



UMTS Linux SDK

Developer's Guide



SIERRA
WIRELESS™

2131102
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	November 2008	New document
1.1	November 2008	General document updates
1.2	April 2009	General update, added Direct IP content

Revision number	Release date	Changes
1.3	October 2009	Added content/wording changes for new Direct IP driver (sierra_net) Added multiple simultaneous host applications support content Updated API initialization section Replaced Rebuilding SDK packages Added pkgs/cl description
2	June 2011	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 drivers	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	29
Shutting down the API	30

Host application API usage	31
Opening the host application	31
Checking that the modem is available	31
Powering the modem down and up	32
Handling notifications	32
Enabling network registration	33
Closing the host application	33
SIM Authentication and Codes	35
Using these API functions	35
Using a MEP code to unblock the modem.	35
SIM security	36
Account Profile Management	41
Using these API functions	41
Account profile overview	41
Profile maintenance functions	41
Identifying account profiles	42
Reading profiles	42
Creating and editing profiles	42
Setting a default profile to autoactivate	42
Activating a profile	42
Deleting profiles	42
Network registration	43
Using these API functions	43
Registering on a network.	43
Location-based services	47
Using these API functions	47
Retrieving operational settings	47
Position fix / tracking sessions.	48
Ephemeris / almanac data.	50

Phone Book Maintenance	51
Using these API functions	51
Supported phone books	51
Using over dial numbers	52
Using the phone book functions	53
Application start-up	53
Maintaining ADN, FDN, MSISDN phone books	53
Using the FDN phone book	53
SIM phone book statistics	54
Retrieving phone numbers	54
Retrieving emergency numbers	54
Phone book retrieval	54
Data Connections	57
Using these API functions	57
Establishing a point-to-point (dialup) data connection	57
Modem and SIM characteristics	59
Demultiplexing APIs	61
Using these API functions	61
Process model	62
Supported services	62
Startup, normal operation, and shutdown	62
Error handling	65
Error codes	65
Handling errors	65
Building an Application	67
Building an application – SDK libraries	67

- Package breakdown 67
 - Package naming convention 68
 - Package services 68
- SDK directory tree contents. 69
 - Packages 72
- SDK Portability 81**
 - Operating system wrapper layer 81
 - Porting to other Linux versions 82
 - Porting to other embedded operating systems 82
 - SDK process model. 82
 - SDK thread model 83
 - API call handling 83
 - Device detection 84
 - USB access 84
 - Sierra Wireless driver considerations 84
- Index. 85**

List of Tables

Table 2-1: Locations of information in API documentation	17
Table 10-1: Supported phone books	51
Table 12-1: Component characteristics functions	59

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 6-1: CHV1 verification process.	37
Figure 6-2: Enabling/disabling CHV1	38
Figure 8-1: Manual PLMN selection	45
Figure 9-1: Initiating a single position fix	49
Figure 10-1: Retrieving an entire phone book	55
Figure 13-1: Process model.	62
Figure 13-2: Data flow: Demux application	64
Figure 15-1: SDK file folder structure	69

>> 1: About This Guide

1

SDK introduction

The Linux Software Development Kit (SDK) allows software developers to create Linux-based applications for Sierra Wireless' UMTS 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 drivers. The drivers are open source and are distributed independently of the SDK. For information on obtaining and installing the drivers, 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 Linux 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

These additional references may help you use the SDK to develop your applications:

- 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.
- GSM Association (www.gsmworld.com)
- Palowireless Resource Control (www.palowireless.com)—Links to sources for GPRS/EDGE/UMTS articles, white papers, definitions, etc.

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](#) outlines 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
Header files	Files > File List
Functions	Files > Globals > Functions
Typedefs	Files > Globals > Typedefs
Enumerations	Files > Globals > Enumerations
Defines	Files > Globals > Defines

>> 3: SDK Installation and Setup

Sierra Wireless delivers the Linux SDK as a zipped .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 69.

Installing the device drivers

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

Distributing files

When you distribute your host applications 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 drivers interface with Linux.

Software architecture

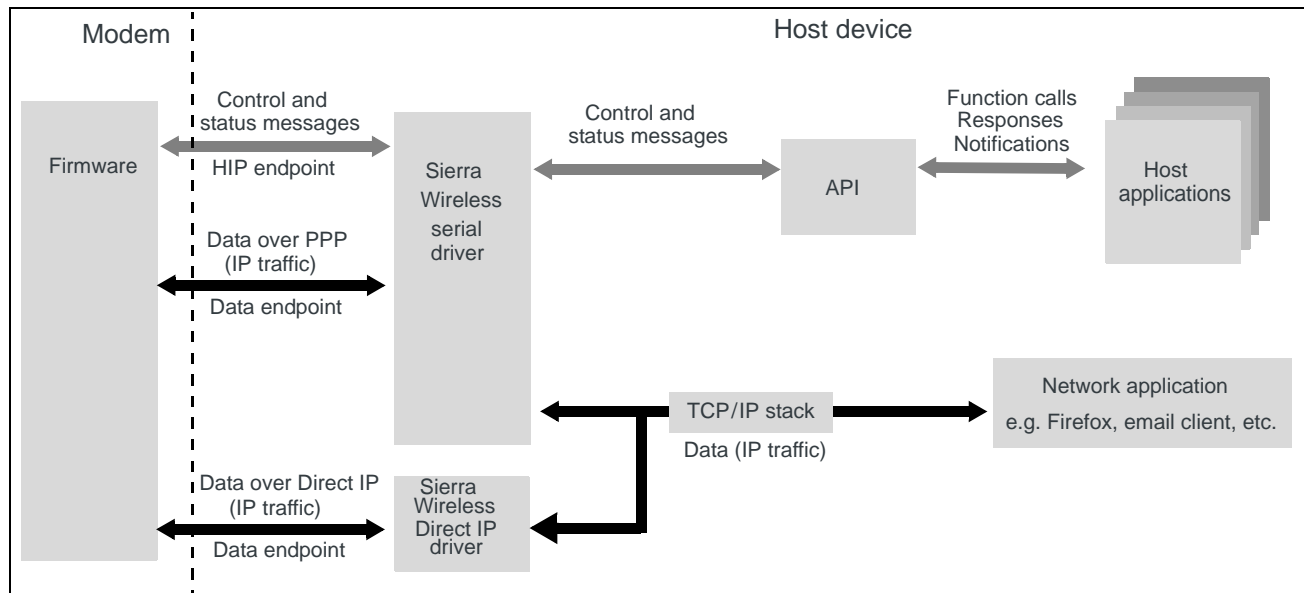


Figure 4-1: Software layers

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, as shown in [Chapter 13 on page 61](#). 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 drivers, 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 the application's responsibility to establish a data connection. For Direct IP modems, this can be done using the **SwiActivateProfile()** API.*

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](#) 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.

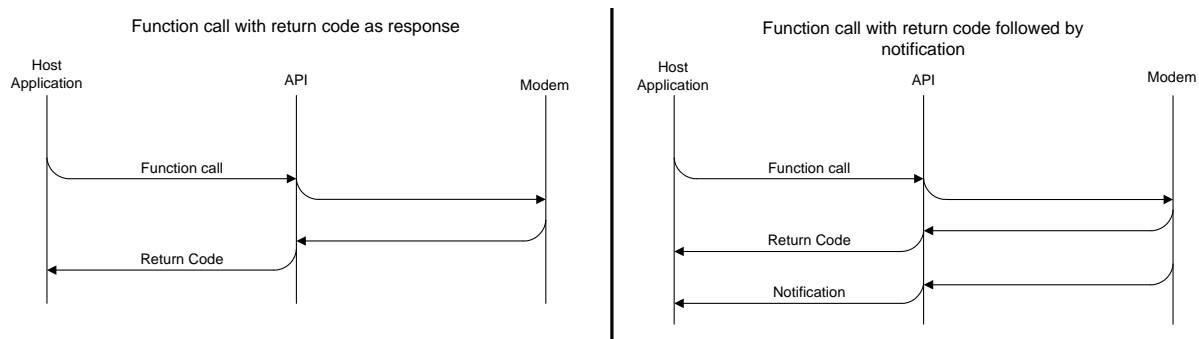


Figure 4-2: Request-response patterns

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_NetworkStatus** rather than frequently calling the function **SwiGetServiceStatusEx**.*

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 a SIM notification before calling additional SIM functions, but can call other functions (such as LBS) 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.
- These notifications occur when specific system state changes occur (for example, available networks, modem temperature, etc.).

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.

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 32 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

The following figure shows the interaction between the API and other software and hardware components.

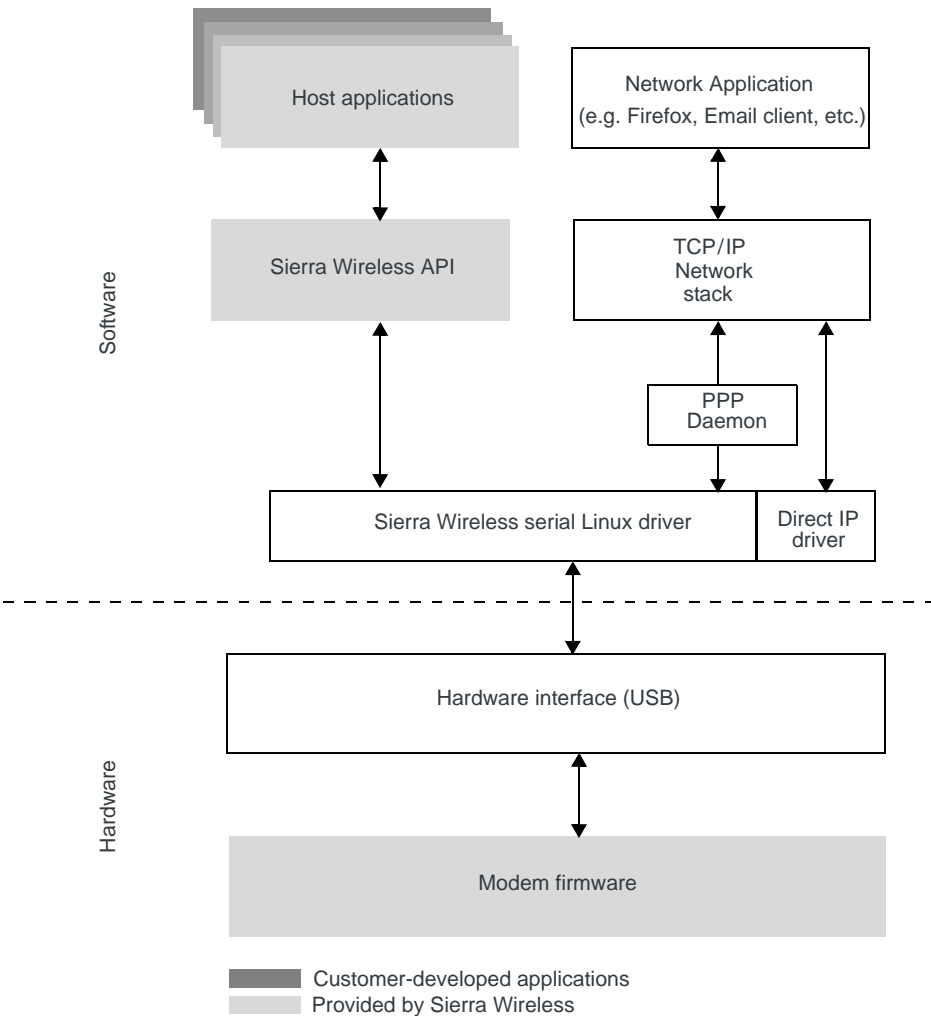


Figure 4-3: API/software/hardware interaction

>> 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 supports 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 API functions, 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.

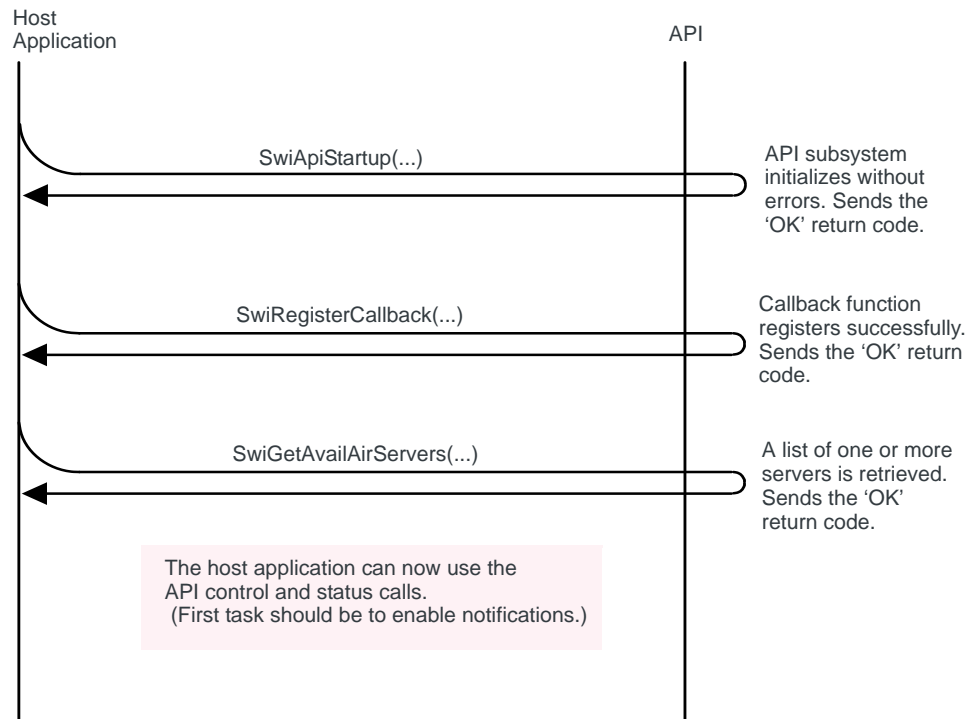


Figure 5-1: API initialization when air servers are available

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.

4. Call **SwiRegisterCallback** again to re-register the callback function.

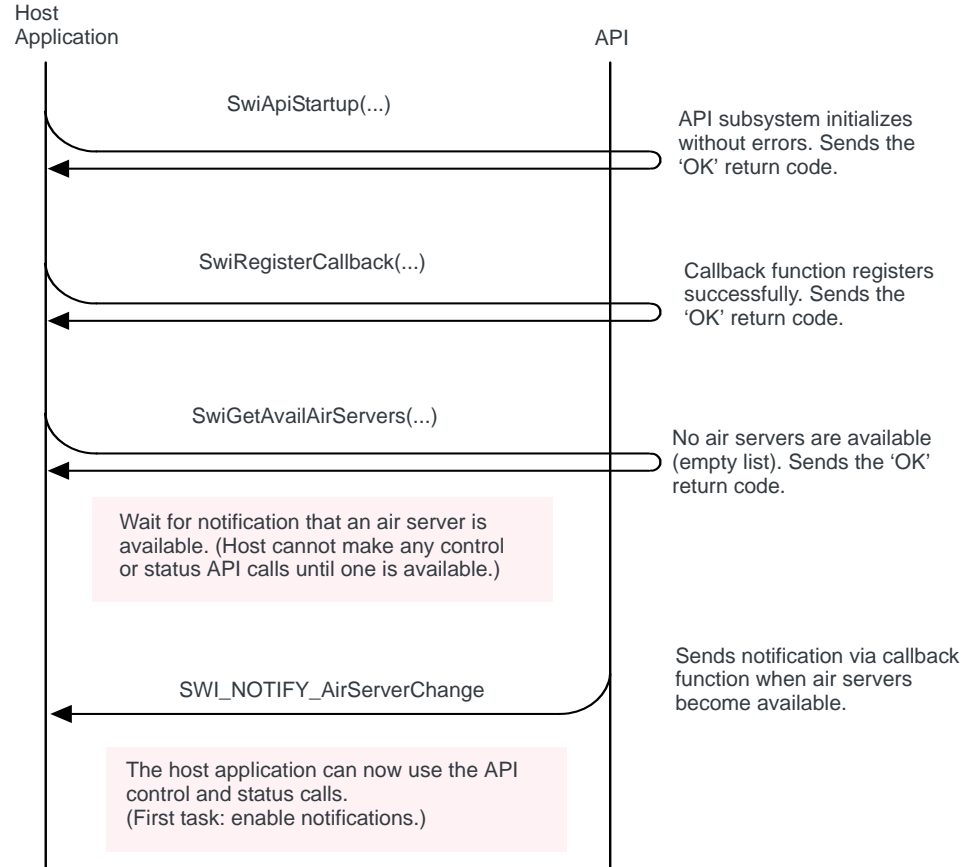


Figure 5-2: API initialization when no air servers are available

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.

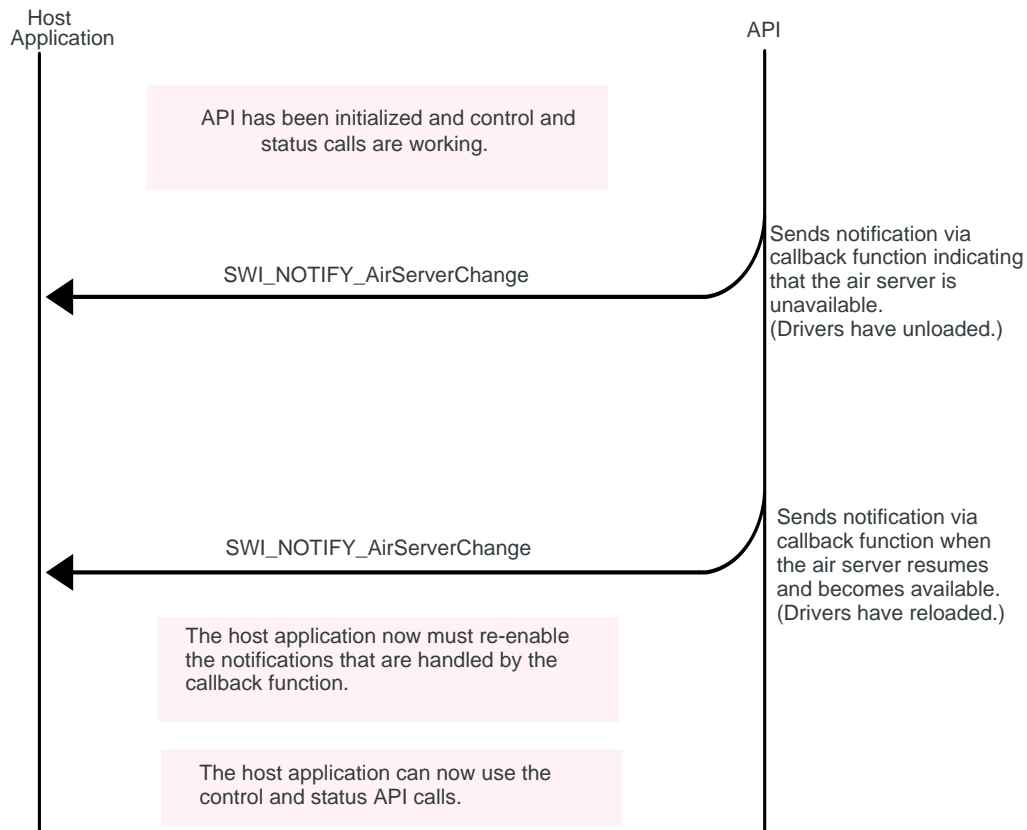


Figure 5-3: Handling modem reset

Shutting down the API

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

1. Call **SwiStopAllNotif** to disable all notifications.
2. Call **SwiDeRegisterCallback** so the API will stop using the callback function.

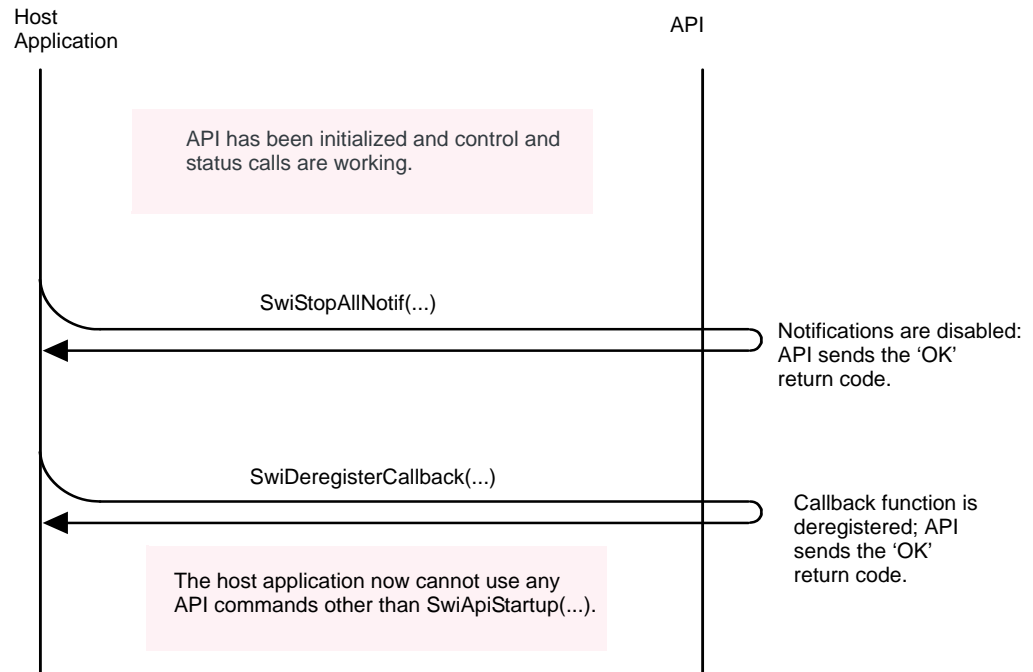


Figure 5-4: Shutting down the API

Host application API usage

Opening the host application

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

1. Initialize the API and register a notification callback function as shown in [Figure 5-1](#) on page 28.
2. Call **SwiSetHostStartup** to ensure the modem is able to register on a network.

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

Checking that the modem is available

To check if the modem is available (driver has been loaded), call **SwiGetAvailAirServers**.

To make sure the modem is ready to use (not in boot and hold mode by calling **SwiGetBootAndHoldMode**, for example), call any API function that returns a modem parameter. If the function fails to return a value within a reasonable number of attempts, the modem is not available.

*Note: Refer to the API online reference guide to determine whether your host application should use **SwiSetModemDisable** or **SwiSetHostStartup**.*

Powering the modem down and up

Use the following functions to power the modem down and up:

- **SwiGetModemDisable**—Indicates the current modem state (in low power mode or not).
- **SwiSetModemDisable**—Enables or disables the modem. If you disable the modem with this function, you must also use the function to re-enable it—resetting the modem does not re-enable it. (Modem disable setting is persistent across modem power cycles.)
- **SwiSetHostStartup**—Enables or disables the modem. (Powered-down state is not-persistent across modem power cycles.)

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.

Disabling notifications

Call the function **SwiStopNotify** to disable individual notifications, or **SwiStopAllNotify** to disable all notifications.

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.

Using the “ready” notifications

The following notifications are received when services become available:

- **SWI_NOTIFY_PlmnReady**—Modem can switch modes between automatic and manual PLMN selection. (See [Registering on a network](#) on page 43.)
- **SWI_NOTIFY_SimStatusExp**—Application must wait for this notification before using SIM functions. (See [Chapter 6](#) on page 35.)

The host application should wait to receive these notifications before calling related functions.

Enabling network registration

Each time the host application starts, call **SwiSetHostStartup** to ensure the modem is able to register on a network.

Closing the host application

When the host application closes, disable notifications from the modem and possibly shut down your application as well—see [Disabling notifications](#) on page 32.

>> 6: SIM Authentication and Codes

This chapter describes how to use API functions to perform the following SIM-related tasks on GSM modules:

- Unlocking the MEP feature
- Managing SIM security (CHV1 and CHV2)
- Unlocking CHV1 and CHV2

Using these API functions

The following are key usage notes for functions described in this chapter (see the API online reference guide for additional details):

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)
- Host must receive the **SWI_NOTIFY_SimStatusExp** notification indicating that the SIM is ready before issuing any calls. It is issued after every SIM function call to report the function call result and the SIM's status.
- If a call requires a **SWI_NOTIFY_SimStatusExp** notification, do not repeat the call until you receive it or it times out.

Using a MEP code to unblock the modem

A MEP (Mobile Equipment Personalization) code is used to deactivate the GSM MEP feature, which restricts user equipment to a specific service provider's SIMs.

To deactivate the GSM MEP feature on the modem:

1. Call **SwiSetMEPUnlock** with the correct MEP unlocking code.
2. Wait for the **SWI_NOTIFY_SimStatusExp** notification indicating whether the modem was unlocked successfully.
3. If the unlock attempt failed, verify that you are using the correct code and repeat this procedure.

SIM security

The SIM is protected by two levels of security (if enabled) to prevent unauthorized access of the SIM and its features:

Note: In a typical host application interface, the user should have to enter the code twice to make sure it is entered correctly.

- CHV1—When enabled, a voice-enabled modem can call only emergency numbers unless the correct unlocking code (CHV1) is used.
- CHV2—Always enabled. The correct unlocking code (CHV2) is required to use special features such as the FDN phonebook.

Checking / setting CHV1 enabled status

To determine if CHV1 is enabled, check the **SWI_NOTIFY_SimStatusExp** notification, or call **SwiGetSimLock**.

To enable or disable CHV1, call **SwiSetSimLock**.

CHV1 verification

If CHV1 security is enabled, the CHV1 code must be entered:

1. When the modem is restarted or reset, the **SWI_NOTIFY_SimStatusExp** notification is sent to the host.
2. The host gets the CHV1 PIN and calls **SwiSetSimVerify** to compare the entered PIN with the CHV1 code on the SIM.
3. The modem sends the **SWI_NOTIFY_SimStatusExp** notification to the host, indicating success (correct PIN) or failure (incorrect PIN). If the PIN was wrong, the notification also indicates that the PIN has to be re-entered, and reports the number of retries remaining before the SIM is blocked. (See [Unblocking CHV1 or CHV2](#) on page 38 for details.)

Note: Do not hard code the number of allowed attempts—use the information in the notification. (The number of allowed attempts is network-dependent.)

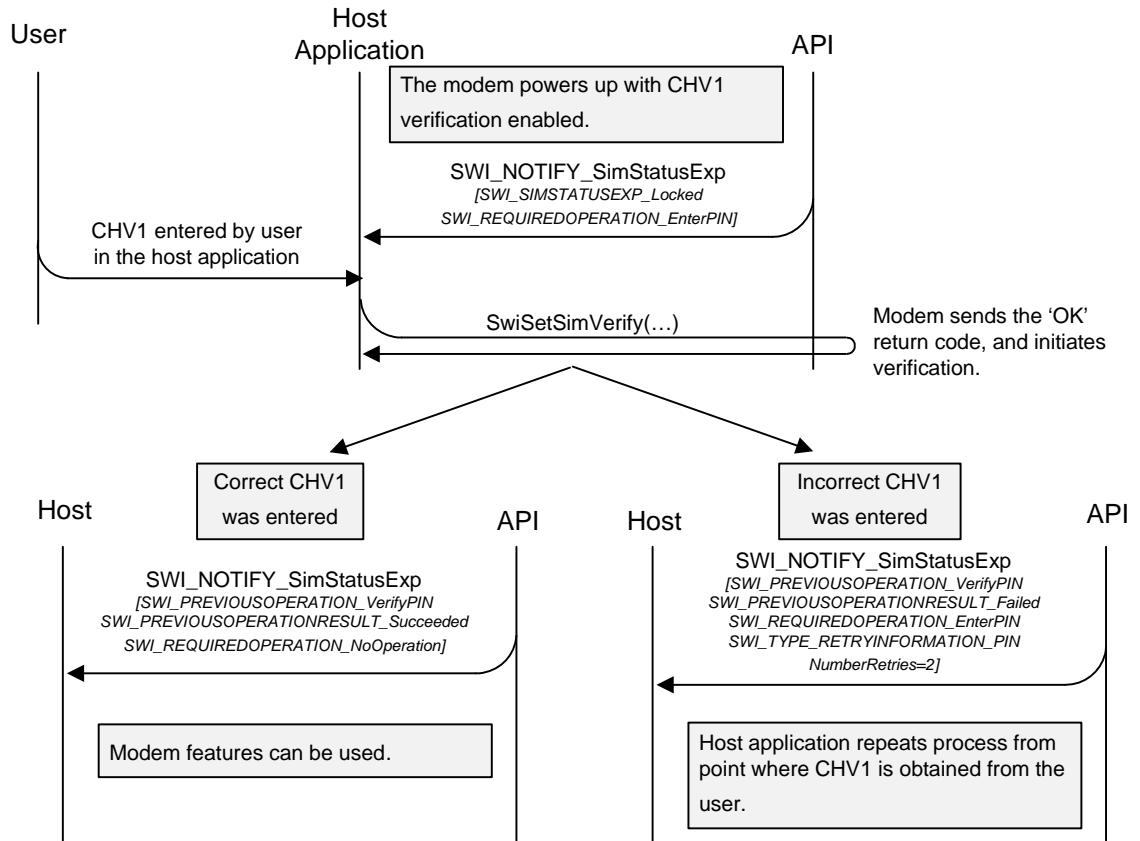


Figure 6-1: CHV1 verification process

CHV2 verification

CHV2 is always enabled—the CHV2 code must be sent to the modem to access special features such as the FDN phonebook and the ACM (Accumulated Call charge Meter) feature.

1. Call **SwiChv2StatusKick** to tell modem to request CHV2 verification is required.
2. Receive **SWI_NOTIFY_SimStatusExp**.
3. Call **SwiSetSimVerify** with the CHV2 code.
4. Receive **SWI_NOTIFY_SimStatusExp** indicating success.

Note: Just like CHV1, if an incorrect CHV2 code is entered repeatedly, CHV2-restricted functionality becomes blocked. See [Unblocking CHV1 or CHV2](#) on page 38 for details on unblocking CHV2.

Enabling / disabling CHV1

Note: Only CHV1 can be disabled—CHV2 is always enabled.

To enable or disable CHV1:

1. After getting the CHV1 code from the user, call **SwiSetSimLock**, passing the CHV1 code and setting the Enable flag to enable or disable.
2. Wait for the modem to send the **SWI_NOTIFY_SimStatusExp** notification indicating success or failure (likely because the wrong code was entered).

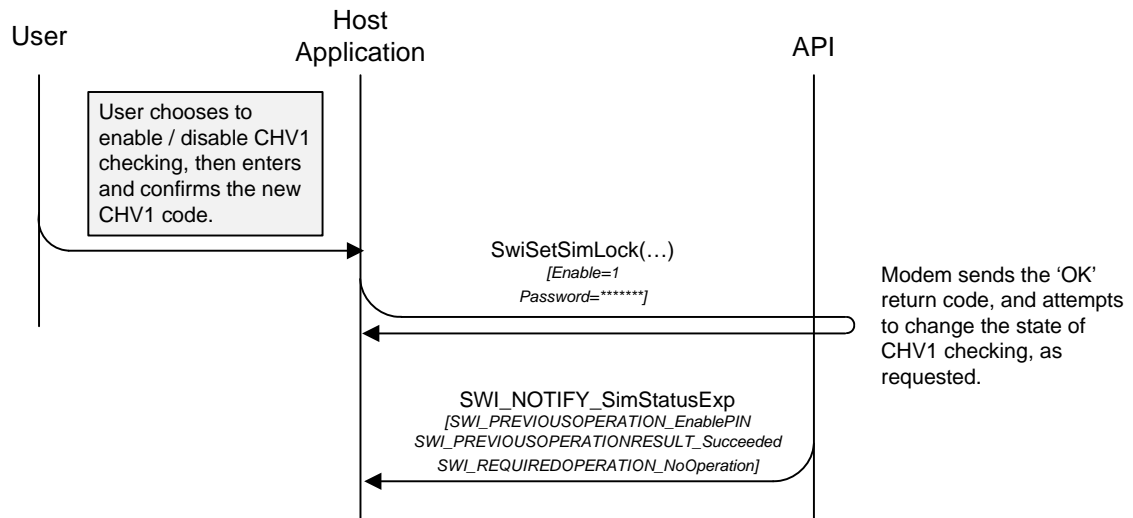


Figure 6-2: Enabling/disabling CHV1

Changing CHV1 or CHV2

Note: You cannot use an emergency number as the CHV1 code (or as the beginning of the CHV1 code).

To change the CHV1 code:

1. Call **SwiSetSimPassword** with the new CHV1 code.

To change the CHV2 code:

1. Call **SwiChv2StatusKick** with the parameter **SWI_TYPE_CHV2KICKTYPE_Change**. This triggers the **SWI_NOTIFY_SimStatusExp** notification.
2. Receive the **SWI_NOTIFY_SimStatusExp** notification.
3. Call **SwiSetSimPassword** with the new CHV2 code.

Unblocking CHV1 or CHV2

When either CHV1 or CHV2 is blocked because the wrong code was entered too many times in a row, a PUK (Pin Unblocking Code) can be used to unblock it. (PUK1 is used for CHV1, and PUK2 is used for CHV2.)

Note: If CHV1 is permanently blocked, voice-enabled modems can dial only emergency numbers.

PUK codes are obtained from the service provider. If an incorrect PUK code is used too many times in a row, CHV1 (for PUK1) or CHV2 (for PUK2) becomes permanently blocked.

To unblock either CHV code, use the process described for CHV1 verification, replacing the CHV code with the appropriate PUK code.

Sample application

The SDK includes a sample application (SIMLock) that demonstrates SIM security functionality. The following files are available:

- SIMLock executable—Located in \$INSTALL_FOLDER/build/bin/i386
- Source code—Located in \$INSTALL_FOLDER/Sample_Code/gsm/SIMLock
- User guide—Located in \$INSTALL_FOLDER/docs

>> 7: Account Profile Management

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

- Read, create, edit, and delete profiles
- Assign a default profile
- Activate profiles

Using these API functions

The following are key usage notes for functions described in this chapter (see the API online reference guide for additional details):

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)
- **SWI_NOTIFY_GsmProfileChange** reports any changes made to profiles.

Account profile overview

Account profiles contain information used by the network to verify access to network services. The profile information is obtained from the service provider (usually with the SIM) and may contain:

- A username
- A password
- An APN (Access Point Name)
- An IP address (if not automatically assigned by the network)
- Indication as to whether IP header compression is used
- DNS address(es)

Number of supported profiles

UMTS modules support several profiles (labeled 1, 2, etc.). Refer to your modem's Product Specification Document for the actual number supported.

Profile maintenance functions

Using these API functions, you can identify, read, create, update, activate, and delete account profiles. The sample program **SwiProfileManipulateUMTSi386/arm9** demonstrates how to program some of these operations.

Identifying account profiles

To get a list of all account profiles, call **SwiGetGsmProfileSummary**. The modem returns a profile list, which includes the status of each profile, and identifies the default profile and active profile.

Reading profiles

To read the full details for a specific profile:

1. Call **SwiGetGsmProfileBasic** to read basic details.
2. Call **SwiGetGsmProfileDns** to read DNS details.

Creating and editing profiles

Creating profiles

*Note: After each call, the modem sends the **SWI_NOTIFY_GsmProfileChange** notification. You do not have to wait for this notification to arrive before calling the next function.*

To create a new profile, call the following functions (and pass the new profile's index number in each call):

1. Call **SwiSetGsmProfileBasic** to set basic details.
2. Call **SwiSetGsmProfileDns** to set DNS details.

Setting a default profile to autoactivate

If the profile designated as the default profile is set to autoactivate, the modem initiates a connection using it as soon as the modem is reset.

To set the default profile, call **SwiSetDefaultProfile** using the profile's index number.

Activating a profile

Activating a profile initiates a packet data connection. To activate a profile, call **SwiActivateProfile**. (See [Establishing a point-to-point \(dialup\) data connection](#) on page 57 for details.)

Deleting profiles

To delete a profile and set it back to factory default values, call **SwiEraseProfile**.

>> 8: Network registration

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

- Register on a network
- Select frequency bands
- Manually select a network

Using these API functions

The following are key usage notes for functions described in this chapter (see the API online reference guide for additional details):

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)
- To register, an unlocked SIM with a valid account (with no restrictions affecting registration) must be installed in the modem, the modem must be configured to work on appropriate bands/networks, and it must be in range of a network with adequate signal strength.

Registering on a network

The modem must be registered on a network before data, voice, or other connections can be established.

To register the modem on a network:

1. Make sure the modem is powered up. Call one of the following functions (see the API to determine which is appropriate for your application):
 - **SwiSetHostStartup** with Startup = True.
 - **SwiSetModemDisable** with ModemDisable = False
2. Set the frequency band(s), if necessary, on which the modem will operate. (See your modem's Product Specification Document for a list of supported bands.). See [Setting the frequency band\(s\)](#) on page 44.
3. Select the network on which to register the modem, if necessary. Once the network has been selected, the modem registers automatically.
4. If you want to change the network manually, see [Selecting and registering on a network](#) on page 44.

Setting the frequency band(s)

To get a list of supported bands for your modem:

1. Call **SwiGetBandInfo**.

To set the bands that the modem should use (or to 'autoband'):

1. Call **SwiSetRadioBandCfg**.
2. Wait for the **SWI_NOTIFY_BandWrite** notification, which indicates if the band change is successful.

*Note: **SWI_NOTIFY_Band** reports the new frequency band. It is received whenever the modem switches between bands (after a function call, if the modem is set to 'autoband', or after a non-API action such as an AT command).*

Selecting and registering on a network

The modem can be set to select a network automatically or manually by calling **SwiSetPLMNMode**. To retrieve the current setting, call **SwiGetPLMNMode**.

To manually select a network on which to register:

1. Wait for the **SWI_NOTIFY_PlmnReady** notification. This indicates that manual selection is available and can begin.
2. Call **SwiStartPLMNSearch** to identify available PLMNs.
3. Wait for the **SWI_NOTIFY_PlmnAvailable** notification, indicating that a list of PLMNs can be read from the modem.
4. Call **SwiGetPLMNSelection** to read a PLMN from the modem. Repeat until all PLMNs are read. (The PLMN data structure sets the flag `MorePlmn = 1` if there are more PLMNs to read.)
5. Call **SwiSetPLMNMode**, set the mode to 'manual', and identify the PLMN on which to register.
6. Wait for the **SWI_NOTIFY_PlmnMode** notification indicating whether the registration attempt succeeded.

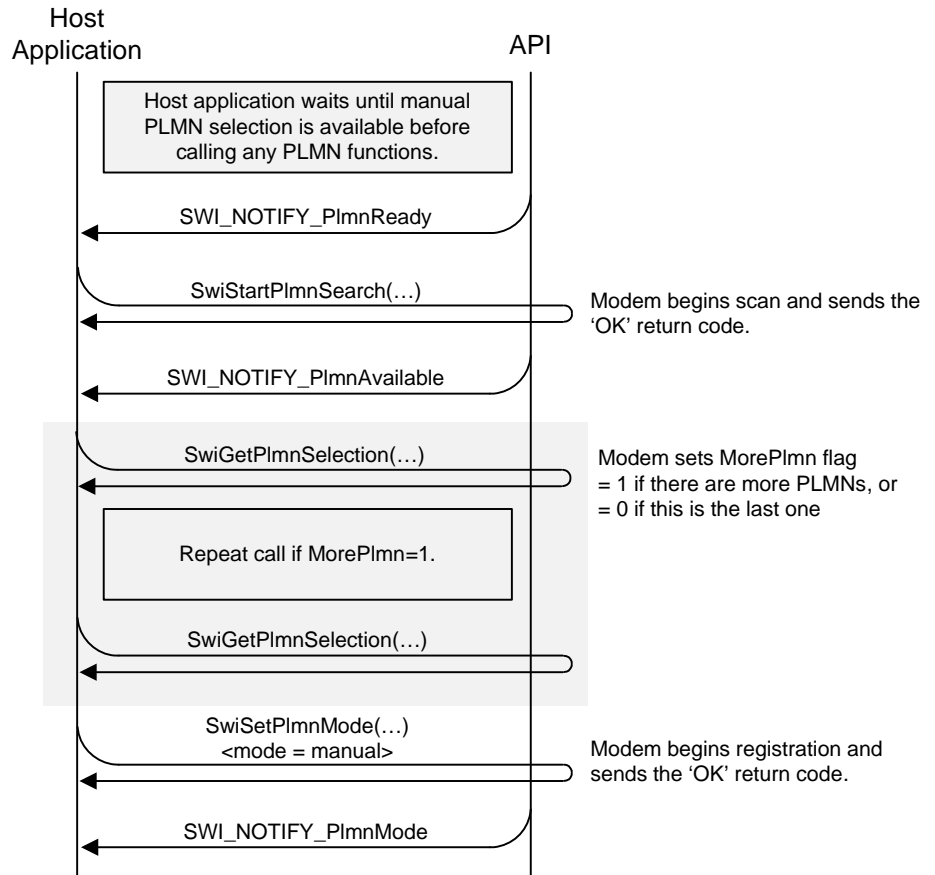


Figure 8-1: Manual PLMN selection

>> 9: 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

Using these API functions

The following are key usage notes for functions described in this chapter:

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)

See the API online reference guide for additional details.

Retrieving operational settings

Get/set modem default and current operational parameters

To report the modem's default operational parameters, and to get and set the current values of these parameters, call the following functions:

- **SwiGetLbsPaParam**—Get the modem's default operational parameters.
- **SwiGetLbsPaPortId/SwiSetLbsPaPortId**—Get and set the SUPL Server port ID
- **SwiGetLbsPalpAddr/SwiSetLbsPalpAddr**— Get and set the SUPL Server IP address

Get satellite details

To read satellite information for all satellites in view (azimuth, elevation, SNR, etc.):

1. Call **SwiGetLbsSatInfo**.

Get LBS status

To determine the status of the most recent fix session:

1. Call **SwiGetLbsPdStatus**.

Get/set user-selected LBS fix settings

To get or set LBS fix settings (fix type, performance, accuracy, etc.):

1. Call **SwiGetLbsFixSettings** or **SwiSetLbsFixSettings**.

Position fix / tracking sessions

You can use LBS functions to initiate single position fixes and tracking sessions (multiple position fixes).

Report modem's last known location

To get the result of the modem's most recent position fix:

1. Call **SwiGetLbsPdData**.

Get modem's current location (Initiate single position fix)

To get the modem's current location:

1. Call **SwiSetLbsPdGetPos** to initiate a position fix. The return code indicates if the fix is initiated or if an error occurred.
2. Wait for notifications reporting the progress of the fix.

The following notifications are received when a fix is successful:

- **SWI_NOTIFY_LbsPdBegin**
- **SWI_NOTIFY_LbsPdData** (Data is available.)
- **SWI_NOTIFY_LbsSatInfo** (Data is available.)
- **SWI_NOTIFY_LbsPdDone** (Fix is complete.)

The fix may not complete successfully, in which case other notifications are received:

- If the fix fails due to an error, or is stopped via the API or another interface (AT, CnS, etc.), receive **SWI_NOTIFY_LbsPdUpdateFailure**
- If the fix times out, receive **SWI_NOTIFY_LbsPdEnd**

3. If the fix was successful, call **SwiGetLbsPdData** to get the fix results.

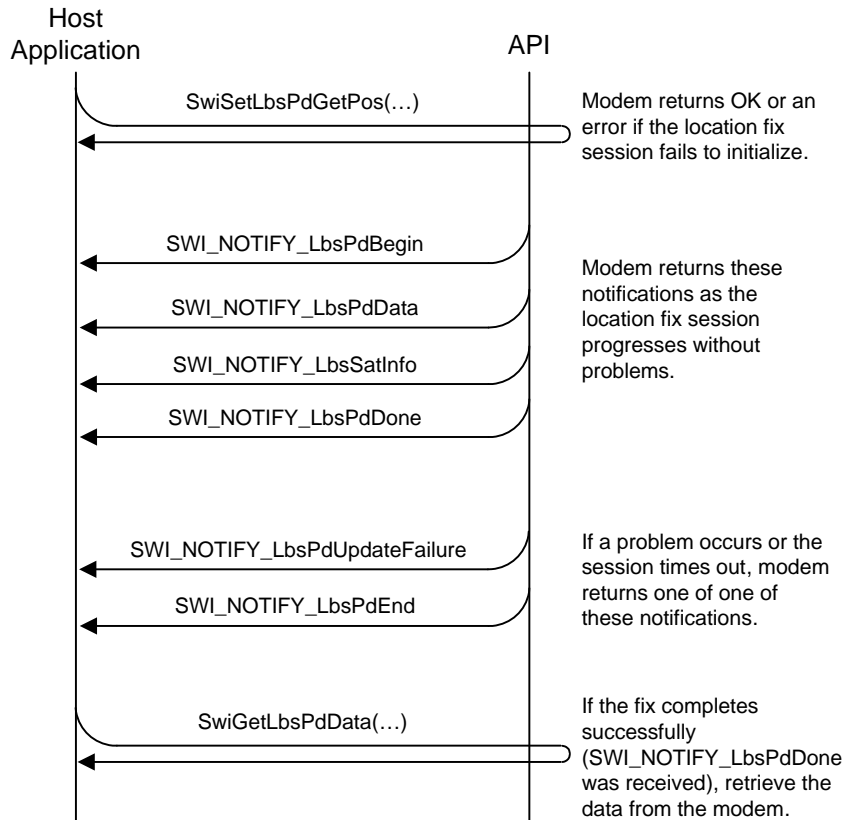


Figure 9-1: Initiating a single position fix

Initiate tracking session

A tracking session consists of a number of individual position fixes repeated a number of times (the fix count) at a specific rate (the fix rate). If the fix count is 1000, the session keeps running until it is explicitly stopped (see [End tracking session](#) on page 49).

To initiate and process a tracking session:

1. Call **SwiSetLbsPdTrack** to initiate the session. The return code indicates if the session is initiated or if an error occurred.
2. Wait for notifications reporting the progress of each location fix. (See [Get modem's current location \(Initiate single position fix\)](#) on page 48—note that the **SWI_NOTIFY_LbsPdDone** notification is sent only after the final fix).
3. When the final fix is finished (based on the fix count), the host receives the **SWI_NOTIFY_LbsPdDone** notification.

End tracking session

To end the tracking session prematurely (before it reaches the fix count):

1. Call **SwiSetLbsPdEndSession**.
2. Receive the **SWI_NOTIFY_LbsPdEnd** notification (indicating why the session ended) and the **SWI_NOTIFY_LbsPdDone** notification (indicating the session is finished).

Respond to network-initiated fix request

If the network requests the modem's location (using the **SWI_NOTIFY_LbsNiReq** notification):

1. The host receives the **SWI_NOTIFY_LbsNiReq** notification. If the notification's **NotifType** value is **LBSNIREQNOTIF_UserRespReq**, the host must respond, otherwise the modem's location is returned to the network automatically.
2. If the host must respond, it calls **SwiSetLbsNiReq** indicating whether the request is accepted or rejected.
3. If the request is accepted, the modem's location is returned to the network.

Ephemeris / almanac data

The API includes commands that affect the downloading of assistance data.

Enabling / disabling 'Keep Warm' processing

While GPS is required by the host, the GPS assistance data can be kept current by enabling 'Keep Warm' processing. When enabled, the modem periodically downloads GPS assistance data.

To enable Keep Warm processing:

1. Call **SwiSetLbsPaKeepWarmStart**. The modem determines how often to download assistance data.
2. The host receives the **SWI_NOTIFY_LbsPaWarmBegin** notification indicating that Keep Warm has begun. The host also receives (periodically) the **SWI_NOTIFY_LbsPaWarmStatus** notification indicating the current status of Keep Warm processing.

To disable Keep Warm processing:

1. Call **SwiSetLbsPaKeepWarmStop**.
2. The host receives the **SWI_NOTIFY_LbsPaWarmDone** notification indicating that Keep Warm is finished.

To check the status of Keep Warm processing (enabled/disabled):

1. Call **SwiGetLbsPaWarmStatus**.

Simulating a coldstart to force assistance data download

To simulate a coldstart:

1. Call **SetLbsClearAssistance** to clear the location parameters.

>> 10: Phone Book Maintenance

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

- Use over dial numbers
- Maintain phone books (add, edit, delete entries)
- Use the FDN phone book
- Retrieve emergency phone numbers
- Retrieve phone numbers from any phone book

Using these API functions

*Note: The functions in this section are found in the **Phonebook** API module. For additional network registration-related functions, refer to the API documentation.*

The following are key usage notes for functions described in this chapter (see the API documentation for additional details):

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)
- Host must receive the **SWI_NOTIFY_PhonebookReady** notification before calling any phone book-related functions.
- Host must receive the **SWI_NOTIFY_SimStatusExp** notification before forcing the modem to request CHV2 verification.

Supported phone books

The API supports the phone books listed in [Table 10-1](#) on page 51. (Availability of phone books is carrier-dependent.)

Table 10-1: Supported phone books

Abbreviation	Name	Description	Storage	Actions
ADN	Abbreviated Dialing Numbers	<ul style="list-style-type: none"> • Stores names / numbers for making phone calls • Number of entries: carrier-dependent, typically 255 max. • Supports over dial numbers. See Using over dial numbers on page 52. 	SIM	Add Edit Delete
CPHS (Voice modems only)	CPHS Mailing Numbers	<ul style="list-style-type: none"> • Stores up to four voice mailbox numbers; carrier-dependent • Example: Could be used to call a voice mailbox if messages are waiting. • Read-only 	SIM	Edit

Table 10-1: Supported phone books (Continued)

Abbreviation	Name	Description	Storage	Actions
FDN	Fixed Dialing Numbers	<ul style="list-style-type: none"> Stores numbers that user is restricted to for dialing (when FDN is enabled). Number of entries: carrier-dependent, typically 100 max. Supports overdial numbers. See Using overdial numbers on page 52. <p>See Using the FDN phone book on page 53 for additional details.</p>	SIM	Add Edit Delete
LND (Voice modems only)	Last Numbers Dialed	<ul style="list-style-type: none"> Stores most recent numbers dialed (typically 10). Example: Could be used to implement a redial feature, or be added to ADN Read-only phone book 	SIM	Add Edit Delete
LNM (Voice modems only)	Last Numbers Missed	<ul style="list-style-type: none"> Stores numbers of most recent missed incoming calls (typically 10). Example: Could be used to implement a call-back feature, or be added to ADN. Read-only phone book 	Modem (NVRAM)	Delete (entire book)
LNR (Voice modems only)	Last Numbers Received	<ul style="list-style-type: none"> Stores numbers of most recent answered incoming calls (typically 10). Example: Could be used to implement a call-back feature, or be added to ADN. Read-only phone book 	Modem (NVRAM)	Delete (entire book)
MSISDN	Mobile Subscriber International Subscriber Identity Number	<ul style="list-style-type: none"> Stores the account's phone number(s) SIM may be pre-configured by carrier 	SIM	Edit
SDN	Service Dialing Numbers	<ul style="list-style-type: none"> Carrier-provisioned numbers Examples: billing enquiries, emergency numbers, technical support, etc. Read-only phone book 	SIM	View only

Using overdial numbers

Note: The ADN and FDN phone books support overdial numbers.

Overdial numbers are numbers and symbols dialed after establishing a connection. They are part of the stored phone number, and begin with a comma.

Valid numbers and symbols include:

- '0'-'9'
- '*'
- '#'

-
- ‘;’ (first occurrence in phone number)
Indicates the beginning of the overdial phone number, and forces a three-second pause.
 - ‘;’ (subsequent appearances in phone number)
Three-second pause.
 - ‘?’
Wildcard character. Indicates a missing digit that the user needs to enter.

For example, if the phone book number is 6045551212,,112,4:

- 6045551212 is dialed.
- Six-second pause after the connection is established
- 112 is dialed
- Three-second pause
- 4 is dialed

Using the phone book functions

Application start-up

When the host application starts up, it should:

1. Wait for **SWI_NOTIFY_PhonebookReady** before calling any phone book functions.
2. Call **SwiGetPhonebookAvailable** to see which phone books can be used.

Maintaining ADN, FDN, MSISDN phone books

Entries in the ADN, FDN, and MSISDN phone books can be updated using API functions.

Note: CHV2 must be verified before adding or editing FDN phone book numbers as shown in [Using the FDN phone book](#) on page 53.

To add a phone book entry:

1. Call **SwiAddPhonebookEntry**.

To edit a phone book entry:

1. Call **SwiEditPhonebookEntry**.

To delete a phone book entry:

1. Call **SwiDeletePhonebookEntry**.

Using the FDN phone book

Phone book entries:

When the FDN feature is enabled, users can call only numbers that match the format of the entries in the phone book, as shown in the following examples of FDN phone book numbers.

- 60"—User can call any number beginning with "604" (e.g. 6045551234)
- 2505558989—User can call only 2505558989.

Note: For information about CHV2 codes, see [CHV2 verification](#) on page 37.

Enabling/disabling FDN

To check if FDN is enabled or disabled:

1. Call **SwiGetFdnMode**.

To enable or disable the FDN phone book:

2. Wait for the **SWI_NOTIFY_SimStatusExp** notification indicating that the SIM phone books are available.
3. Call **SwiChv2StatusKick** to tell the modem that CHV2 verification is required. This will trigger a notification to enter the CHV2 code.
4. Wait for the **SWI_NOTIFY_SimStatusExp** requesting the code.
5. Call **SwiSetFdnMode**, passing in the CHV2 code.

SIM phone book statistics

To get the current sizes and remaining space for all SIM-based phone books:

1. Call **SwiGetPhonebookSize**.

Retrieving phone numbers

Retrieving emergency numbers

Emergency phone numbers are stored on both the modem and the SIM.

To obtain all of these numbers:

1. Call **SwiGetEmergencyEntry** to retrieve a single phone number.
2. Repeat until the **MoreEntries** flag in the returned data structure is 0 (no more entries remain).

Note: Duplicate numbers are returned if a number is stored on both the SIM and the modem.

Phone book retrieval

To read the first entry from any phone book:

1. Call **SwiGetPhonebookEntry** with **ReadFromStart** = true.

To read the next entry from any phone book (based on the most recent entry read):

1. Call **SwiGetPhonebookEntry** with **ReadFromStart** = false.

To read all entries from a phone book:

1. Call **SwiGetPhonebookEntry** with **ReadFromStart** = true.
2. Repeat (with **ReadFromStart** = false) until **MoreEntries** = false (no more entries remain).

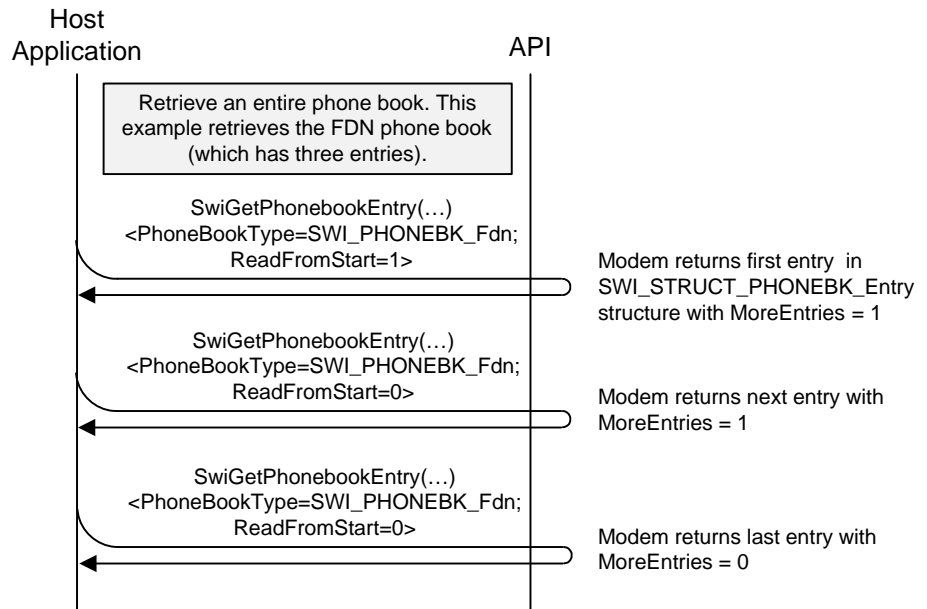


Figure 10-1: Retrieving an entire phone book

Retrieving all entries in the ADN phone book

As soon as phone book service is available, the modem begins returning every entry in the ADN phone book if:

- The ADN phone book is available, and
- FDN is disabled (If FDN is enabled, only FDN entries can be used. This effectively disables the ADN phone book.), and
- The **SWI_NOTIFY_PhonebookEntry** notification is enabled.

One entry is returned with each **SWI_NOTIFY_PhonebookEntry** notification, and the last entry returns with **MoreEntries** = false.

If the ADN phone book has no entries, a single **SWI_NOTIFY_PhonebookEntry** notification is returned with **MoreEntries** = false and **Valid** = false.

>> 11: Data Connections

11

This chapter describes how to use API functions to establish a data connection.

Using these API functions

The following are key usage notes for functions described in this chapter (see the API online reference guide for additional details):

- Host must enable notifications before they can be received. (See [Managing notifications](#) on page 25.)
- The modem must be registered on a network before you try to establish a data (GPRS / EDGE / UMTS) connection. See [Chapter 8](#) starting on page 43 for details.
- (Voice-enabled modems only) A circuit-switched (CS) data connection cannot be initiated if a voice call is in progress.
- (Voice-enabled modems only) If a voice call occurs during a CS data connection, the CS data connection is maintained, but suspended (the IP address is not lost). The CS data connection resumes when the voice call completes. (Note that your host application may have timed out by this point.)
- If you are using a Sierra Wireless HSPA+ modem, you must install the `sierra_net.c` driver in addition to the `sierra.c` driver. For information on obtaining and installing the drivers, refer to the Linux SDK Integration Guide included in the SDK.

Establishing a point-to-point (dialup) data connection

The modem can establish packet-switched data connections on GPRS, EDGE, and UMTS networks. See the modem's Product Specification Document for maximum data rates.

To establish a packet-switched data connection with a modem that is already registered on a network:

1. Ensure your modem has a usable profile configured before you start a connection; you can use the APIs to manage your modem's profiles programmatically. (See [Account Profile Management](#) on page 41.)
2. If not already enabled, enable notifications that are received when conditions that can affect the connection occur. Some of these notifications include:
 - **SWI_NOTIFY_RegistrationExp**—Modem is registered on a network (and identifies the PLMN and/or SPN)
 - **SWI_NOTIFY_Band**—Modem switched bands (due to function call or autoband)

- **SWI_NOTIFY_SimStatusExp**—SIM status changes
 - **SWI_NOTIFY_NetworkStatus**—Network status changes
 - **SWI_NOTIFY_ServiceIcon**—Available services change (GPRS, EDGE, UMTS)
 - **SWI_NOTIFY_Rssi**—RSSI value changes
 - **SWI_NOTIFY_Temperature**—Modem is overheating; data transmission is suspended until the temperature drops
 - **SWI_NOTIFY_TransmitAlert**—Problem with the antenna
3. Wait for a **SWI_NOTIFY_PktSessionCall** notification, which indicates if the connection was successful.

Your application must establish a separate dialup networking connection using a suitable mechanism:

- **PPP connection**—The sample application provided in the `$INSTALL_FOLDER/Samples_Code/gsm/ConnectGSM` provides an example of how to accomplish this using the built-in Linux PPP Daemon called “pppd”.
- **Direct IP**—The API call **SwiActivateProfile()**, described in the API online reference guide, is used to establish the connection over the Direct IP interface.

>> 12: Modem and SIM characteristics

This chapter details several functions that allow the host application to identify account parameters, modem component details, and SIM details.

The following table provides an overview of these elements—for detailed explanations, refer to the API online reference guide.

Table 12-1: Component characteristics functions

Function	Description
Firmware details	
SwiGetBootVersion	Returns the version of the bootloader (a component of the firmware)
SwiGetBootloaderBuildDate	Returns the date the bootloader was built
SwiGetFirmwareVersion	Returns the version of the modem firmware
SwiGetFirmwareBuildDate	Returns the date the firmware was built
SwiGetFlashImgInfo	Returns firmware flash image details
Modem details	
SwiGetHardwareVersion	Returns the version of the modem hardware
SwiGetUsbdInfo	Returns USB descriptor build details
SwiGetDeviceID	Returns the device's unique identify number (the ESN or EID)
SwiGetPriInfo	Returns the device's PRI details
SwiGetImei	Returns the International Mobile Equipment Identity (a number that uniquely identifies every GSM device) from the modem
SwiGetSerialNumber	Returns the factory serial number (FSN) of the modem
Modem features	
SwiGetAvailableFeatures	Returns the modem's available features, including the PDP context type, and support for voice, tri-band
SwiGetFeatureCustomizations	Returns the modem's customizable features
SIM identification numbers	
SwiGetGsmIMSI	Returns the International Mobile Subscriber Identity (a number used to identify the account holder) from the SIM
SwiGetIccId	Returns the Integrated Circuit Card ID (a unique number used to identify the SIM) from the SIM
Available air servers	
SwiGetAvailAirServers	Returns a list of available air servers
SwiSelectAirServer	Binds the modem to a specific air server

>> 13: 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 move through 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 API functions

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, which 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 that time.

- External applications are free to access the ttys directly for Sierra Wireless modems that 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.

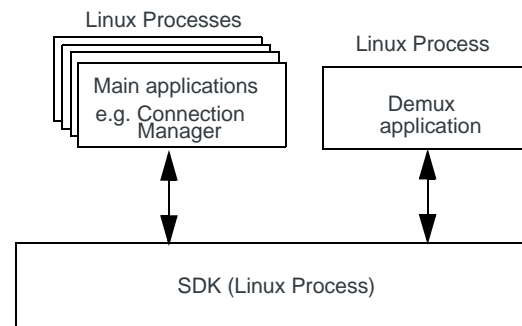


Figure 13-1: Process model

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 written to the buffer, the caller should then submit the buffer for sending (by calling **SwiApiDxSend()**) 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.

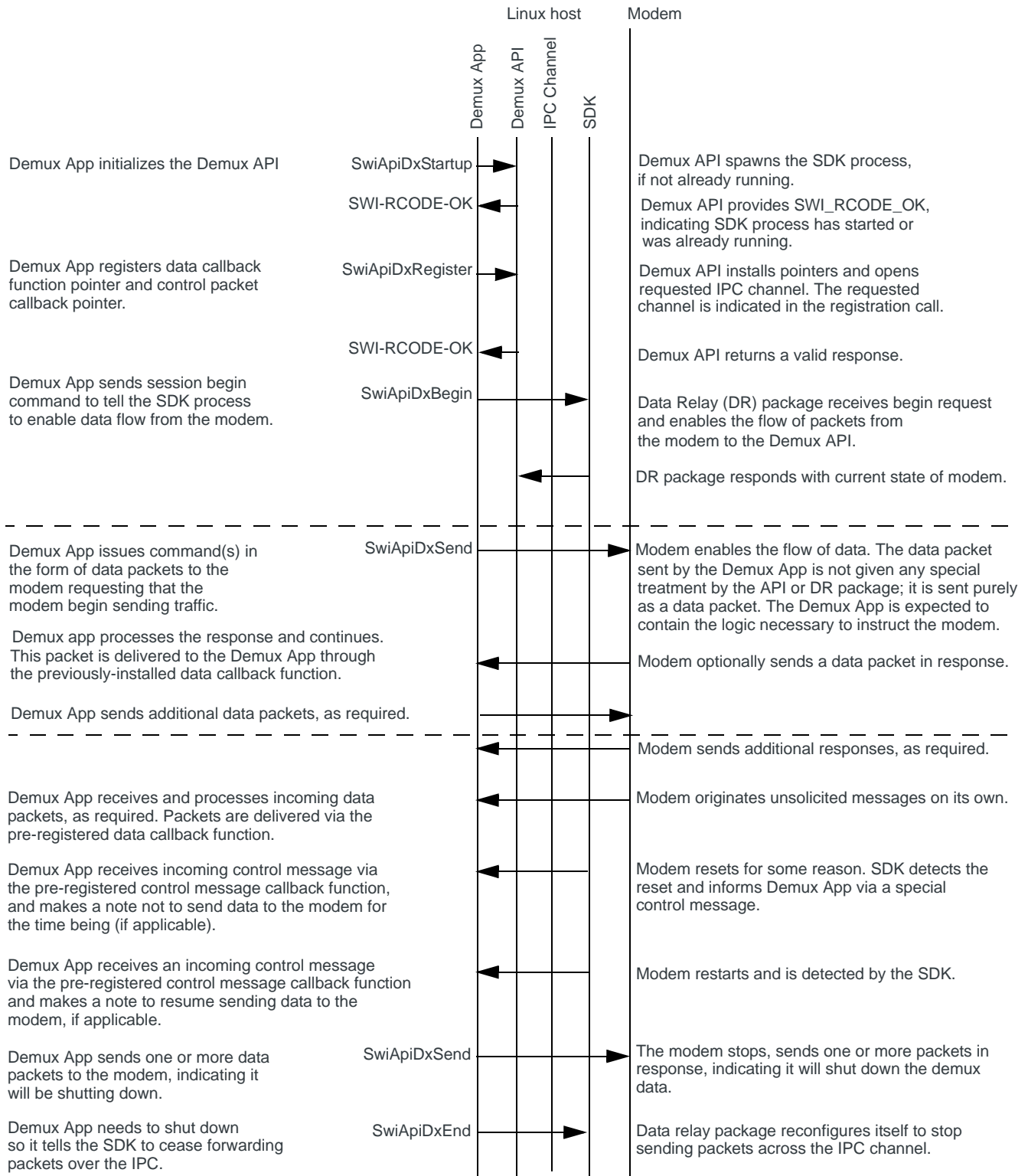


Figure 13-2: Data flow: Demux application

>> 14: Error handling

14

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.)

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 application 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 and they are located in subdirectories under the directory:

`$INSTALL_FOLDER/pkgs`

Package naming convention

Each package has its own directory that 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 keyword *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 only have to include *xxdefs.h* to automatically receive package level function prototypes.

xxdefs.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 *xxdefs.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 *xxdefs.h* file, callers are generally forbidden from accessing its members directly, even though the structure is clearly published within the header file. Instead, callers 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 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.

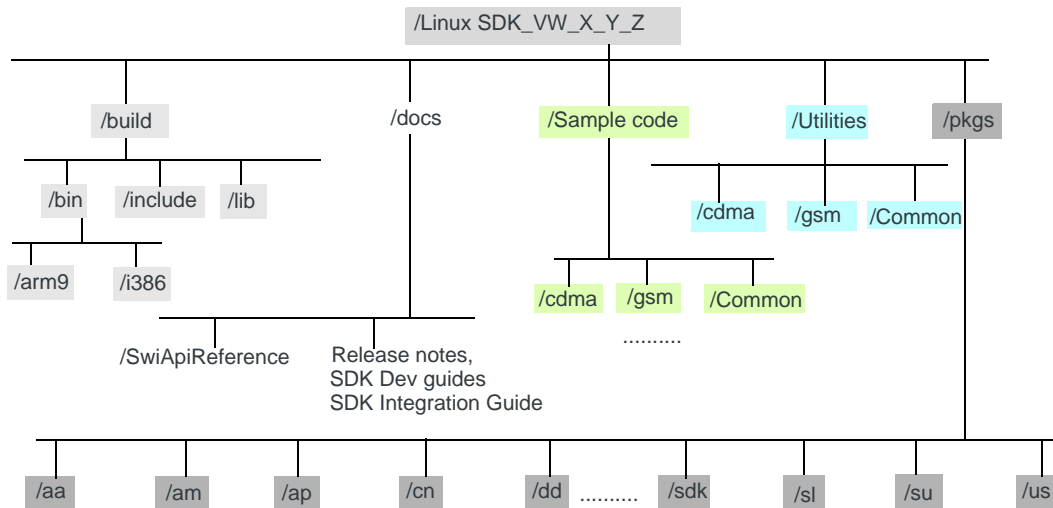


Figure 15-1: SDK file folder structure

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 Download 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 73.

If you need to recompile the Firmware Download utility, see [Utilities/gsm](#) on page 72.

build/bin

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

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

- Sample applications
- CDMA (FWUpdaterCDMA...) and GSM/UMTS (FirmwareDownloadUMTS...) firmware updater utilities

- The SDK (swisdk), which is used as the default by all sample code executables if no path is specified.

The executables in the subdirectories (as well as those in the Sample_Code and Utilities directories) are identical to the ones stored in the pkgs/sdk directory.

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

When running the Firmware Download utility, you must specify the firmware file (CWE image) to download and its location (absolute path); optionally, you can specify the path to 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 GSM Network-related APIs it would need to include either or both of:

- SwiApiGsmNetwork.h
- SwiApiGsmBasic.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, 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
- CDMA Linux SDK Developer's Guide
- Linux SDK Integration Guide
- Licensing agreement
- Release notes for the particular version of the SDK
- A subdirectory structure SwiApiReference (described below).
- SimLock Sample Application Notes

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 for CDMA products. For more information, see the CDMA Developer's Guide (in \$INSTALL_FOLDER/docs).

Sample_Code/gsm

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. 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, review the sample applications. Examples of sample apps contained in this tree are:

- PlmnSelect
- ProfileManipulate
- RadioOnOff
- SignalStrength
- ConnectGSM
- SIMLock
- Notification

Sample_Code/Common

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

Utilities

This directory 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 69](#)) 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 utility for CDMA products. For more information, see the CDMA Developer's Guide (in \$INSTALL_FOLDER/docs).

Utilities/Common

This directory contains utilities that can be used by both CDMA and UMTS modems, such as the Diagnostic and Relay Agent utilities.

Utilities/gsm

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

An example of this tree's contents is the FirmwareDownload utility. If you need to recompile this utility, enter the FirmwareDownload subdirectory and use the following command for the appropriate platform:

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

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 15-1 on page 69]). Packages are usually named with a 2- to 4-character identifier prefix 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 15-1](#) on page 69, 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 74](#) 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 82.

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 68.
- **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]]
[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 xxtest.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, which 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 are 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.h`¹. 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.

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

`SwiApixxx.*`

`apxxx.*`

Files whose names begins 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
```

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:

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

`SwiApixxx.c`—For APIs that your application calls.

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.

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 SIM functions on GSM systems have *Sim* as their category name and APIs for Location Based Services have *Lbs* 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, 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/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, and 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 `dldefs.h`. Therefore any package needing to use the logging facility also needs to include `dldefs.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 (`dlmasterenable(..., 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 manages communications between the host and modem to obtain NMEA and modem diagnostic traffic. To interact with external applications, it uses a specialized set of APIs from the APDX package, and also uses the services of the DS abstraction layer, which interacts directly with the IPC channel via sockets.

pkgs/ds

The DS (Data Services) 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 (via sockets).

The DS package does not interpret the data in any way; 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 only be done from inside the thread context of the superior thread.

pkgs/er

Contains an *erAbort()* facility. By convention, whenever a situation arises in the code that 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 81.)

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.

pkgs/swi_sdk

This package is deprecated and will disappear in subsequent releases.

>> 16: SDK Portability

16

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 Linux versions

Porting the SDK to other Linux versions 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
- Device detection
- Sending/Receiving via USB
- The USB driver

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 ([Operating system wrapper layer](#) on page 81) and the IC package ([pkgs/ic](#) on page 78) 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.

If 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, and 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 detects the presence of a modem and creates links to it in the form of file handles in the `/dev` directory.

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

A

- abstraction, package, [75](#)
- account profiles
 - activating, [42](#)
 - default, autoactivate, [42](#)
 - deleting, [42](#)
 - description, [41](#)
 - editing, [42](#)
 - identifying, [42](#)
 - number supported, [41](#)
 - reading, [42](#)
- accuracy (LBS), get/set, [48](#)
- air servers
 - available, [28, 59](#)
 - register callback, [27](#)
 - select, [59](#)
 - unavailable, [28](#)
- almanac, [50](#)
- antenna, problem notification, [58](#)
- API
 - initialization, [28](#)
 - initialize with SwiApiStartup, [27](#)
 - layer, [23](#)
 - notifications, [23](#)
 - reference, online, [17](#)
 - shut down, [30](#)
- API-side, [73, 82](#)
- APN, profile, [41](#)
- application
 - building, [67](#)
 - host, closing, [33](#)
- architecture, software, [21](#)
- assistance data, [50](#)
- asymmetric notifications, [23](#)
- available services, change notification, [58](#)
- azimuth, read, [47](#)

B

- band, setting, [43, 44](#)
- blocking call, [83](#)
- bootloader
 - build date, [59](#)
 - version, [59](#)

C

- call handling, [83](#)
- callback function
 - deregistering for shutdown, [30](#)
 - notifications, processing, [32](#)
 - register, [25](#)
 - registering, [63](#)

CHV1

- change code, [38](#)
- description, [36](#)
- enabling/disabling, [36, 38](#)
- unblocking, [38](#)
- verification, [36](#)

CHV2

- change code, [38](#)
- description, [36](#)
- unblocking, [38](#)
- verification, [37](#)

- CnS layer, package, [76](#)

- code samples, [71, 72](#)

- communication

- host application layer, [23](#)
 - inter-process, package, [78](#)

- ConnectGSM, sample application, [71](#)

D

- data connection
 - establishing, [57](#)
 - overview, [57](#)
- data relay, package, [77](#)
- data services, package, [77](#)
- Debian, supported distribution, [15](#)
- demultiplexing APIs, [61](#)
- Demux application, [62](#)
- development systems, system requirements, [16](#)
- device detection, [84](#)
- device driver, installation, [19](#)
- dialup networking connection, [58](#)
- distribution, files, [19](#)
- DNS address, profile, [41](#)
- driver considerations, [84](#)
- driver, unloading and reloading, [29](#)

E

- EID, [59](#)
- elevation, read, [47](#)
- environment, setting up, [19](#)
- ephemeris, [50](#)
- error codes, [65](#)
- error handling, [65](#)
- ESN, [59](#)

F

- factory serial number, [59](#)
- files, distribution, [19](#)
- firmware
 - build date, [59](#)
 - flash image, [59](#)
 - layer, [23](#)
 - version, [59](#)

fix

See *tracking session*

fix, GPS

current, [48](#)

initiate, [48](#)

most recent, [48](#)

type, get/set, [48](#)

frequency band, setting, [43](#), [44](#)

FSN, [59](#)

H

header files

SDK, /build, [69](#)

xxidefs.h, [68](#)

xxiproto.h, [68](#)

xxudefs.h, [68](#)

xxuproto.h, [68](#)

HIP layer, package, [78](#)

host application

closing, [33](#)

layer, [23](#)

opening, [31](#)

host-initiated requests/responses, [24](#)

I

ICCID, [59](#)

IMEI, [59](#)

initialization, API, [28](#)

installation

device driver, [19](#)

SDK, [19](#)

IP address, profile, [41](#)

IP header compression, profile, [41](#)

IPC channels, [83](#)

K

keep warm

disabling, [50](#)

enabling, [50](#)

status, [50](#)

L

layers

API, [23](#)

firmware, [23](#)

host application, [23](#)

modem driver, [23](#)

software, [21](#)

LBS

almanac, [50](#)

default parameters, modem, [47](#)

ephemeris, [50](#)

fix settings, [48](#)

force assistance data download, [50](#)

keep warm processing, [50](#)

location, current, [48](#)

location, last known, [48](#)

network initiated fix request, [50](#)

overview, [47](#)

satellite information, [47](#)

status, [47](#)

tracking session, ending, [49](#)

tracking session, initiating, [49](#)

tracking sessions, [48](#)

Linux, distributions supported, [15](#)

location based services. See *LBS*

logging, package, [77](#)

M

main(), pkgs/sdk, [79](#)

make files, library location, [67](#)

MEP code, [35](#)

message layer, package, [75](#)

modem

availability, [31](#)

band switch notification, [57](#)

characteristics, function calls, [59](#)

driver layer, [23](#)

features, available, [59](#)

features, customizable, [59](#)

hardware version, [59](#)

information, package, [77](#)

LBS parameters, default, [47](#)

location fix, current, [48](#)

location fix, most recent, [48](#)

network registration, [44](#)

network registration status, [57](#)

network selection, [44](#)

power up/down, [32](#)

reset, [29](#)

status, [43](#)

unblocking, MEP code, [35](#)

modem-initiated notifications, [24](#)

N

network

registering modem, [44](#)

registration, [43](#)

registration, enabling, [33](#)

NMEA traffic, [63](#)

Notification, sample application, [71](#)

notifications

- API support, 83
- asymmetric, 23
- asymmetric, modem-initiated, 24
- disabling, 32
- disabling for shutdown, 30
- enabling, 32
- managing, 25
- processing, 32
- symmetric, 23

O

OS wrapper layer, 81

P

packages. See *pkgs*

password, profile, 41

performance (LBS), get/set, 48

pkgs

- /, root directory, 74
- /aa, SDK data type abstraction, 75
- /am, message layer, 75
- /ap, API source code, 75
- /cn, CnS layer, 76
- /dd, modem information, 77
- /dl, logging, 77
- /dr, data relay, 77
- /dr, data services, 77
- /er, abort facility, 78
- /hd, packet frame/deframe package, 78
- /hp, HIP layer, 78
- /ic, inter-process communication, 78
- /mm, memory allocation, package, 78
- /os, O/S abstraction layer, 79
- /pi, packing facilities, 79
- /ql, queue facility, 79
- /sdk, main() function, 79
- /sl, string manipulation, 79
- /su, thread startup/initialization, 79
- /us, endpoint access and device detection/scanning, 79
- names, conventions, 73
- naming convention, 68
- overview, 72
- services, overview, 68

PLMN

- availability, 44
- mode, set, 44
- read, 44
- selection modes, automatic/manual, 32

PlmnSelect, sample application, 71

portability, SDK, 81

porting, to other embedded OS, 82

power saving via notifications, 24

PRI details, 59

process model, 62, 82

process, definition, 82

ProfileManipulate, sample application, 71

profiles. See *account profiles*

PUK, definition, 38

R

RadioOnOff, sample application, 71

registration, network, 33, 43

release notes, SDK, 70

request/response

- host-initiated, 24

- patterns, 23

requests, host-initiated, 24

requirements

- development system, 16

- system, 15

responses, symmetric notifications, 24

RSSI, change notification, 58

S

satellite information, read, 47

SDK

- /build, 69

- /build/bin, 69

- /build/include, 70

- /build/lib, 70

- /docs, 70

- /docs/SwiApiReference, 71

- /pkgs. See *pkgs*

- /Sample_Code, 71

- /Sample_Code/Common, 71

- /Sample_Code/gsm, 71

- /Utility/cdma, 71, 72

- /Utility/common, 72

- /Utility/gsm, 72

- directory structure, 69

- installation, 19

SDK-side, 73, 82

services

- availability, 32

- supported, 62

SetLbsClearAssistance, 50

setup of SDK, 19

shared memory messaging facility, 83

shut down API, 30

SignalStrength, sample application, 71

SIM

- availability, 32

- block, retries remaining, 36

- calls, notification required before starting, 35

- characteristics, function calls, 59

- CHV1, description, 36

- CHV2, description, 36

- security, 36

- status change notification, 58

SIMLock, sample application, 71

SNR, read, 47

- software architecture, [21](#)
- source code, API, [75](#)
- state changes, modem-initiated notifications, [24](#)
- SUPL server
 - IP address, get/set, [47](#)
 - port ID, get/set, [47](#)
- SWI_NOTIFY_AirServerChange, [28](#), [29](#)
- SWI_NOTIFY_Band, [44](#), [57](#)
- SWI_NOTIFY_BandWrite, [44](#)
- SWI_NOTIFY_GsmProfileChange, [41](#), [42](#)
- SWI_NOTIFY_LbsNiReq, [50](#)
- SWI_NOTIFY_LbsPaWarmBegin, [50](#)
- SWI_NOTIFY_LbsPaWarmDone, [50](#)
- SWI_NOTIFY_LbsPaWarmStatus, [50](#)
- SWI_NOTIFY_LbsPdBegin, [48](#)
- SWI_NOTIFY_LbsPdData, [48](#)
- SWI_NOTIFY_LbsPdDone, [48](#), [49](#)
- SWI_NOTIFY_LbsPdEnd, [48](#), [49](#)
- SWI_NOTIFY_LbsPdUpdateFailure, [48](#)
- SWI_NOTIFY_LbsSatInfo, [48](#)
- SWI_NOTIFY_NetworkStatus, [58](#)
- SWI_NOTIFY_PktSessionCall, [58](#)
- SWI_NOTIFY_PlmnAvailable, [44](#)
- SWI_NOTIFY_PlmnMode, [44](#)
- SWI_NOTIFY_PlmnReady, [32](#), [44](#)
- SWI_NOTIFY_RegistrationExp, [57](#)
- SWI_NOTIFY_Rssi, [58](#)
- SWI_NOTIFY_Servicelcon, [58](#)
- SWI_NOTIFY_SimStatusExp, [32](#), [35](#), [58](#)
- SWI_NOTIFY_Temperature, [58](#)
- SWI_NOTIFY_TransmitAlert, [58](#)
- SWI_RCODE, [65](#)
- SwiActivateProfile, [42](#)
- SwiApiCmBasic.h, [70](#)
- SwiApiDemuxBegin, [63](#)
- SwiApiDemuxEnd, [63](#)
- SwiApiDemuxReg, [63](#)
- SwiApiDemuxSend, [63](#)
- SwiApiDemuxSetheader, [63](#)
- SwiApiDemuxStartup, [63](#)
- SwiApiGsmBasic.h, [70](#)
- SwiApiGsmNetwork.h, [70](#)
- SwiApiStartup, [27](#), [28](#), [82](#)
- SwiApiWaitNotification, [25](#), [32](#)
- SwiChv2StatusKick, [37](#), [38](#)
- SwiDataTypes.h, [70](#)
- SwiDeRegisterCallback, [30](#)
- SwiEnableNotif, [32](#)
- SwiEraseProfile, [42](#)
- SwiGetAvailableFeatures, [59](#)
- SwiGetAvailAirServers, [27](#), [28](#), [31](#), [59](#)
- SwiGetBandInfo, [44](#)
- SwiGetBootAndHoldMode, [31](#)
- SwiGetBootloaderBuildDate, [59](#)
- SwiGetBootVersion, [59](#)
- SwiGetDeviceID, [59](#)
- SwiGetFeatureCustomization, [59](#)
- SwiGetFirmwareBuildDate, [59](#)
- SwiGetFirmwareVersion, [59](#)

- SwiGetFlashImgInfo, [59](#)
- SwiGetGsmProfileBasic, [42](#)
- SwiGetGsmProfileDns, [42](#)
- SwiGetGsmProfileSummary, [42](#)
- SwiGetHardwareVersion, [59](#)
- SwiGetIccId, [59](#)
- SwiGetImei, [59](#)
- SwiGetLastError, [65](#)
- SwiGetLbsFixSettings, [48](#)
- SwiGetLbsPalpAddr, [47](#)
- SwiGetLbsPaParam, [47](#)
- SwiGetLbsPaPortId, [47](#)
- SwiGetLbsPdData, [48](#)
- SwiGetLbsPdStatus, [47](#)
- SwiGetLbsSatInfo, [47](#)
- SwiGetModemDisable, [32](#)
- SwiGetPLMNMode, [44](#)
- SwiGetPLMNSelection, [44](#)
- SwiGetPriInfo, [59](#)
- SwiGetSerialNumber, [59](#)
- SwiGetSimLock, [36](#)
- SwiGetUsbdInfo, [59](#)
- SwiNotify, [27](#), [83](#)
- SwiRcodes.h, [70](#)
- SwiRegisterCallback, [25](#), [27](#), [28](#)
- SwiSelectAirServers, [59](#)
- SwiSetDefaultProfile, [42](#)
- SwiSetGsmProfileBasic, [42](#)
- SwiSetGsmProfileDns, [42](#)
- SwiSetHostStartup, [31](#), [32](#), [43](#)
- SwiSetLbsFixSettings, [48](#)
- SwiSetLbsNiReq, [50](#)
- SwiSetLbsPalpAddr, [47](#)
- SwiSetLbsPaKeepWarmStart, [50](#)
- SwiSetLbsPaKeepWarmStop, [50](#)
- SwiSetLbsPaPortId, [47](#)
- SwiSetLbsPdEndSession, [49](#)
- SwiSetLbsPdGetPos, [48](#)
- SwiSetLbsPdTrack, [49](#)
- SwiSetMEPUnlock, [35](#)
- SwiSetModemDisable, [32](#), [43](#)
- SwiSetPLMNMode, [44](#)
- SwiSetRadioBandCfg, [44](#)
- SwiSetSimLock, [36](#), [38](#)
- SwiSetSimPassword, [38](#)
- SwiSetSimVerify, [36](#), [37](#)
- SwiStartPLMNSearch, [44](#)
- SwiStopAllNotif, [30](#), [32](#)
- SwiStopNotify, [32](#)
- symmetric notifications, [23](#)
- system requirements, [15](#)

T

- temperature, notification, [58](#)
- thread
 - creation, [83](#)
 - definition, [82](#)
 - model, SDK, [83](#)

timeouts, host-initiated requests/responses, [24](#)
tracking session
 ending, [49](#)
 initiating, [49](#)
 network initiated, [50](#)

U

Ubuntu, supported distribution, [15](#)
unblocking, CHV1/CHV2, [38](#)
USB access, [84](#)
USB descriptor build details, [59](#)
username, profile, [41](#)



SIERRA
WIRELESS™