, 34
 SwiGetHdrRoamStatus, 60
 SwiGetHdrSrvState, 54, 60
 SwiGetHeadsetState, 47
 SwiGetIOTALog, 37
 SwiGetLastError, 73
 SwiGetLbsDloadNotifyStatus, 33
 SwiGetLbsLocNotifyStatus, 33
 SwiGetLbsPaGpsLock, 66
 SwiGetLbsPaPalAddr, 66
 SwiGetLbsPaNetAccess, 66
 SwiGetLbsPaParam, 66
 SwiGetLbsPaPortId, 66
 SwiGetLbsPaPrivacy, 66
 SwiGetLbsPaPtmMode, 66
 SwiGetLbsParamNotifyStatus, 34
 SwiGetLbsPdData, 63
 SwiGetLbsSatInfo, 67
 SwiGetLockStatus, 51
 SwiGetMEIDESN
 description, 57
 during activation, 35
 SwiGetMICMuteStatus, 48
 SwiGetPhoneNumber, 37
 SwiGetProvisioningDate, 36
 SwiGetRoamingStatus, 60
 SwiGetRssi, 59
 SwiGetRssiEcio, 59
 SwiGetServiceIndication, 54, 60
 SwiGetSpeakerMuteStatus, 47
 SwiGetSystemTime, 61
 SwiGetTechBSInfo, 61
 SwiGetTechHdrNotificationStatus, 34
 SwiGetTechPowersaveMode, 58
 SwiGetTemperature, 58
 SwiGetToneLevel, 48
 SwiHWGetPower, 58
 SwiInt*.c, 84
 SwiIsModemReady, 31
 SwiLockModem, 51
 SwiNotify, 27, 31, 33
 SwiPowerSaveExit, 58
 SwiPstGetCurrentNam, 37
 SwiPstGetHdrPassword, 38
 SwiPstGetHdrUserId, 38
 SwiPstGetMinuteCallBeep, 49
 SwiPstGetServiceAreaAlert, 49
 SwiPstGetSipUserId, 38
 SwiPstLock, 36
 SwiPstSetHdrPassword, 38
 SwiPstSetHomeSidNid, 36
 SwiPstSetMinuteCallBeep, 49
 SwiPstSetServiceAreaAlert, 49
 SwiPstSetSipPassword, 38
 SwiPstSetSipUserId, 38
 SwiPstUnlock, 35
 SwiRcodes.h, 24, 78
 SwiRegisterCallback, 25, 27, 29, 31
 SwiResetModem
 in activation process, 36
 SwiRetrieveCdmaSMSMessage, 40
 swisdk executable, 19

- SwiSendCdmaSMS, 40
- SwiSetAnswerState, 43, 55
- SwiSetAutoLock, 51
- SwiSetCallAnswerCmd, 43
- SwiSetCallDisconnectCmd, 45, 55
- SwiSetCallFlashCmd, 44
- SwiSetCallOriginateCmd, 44, 54
- SwiSetCdmaSpeakerVolume, 47
- SwiSetDTMFKey, 49
- SwiSetIMSI, 36
- SwiSetKeyPressed, 49
- SwiSetKeyReleased, 49
- SwiSetLbsClearAssistance, 64
- SwiSetLbsDloadNotifyStatus
 - notifications it enables/disables, 33
- SwiSetLbsLocNotifyStatus
 - notifications it enables/disables, 33
- SwiSetLbsPaGpsLock, 67
- SwiSetLbsPaPalAddr, 67
- SwiSetLbsPaNetAccess, 67
- SwiSetLbsPaPortId, 67
- SwiSetLbsPaPrivacy, 67
- SwiSetLbsPaPtlmMode, 67
- SwiSetLbsParamNotifyStatus
 - notifications it enables/disables, 34
- SwiSetLbsPdDownload, 64
- SwiSetLbsPdEndSession, 66
- SwiSetLbsPdGetPos, 64
- SwiSetLbsPdTrack, 65
- SwiSetMICMuteStatus, 48
- SwiSetMobileDirNum, 35
- SwiSetSpeakerMuteStatus, 47
- SwiSetToneLevel, 48
- SwiStartIOTASession, 37
- SwiStopIOTASession, 37
- SwiStopNotify, 30, 33, 34
- SwiStoreCdmaSMSMessage, 40
- symmetric notifications, 23, 24
- system calls, Linux, 87
- System Identifier (SID), 35, 36
- system requirements, 15

T

- Tech HDR notifications, 34
- temperature, 58
- terminal location
 - request, 63
 - track, 64
- terminate (answer) a call, 43

- test program, 84
- *test.c, 81
- text messaging
 - See SMS.
- thread model, 91
- three-way calling, 44
- time, CDMA network, 61
- timeouts, 24
- tone volume, DTMF, 48
- track the location of terminal, 64
- tty's, accessing directly, 70
- ttyUSB0, 55
- *.txt, 81
- Typedefs, in online documentation, 17

U

- *udefs.h, 76
- upgrading from a previous version of the SDK, 19
- *uproto.h, 76
- USB access, 92
- user ID
 - HDR, 38
 - Simple IP, 38
 - to access service provider's data services, 35
- Utilities folder, 80
- utility, Diagnostic, 80
- utility, Firmware Updater
 - executables, 78
 - source code, 80
- utility, Relay Agent, 80

V

- variables, in online documentation, 17
- version information
 - firmware, 57
- voice functions, 47
- volume
 - DTMF tone, 48
 - microphone, 48
 - mute, 47
 - speaker, 47

W

- Wireless Local Number Portability (WLNP), 36
- WLNP (Wireless Local Number Portability), 36
- wrapper layer, 89



SIERRA
WIRELESS™