System Services Reference

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Tuesday, August 19, 2014

Table of Contents

About This Reference **7**
Typographical conventions 9
Technical support 11

Chapter 1: Artwork Client (artwork_client_car2) **13**

Chapter 2: Audio Management **15**
Audio manager PPS objects 17

Chapter 3: Certificate Management **19**

Chapter 4: Geolocation **23**

Chapter 5: Handsfree Telephony **25**
Handsfree telephony in QNX CAR 26
Processing the handsfree call 29
io-acoustic 30
Configuring io-acoustic 31
Configuration keys 32
Example configurations 36
Acoustic processing tuning files (.qcf) 38
Remote control server (RCS) 39
Using the io-acoustic API 42
io-acoustic API 44
IOAP_* type definitions 44
IOAP_HF_EVENT_* 46
ioap_device_t 47
ioap_event_t 48
ioap_event_next() 49
ioap_hf_attach() 50
ioap_hf_config() 51
ioap_hf_get_latency_estimate() 52
ioap_hf_get_log_level() 54
ioap_hf_get_output_volume() 55
ioap_hf_go() 56
ioap_hf_latency_estimate_t 58
ioap_hf_latency_test_t 59
ioap_hf_mute() 59
ioap_hf_prepare() 61

ioap_hf_read_events() 63
ioap_hf_register_events() 64
ioap_hf_route() 66
ioap_hf_set_log_level() 68
ioap_hf_set_output_volume() 69
ioap_hf_setup() 70
ioap_hf_setup_t 72
ioap_hf_start_latency_test() 72
ioap_hf_stop() 74
ioap_io_map_t 75

Chapter 6: Image Generation 77
gen-ifs 78
gen-osversion 80
mkimage.py 82
mksysimage.py 84
mktar 87

Chapter 7: Keyboard 91
Keyboard (keyboard-imf) 93

Chapter 8: MirrorLink 95
The mlink-daemon service—discoverer, launcher, and audiorouter 98
The mlink-rtp service—RTP audio streaming 101
The mlink-viewer service—MirrorLink viewer app 102

Chapter 9: Navigation Engine 105

Chapter 10: Network Manager (net_pps) 107

Chapter 11: Now Playing Service 109
Using the now playing service 111
Now playing service PPS objects 113
Now Playing Service (nowplaying) 114

Chapter 12: Radio 115
RadioApp 116

Chapter 13: Realtime Clock Synchronization 117

Chapter 14: Shutdown service (coreServices2) 119

Chapter 15: Software Updates 123
Software update core library 124
Architecture of swu-core library 124
Key concepts of the library 126
How software update applications integrate with swu-core 129
Manifest file 139
SWU library API 142
Software update daemon 235
swud 235
Loading swud modules 236
Developing swud modules 237
Reference modules 238
Generating a delta file 241
SWU module API 244

Chapter 16: System Launch and Monitor (SLM) 247

Chapter 17: Tether Manager (tetherman) 249

Chapter 18: Wi-Fi configuration (wpa_pps) 251

About This Reference

The *System Services Reference* lists the main services available on the QNX CAR platform. This reference describes each of these services and, where applicable, how to configure and run them.

To find out about:	See:
The artword client (`artwork_client_car2`) for retrieving album art	*Artwork client* (p. 13)
Audio services (`audioman`, `nowplaying`)	*Audio management* (p. 15)
Certificate management	*Certificate management* (p. 19)
Geolocation	*Geolocation* (p. 23)
Handsfree telephony with acoustic processing, including acoustic echo cancellation	*Handsfree Telephony* (p. 25)
Image generation (`mksysimage.py` and other utilities)	*Image Generation* (p. 77)
Keyboard (`keyboard-imf`)	*Keyboard* (p. 91)
Using the car's HMI to view and control MirrorLink apps on a smartphone	*MirrorLink* (p. 95)
Enabling the navigation engine	*Navigation Engine* (p. 105)
Setting up networking	*Network manager (`net_pps`)* (p. 107)
Making sure that different media players (including phones) and media controllers don't clash	*Now Playing Service (`nowplaying`)* (p. 109)
Radio (TI SDR)	*Radio* (p. 115)
Using the realtime clock	*Realtime Clock Synchronization* (p. 117)
Shutting down	*Shutdown service (`coreServices2`)* (p. 119)
Writing a software update application based on the update library shipped with the platform	*Software updates* (p. 123)
Managing the launch order of processes at startup	*System Launch and Monitor (SLM)* (p. 247)

To find out about:	See:
Tethering portable devices	*Tether manager (`tetherman`)* (p. 249)
Wi-Fi (WPA) configuration	*Wi-Fi configuration (`wpa_pps`)* (p. 251)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Artwork Client (`artwork_client_car2`)

Retrieve multimedia artwork.

Syntax:

```
artwork_client_car2 -p tmp_path -c cache_path
                    [-n client_name] [-s cache_size]
                   [-w write_frequency] [-l max_lookups] [-t]
```

Runs on:

QNX Neutrino

Options:

`-p` ***tmp_path***

The path to the temporary location where artwork will be placed.

`-c` ***cache_path***

The path to the artwork cache to be used.

`-n` ***client_name***

The name of the artwork client. Default is `albumart_client`.

`-s` ***cache_size***

The size of the artwork cache, in bytes. Default is 1 MB.

`-w` ***write_frequency***

The frequency, in milliseconds, for the artwork client to write to persistent storage. Default is 100.

`-l` ***max_lookups***

The maximum number of artwork items that the artwork client will look up and store. Default is 100.

`-t`

Generate thumbnails. Default is `false`.

Description:

The artwork_client_car2 service retrieves album artwork. SLM starts this service at system startup.

Chapter 2
Audio Management

The QNX CAR platform uses the audio manager to route audio streams to output devices, and the audio manager and the now playing service to manage the behavior of concurrent audio streams.

Overview

The QNX CAR platform supports diverse audio stream types (sound effects, ringtones, media from a media player, etc.). Each audio stream type must be routed to the most appropriate output device (speakers, headphones, etc.). When two or more audio streams request access to an output device, applications must know which audio stream takes priority, and what should be done with the other audio streams so that:

- the audio stream with the highest priority takes precedence and has access to the preferred output device(s)
- other audio streams are attenuated or muted, as required by the system configuration
- media playback is stopped or paused when an audio stream is “ducked” in favor of a higher priority audio stream, and restarted when appropriate

Applications implemented on the QNX CAR platform should use the audio manager to route audio streams to their preferred output devices, and to attenuate or mute audio streams when higher-priorty audio stream types open, based on the configuration. The audio manager doesn't pause or stop audio playback, however. For example, the audio manager may mute media playback when the telephone rings, but the media will continue playing. Applications should use the now playing service to manage pausing and stopping playback.

Audio manager

The audio manager and its library of functions and data structures provide:

- automatic routing, and manual routing of the PCM *preferred* path
- audio stream type identification
- audio concurrency policy (ducking) management
- audio device monitoring and mounting (e.g., headset, A2DP, HDMI)

For more detailed information about using the audio manager, see *Audio Manager Library Reference*. For information about the PPS objects used by the audio manager in QNX CAR, see “Audio manager PPS objects”.

Now playing service

The now playing service (`nowplaying`) can be used along with the audio manager to manage audio stream concurrency. It supports two distinct PPS interfaces, one for media players (including phones), and one for media controllers.

All media players and media controllers on a system should register with the now playing service and subscribe and publish to the relevant PPS objects in order to know what other media players and controllers are doing, and to let other players and controllers know what they are doing.

For more detailed information about using the now playing service, see "Now Playing Service".

Audio manager PPS objects

The audio manager uses PPS objects to communicate with other system components and third-party applications.

The audio manager uses the PPS objects listed below. For information about these objects, see the relevant pages in the *PPS Objects Reference*.

- `/pps/services/audio/audio_router_control`
- `/pps/services/audio/audio_router_status`
- `/pps/services/audio/control`
- `/pps/services/audio/devices/`
- `/pps/services/audio/mixer`
- `/pps/services/audio/status`
- `/pps/services/audio/types/`
- `/pps/services/audio/voice_status`

Chapter 3
Certificate Management

Certificate management is handled by a service available to client applications that need to validate certificates and private keys for operations such as VPN access, Wi-Fi access, and SSL webpage access.

Overview

The certificate manager service (`certmgr_pps`) provides a centralized service that offers certificate and private key-related operations to services and applications, such as S/MIME, VPN, Wi-Fi and the web browser. In this QNX CAR release, certificate management is used only by the web browser for authenticating SSL website certificates.

Adding a certificate

Certificates are stored at `/var/certmgr`. This directory includes subdirectories for the various services and applications that require certification management. Each sub-directory contains `user_trusted directory` sub-directories with the trusted certificates. For example: `/var/certmgr/web/user_trusted/`.

PPS objects

The certification manager uses the following PPS object: `/pps/services/certmgr/control`. With the current release, this object is used only for QNX CAR internal communications; third-party applications don't need to publish or subscribe to it.

Browser behavior

The images below show how the browser displays information about certificates to the user.

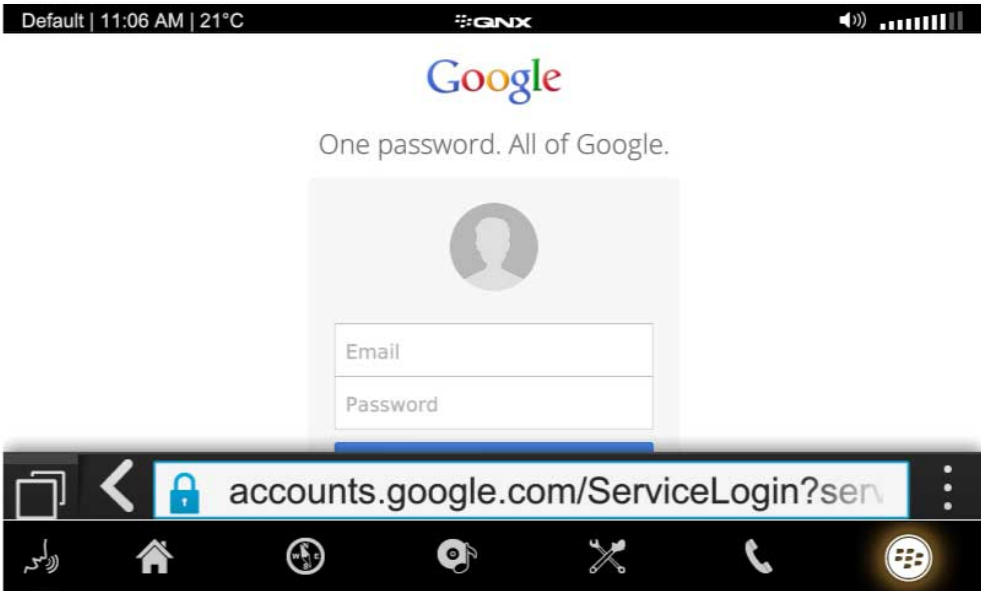


Figure 1: The browser on a page with an authenticated certificate. To the left of the URL, the blue lock icon indicates that the certificate manager authenticated the page's certificate.

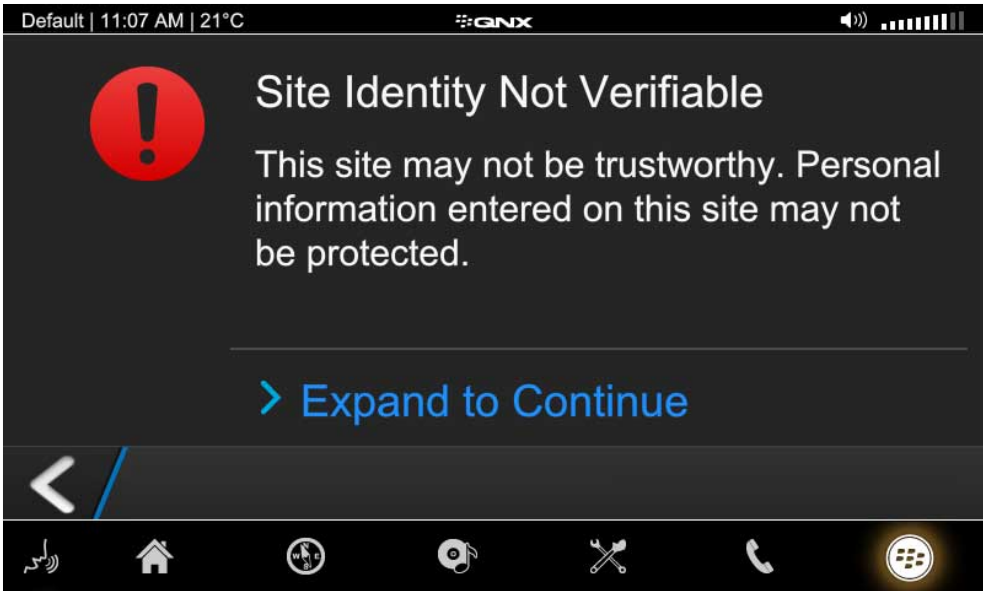


Figure 2: The warning shown by the browser when it encounters a website whose certificate it can't authenticate.

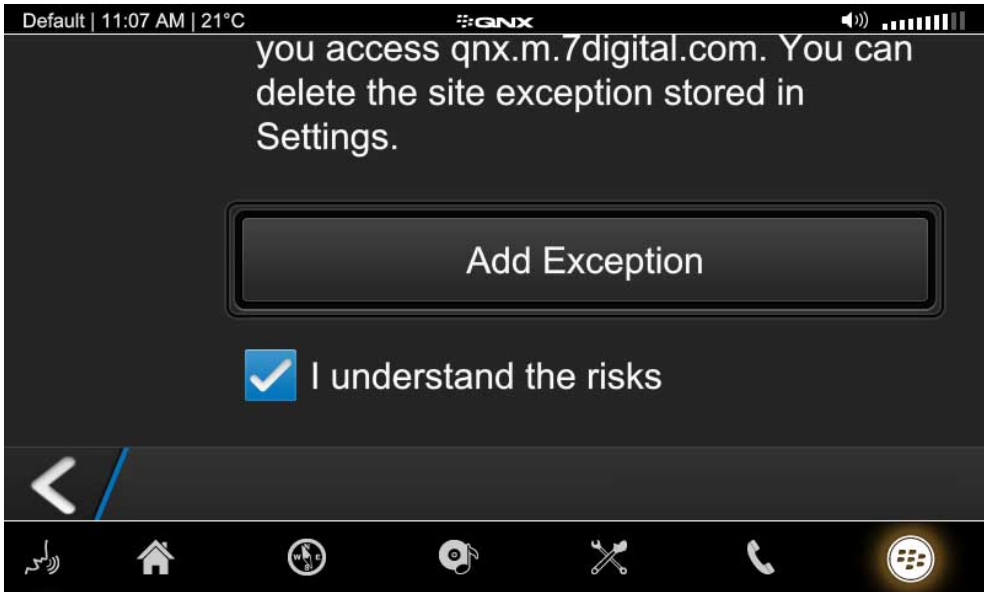


Figure 3: The browser inteface that allows the user to choose to allow a webpage with an unauthenticated certificate to load.

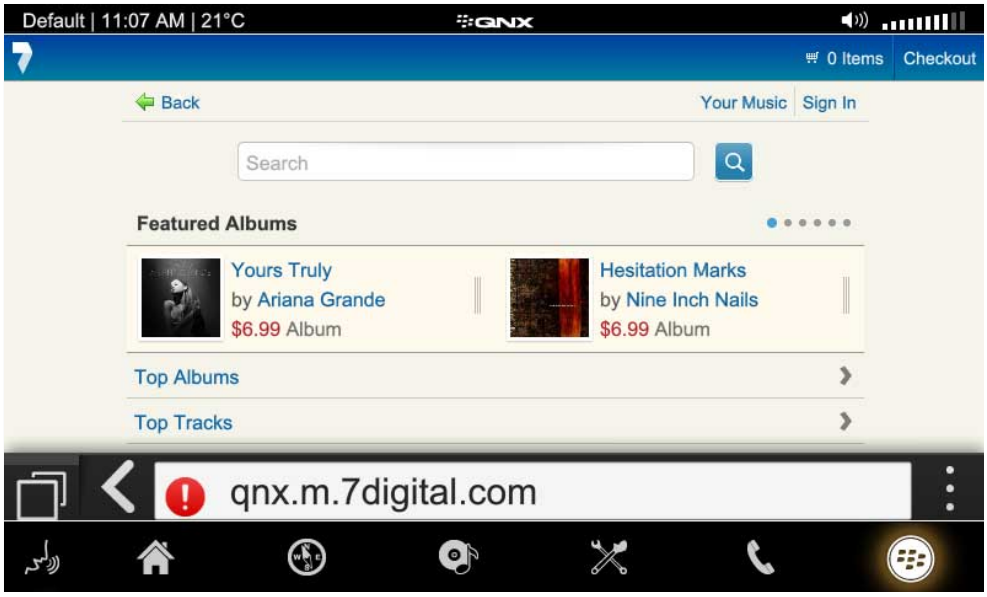


Figure 4: A webpage with an unauthenticated certificate. To the left of the URL, the red exclamation mark indicates that the certificate manager couldn't authenticate the page's certificate.

Chapter 4
Geolocation

The Geolocation service provides the current location of the client based on its IP address.

Upon receipt of a `location` request message from the client, the Geolocation service queries *http://www.hostip.info* to get the current location, based on the client's IP address. The correctness of the result depends on the contents of the database that `hostip.info` provides. If the client's IP isn't in the database, an incorrect location might be returned.

Client queries about location information can be made using the following PPS command:

```
(exec 3<>/pps/services/geolocation/control?wait && echo 'msg::
location\nid::test\ndat:json:
{"period":5.0,"provider":"network","fix_type":"wifi"}
' >&3 &&
cat <&3)
```

where *period* specifies the interval between updates from the server. If the period is 0 then the update is provided only once. The Geolocation service responds to the client in the following format in the `control` object:

```
@control
res::location
id::test
dat:json:{"accuracy":60,"latitude":45.3333,"longitude":-75.9}
```

The QNX CAR browser also uses the Geolocation service to query the location as shown above.

The Geolocation service is multithreaded and can handle requests from multiple clients at the same time.

For more information about the PPS objects the Geolocation service uses, see these entries in the *PPS Objects Reference*:

- /pps/services/geolocation/control
- /pps/services/geolocation/status

Chapter 5
Handsfree Telephony

The QNX CAR platform includes support for handsfree telephony using the io-acoustic resource manager.

Why is acoustic processing necessary?

During an automotive handsfree call, the voice of the far-side speaker (the caller at the remote end of the conversation) is played out over the vehicle loudspeakers and is picked up on the near-side microphone along with the driver's voice. If they are not attenuated, the far-side speaker's words as well as background noise are audible to the far-side speaker as an echo repeating his or her voice and the noise. Acoustic echo cancellation is therefore required to maintain acceptable sound quality and intelligibility during a handsfree call.

The automobile can be an acoustically hostile environment for capturing speech signals. Road, traffic, and engine noises tend to mask speech signals, and requirements for microphone positioning can leave the voice almost hidden by noise. The result can be a telephone conversation that is tiring to listen to or difficult to understand. To further complicate matters, the noise in a car is dynamic, changing its loudness and frequency content due to many factors, including road surface, vehicle speed, open or closed windows, and so on. Acoustic noise reduction improves the subjective quality and the objective intelligibility of speech signals by removing unwanted noise and distortion and enhancing the speech.

In automotive environments, variation in loudspeaker and microphone placement can create a very complicated acoustic path. For instance:

- The echo may be several times louder than the near-side speaker's speech.
- The echo may traverse moving bodies before arriving at the microphone, making it difficult to prediction and cancel.

Acoustic echoes can also be a problem in more conventional handsfree speaker phones and handheld phones (including mobile phones), where the loudspeaker and microphone are in the same physical housing.

Acoustic echo cancellation is part of the QNX Acoustic Processing Suite 2.0. Your QNX representative can provide you with documentation for the complete suite, as well as assistance implementing and tuning AEC and other acoustic processing features in your project and production vehicles.

Handsfree telephony in QNX CAR

Handsfree telephony in the QNX CAR platform uses the `io-audio`, `io-acoustic`, and `io-bluetooth` services.

Overview

The QNX CAR platform includes support for handsfree telephony, including acoustic echo cancellation (AEC), and handsfree Bluetooth phone support implemented on its reference boards. It uses acoustic echo cancellation to:

- extract voice from cabin noise created by road services, construction, engines, wind, rain, and other vehicles
- improve the clarity, quality, and accuracy of voice communication
- enhance the performance of in-car handsfree communication and speech recognition

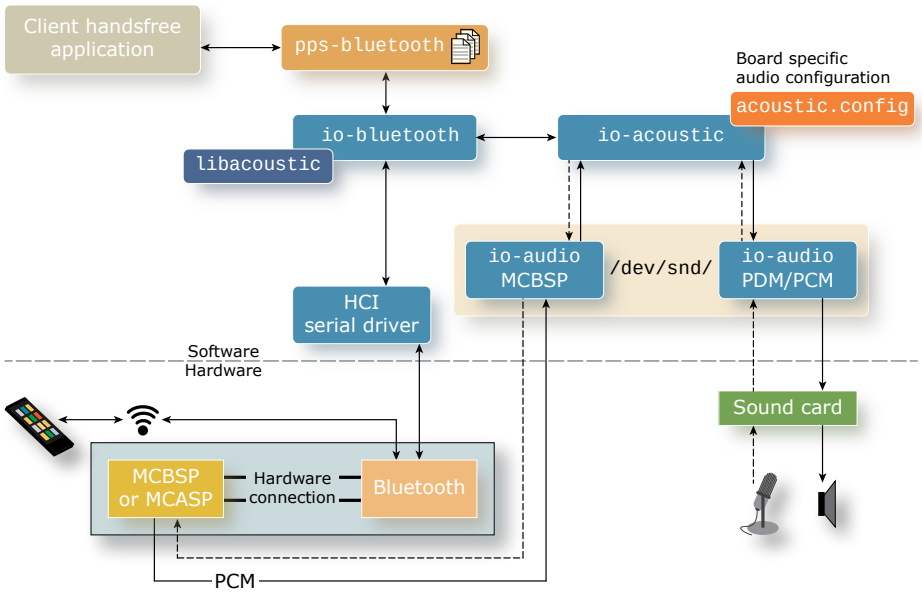


Figure 5: Overview of echo cancellation for a Bluetooth handsfree connection

The figure above shows how handsfree telephony is implemented on the QNX CAR platform.

You may want to configure your implementation by adjusting the parameters in the `io-acoustic` configuration file. You will only need to use the `io-acoustic` *API* (p. 44) if you replace `io-bluetooth` with your own custom Bluetooth service.

Far-side speech

The far-side speech is processed as follows:

1. The `io-bluetooth` process uses Bluetooth to connect the in-vehicle system and the cell phone, and uses the `libacoustic` library to configure and control `io-acoustic`.
2. The audio stream passes through the board's on-chip Bluetooth module and its on-chip multi-channel buffered serial port (MCBSP).
3. The `io-acoustic resource` manager uses QNX acoustic processing to enhance the received audio signal to compensate for vehicle cabin acoustics and noise, and increase intelligibility. From the MCBSP, the audio stream is routed through `io-audio`, then `io-acoustic`, which uses a board-specific configuration file (`acoustic.conf`). This file sets parameters such as the number of input and output channels and devices, the audio stream route, and the compensation for system latency. It also specifies an acoustic tuning file, which can be tailored for the expected acoustic environment.
4. `io-acoustic` returns the stream to `io-audio`, which passes it on to the sound card for output to the loudspeakers.

Near-side speech and acoustic echo

The near-side speech and the acoustic echo are processed as follows:

1. Near-side speech and the acoustic echo from the loudspeakers picked up by the in-vehicle microphones are routed back through `io-audio`, `io-acoustic`, and the on-chip MCBSP and Bluetooth processing.
2. The `io-acoustic` resource manager uses the QNX AAP component to attenuate the echo and clarify the speech. See "*Processing the handsfree call* (p. 29)" for more detailed information about how the QNX acoustic processing library processes the input and output signals of a handsfree call.

Who needs to use the `io-acoustic` API?

The QNX CAR platform is delivered with ready-to-use handsfree telephony. The included `io-bluetooth` service included in the platform uses the `io-acoustic` API to manage the acoustic processing for handsfree telephony. Thus, you will need to use the `io-acoustic` API only if:

- you replace the included `io-bluetooth` with your own Bluetooth service

or

- you write your own program to do latency tests using the *ioap_hf_latency_*()* functions in the `io-acoustic` API.

If you write an application to perform latency tests, you will still need to use `io-bluetooth` to start and stop the call and its audio.

Only one service should ever use `io-acoustic`. Do *not* attempt to use both `io-bluetooth` and your own Bluetooth service to control `io-acoustic`. Attempting to do so will lead to unpredicable results.

Processing the handsfree call

The QNX CAR acoustic processing module uses reference signals, and processes input and output signals to improve the sound quality and intelligibility of handsfree calls.

The figure below shows a simplified view of handsfree acoustic echo cancellation through the QNX acoustic processing module:

1. On the near-side (in the vehicle), one or more microphone signals are input for send-signal processing.
2. The audio signal received from the far-side of the conversation is processed to improve sound quality before it is output to the local system.
3. The send-signal processing produces an echo-free and noise-reduced output signal, which is sent to the remote system.

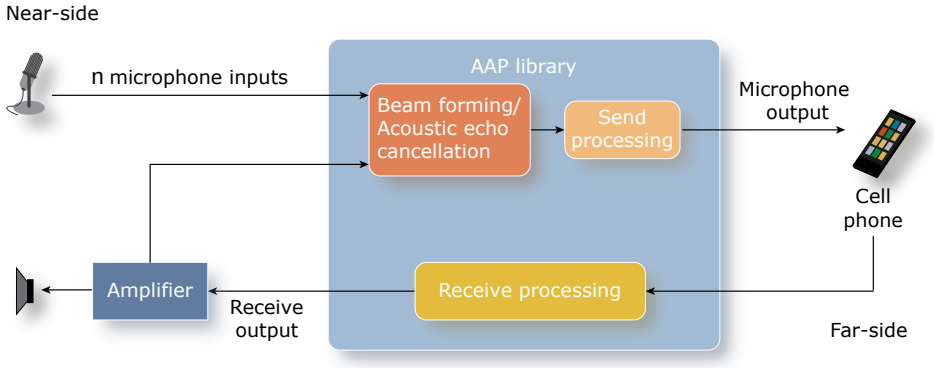


Figure 6: Input and output signals in handsfree telephony

io-acoustic

Provide integrated access to acoustic processing capabilities, including acoustic echo cancellation (AEC) and noise reduction (NR)

Syntax:

```
io-acoustic [-c filepath] [-f] [-n filepath]
            [-t threads] -u UID:GID [-v]
```

Options:

`-c` *filepath*

Location of the `io-acoustic` configuration file. The default location is `/etc/acoustic/acoustic.conf`.

`-f`

Run `io-acoustic` in the foreground, not as a daemon.

`-n`

Set the prefix for the `io-acoustic` mount location. For example: `io-acoustic -n /dev/newplace`. Default mount location is `/dev/io-acoustic`.

`-t` *threads*

The number of threads to reserve for `io-acoustic`. Default is 16.

`-u`

Force `io-acoustic` to run under a specific user ID and group ID, and, optionally, multiple supplementary group IDs (SGIDs). For example, to set the user ID to "devuser" use: `io-acoustic -u devuser:25`. Or, To run as user with UID 10 under primary group ID 80, and inherit the capabilities of groups 80, 81 and 82, use: `io-acoustic -u 80:10:80,81,82`.

`-v`

Increase output verbosity (messages are written to `sloginfo`). The `-v` option is cumulative; each additional `v` adds a level of verbosity, up to 7 levels. For example, `io-acoustic -vvv` sets verbosity to 3. Default is 2.

Description:

The `io-acoustic` resource manager integrates QNX acoustic echo cancellation (AEC) and noise reduction (NR) into the QNX CAR platform. It uses QNX's acoustic processing (AAP) library to perform multichannel signal preprocessing for handsfree (HF) systems.

The QNX CAR platform includes configuration files for each reference board on which it has been implemented. These files are preset for optimal performance and sound quality, given the assumptions QNX was able to make about the environment in which the platform will be used. These settings can be changed by editing the relevant configuration file.

The prefix, number of threads, and verbosity can be set in the configuration file. If these options are set in the configuration file, `io-acoustic` ignores the values entered on the command line.

For information, see "Configuring `io-acoustic`".

Configuring `io-acoustic`

The QNX CAR platform acoustic echo cancellation component includes a configuration file with keys for configuring `io-acoustic`.

About configuring `io-acoustic`

The `io-acoustic` resource manager gets its configuration keys from a configuration file. This file sets up the mapping between the hardware devices on your system and the acoustic processing inputs and outputs, as well as other operational parameters. It is a plain-text file with a series of key-value pairs, separated by module markers.

The QNX CAR platform is shipped with a default configuration file for the current reference board. The keys in this file are preset for optimal performance and sound quality, given the assumptions QNX was able to make about the environment in which the platform will be used. The default configuration file is `/etc/acoustic/acoustic.conf`.

You can specifiy another configuration file by using the `-c` at startup. You can also change some other keys at startup. For more information, see the `io-acoustic` *startup options* (p. 30).

The prefix, number of threads, and verbosity can be set in the configuration file or with the command line at startup. However, if these options are set in the configuration file, `io-acoustic` ignores the values entered at startup.

About routing

When `io-acoustic` starts up it uses the routing specified in its configuration file. If the routing specified in this file fails, then the routing will fall back to the following configuration, using the system-preferred input and output devices:

Direction		
Input	Microphone in	Phone in
	Channel 1 mapped to `IOAP_MIC_IN_1`.	Channel 2 mapped to `IOAP_PHONE_IN_1`.
Output	Speaker out	Phone out
	Channel 2 mapped to `IOAP_SPK_OUT_1`.	Channel 1 mapped to `IOAP_PHONE_OUT_1`.

Since the name of the Bluetooth audio interface varies from system to system, this default routing doesn't configure audio to flow through Bluetooth. However, the routing specified in `acoustic.conf` should be correct for the reference board on which the QNX CAR platform has been implemented, given the assumptions QNX was able to make about the board and the environment in which it will be used.

For information about configuring other acoustic processing parameters, see "*Configuration keys* (p. 32)" below.

Configuration keys

The `io-acoustic` resource manager configuration file keys can be edited to tune acoustic echo cancellation for handsfree telephony.

The tables below list and describe the keys that can be included in an `io-acoustic` configuration file, their default values, and valid values or ranges of values:

Global

Key	Description	Default
`# comment`	Comment line. Text on a line starting with "#" is considered a comment and ignored.	n/a
logfd logfile_descriptor	Log location. 1=stdout, 2=stderr, -1=slog. Default is -1, system log.	-1
verbose verbosity_level	Verbosity level: 0-7. Higher values mean more information is logged by `io-acoustic` regarding warnings, errors and operational events. Default is 2, errors only.	2
threads number_of_threads	Number of threads in thread pool.	16

Key	Description	Default
prefix module_prefix	Installation path prefix for `io-acoustic` modules.	`/dev/io-acoustic`
audiothreadpriority priority	Priority of the audio processing threads.	18

Module markers

Key	Description	Default
<hf apm-aap-hf.so>	Start marker for handsfree configuration, with `apm-aap-hf.so` as the associated DLL name, and `hf` as the mount location. If the default prefix is used, the HF module will be mounted at `/dev/io-acoustic/hf`.	n/a
</hf>	Handsfree configuration end marker.	n/a

HF module audio routing

Key	Description	Default
inputs number_of_devices	Number of input devices to open (1 to 4). Default is 1, with 2 channels open, the first mapped to *IOAP_MIC_IN_1* and the second to *IOAP_PHONE_IN_1*.	1
ipathX device_path	Path for input device *X*. If no path is specified, the preferred device will be used. If there are multiple devices with both microphone and reference inputs to route, devices with microphone inputs should be routed first.	Preferred device
ichannelsX number_of_channels	Number of audio channels to open from input device *X* (1 to 4). This key must be specified for each input device. The minimum number of channels across all devices is 2 (one microphone, and one phone or reference). The maximum number of channels for the system 4 (two microphones, one phone and one reference).	See *inputs*
irouteX_Y route	Acoustic processing input to route from channel *Y* of input device *X*. This key must be specified for each input device channel. Can be any of *IOAP_MIC_IN_1 IOAP_MIC_IN_2*, *IOAP_REF_IN_1*, or *IOAP_PHONE_IN_1*.	See *inputs*
outputs number_of_devices	Number of output devices to open (1 or 2). Default is 1, with 2 channels open, the first mapped to *IOAP_PHONE_OUT_1* and the second to *IOAP_SPKR_OUT_1*.	1
opathX device_path	Path for output device *X*. If no path specified, the preferred device will be used.	Preferred device

Key	Description	Default
ochannelsX number_of_channels	Number of audio channels to open from output device *X*. Can be 1 or 2. This key must be specified for each output device. The minimum number of channels across all devices is 1 (one phone out). The maximum number is 2 (one phone out and one speaker out).	See *outputs*
orouteX_Y route	Acoustic processing output to route to channel *Y* of output device *X*. This key must be specified for each output device channel. It can be either *IOAP_SPKR_OUT_1* or *IOAP_PHONE_OUT_1*.	See *outputs*

Timing

Key	Description	Default
msprime prime_time_in_ms	The time (in milliseconds) to prime speaker output on “go” in order to compensate for system latency. This time period is platform specific. To achieve best system latency, it should be adjusted to the shortest time possible (that is, the shortest time period that is sufficient to prevent underruns).	100
msphoneprime prime_time_in_ms	The time (in milliseconds) to prime phone output on “go” in order to compensate for system latency. This time period is platform specific. To achieve best system latency, it should be adjusted to the shortest time possible (that is, the shortest time period that is sufficient to prevent underruns).	30
msrefdelta_default delta_time_in_ms	The time (in milliseconds) to add to the *msprime* value to compensate for audio system latencies. This time is the difference between the value measured using the latency test and the value specified by *msprime*.	0

Fragment size

Key	Description	Default
framesperfrag number_of_ap_frames	Number of acoustic processing frames per audio fragment. The larger the number, the greater the delay; the lower the number, the higher the CPU load. Must be 1 or greater.	1

Tuning

Key	Description	Default
qcf file_path	Path to acoustic processing tuning file.	n/a

Key	Description	Default
qcf_rcs file_path	Path to an alternate acoustic processing tuning file to load instead of the file specified by the *qcf* parameter, if *RCS* (p. 39) is enabled.	n/a
dbminvol_default volume_in_dB	Minimum (0%) volume to use if no device-specific key is available.	-20
dbmaxvol_default volume_in_dB	Maximum (100%) volume.	10

RCS

Key	Description	Default
rcsactive `true`	Set to "true" to enable the RCS server in `io-acoustic`; any other value is ignored. The server by default is disabled; only enable it if you are using a `io-acoustic` build with RCS enabled and are using QWALive to tune the acoustic processing.	`n/a`
rcsport port_number	TCP/IP port for RCS server.	4000

Other

Key	Description	Default
autoroute `true`|`false`	Set to "true" to enable the `io-audio` router plugin. When the router plugin is enabled, the preferred device will be automatically changed when a new device is connected.	`false`
volumecontrol `true`|`false`	Enable manual volume control. The default (`false`) puts the volume level at a fixed-value specified by the tuning file. service. Set to `true` to enable the "set volume" API function and allow the application to get and set the volume level during and between calls.	`false`
onaudioerr `reprepare`|`restart`	Set how audio overruns and underruns are handled. The default `reprepare` only resets audio processing back to a default state after an audio error. Set to `restart` to force a full audio restart on an audio error. The `restart` mode is slower, but is safer when using an unstable audio driver because in some cases, a reprepare operation can lock the audio device.	`reprepare`

Example configurations

Examples of `io-acoustic` configuration files are useful when learning to tune acoustic echo cancellation.

You can refer to the examples below to help you better understand how to edit your `io-acoustic` configuration file.

Single device input and output

The following simple configuration file configures only the `io-acoustic` handsfree module.

```
# Single device input and output

logfd 2
verbose 6
<hf apm-aap-hf.so>
    inputs 1
    ichannels1 2
    iroute1_1 IOAP_MIC_IN_1
    iroute1_2 IOAP_PHONE_IN_1
    outputs 1
    ochannels1 2
    oroute1_1 IOAP_PHONE_OUT_1
    oroute1_2 IOAP_SPKR_OUT_1
    qcf /etc/acoustic/mono.qcf
    msprime 30
</hf>
```

This example assumes that the `mono.qcf` acoustic processing configuration file specifies mono operation. It instructs `io-acoustic` to open one stereo input device and one stereo output device. For more information , see "*Acoustic processing tuning files (`.qcf`)* (p. 38)".

The two input device channels are mapped to microphone 1 and phone input 1 of the acoustic processing library, and the two output device channels are mapped to speaker output 1 and phone output 1. Because no path to a device is specified in the routings, the preferred input and output devices are used.

In addition, this file configures logging, specifies the location of an acoustic processing configuration file, and configures the speaker output zero-priming.

The table below provides a line-by-line description of the configuration file:

Key	Effect of setting
`logfd 2`	Direct logging to standard error.
`verbose 6`	Increase verbosity to level 6.
`hf apm-aap-hf.so`	Start marker for hands-free configuration.
`inputs 1`	Open one input device.

Key	Effect of setting
`ichannels1 2`	Open 2 channels for input device 1.
`iroute1_1 IOAP_MIC_IN_1`	Route channel 1 of input device 1 to acoustic processing input *IOAP_MIC_IN_1*.
`iroute1_2 IOAP_PHONE_IN_1`	Route channel 2 of input device 1 to acoustic processing input *IOAP_PHONE_IN_1*.
`outputs 1`	Open one output device.
`ochannels1 2`	Open 2 channels for output device 1.
`oroute1_1 IOAP_PHONE_OUT_1`	Route acoustic processing output *IOAP_PHONE_OUT_1* to channel 1 of output device 1.
`oroute1_2 IOAP_SPKR_OUT_1`	Route acoustic processing output *IOAP_SPKR_OUT_1* to channel 2 of output device 1.
`qcf /etc/acoustic/mono.qcf`	Use /etc/acoustic/mono.qcf as the acoustic processing tuning file.
`msprime 30`	Prime speaker output with 30 milliseconds of zeroes on “go” to compensate for system latency.
`/hf`	End marker for handsfree configuration.

Two input devices, two output devices

This configuration file uses two separate input devices and two separate output devices.

```
# Mono input, mono output

<hf apm-aap-hf.so>
   inputs 2
      ichannels1 1
      iroute1_1 IOAP_MIC_IN_1
      ipath2 /dev/snd/pcmC1D0c
      ichannels2 1
      iroute2_1 IOAP_PHONE_IN_1
   outputs 2
      ochannels1 1
      oroute1_1 IOAP_SPKR_OUT_1
      opath2 /dev/snd/pcmC1D0p
      odevice2 0
      ochannels2 1
      oroute2_1 IOAP_PHONE_OUT_1
   qcf /etc/acoustic/mono.qcf
   msprime 30
</hf>
```

Two microphone inputs, a reference input, separate phone and output devices

This configuration file use two microphone inputs, a reference input, and separate phone input and output devices. The `mixer.qcf` tuning file should configure the acoustic processing to mix the two microphone inputs as part of its processing.

```
# Dual mic input, mono output

<hf aap-apm-hf.so>
   inputs 3
      ichannels1 2
      iroute1_1 IOAP_MIC_IN_1
      iroute1_2 IOAP_MIC_IN_2
      ipath2 /dev/snd/pcmC1D0c
      ichannels2 1
      iroute2_1 IOAP_REF_IN_1
      ipath3 /dev/snd/pcmC2D0c
      ichannels3 1
      iroute3_1 IOAP_PHONE_IN_1
   outputs 2
      ochannels1 1
      oroute1_1 IOAP_SPKR_OUT_1
      opath2 /dev/snd/pcmC1D0p
      ochannels2 1
      oroute2_1 IOAP_PHONE_OUT_1
   qcf /etc/acoustic/mixer.qcf
   msprime 30
   </hf>
```

Acoustic processing tuning files (`.qcf`)

The QNX CAR platform handsfree telephony component includes a configuration file with keys for tuning `io-acoustic`.

The QNX acoustic processing module uses binary configuration files to tailor the acoustics (AAP) library for specific vehicle environments. These files are placed in the `/etc/acoustic/` directory, and are identifiable by their `.qcf` extension. You are note required to use an acoustic processing file when implementing a system built on the QNX CAR platform. However, we recommend that you use one in order to be able to tune your acoustic processing for the best possible performance in your vehicles' acoustic environments.

If your acoustic environment is substantially different from the one for which the default acoustic tuning file was defined, you may want to use another tuning file. In this case, you must set the `qcf` key in the `io-acoustic` configuration file to the path to your custom file. For example: `qcf /etc/acoustic/mono.qcf`, or `qcf /etc/acoustic/custom.qcf`.

You may be required to sign additional royalty agreements if you plan to use the full suite of QNX advanced acoustic processing features in your production vehicles. Please contact your QNX representative for more information.

Remote control server (RCS)

QNX handsfree telephony includes a remote control server (RCS) that gives access to the QNX acoustic processing library over a network connection.

Overview

The QNX remote control server (RCS) allows you to use the QWALive tuning application to connect to the acoustic processing library over a network connection. When you use the RCS with QWALive you can:

- change switches to enable and disable features, or to adjust parameters in the acoustic processing library
- stream library input and output audio to an external file
- use audio injection from external files to override library input and output audio

Enabling RCS

To enable RCS in the handsfree telephony module you need to make two changes to the `io-acoustic` configuration file:

- Change the name of the handsfree module library to the name of the RCS-enabled library. For example, change `apm-aap-hf.so` to `apm-aap-rcs-hf.so`.
- Explicitly enable RCS by adding the line `rcsactive true` to the module configuration section in the configuration file.

The sample below shows the module section of the configuration file with the parameters set to enable RCS access:

```
# Single device input and output, RCS enabled

<hf apm-aap-rcs-hf.so>
    inputs 1
    ichannels1 2
    iroute1_1 IOAP_MIC_IN_1
    iroute1_2 IOAP_PHONE_IN_1
    outputs 1
    ochannels1 2
    oroute1_1 IOAP_PHONE_OUT_1
    oroute1_2 IOAP_SPKR_OUT_1
    qcf /etc/acoustic/mono.qcf
    msprime 30
    rcsactive true
</hf>
```

Tuning file load order

At startup, the handsfree module first sets up the acoustic library with built-in defaults. It then uses information from *tuning (`.qcf`) files* (p. 38) to refine the acoustic processing for the given platform and usage scenario. The choice of tuning file depends on whether RCS is enabled.

Normal (RCS not enabled)

If the RCS module isn't enabled, the handsfree module will tune the library in the following order:

1. Load the built-in defaults and the tuning file at the path set by the `qcf` parameter in the `io-acoustic` configuration file (`acoustic.conf`).
2. If loading the specified tuning file fails, load the built-in defaults only.

RCS enabled

If the RCS module is enabled and the `qcf_rcs` field is defined in the `io-acoustic` configuration file, the handsfree module will tune the library in the following order:

1. Load the built-in defaults and the tuning file at the path set by the `qcf_rcs` parameter in the `io-acoustic` configuration file (`acoustic.conf`).
2. If loading the tuning file specifed by the `qcf_rcs` parameter fails, proceed as for a system without RCS enabled.

Changes to tuning parameters

Each time the handsfree module is started by a call to *ioap_hf_go()*, the module tunes the acoustic processing library as defined above, unless QWALive is used to restart the target or *ioap_hf_config()* has been called previously to change the `qcf` parameter.

Behavior when `qcf_rcs` is defined

When the `qcf_rcs` parameter is defined in the `io-acoustic` configuration file, if the RCS client downloads new tuning parameters to the target (QWALive **`Push to Target`**), this download overwrites the contents of the tuning file. All starts made after the download, whether they are initiated through the **`Restart Target`** button in QWALive or by a call to *ioap_hf_go()*, will use the tuning file load order for RCS enabled.

Behavior when `qcf_rcs` is *not* defined

If the `qcf_rcs` parameter isn't defined in the `io-acoustic` configuration file, the QWALive **`Push to Target`** function will overwrite the contents of the tuning file at `/tmp/hf_rcs_config.qcf`. However, when the handsfree module is restarted, it will always use the tuning file load order for a system that doesn't have RCS enabled. This behavior allows you to save the tuning parameters to a file on the target for later reference.

Behavior when the `qcf` parameter is changed

If the tuning file defined by the `qcf` parameter is changed with a call to *ioap_hf_config()*, all starts after this change (initiated through the **`Restart Target`** button in QWALive or by a call to *ioap_hf_go()*) will use the tuning file load order for RCS not enabled.

- To ensure that tuning is consistent across restarts, the handsfree module remembers which method was used to make the most recent change to tuning parameters (QWALive **`Push to Target`** or *ioap_hf_config()*).
- For more information about QWALive, please refer to your QNX Acoustic Processing documentation.

Using the `io-acoustic` API

Projects that use their own Bluetooth service need to use the `io-acoustic` API to manage acoustic echo cancellation and noise reduction.

If you use the `io-acoustic` API, you will have to perform the tasks described below.

Setting up

To enable acoustic echo cancellation on your system, you need to perform a few setup tasks when you start your system:

1. Link to the `libacoustic` libraries.
2. Call *ioap_hf_attach()* to attach `io-acoustic` and get a handle to this resource manager.

Configuration

If you want to use an acoustic tuning file for the AAP library other than the one specified in the `io-acoustic` configuration file, you need to:

1. Call *ioap_hf_config()* to set the path to the AAP library tuning file. For more information about acoustic tuning, see the AAP documentation.

If you want to change the device routing from that specified in the `io-acoustic` configuration file:

1. Call *ioap_hf_route()* to set up the routing between the hardware devices and the acoustic processing inputs and outputs.

Registering for events

When you have completed your initial setup, you need to:

1. Call *ioap_hf_register_events()* to register for `io-acoustic` events.
2. Start a thread to listen for events. In this thread, use the file descriptor returned by *ioap_hf_attach()* to call *select()*.
3. When *select()* returns, call *ioap_hf_read_events()* to process the events.

Starting

To start acoustic processing:

1. Call *ioap_hf_prepare()* to validate the acoustic processing paths set in the configuration file or by the call to *ioap_hf_route()* during setup.
2. Call *ioap_hf_go()* to start the acoustic processing.

Stopping

To stop acoustic processing, call *ioap_hf_stop()*.

Troubleshooting

If you need to get information about your system and how it is configured for acoustic processing you can:

- Call *ioap_hf_setup()* to retrieve the device routes defined in the configuration file or set up by *ioap_hf_route()*.
- Call *ioap_hf_get_log_level()* to get the current `io-acoustic` verbosity level.
- Call *ioap_hf_set_log_level()* to set the `io-acoustic` verbosity level.

Getting and setting the output volume level

Volume control is enabled if *volumecontrol* is set to "true" in the configuration file. Otherwise, the volume level is fixed and specified by the tuning file.

This design supports systems that want the AAP library to control the volume, and systems that want the codec to control the volume. The acoustic echo cancellation performance will be better if the AAP library controls the volume, but this option is more difficult to implement. It requires that your system intercept the volume control change request and pass it into your handsfree application rather than just letting the audio subsystem handle it.

If you configure your system to have the AAP library control the volume, you can use the acoustic processing API to get and set the volume level. The volume level you set with the API remains in force until the next time `io-acoustic` is started:

- Call *ioap_hf_get_output_volume()* to get the acoustic processing output volume.
- Call *ioap_hf_set_output_volume()* to set the acoustic processing output volume.

Estimating system latency

If you do not know your system's latency, you can use *ioap_start_latency_test()* to run a latency test, and *ioap_hf_get_latency_estimate()* to retrieve the results of this test.

`io-acoustic` API

The QNX CAR platform includes APIs for working with *io-acoustic* to manage acoustic processing for handsfree telephony.

This chapter describes the APIs available for managing acoustic processing for handsfree devices in the QNX CAR platform. It describes functions, data structures, type definitions, etc. used for working with *io-acoustic*. The header files for these APIs are:

- `<acoustic/acoustic.h>`
- `<acoustic/hf.h>`

Only one service should ever use `io-acoustic`. Do *not* attempt to use both `io-bluetooth` and your own Bluetooth service to control `io-acoustic`. Attempting to do so will lead to unpredicable results.

IOAP_* type definitions

Handsfree telephony acoustic processing uses a number of pre-defined type definitions.

Limits

Type definitions used by acoustic processing include:

`IOAP_MAX_DEVICES 4`

Maximum number of devices.

`IOAP_MAX_DEVICE_IO 4`

Maximum number of device inputs or outputs.

`IOAP_MAX_DEVICE_PATH 128`

Maximum device path length.

Direction of acoustic processing

`IOAP_OUTPUT 1u`

Output direction.

`IOAP_INPUT 2u`

Input direction.

Input and output channel routing masks

Routing masks are used to manage acoustic processing input and output channels.

IOAP_MIC_IN_1 0x00000001u

First microphone input channel.

IOAP_MIC_IN_2 0x00000002u

Second microphone input channel.

IOAP_REF_IN_1 0x00000010u

First reference input channel.

IOAP_PHONE_IN_1 0x00000100u

First phone input channel.

IOAP_SPKR_OUT_1 0x00001000u

First speaker output channel.

IOAP_PHONE_OUT_1 0x00010000u

First phone output channel.

Actions

IOAP_OFF 0

Disable.

IOAP_ON 1

Enable.

IOAP_NOCHANGE -1

Make no changes.

Logging verbosity levels

IOAP_LOG_SHUTDOWN 0

Shutdown (not used)

IOAP_LOG_CRITICAL 1

Critical (not used)

IOAP_LOG_ERROR 2

Error

IOAP_LOG_WARNING 3

Warning

IOAP_LOG_NOTICE 4

Notice (not used)

IOAP_LOG_INFO 5

Information

IOAP_LOG_DEBUG1 6

Debug detail.

IOAP_LOG_DEBUG2 7

Debug fine detail.

IOAP_LOG_DEBUG3 8

Debug even finer detail.

IOAP_HF_EVENT_*

Handsfree telephony acoustic processing events.

Handsfree telephony acoustic processing delivers the following events:

IOAP_HF_EVENT_STARTED 0x0001

Acoustic processing has started.

IOAP_HF_EVENT_STOPPED 0x0002

Acoustic processing has stopped.

IOAP_HF_EVENT_PREPARED 0x0004

Preparation for acoustic processing is complete.

IOAP_HF_EVENT_ERROR 0x0008

Not currently used.

IOAP_HF_EVENT_RESTART 0x0010

Acoustic processing has restarted.

IOAP_HF_EVENT_FATAL_IO 0x0020

There has been a fatal (non-recoverable) i/o error.

IOAP_HF_EVENT_ALL -1

Enable delivery of all events.

ioap_device_t

Set hardware device routing to acoustic processing input and output.

Synopsis:

```
#include <acoustic/acoustic.h>

typedef struct ioap_device {
    char path[IOAP_MAX_DEVICE_PATH];

    uint32_t nchannels;
    uint32_t route[IOAP_MAX_DEVICE_IO];
} ioap_device_t;
```

Library:

`libacoustic`

Description:

The structure defines a single hardware device, and the routing of a single hardware device's inputs and outputs to the corresponding acoustic processing inputs and outputs.

Member	Type	Description
path	char	The path to the device; null-terminated; `IOAP_MAX_DEVICE_PATH`.
nchannels	uint32_t	The number of populated routes in *route*; must be 1 or 2.
route	uint32_t	The routing for a device channel to the corresponding acoustic processing input or output channel (one of `IOAP_MIC_IN_1`, `IOAP_MIC_IN_2`, etc.). The number populated is specified by *nchannels*; `IOAP_MAX_DEVICE_IO`

For more information about routing, see *ioap_hf_route()* (p. 66).

Classification:

QNX Neutrino

ioap_event_t

Hold acoustic processing event information and data.

Synopsis:

```
#include <acoustic/acoustic.h>

typedef struct ioap_event {
    uint32_t    length;
    uint32_t    type;
    char        data[];
} ioap_event_t;
```

Library:

libacoustic

Description:

The structure `ioap_event_t` is used to hold acoustic processing events. For more information about handsfree events and how to get them, see:

- *ioap_event_next()* (p. 49)
- *ioap_hf_read_events()* (p. 63)
- *ioap_hf_register_events()* (p. 64)
- `IOAP_HF_EVENT_*` (p. 46)

Member	Type	Description
length	uint32_t	The event length, in bytes; includes both the header and the data.
type	uint32_t	The event type. For more information, see software (`*_SW_EVENTS_*`) and handsfree (`IOAP_HF_EVENTS_*`) events.
data[]	char	Event data; optional.

Classification:

QNX Neutrino

ioap_event_next()

Return the location of the next unread acoustic processing event.

Synopsis:

```
#include <acoustic/acoustic.h>

int ioap_event_next(_evt)
    (ioap_event_t*)IOAP_ROUNDUP8((char*)_evt + _evt->length)
;
```

Arguments:

_evt

Pointer to the next event to read.

Library:

`libacoustic`

Description:

The *ioap_event_next()* function returns a pointer to the location of the next unread acoustic processing event. Use this function to retrieve the next event in the event buffer returned by *ioap_hf_read_events()*.

Returns:

`>0`

Success: a pointer to next valid event.

`NULL`

There are no more acoustic processing events to be read.

Errors:

n/a

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

ioap_hf_attach()

Open a connection to the `io-acoustic` handsfree module.

Synopsis:

```
#include <acoustic/hf.h>
int ioap_hf_attach(const char* path_mgr);
```

Arguments:

path_mgr

Path to the instance of `io-acoustic` to which to attach. `NULL` defaults to the standard location for `io-acoustic`. Pass `NULL` unless `io-acoustic` isn't installed in its default location, or there are multiple instances of `io-acoustic` available.

Library:

`libacoustic`

Description:

The *ioap_hf_attach()* function attaches an instance of `io-acoustic` for use with handsfree acoustic processing.

Returns:

`>0`

Success: a handle for the attached `io-acoustic`.

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_config()

Set the next tuning file to use for initializing handsfree processing.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_config(int adp,
                   const char* filename);
```

Arguments:

apd

The handle returned by *ioap_hf_attach()* when it attaches to a process to `io-acoustic`.

filename

Path to the next acoustic processing tuning file to use.

Library:

`libacoustic`

Description:

The *ioap_hf_config()* function sets the next tuning file to use for initializing handsfree processing.

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

EINVAL

Invalid handle or null pointer argument. If there is an issue loading the tuning file, an error is sent through notification events.

ENOMEM

Could not allocate memory to backup tuning file name.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_get_latency_estimate()

Get the latency estimate from a latency test.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_get_latency_estimate(
                int apd,
                ioap_hf_latency_estimate_t* estimate);
```

Arguments:

apd

The handle to `io-acoustic`.

estimate

Pointer to the structure with the results of the latency test.

Library:

`libacoustic`

Description:

The *ioap_hf_get_latency_estimate()* function retrieves the results of latency tests started by *ioap_hf_start_latency_test()* and placed in the `ioap_hf_latency_estimate_t` data structure.

A latency test determines the latency for each of nine clicks that are played back. The two smallest and two largest latency values are discarded. The remaining five latency values are used to calculate the three values returned by the test:

- latency estimate
- latency spread
- cross-correlation

A successful latency test should return a latency-spread of < 4 milliseconds and a cross-correlation of > 500. If the latency spread is greater than 4 milliseconds or the cross-correlation is less than 500, the signal to noise ratio of the clicks should be increased, for example by increasing the playback volume or decreasing the ambient noise level. If the latency is > 100 milliseconds, you should increase the offset specified when starting the test in order to move the estimate calculation into the correct measurement window.

The *msrefdelta_default* key in the `.conf` configuration file should be set to the difference between the *msprime* and the returned latency estimate. See "*Configuration keys* (p. 32)".

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

`EACCESS`

Acoustic processing hasn't been started.

`EAGAIN`

Test result isn't ready yet.

`EFAULT`

An error occurred while retrieving the estimate.

ENOTSUP

Diagnostics processing hasn't been enabled.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_get_log_level()

Get the verbosity level for the handsfree acoustic processing logs.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_get_log_level(int apd, int* level);
```

Arguments:

apd

The handle to `io-acoustic`.

level

Pointer to the location where the log level is stored.

Library:

`libacoustic`

Description:

The *ioap_hf_get_log_level()* function retrieves the current verbosity of the handsfree acoustic processing logs.

The default verbosity is `IOAP_LOG_ERROR`. The log verbosity can be changed by calling *ioap_hf_set_log_level()*, editing the configuration file, or by using the `-v` option when starting `io-acoustic`.

Returns:

0

Success

-1

An error occurred (errno is set).

Errors:

EINVAL

Invalid handle or null pointer argument .

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_get_output_volume()

Get the output volume.

Synopsis:

```
#include <acoustic/hf.h>
int ioap_hf_get_output_volume(int apd,  int32_t* volume);
```

Arguments:

apd

The handle to `io-acoustic`.

volume

Pointer to the handsfree acoustic processing volume.

Library:

`libacoustic`

Description:

The *ioap_hf_get_output_volume()* function retrieves the handsfree acoustic processing volume.

If your system is configured to allow the volume to be set through the API (*volumecontrol* is set to `true` in the *configuration file* (p. 32)), you can set the volume by calling *ioap_hf_set_output_volume()*. The valid range is from 0 (minimum) to 100 (maximum).

Returns:

`0`

Success: the volume setting.

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_go()

Start handsfree acoustic processing.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_go (int apd) ;
```

Arguments:

apd

The handle to `io-acoustic`.

Library:

`libacoustic`

Description:

The *ioap_hf_go()* function starts handsfree acoustic processing, initiating the start of audio transfer. It can only be used after acoustic processing has been prepared by calling *ioap_hf_prepare()*.

When it starts, *ioap_hf_go()* delivers the acoustic processing event `IOAP_HF_EVENT_STARTED`.

If acoustic processing is already running, calling this function has no effect.

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

`ENOTSUP`

The priming interval was too short to be able to start acoustic processing.

`EACCES`

Acoustic processing is stopped, or has not been prepared.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

ioap_hf_latency_estimate_t

Hold data returned by a latency test.

Synopsis:

```
#include <acoustic/hf.h>

typedef struct ioap_hf_latency_estimate {
    int32_t mslatency;
    int32_t msspread;
    int32_t crosscorr;
} ioap_hf_latency_estimate_t;
```

Library:

`libacoustic`

Description:

The structure `ioap_hf_latency_estimate_t` is used to hold latency estimates gathered by *ioap_hf_start_latency_test()* and made available by *ioap_hf_get_latency_estimate()*.

Member	Type	Description
mslatency	int32_t	The latency estimate, in milliseconds, calculated as the average of five separate latency measurements.
msspread	int32_t	The spread of latency measurements; that is, the spread from the minimum latency and the maximum latency recorded during five separate measurements.
crosscorr[]	int_32	The cross-correlation estimate between output and input signals. A value of 0 means no correlation was found; a value greater than 500 means a good correlation was found.

Classification:

QNX Neutrino

ioap_hf_latency_test_t

Configuration data to set up a latency test.

Synopsis:

```
#include <acoustic/hf.h>

typedef struct ioap_hf_latency_test {
    uint32_t msoffset;
    uint32_t channel;
} ioap_hf_latency_test_t;
```

Library:

```
libacoustic
```

Description:

The structure holds information to set up a latency test performed by *ioap_hf_start_latency_test()*.

Member	Type	Description
msoffset	uint32_t	The number of milliseconds to wait after each click before starting data collection for the latency measurements, performed during the next 100 ms, approximately. This offset is used to accommodate systems with longer (>100 ms) latencies.
channel	uint32_t	The microphone or reference input channel to use for the latency determination (`IOAP_MIC_IN_1`, `IOAP_MIC_IN_2`, etc.).

Classification:

QNX Neutrino

ioap_hf_mute()

Set or get the acoustic processing input or output mute status.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_mute(int adp, uint32_t direction, int32_t*
muteaction);
```

Arguments:

apd

The handle to `io-acoustic`.

direction

The direction of the acoustic processing, either `IOAP_INPUT` or `IOAP_OUTPUT`.

muteaction

A pointer to value specifying the mute action to perform. This action can be one of:

- `IOAP_ON`
- `IOAP_OFF`
- `IOAP_NOCHANGE` (Only retrieve the mute state.)

Library:

`libacoustic`

Description:

The *ioap_hf_mute()* function sets and gets the mute state for acoustic processing input and output.

Applying hardware mutes when you are using acoustic processing is not advised since the hardware will mute the microphone audio before it is input to the acoustic processing library. With no input coming from the microphone, the noise and signal estimates that the acoustic processing maintains will decay down to zero. When the mute is later released, acoustic processing requires time to rebuild these statistics, which it requires for correct operation in the current acoustic environment.

When you use the acoustic processing library to apply mutes, if acoustic processing isn't active, the mute action has no effect. However, the new setting is maintained until the next time that acoustic processing is started, and applied at that time.

Returns:

`>0`

Success

`-1`

An error occurred (errno is set).

Errors:

EINVAL

Invalid handle, direction, or null pointer argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_prepare()

Prepare the system for acoustic processing.

Synopsis:

```
#include <acoustic/hf.h>
int ioap_hf_prepare(int apd) ;
```

Arguments:

apd

The handle to `io-acoustic`.

Library:

`libacoustic`

Description:

The *ioap_hf_prepare()* function prepares the system for acoustic processing. You must call this function before you call *io_hf_go()* to start acoustic processing, including acoustic echo cancellation. It can only be called when acoustic processing is stopped.

This function:

1. Validates the acoustic processing routing against the current configuration specified by *ioap_hf_route()* and ensures that there are no inconsistencies.
2. Allocates the memory needed for audio transfer.

3. Opens the audio devices in `START_ON_GO` mode. Audio transfer isn't started until *ioap_hf_go()* is called.

When *ioap_hf_prepare()* finishes preparing the system, it delivers the acoustic processing event `IOAP_HF_EVENT_PREPARED`.

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

`EACCES`

Acoustic processing is not stopped.

`EINVAL`

Invalid handle or null pointer argument.

`EIO`

Setup error with input or output.

`ENODEV`

No memory available for data structures.

`ENOMEM`

No memory available for data structures.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_read_events()

Read handsfree acoustic processing events.

Synopsis:

```
#include <acoustic/hf.h>

ssize_t ioap_hf_read_events(int apd,
                            void *buf,
                            size_t buf_len);
```

Arguments:

apd

The handle to `io-acoustic`.

buf

Pointer to the buffer with the events.

buf_len

Length of the event buffer, in bytes.

Library:

`libacoustic`

Description:

The *ioap_hf_read_events()* function reads handsfree acoustic processing events. Before calling this function, you must register for events and intialize the queue by calling *ioap_hf_register_events()*.

Each event consists of an `ioap_event_t` data structure. This structure may be followed by additional data, then by a variable number of padding bytes to ensure 8-byte alignment. Declaring your event buffer type double will ensure that it is 8-bit aligned.

The *ioap_hf_read_events()* function never returns partial events; it returns only the events that can fit into the length of the buffer that is provided. If the first event does not fit into this buffer, the function returns an error and sets `E2BIG`.

If the event buffer holds more that one event, you should use *ioap_event_next()* to extract the next event, based on the previous event's address and contents.

See `IOAP_HF_EVENT_*` for information about the individual handsfree acoustic processing events.

The `io-acoustic` service won't hold an unlimited number of events. If it reaches its limit, it discards all new events until its events are read and its queue cleared.

Returns:

`0`

Success: the number of events read.

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

`E2BIG`

The buffer is too small for the event data.

`EIO`

The event queue has not been intitialized. Call *ioap_hf_register_events()* to initialize the queue.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_register_events()

Register to receive handsfree acoustic processing events.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_register_events(int apd, int events) ;
```

Arguments:

apd

The handle to `io-acoustic`.

events

Reserved. Set to `IOAP_HF_EVENT_ALL`.

Library:

`libacoustic`

Description:

The *ioap_hf_register_events()* function registers your process to receive handsfree events. After you have registered to receive events, you can call *ioap_hf_read_events()* to get events.

Returns:

`>0`

Success: ID of the opened device.

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

`ENOMEM`

No memory available for data structures.

`EBUSY`

ioap_hf_register_events() is being called twice.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

ioap_hf_route()

Set the routing between the hardware and the acoustic processing inputs and outputs.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_route(int apd,
                  ioap_io_map_t* map);
```

Arguments:

apd

The handle to `io-acoustic`.

map

Pointer to an `ioap_map_io_t` structure specifying the new routing.

Library:

`libacoustic`

Description:

The *ioap_hf_route()* function modifies the audio routing between the hardware and the acoustic processing inputs and outputs.

If the call to *ioap_hf_route()* fails, the routing is unchanged. If the call is successful, the system maintains the routing even after *ioap_hf_stop()* has been called; that is, the new routing will be kept until the next call to *ioap_hf_route()*, or a system restart.

Routing is defined in the structure `ioap_map_io_t`, and must respect the following rules:

- The number of devices must be greater than zero (0).
- For any device, routed channels must be either all input or all output.
- Channels used must be contiguous; you may not use channel 2 if you haven't used channel 1.
- Routes must be contiguous; you may not specify `IOAP_SPKR_OUT_2` if you haven't also specified `IOAP_SPKR_OUT_1`.

- Duplicate inputs or outputs aren't allowed.

For more information about the default routing, see Configuring `io-acoustic`. For information about the current routing configuration, call *ioap_hf_setup()*.

Returns:

0

Success

-1

An error occurred (errno is set).

Errors:

E2BIG

Too many input or output groups were specified, or there are too many channels in a hardware group.

EACCESS

Attempt to modify routing while data is being processed.

EINVAL

Invalid handle or null pointer argument.

EIO

The channel mapping is not contiguous, or a required channel is missing.

ENOMEM

No memory available for data structures.

EINOTSUP

Invalid I/O type.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_set_log_level()

Set the verbosity level for the handsfree acoustic processing logs.

Synopsis:

```
#include <acoustic/hf.h>
int ioap_hf_set_log_level(int apd, int* level) ;
```

Arguments:

apd

The handle to `io-acoustic`.

level

Pointer to the location where the log level is stored.

Library:

`libacoustic`

Description:

The *ioap_hf_set_log_level()* function sets the verbosity of the acoustic echo cancellation logs.

The default verbosity is `IOAP_LOG_ERROR`. You can change this verbosity by editing the configuration file, by using the `-v` option when starting `io-acoustic`, or by calling *ioap_hf_set_log_level()*.

You can check the current verbsoity level by calling *ioap_hf_get_log_level()*.

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument .

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_set_output_volume()

Set the output volume.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_set_output_volume(int apd,
                              int32_t* volume);
```

Arguments:

apd

The handle to `io-acoustic`.

volume

Pointer to the acoustic processing volume.

Library:

`libacoustic`

Description:

The *ioap_hf_set output_volume()* sets the handsfree acoustic processing volume. The valid range is 0 (minumum) to 100 (maximum). Values outside this range are clamped; for example, if you try to set the volume to 120, the system will set it at 100.

If acoustic processing is active, the volume change takes effect immediately. If acoustic processing isn't active, the change takes effect the next time acoustic processing is started.

To retrieve the volume setting, call *ioap_hf_get_output_volume()*.

To use this function, acoustic processing on your system must be configured to allow the volume to be set through the API (*volumecontrol* is set to `true` in the *configuration file* (p. 32)).

Returns:

`0`

Success: the volume setting.

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

`EIO`

Acoustic processing is configured for fixed volume control, so the volume can't be set through the API.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_setup()

Retrieve the current routing configuration.

Synopsis:

```
#include <acoustic/hf.h>
int ioap_hf_setup(int apd, int events) ;
```

Arguments:

apd

The handle to `io-acoustic`.

setup

Pointer to an `ioap_hf_setup_t` structure to load with with the current acoustic processing configuration.

Library:

`libacoustic`

Description:

The *ioap_hf_setup()* function retrieves the acoustic processing routing configuration.

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

`EINVAL`

Invalid handle or null pointer argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_setup_t

Define handsfree module routing setup.

Synopsis:

```
#include <acoustic/hf.h>

typedef struct ioap_hf_setup {
     ioap_io_map_t mapin;
     ioap_io_map_t mapout;
} ioap_hf_setup_t;
```

Library:

`libacoustic`

Description:

The structure contains maps to define the handsfree module's routing. For more information, see *ioap_hf_setup()*.

Member	Type	Description
mapin	`ioap_io_map_t`	Map for the input device.
mapout	`ioap_io_map_t`	Map for the output device.

Classification:

QNX Neutrino

ioap_hf_start_latency_test()

Start a latency test.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_start_latency_test(int apd, const
ioap_hf_latency_test_t* test) ;
```

Arguments:

apd

The handle returned by *ioap_hf_attach()* when it attaches to a process to `io-acoustic`.

test

Pointer to the latency test configuration structure.

Library:

`libacoustic`

Description:

The *ioap_hf_start_latency_test()* function starts a latency test. A latency test injects a series of nine clicks into the receive output and gathers statistics from the signal captured on the specified output.

To retrieve the results of a latency test, call *ioap_hf_get_latency_estimate()* and look at the values in the `ioap_hf_latency_estimate_t` data structure.

The results of a latency test can be used to configure the time (in milliseconds) to prime speaker or phone output on "go" in order to compensate for system latency. This time period is platform specific, and should be adjusted to the shortest time possible (that is, the shortest time period that is sufficient to prevent underruns).

A latency test determines the latency for each of nine clicks that are played back. The two smallest and two largest latency values are discarded. The remaining five latency values are used to calculate the three values returned by the test:

- latency estimate
- latency spread
- cross-correlation

A successful latency test should return a latency-spread of < 4 milliseconds and a cross-correlation of > 500. If the latency-spread is greater than 4 milliseconds or the cross-correlation is less than 500, the signal-to-noise ratio of the clicks should be increased, for example by increasing the playback volume or decreasing the ambient noise level. If the system latency is > 100 milliseconds, you should increase the offset specified when starting the test in order to move the estimate calculation into the correct measurement window.

The *msrefdelta_default* key in the `.conf` configuration file should be set to the difference between the *msprime* and the returned latency estimate. See "*Configuration keys* (p. 32)".

Returns:

`0`

Success

`-1`

An error occurred (errno is set).

Errors:

EINVAL

Invalid handle, null pointer argument or test argument.

EACCESS

Acoustic processing hasn't been started.

EBUSY

Test already in progress.

EFAULT

An error occurred during the test.

ENOTSUP

Diagnostics processing hasn't been enabled.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_hf_stop()

Stop acoustic processing.

Synopsis:

```
#include <acoustic/hf.h>

int ioap_hf_stop(int apd, int events) ;
```

Arguments:

apd

The handle to `io-acoustic`.

Library:

libacoustic

Description:

The *ioap_hf_stop()* function stops acoustic processing and delivers the event IOAP_HF_EVENT_STOPPED when it has completed.

Calling *ioap_hf_stop()* when acoustic processing is not running will have no effect.

Returns:

>0

Success

-1

An error occurred (errno is set).

Errors:

EINVAL

Invalid handle or null pointer argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

ioap_io_map_t

Define handsfree module routing setup.

Synopsis:

```
#include <acoustic/acoustic.h>

typedef struct ioap_io_map {
     ioap_io_map_t mapin;
     ioap_io_map_t mapout;
     } ioap_io_map_t;
```

Library:

`libacoustic`

Description:

The `ioap_io_map_t` structure specifies a list of hardware devices and the mapping of their inputs or outputs to the corresponding acoustic processing inputs or outputs. For more information about routing, see *ioap_hf_route()*.

Member	Type	Description
direction	`uint32_t`	Direction, either input (`IOAP_INPUT`) or output (`IOAP_OUTPUT`).
ndevices	`uint32_t`	The number of populated devices in *devicelist*.
devicelist [`IOAP_MAX_DEVICES`]	`ioap_device_t`	Array of devices. The number of populated devices, is specified by *ndevices*. See *ioap_hf_route()* for routing rules.

Classification:

QNX Neutrino

Chapter 6
Image Generation

The QNX CAR platform includes tools to assist you with building images for your target.

The tools included with the QNX CAR platform are in addition to the utilities available with the QNX Neutrino RTOS SDP. For information about the SDP utilities, see the QNX SPD *Utilities Reference*.

For information about building images, see:

- *Building and Customizing Target Images* in the QNX CAR documentation.
- *Building Embedded Systems* in the QNX SPD documentation.
- The *BSP User Guide* for your board.

gen-ifs

Perform setup activities and then run `mkifs` to include IFS files in an image.

Syntax:

```
gen-ifs  [-o] output [options]
```

Options:

`-c, --config`

Read the specified IFS configuration file.

`-d, --default-ifs`

Specify which IFS file is the default IFS (`qnx-ifs`).

`--defaults`

Include default directories in the search path.

`-f, --input`

Include the specified input file.

`-N--no-defaults`

Don't include default directories in the search path.

`-o, --output`

Write to the specified IFS file.

`--output-path`

Write IFS(s) to the specified location.

`-r, --root`

Include the specified directory as a root directory.

`-v, --verbose`

Increase verbosity.

Description:

The `gen-ifs` utility sets up the ***`MKIFS_PATH`*** for the specified board type, locates and concatenates the specified buildfiles, and then runs `mkifs` to include the `.ifs` files in an image.

The `gen-ifs` utility calls the `mkifs` utility to create the `.ifs` file(s) that are included in the final target image. An IFS is a bootable image filesystem that contains the `procnto` module, your boot script, and possibly other components such as drivers and shared objects.

If you generate only a single `.ifs` file, then you need to specify just the `-o` and `-f` options. However, if you want to generate multiple `.ifs` files, then you must specify at least the `-c`, `-d`, and `--output-path` options.

The following example uses the `omap5uevm` platform and creates an `ifs` file. The information in the resulting file is used by the image-generation script *`mksysimage.py`* (p. 84).

Examples:

The following example creates a single `ifs` file as output:

```
%QNX_CAR_DEPLOYMENT%/deployment/scripts/gen-ifs -o omap5uevm.external
```

Exit status:

0

The setup activities and inclusion of the IFS image files completed successfully.

>0

An error occurred.

Caveats:

None.

gen-osversion

Generate the `/etc/os.version` file based on build environments.

Syntax:

```
gen-osversion [options] ... <platform>.<variant>
```

Options:

`-q, --quiet`

Prevent output.

`-p, --additionalParameters`

Include additional parameters <*parameter*>=<*value*>.

`-v, --verbose`

Increase verbosity.

Description:

You can use the `gen-osversion` utility on its own or as part of the `mksysimage.py` utility script to generate the `os.version` file based on build environments.

To run `gen-osversion`, you need to specify the platform and its variant.

Examples:

The following example uses the `omap5uevm` platform and creates an `os.version` file. The information in the resulting file is used by the image-generation script `mksysimage.py`.

```
%QNX_CAR_DEPLOYMENT%/deployment/scripts/gen-osversion omap5uevm.external
```

The resulting output might look something like this:

```
date:  Thu Aug 30 13:23:48 2013
project: Local Build
buildHost: ubuntu
buildID: Local Build
buildNum: Local Build

platform: omap5uevm.external
MyBranch: trunk
MyRev: 6998
externalBranch: main
externalRev: 2732
```

Exit status:

0

The OS version file was generated successfully.

>0

An error occurred.

Caveats:

None.

mkimage.py

Generate an image from existing `.tar` files.

Syntax:

```
mkimage  [-o] outputpath ... [options]
```

Options:

`-c`, `--config`

Use the specified configuration file. See "Configuration file for `mkimage.py`" in the *Building and Customizing Target Images* guide.

`-o`, `--output`

Write to the specified image file.

`-t`, `--tar_path`

Specify the path to the `.tar` files.

`--tars`

Specify a list of `.tar` files.

`-v`, `--verbose`

Specify the verbosity up to a maximum of 4.

Description:

The `mkimage.py` script creates an image from the `.tar` files generated by `mktar`. This script extracts from the `.tar` file, creates the buildfiles, creates temporary `.image` files for each partition, and then creates the diskimage config file to create the final disk image. The bootable image file(s) (`.image`) are placed in the specified output file, followed by the embedded filesystem files (or any other type of image). With the exception of bootable images, all image files are padded up to the block size specified on the command line. If you don't specify a blocksize, the default is 64K.

Examples:

To run `mkimage.py`, you must specify the `-o` option and run the script from the location where the `.tar` files are located.

The following example shows have to generate an image:

```
%QNX_TARGET%/scripts/mkimage.py -o /tmp/
```

Exit status:

0

An image file was generated successfully.

>0

An error occurred.

Caveats:

None.

mksysimage.py

Create a QNX CAR image and generate other supporting files, such as `.ifs` and `.tar`.

Syntax:

```
mksysimage  [-o output] [options]... [board_name.external]
```

Options:

`-c, --mksysimage-config-file`

Specify the configuration file used for the `mksysimage.py` utility.

`-f, --force`

Force the overwriting of existing `tar` files.

`-g, --osversion-content`

Specify any additional content for the `os.version` file.

`-G, --no-gen, --no-generation`

Run only `mktar` and imaging components.

`--gen-ifs-options`

Specify the options for `gen-ifs.py` (see `gen-ifs.py --help`).

`--image-config-path`

Specify the path of the configuration files for the `mkimage` utility.

`-k, --mkimage-options`

Specify the options for `mkimage.py` options (see `mkimage.py --help`).

`-m, --mktar-options`

Specify the options for `mktar.py` (see `mktar.py --help`).

`--no-gen-ifs`

Don't generate IFS files.

`--no-mkimage`

Don't generate the `mkimage` part of the process.

`--no-mktar`

Don't generate the `mktar` part of the process.

`--no-gen-osversion`

Don't generate an `os.version`.

`-o, --output-path`

Write image and `tar` files to the specified path. If a `-t` option is specified, `tar` files are written to the path specified.

`-q, --quiet`

Prevent any output.

`-t, --tar-file-path`

Read and write `tar` files to and from the specified path.

`-v, --verbose`

Increase the verbosity.

If you specify both the `-p` and `-m` options, any intermediate directories you have created have mode `u+wx`.

Description:

The `mksysimage.py` utility is a Python script that invokes other utilities to generate `tar` files and images for each platform. By default, `mksysimage.py` reads a configuration file from ***%QNX_TARGET%***`/<platform>/sd-boot/config/<platform>-mksysimage.cfg`. This configuration file defines the `.tar` files and images created during the image-generation process. The image variants for each platform are defined within the configuration file. By default, for each image variant, `mksysimage.py` generates two tar files and one image. The `tar` file <*platform*>-os.tar contains two QNX CAR2 filesystems that include all files except MLO and IFS files. The `tar` file <*platform*>-dos-<*image_variant*> contains a FAT16 filesystem that includes all bootup files, such as MLO and IFS files. The final generated image includes these two `tar` files.

You can change the default configuration file associated with `mksysimage.py` (where the default file is located at ***%QNX_CAR_DEPLOYMENT%***`/<platform>/sd-boot/config/platform-mksysimage.cgf`), or specify your own by using the `-c` option in `mksysimage.py`. Setting this option will enable you to further customize your `tar` files and images. For more information about changing the configuration file for `mksysimage.py`, see "Configuration file for `mksysimage.py`" in *Building and Customizing Target Images*. For information about calculating the size of images and partitions, see "Calculate the size of an

image/partition" in in *Building and Customizing Target Images*. If you want `mksysimage.py` to generate only certain file types, use the following options:

To generate only:	Use the following options on the command line for `mksysimage.py`:
IFS files	`mksysimage -o outputpath <board>.external --no-mkimage --no-mktar --no-gen-osversion`
TAR files	`mksysimage -o outputpath <board>.external --no-mktar`
`mkimage`	`mksysimage -o outputpath <board>.external --no-gen-ifs --no-gen-mktar --nogen-osversion`

Examples:

To run `mksysimage.py`, you need to specify the platform, its variant, and the output path.

The following example reads the default configuration file for the `omap5uevm` platform and creates three images and their corresponding `tar` files in the specified output path called `/tmp`.

```
%QNX_TARGET%/scripts/mksysimage.py -o /tmp/ omap5uevm.external
```

Exit status:

0

The specified image file was created successfully.

>0

An error occurred.

Caveats:

None.

mktar

Create a `tar` file containing a filesystem for a specified board variant

Syntax:

```
mktar  [-o output] [options] ... [board_name_argument]
```

Options:

`--bars`

Include new-style (BAR) applications (implies `--no-defaults`)

`--compress`

Use the given compression method: `auto (default)`, `none`, `gzip`, or `bzip2`.

`--cpu`

Set the system architecture (e.g., `\"armle-v7\"`).

`-f`, `--fileset`

Include the specified fileset.

`--help`

Display a help message showing `mktar` usage information.

`--no-defaults`

Don't include the board's default filesets/applications.

`-o` *output*

Write to the specified output file.

`-p`, `--package`

Include the given package (implies `--no-defaults`).

`--prefix`

Prefix each path with the given string.

`--profile`

Specify the profile `xml` file (default: `\"profile.xml\"`).

`-s`, `--symbols`

Search in the `runtime-symbols/` folder for the unstripped binaries.

`-v, --verbose`

Increase verbosity.

`-z`

Compress with `gzip`.

If you specify both the `-p` and `-m` options, any intermediate directories you have created have mode `u+wx`.

Description:

The `mktar` utility creates a `tar` file containing the filesets listed in the board-specific profile, resulting in proper ownership and permissions. Don't use `mktar` on its own to create a `tar` file. Instead, use the `mksysimage.py` Python script, with the `--no-mkimage` option to create `.tar` files and no image.

If you use `mktar` without specifying the profile option `--profile`, then by default, the `mktar` utility will use the file `profile.xml` located in `/boards/`*`board`*`.external`.

The `mktar` utility creates a `tar` file containing the filesystem for a specified variant, typically used by `mksysimage.py` to include in the resulting image that's generated. It currently requires binary content from your installed software as well as source and deployment files.

Before you can use this utility, you must have the following prerequisites:

- Linux — Ubuntu Host or VM from *`http://www.ubuntu.com/download`*
- QNX CAR Platform for Infotainment 2.1 host development environment from *`http://www.qnx.com/download`*
- Java v1.6.0_35 — use `sudo apt-get install openjdk-6-jre`
- SVN external repository
- Hardware-specific firmware — `mktar` will return the following warning if the firmware files are not present for `imx61sabre`:
  ```
  skipped missing item: lib/firmware/vpu/vpu_fw_imx6q.bin
  ```

The `mktar` uses Python's `tarfile` module to generate a `tar` file with the appropriate permissions. The list of included files is determined by a board profile, which is stored as `profile.xml` in the `board` directory (e.g., ***`%QNX_CAR_DEPLOYMENT%`***`/boards/omap5uevm.car2/profile.xml`). For more information and for details about dependencies, see the file located at ***`%QNX_TARGET%`***`/pymodules/qnxcar/config.py`.

You can overwrite the default profile by using the `--profile` option.

The `mktar` utility with the `-vvv` option will show the search path used to find files. The typical search path (implemented in *deployment*`/pymodules/qnxcar/path.py:get_stage_locator`) is as follows:

- ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname.variant`*`/cpudir/`
- ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname.variant`*`/cpudir/`
- ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname.variant/`*
- ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname`*`/cpudir/`
- ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname/`*
- ***%QNX_CAR_DEPLOYMENT%***`/cpudir/`
- ***%QNX_CAR_DEPLOYMENT%/***
- ***%QNX_CAR_DEPLOYMENT%***`/runtime-external/cpudir/`
- ***%QNX_CAR_DEPLOYMENT%***`/runtime-external/`
- `%QNX_TARGET%/cpudir/`
- `%QNX_TARGET%/`

Overriding files to create custom `mktar` images

The `mktar` utility will search the search paths identified above to find all files defined in a particular board's `profile.xml` file. To include a new version of any file (e.g., a driver, configuration file, etc.), you would copy the file into the appropriate search path, and then rerun `mktar`.

To choose the appropriate search path, you need to determine the following:

- Does this change affect all boards? If the answer is **yes**, put the file in ***%QNX_CAR_DEPLOYMENT%***`/cpudir/` or *`car2-target/`*.
- Does this change affect only a specific board(s)? If the answer is **yes**, put the file in ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname`*`/cpudir/` or ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname/`* **for all affected boards**.
- Does this change affect only a specific variant of a given board(s)? If the answer is **yes**, put the file in ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname.variant`*`/cpudir/` or ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`boardname.variant/`* **for all affected board/variants**. For example, if you wanted to swap in an updated `devi-hid` driver that applies to all `armle-v7`-based boards you're working with, you'd copy it to ***%QNX_CAR_DEPLOYMENT%***`/armle-v7/usr/bin/devi-hid`. However, if you had a new audio driver for some target board (*target_name*) you would include the file in ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`target_name`* `/armle-v7/lib/dll/deva-ctrl-omap4pdm.so`.

If you're attempting to update, for example, ***%QNX_CAR_DEPLOYMENT%***`/armle-v7/usr/bin/devi-hid`, but the

same file is currently overridden for the board you're interested in (e.g. ***%QNX_CAR_DEPLOYMENT%***`/boards/`*`target_name`*`/armle-v7/usr/bin/devi-hid`), you'd have to remove the overridden file for your new copy in ***%QNX_CAR_DEPLOYMENT%***`/armle-v7/usr/bin/devi-hid` to be picked up. This is because the utility searches the paths in a certain order for any given file defined in `profile.xml`.

Examples:

To create a `tar` file for OMAP5432:

```
%QNX_TARGET%/scripts/mktar -o omap5uevm_external.tar
    omap5uevm.external
```

This command creates a file named `omap5uevm_external.tar` that contains the QNX CAR filesystem for the OMAP5432 board.

To create a `tar` file for SABRE Lite:

```
%QNX_TARGET%/scripts/mktar -o imx61sabre_external.tar
    imx61sabre.external
```

This command creates a file named `imx61sabre_external.tar` that contains the QNX CAR filesystem for SABRE Lite. To list all available *platform.variant* pairs, you can run:

```
ls -d %QNX_CAR_DEPLOYMENT%/boards/*.*
```

The output is compressed if the filename in the `mktar` command ends with `.gz` or `.bz2`. The utility can also be used without a board configuration, by manually specifying filesets (and a CPU type if applicable). You can also add extra filesets to a board's default configuration.

Exit status:

0

A `tar` file was generated successfully.

>0

An error occurred.

Caveats:

None.

Chapter 7
Keyboard

The keyboard service works with the keyboard provided by the HMI to display and manage the on-screen keyboard, or with a physical keyboard to enable input from that keyboard.

Overview

The keyboard service (*keyboard-imf* (p. 93)) lets applications communicate with the on-screen keyboard through PPS objects. It allows them to:

- show and hide the keyboard
- know the keyboard height, in pixels, so they can, if necessary, adjust their displays to fit into the remaining available screen area
- accept text entries and know how many characters have been entered

Interaction of HMI, keyboards, and applications

The diagram below shows how `keyboard-imf` interacts with the HMI virtual keyboards:

- The HMI virtual keyboards (HTML inside the Navigator, or Qt standalone) create the `/pps/system/keyboard/control` and `/pps/system/keyboard/status` PPS objects, and publish their activities to them.
- The `keyboard-imf` service subscribes to these objects to be able to receive information from the HMI.
- The `keyboard-imf` service creates the `/pps/services/input/control/` PPS object, to which it publishes information received from the HMI and the applications. This object is for internal communication between `keyboard-imf` and the HMI; other components and applications don't need to publish or subscribe to this object.
- Applications, such as Weblauncher and QT runtime, subscribe to `/pps/services/input/control/` to learn about keyboard presence, height, etc., and publish information such as the user input and the number of characters entered.

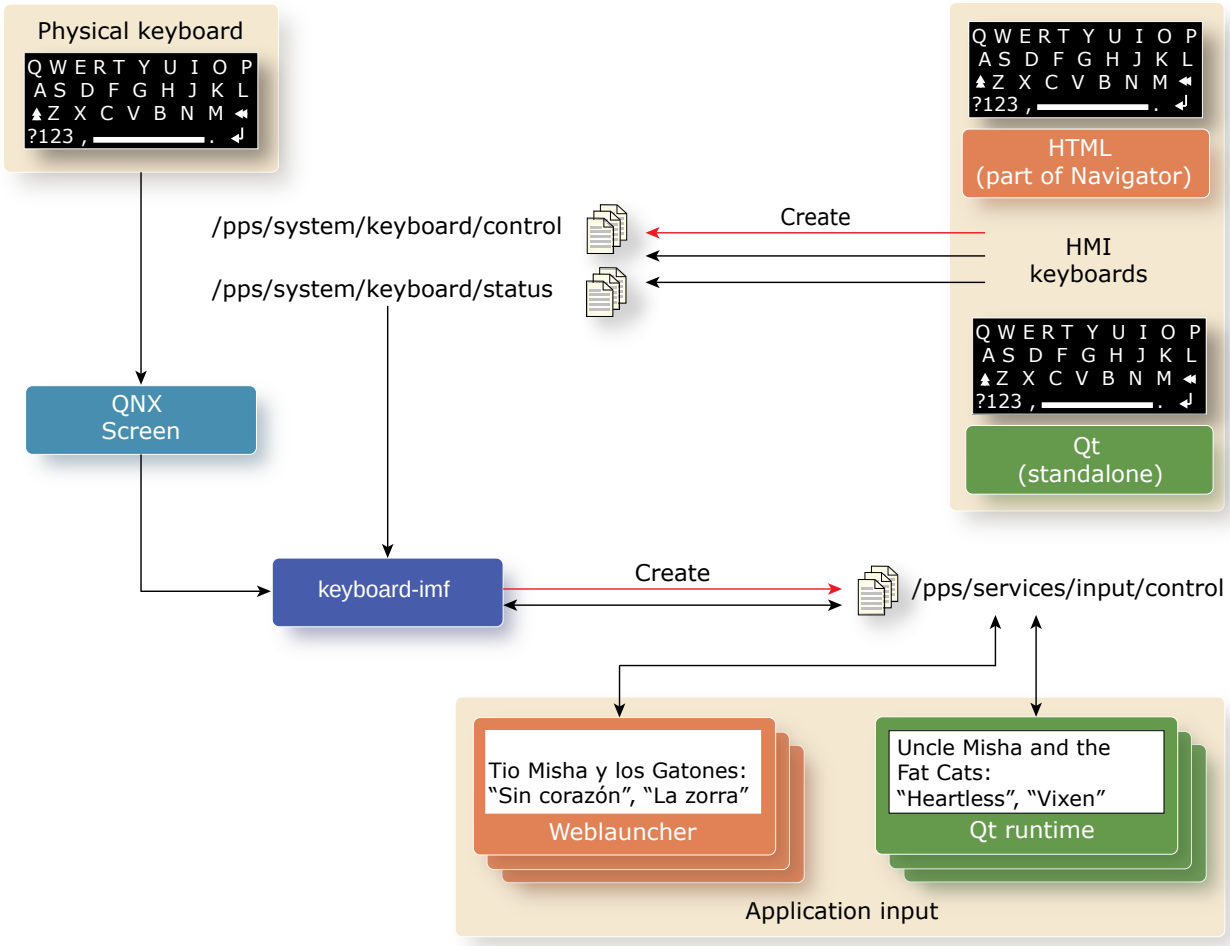


Figure 7: Keyboards, `keyboard-imf` and applications.

Physical keyboard

To use a physical keyboard (connected through a USB port), you need to:

- Configure the QNX Screen `globals` *input* parameter in the Screen configuration file (`graphics.conf`) to accept input from a physical keyboard. For more information, see "Configuration parameters for `globals`" in the *Screen Graphics Subsystem Developer's Guide*.
- Make sure that your system has the language-specific key mapping files for the languages you will support. These files should be in the `/usr/share/keyboard/` directory.

Keyboard (keyboard-imf)

Display and manage the on-screen keyboard

Syntax:

```
keyboard-imf [-d device] [-U group:user]
```

Options:

-d

The display required for the board. See "*Display types* (p. 93)" below.

-U *UID*:*GID*

The user ID and the group ID under which to run keyboard-imf.

Description:

The keyboard-imf service acts as an intermediary between applications requiring keyboard functionality and underlying keyboard services.

For more information about how keyboard-imf interacts with applications and underlying keyboard services, see "*Keyboard* (p. 91)".

Display types

The -d must be set for your board's display. For OMAP5432 boards, set it to hdmi; for SABRE Lite boards, set it to internal. Possible display types are listed in the etc/graphics.conf configuration file. When you are configuring your system, you will need to edit this file and enter the display type(s) supported on your board, as well as other graphics configuration values.

For more information about the etc/graphics.conf configuration file and how to configure it, see the chapter "Screen Configuration" in the *Screen Graphics Subsystem Developer's Guide*.

PPS objects

Applications must subscribe to this PPS object to communicate with the keyboard.

The HMI keyboard service creates these PPS objects to communicate with keyboard-imf:

- /pps/system/keyboard/control
- /pps/system/keyboard/status

In addition, the keyboard-imf service creates this PPS object:

- `/pps/services/input/control/`. This object is for internal communication between `keyboard-imf` and the HMI; other components and applications don't need to publish or subscribe to this object.

Chapter 8
MirrorLink

When a MirrorLink device is connected, MirrorLink apps are available to launch.

Overview

The QNX CAR platform supports the *MirrorLink* technology standard (version 1.1) to enable MirrorLink apps on a smartphone to work with the car's HMI.

This document covers the following topics:

- *Devices supported* (p. 95)
- *Network sandbox* (p. 95)
- *PPS interface* (p. 96)
- *Licensing* (p. 96)
- The MirrorLink services (`mlink-daemon`, `mlink-rtp`, and `mlink-viewer`)

Devices supported

Although any MirrorLink Certified *server device* (phone) should work, we have tested the following phones with the QNX CAR platform 2.1:

- Samsung Galaxy S III (with DriveLink app)
- Nokia 701 and E7 (with Nokia's Car Mode and MirrorLink app installed—note that the free non-MirrorLink version of Car Mode will *not* work.)

For the current list of MirrorLink Certified server devices, see the following page at the Car Connectivity Consortium (CCC) site (using the **Servers** search filter):

MirrorLink™ Certified Product Listing

You need to have the appropriate MirrorLink app for the device installed. Once you plug in the phone, the MirrorLink apps should appear in the HMI's apps section (under **ALL**). On some phones you must unlock the phone and start the MirrorLink app manually before connecting the phone to the head unit.

Network sandbox

To access a device, MirrorLink services need the `devnp-ncm.so` driver to be loaded into an `io-pkt` network stack. You should use a separate network stack for MirrorLink—by convention, this stack should use the `/mirrorlink_sandbox` prefix.

It's important to set an instance number for this network stack to prevent `mount` commands from loading drivers into this stack.

To use SLM to start the network sandbox, add this component section to the SLM configuration file:

```
<SLM:component name="mirrorlink-sandbox">
     <SLM:command>io-pkt-v6-hc</SLM:command>
     <SLM:args> -i1 -d ncm pnp -ptcpip prefix=/mirrorlink_sandbox</SLM:args>
     <SLM:stop stop="signal">SIGTERM</SLM:stop>
     <SLM:waitfor wait="pathname">/mirrorlink_sandbox/dev/socket</SLM:waitfor>
     <SLM:depend>usb</SLM:depend>
     <SLM:depend>pps-setup</SLM:depend>
</SLM:component>
```

To debug the network sandbox, you can use `ifconfig`, `dhcp.client`, and other network utilities by setting the ***SOCK*** environment variable to the sandbox's prefix:

```
SOCK=/mirrorlink_sandbox/ ifconfig
```

PPS interface

The `mlink-daemon` service uses these PPS objects:

This PPS object:	Contains:
`/pps/services/mirrorlink/applications`	Information about the VNC-typed MirrorLink apps that are currently available.
`/pps/services/mirrorlink/entities`	Information about the MirrorLink entities (devices) that are currently available.
`/pps/services/mirrorlink/rtp`	RTP audio-streaming information for MirrorLink apps.
`/pps/system/navigator/applications/applications`	The list of apps installed on the system.

Licensing

The three MirrorLink services use RealVNC licensing as follows:

`mlink-daemon`

Although this service doesn't need the license file for discovering devices, it uses the file for audio (the Audio SDK has an implicit dependency on the Viewer SDK, which requires the license file).

`mlink-rtp`

Doesn't need a license file.

`mlink-viewer`

This service uses the Viewer SDK; it cannot work at all without the license file.

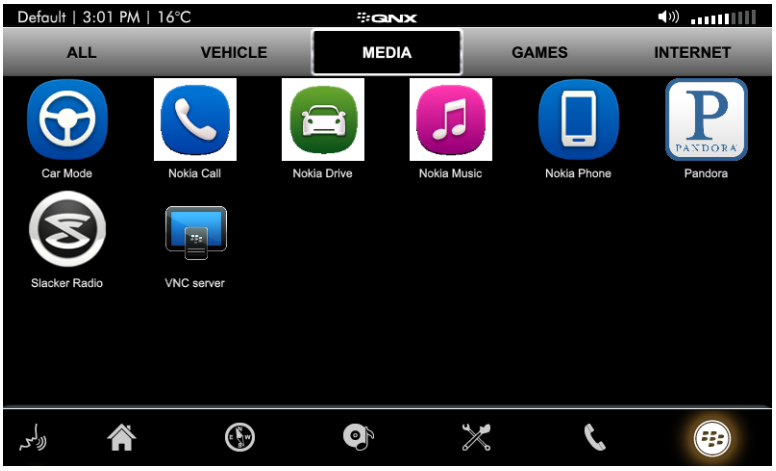
The license file resides under `/etc/vnclicense` by default, but you can use the `-L` command-line option (to both `mlink-daemon` and `mlink-viewer`) to change this location.

Although some features of these services will work without a license file, all of these services depend on the RealVNC SDKs and require proper licensing from RealVNC.

The `mlink-daemon` service—discoverer, launcher, and audiorouter

Overview

The `mlink-daemon` service uses the RealVNC Discovery SDK to detect new MirrorLink devices.



When a new device is detected, the device's MirrorLink applications are published to a PPS object (`/pps/system/navigator/applications/applications`). The service also creates a shortcut for the application (in the `/apps/` directory). Note that there's a limit of 10 MirrorLink applications.

The `mlink-daemon` service also negotiates RTP audio connections with the device and notifies `mlink-rtp` of these connections. Note that the `mlink-daemon` service doesn't need a RealVNC license file for discovering devices, but it uses the license for audio connections.

The service also has a native message-passing interface, which is used by `mlink-viewer` for requesting a MirrorLink application to be launched.

Command line

`mlink-daemon [-A` *`path_to_applications_object`*`] [-a` *`path_to_shortcut`*`] [-D] [-I lo0;en0...] [-i` *`path_to_icon.png`*`] [-L` *`path_to_vnclicense`*`] [-P` *`pps_dir`*`] [-S]`

Options

-A path_to_applications_object

Override location of the /pps/system/navigator/applications/applications PPS object.

-a ***path_to_shortcut***

Override installation path for shortcuts for MirrorLink apps (default: /apps/).

-D

Don't run in the background after successful initialization.

-I

Semicolon-separated list of interfaces to ignore (default: lo0).

-i ***path_to_icon.png***

Set default icon for MirrorLink apps. If multiple -i options are used, the service will use the last icon that was successfully loaded.

-L ***path_to_vnclicense***

Specify location of VNC license file (default: /etc/vnclicense).

-P ***pps_dir***

Base directory for PPS objects (default: /pps/mirrorlink/).

-S

Strictly enforce sandboxing, i.e., don't start with the default network stack. To set the network sandbox, use the ***SOCK*** environment variable. If the -S option is used, the service won't start if this environment variable isn't set. A network sandbox is *highly* recommended!

USB device enumeration

When mlink-daemon starts up, the RealVNC Discovery SDK listens for new devices. The SDK relies on PPS objects (under /pps/services/vnc/discovery/usb/) to detect new USB devices. The SDK also sends the MirrorLink NCM USB command to the device to start the MirrorLink server.

The following scripts are used for enumeration:

Script	Description
/etc/usblauncher/rules.lua	Rules for USB enumeration with whitelist for known devices.
/scripts/vncdiscovery/usb-device-attached.sh	This script is used by usblancher to create the PPS object (under

Script	Description
	`/pps/services/vnc/discovery/usb/`) for the USB device and to request an IP address from the NCM device. When the device is disconnected, the PPS object is removed. The script also contains the blacklist for known incompatible devices.

Using the whitelist isn't future proof—you'll need to manually add new MirrorLink devices to the whitelist. Remember to disable the whitelist and improve the blacklist for deployment.

Using SLM to start the service

To use SLM to start `mlink-daemon`, add this component section to the SLM configuration file:

```
<SLM:component name="mlink-daemon">
     <SLM:command>mlink-daemon</SLM:command>
     <SLM:args>-S -i /usr/mlink/default.png -L /usr/mlink/vnclicense</SLM:args>
     <SLM:envar>SOCK=/mirrorlink_sandbox/</SLM:envar>
     <SLM:stop stop="signal">SIGTERM</SLM:stop>
     <SLM:depend>mirrorlink-sandbox</SLM:depend>
</SLM:component>
```

The mlink-rtp service—RTP audio streaming

Command line

```
mlink-rtp [-D] [-P pps_dir] [-S]
```

Options

-D

Don't run in the background after successful initialization.

-P

Base directory for the rtp PPS object (default: /pps/mirrorlink/).

-S

Strictly enforce sandboxing, i.e., don't start with the default network stack. To set the network sandbox, use the ***SOCK*** environment variable. If the -S option is used, the service won't start if this environment variable isn't set. A network sandbox is *highly* recommended!

PPS object

The mlink-rtp service reads the rtp object for audio-streaming information published by the mlink-daemon service. For details, see the entry for /pps/services/mirrorlink/rtp in the *PPS Objects Reference*.

Using SLM to start the service

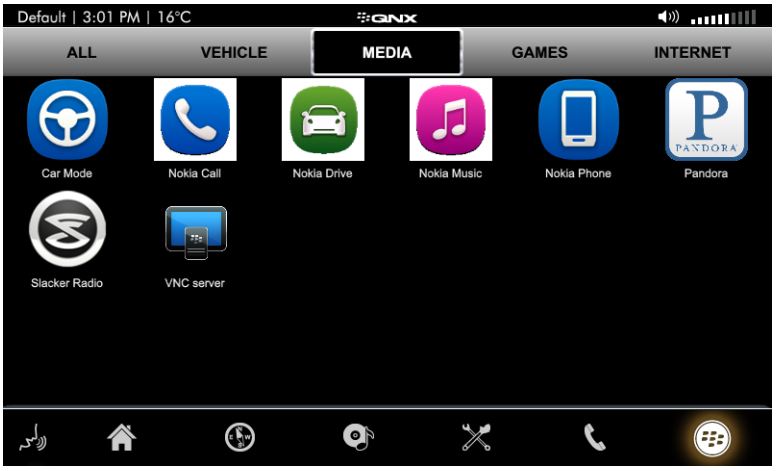
To use SLM to start mlink-rtp, add this component section to the SLM configuration file:

```
<SLM:component name="mlink-rtp">
     <SLM:command>mlink-rtp</SLM:command>
     <SLM:args>-S</SLM:args>
     <SLM:envar>SOCK=/mirrorlink_sandbox/</SLM:envar>
     <SLM:stop stop="signal">SIGTERM</SLM:stop>
     <SLM:depend>mirrorlink-sandbox</SLM:depend>
</SLM:component>
```

The `mlink-viewer` service—MirrorLink viewer app

Overview

The MirrorLink viewer app (`mlink-viewer`) lets the car's HMI view and control the content on a smartphone or other supported MirrorLink device.



Command line

`mlink-viewer [-D] [-H] [-h` *`number`*`] [-L` *`path_to_vnclicense`*`] [-l` *`app_number`*`] [-R] [-S] [-w` *`number`*`]`

Options

`-D`

Test mode: launch the application and show the command string, but don't start the viewer.

`-H`

Show simple HMI, which has to be an image with a width of 123 pixels. This will be visible on the right side of the screen. The top half of this region functions as the device's **home** button, the bottom half as a **back** button.

`-h`

Height (in pixels) of the main window (default: full screen; the actual size depends on the attached screen and system configuration).

`-L`

Specify location of VNC license file (default: `/etc/vnclicense`).

`-l`

Launch the specified MirrorLink app using its *app_number* (`0` to `9`; default: `0`). See `/pps/services/mirrorlink/applications` for more information.

`-R`

Reduced resolution mode. If the device supports server-side scaling, the requested resolution will be only half of the available resolution of the VNC window. On high-resolution or low-performance devices, this mode has significant performance advantages.

`-S`

Strictly enforce sandboxing, i.e., don't run the viewer with the default network stack. To set the network sandbox, use the ***SOCK*** environment variable. If the `-S` option is used, the service won't start if this environment variable isn't set. A network sandbox is *highly* recommended!

`-w`

Width (in pixels) of the main window (default: full screen; the actual size depends on the attached screen and system configuration).

The viewer doesn't quit when the device is disconnected.

Chapter 9
Navigation Engine

The QNX CAR platform is designed to support a variety of third-party navigation engines.

Overview

For this release, the QNX CAR platform includes the Elektrobit EB *street director* as the default navigation engine. The platform provides a reference navigation integration, demonstrating the "command and control" between the HMI and the participating components—navigation engine, window manager, PPS interface, and so on.

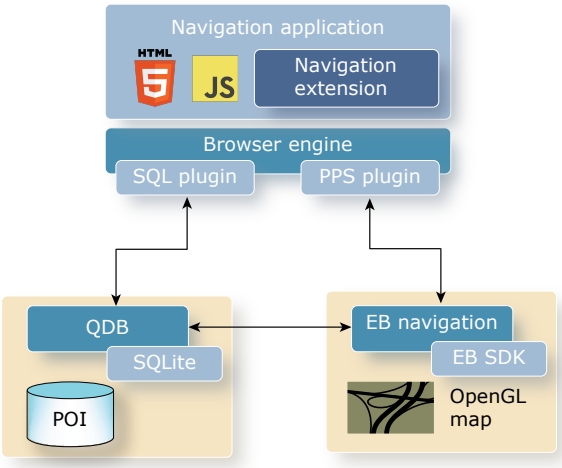


Figure 8: High-level view of navigation integration in the QNX CAR platform.

For more information about the QNX CAR navigation engine, see "Navigation" in the QNX CAR *Architecture Guide*. For more information about using the navigation engine HMI, see "Navigation" in the QNX CAR *User's Guide* chapter "A Guided Tour of the HMI".

Configuration

The QNX CAR platform checks the `/target/board.variant/var/etc/services-enabled` configuration file to know which services are enabled. To enable navigation on your system, set the `NAVIGATION` attribute in this file to `true`. For example:

```
APKRUNTIME:true
DLNA:false
WIFI:true
NAVIGATION:true
APPUPD:false
```

If you are working with a configuration that has had the navigation engine disabled, as well as configuring the `services-enabled` file you may need to change the

`tablist.json` file so that it displays a navigation tab correctly. To do this, you need to edit the `tablist.json` file:

1. Run the following command: `mount -uw /base` to remount the `/base` filesystem as read-write (instead of the default read-only).
2. Open the `/target/usr/hmi/common/js/tablist.json` file and edit the configuration information. For example, to make Tab 3 the navigation tab, your file should have an entry like this:

```
"tab3": {
"id": "navigation",
"type": "local",
"order": "3",
"name": "Navigation",
"opts": {
    "class": "nav"
    }
}
```

When you have finished your edits, don't forget to remount the `/base` filesystem so that it's read-only again (`mount -ur /base`).

Elektrobit street director

The current QNX CAR release comes with the Elektrobit (EB) street director.

The EB street director navigation engine is called `/apps/eb-navigation`. If you are debugging your implementation and need more detailed information in the logs, just start the navigation engine with greater verbosity. For example: `-vvvvvvvv`.

The configuration file for the EB street director is `/apps/eb-navigation/ebnav.conf`. For more information about this file and configuring the EB street director, please see your Elektrobit documentation.

PPS objects

The navigation engine uses the following PPS objects:

- `/pps/qnxcar/navigation/control`
- `/pps/qnxcar/navigation/geolocation`
- `/pps/qnxcar/navigation/options`
- `/pps/qnxcar/navigation/status`

Chapter 10
Network Manager (net_pps)

PPS interface to the network manager

Syntax:

```
net_pps [-A addr:port][-a]
[-c file] [-d] [if0...] [-m]
[-P script] [-p prefix] [-r name [if0...]]
[-S uid] [-s] [-u] &
```

Options:

-A *addr*:*port*

Proxy to publish when proxy authentication is required.

-a

Automatically configure any discovered interfaces when the link state indicates that the interfaces are connected.

-c *file*

Specify configuration file (default: `/etc/net_pps.conf`).

-d

Enable debug messages (to `stdout`).

if0...

Specify prioritized list of interfaces to be considered for multihomed operation, preference of default routes, `confstr` resolver configuration, etc.

-m

Use multipath routes.

-P *script*

Specify the script to run for updating proxy settings.

-p *prefix*

Add this prefix to all executable paths. This has no effect without the -s option.

-r *name* [if0...]

Create another routing domain (called *name*) with the following prioritized interface list.

-S *uid*

Run subprocesses as this *uid*.

-s

Use standard file paths for subprocess executables (defaults: system=/usr/sbin/ user=/usr/bin/).

-u

Automatically assume the interface is connected based on its *up* state. This allows shim drivers as well as drivers that don't issue link state changes to work.

Description:

The net_pps service offers a PPS interface for communicating with the QNX network manager. For more information about QNX support for networking, see "Networking Architecture" in the *System Architecture Guide*.

For more information about the instructions net_pps can send to the network manager, and the information it can receive, see the relevant PPS object descriptions:

- /pps/services/networking/all/interfaces/<interface>
- /pps/services/networking/all/proxy
- /pps/services/networking/all/status_public
- /pps/services/networking/control
- /pps/services/networking/proxy
- /pps/services/networking/status
- /pps/services/networking/status_public

Configuration

The net_pps service uses the /etc/net_pps.conf configuration files. To configure network manager behavior, edit the parameters in this plain-text file.

Chapter 11
Now Playing Service

The now playing service arbitrates between media players (including phones) and media controllers on the system, so that their activities do not clash, and publishes information about their activities. Applications can use this information to help them manage concurrent audio streams.

Overview

The now playing service arbitrates between the different media players (including phones) and media controllers on the system, and publishes information about media player activity. The audio manager is used to control audio stream routing, volume, and ducking behavior.

The now playing service (`nowplaying`) can be used along with the audio manager to manage audio stream concurrency and ensure that:

- the audio stream with the highest priority takes precedence and has access to the preferred output device(s)
- other audio streams are attenuated or muted, as required by the system configuration
- media playback is stopped or paused when an audio stream is "ducked" in favor of a higher priority audio stream, and restarted when appropriate

The now playing service supports two distinct PPS interfaces, one for media players (including phones), and one for media controllers. Each interface has its own *PPS objects* (p. 113).

Media players

All media players on a system should register with the now playing service and subscribe to the media player PPS objects in order to receive information about the other media players on the system. They should also publish information about their activities to these objects, so that the now playing service can know what they are doing and make this information available to the other media players. Media players can also use these objects to publish metadata about the media they are playing, and receive control commands, such as pause and resume.

For example, the phone application (which generally has precedence over other media players) should use the now playing service to inform other media players when a call comes in, so that they can pause playback during the call.

Media controllers

Media controllers on the system (such as a controller tied to the physical buttons on a phone) should subscribe to the media controller PPS object, so that they can receive information about the content (if any) that is currently playing on the system. They can also publish requests to this object when they need to control playback of the currently active media player in the system.

Using the now playing service

To use the now playing service, applications should register with the service, and publish and subscribe to the relevant PPS objects.

Managing audio

The audio manager looks after audio routing to output devices (and from input devices), and attenuating or muting output according to configured audio type priorities. When your application opens a PCM stream, it should also get an audio manager handle so that the audio manager will know about the stream. You should specify the same audio stream type for the audio manager handle as for the PCM stream handle to ensure that audio manager routes, and attenuates or mutes the audio stream as specificied by your system's configuration.

To get a PCM stream handle and an audio manager handle, call *audio_manager_snd_pcm_open()*, then *audio_manager_get_handle()*; or *audio_manager_snd_pcm_open_name()*.

Managing playback

The audio manager doesn't look after your application's media playback, however. If you open a PCM stream and get an audio manager handle for that stream but do not use the now playing service, when your application's audio is muted or attenuated (ducked), playback will continue. To ensure that the preferred behavior can be implemented, your application should:

1. Register with the now playing service, and publish its activities to the appropriate PPS objects:
   - `/pps/services/multimedia/mediacontroller/control`
   - `/pps/services/multimedia/mediaplayer/control`
   - `/pps/services/multimedia/mediaplayer/phone`.
2. Subscribe to `/pps/services/multimedia/mediaplayer/status` to receive the status updates that the now playing service publishes to this object.
3. When it learns that the status of its audio stream has changed, stop, pause, resume, or allow playback to continue, as required.

The now playing service *does not* control audio output or playback. It informs applications that have registered with the PPS objects it uses of changes in the status of audio streams. The decision of what to do when the status of an audio stream changes is the responsibility of your application.

For more information about PPS, see the *QNX Persistent Publish/Subscribe Developer's Guide*

Now playing service PPS objects

The now playing service uses PPS objects to communicate with applications.

The now playing service uses the PPS objects listed below. For information about these objects, see the relevant pages in the *PPS Objects Reference*.

- `/pps/services/multimedia/mediacontroller/control`
- `/pps/services/multimedia/mediaplayer/control`
- `/pps/services/multimedia/mediaplayer/phone`
- `/pps/services/multimedia/mediaplayer/status`

Now Playing Service (`nowplaying`)

Arbitrate between the different media players (including phones) and media controllers on the system so that they do not clash

Syntax:

```
nowplaying [-1] [-S] [-U UID:GIG] [-v]
```

Options:

`-1`

("one") Display only one volume dialog at a time.

`-S`

Write log to the `stderr` in addition to `sloginfo`.

`-U` *number*

The user ID and group ID which `nowplaying` should assume so that it doesn't have to continue running as root.

`-v`

Set verbosity of output to `slog2`.

Description:

The `nowplaying` service monitors media players, media controllers, and phones, and publishes PPS objects with information about their activities. Applications can use this information to help them manage concurrent audio streams.

See also:

- *Audio Manager Library Reference*
- `/pps/services/multimedia/mediacontroller/control`
- `/pps/services/multimedia/mediaplayer/control`
- `/pps/services/multimedia/mediaplayer/phone`
- `/pps/services/multimedia/mediaplayer/status`

Chapter 12
Radio

The QNX CAR platform includes a reference radio service, which runs on the Texas Instruments J5 ECO EVM811x EVM board.

Overview

The QNX CAR platform's reference radio service, `RadioApp`, is provided in collaboration with Texas Instruments and runs on the J5 ECO board. It interacts with the DSP on this board to offer tuning, band selection, scanning and RDS (Radio Data System). It uses the QNX PPS service to communicate with the HMI.

For information about starting up and running `RadioApp`, see *RadioApp* (p. 116).

If you create a custom image, you must use the `RadioApp`'s `-L` option to reserve memory for the service.

Connecting external components

To use the reference radio, you will need to connect an antenna, and headphones or speakers to the board. If you consider the top of the board to be the edge with the AM-FM switch:

- The antenna should be connected to the J1 AM-FM port.
- The headphones or speakers should be connected to the third jack from the bottom, on the left-hand side of the board.

PPS objects

The QNX CAR platform uses these PPS objects to communicate with the reference radio:

- `/pps/radio/ti_control`
- `/pps/radio/ti_rds`
- `/pps/radio/ti_statsus`

The platform also has the following PPS objects to exchange simulated radio information, which the QNX CAR HMI can use for its radio displays. These objects are not currently used by the radio service. They are:

- `/pps/radio/command`
- `/pps/radio/status`
- `/pps/radio/tuners`

RadioApp

Support a reference radio service on the TI Jacinto 5 board

Syntax:

```
RadioApp -p 0 -f path -h pps [-L address,size]
```

Options:

`-p 0`

TI command-line option; must be "0" (zero).

`-f path`

The path to the ELF configuration file.

`-h`

Instruct the service to use PPS. Must be "pps".

`-L address,size`

Reserve memory for the service. Use only if creating a custom image. If used, values must be "0x96C00000,0x8D00000". The first number is the address and the second is the size of the memory allocation.

Description:

The `RadioApp` service is provided by Texas Instruments. Note the following about using this service:

- `RadioApp` uses the ELF configuration file `jive_dsp_app_elf.out`. This is a binary file, and it should not be altered. Use the `-f` at startup to indicate the location of this file.
- The following processes must be running before starting `RadioApp`: `mq` and `syslink_drv`.
- If you create a custom image, you must use the `-L` option to reserve memory for the service.

For more information about the QNX CAR reference radio implementation, see *Radio* (p. 115).

Chapter 13
Realtime Clock Synchronization

To ensure time-dependent applications run successfully, the system clock synchronizes with a reliable, Internet-based time source when the system is booted.

You must have a working Internet connection for the system clock to be properly set.

Applications such as The Weather Network and Pandora use SSL to securely access websites they need for their operations. When the system clock deviates from the actual (correct) time, those applications can fail because they can't validate a website's certificate. Changing the time “on-the-fly” can cause key platform components, notably the JavaScript core, to fail. As a result, this operation isn't allowed once the HMI has loaded, so the system clock must be set during startup, before the HMI loads.

On systems without battery-powered backup for their realtime clock (RTC)—such as the Freescale i.MX6Q SABRE Lite board—the clock setting is lost when the hardware is powered off. To address this problem, the system startup script sets the system clock to the RTC's setting. If the date and time are *earlier* than a threshold time value that's hardcoded for the QNX CAR system, the system assumes the RTC isn't set properly and then tries to synchronize both the RTC and the system clock with an NTP server (which requires an Internet connection).

On hardware that does have battery backup, the RTC is typically set even before the OS image is installed. In this case, the date and time will be later than the threshold time value, so the startup script won't try to synchronize the RTC and system clock.

For more information about the QNX realtime clock, see `rtc` in the QNX Software Development Platform *Utilities Reference*.

Chapter 14
Shutdown service (coreServices2)

Provide access through PPS communication to a variety of services, including shutdown.

Syntax:

```
coreServices2 [-r path] [-U UID:GIG] [-S UID:GIG] [-M UID:GIG]
[-l none|module[,module]*] [-v]* [-d] [-C
    configuration file]
```

Options:

-r ***path***

Specify the root path to the PPS service. Default is `/pps/services`.

-U

The username or the *UID:GID,GID* specifying the user and group IDs of the main server process.

-S

The username or the *UID:GID,GID* specifying the user and group IDs of the spawner process.

-M

The username or the *UID:GID,GID* specifying the user and group IDs of the monitor process.

-l ***none|module[,module]****

If specified, use this list of dynamic modules instead of the dynamic modules listed in the configuration file.

-d

Run in foreground instead of as a daemon (default).

-v

Set verbosity of output to `sloginfo`.

-C ***filename***

The filepath and filename of the configuration file.

Description:

The coreServices2 utility provides a single point from which to ask the system to run a variety of services. It handles the house-keeping, while communication is through PPS. This design means that the requesting component or application only needs to publish and subscribe to the relevant PPS objects.

The QNX CAR platform uses coreServices2 to provide access from the HMI to the shutdown service.

coreServices2 objects

The coreServices2 service maintains an object for every service to which it gives access. Each object represents a single service. An object may be static (compiled into the coreServices binary) or dynamic (loaded through *dlopen()* at runtime).

Most basic services are static, but some services that are large (or that require large shared libraries) and are not needed by all implementations are made available as dynamic modules.

In the QNX CAR platform, coreServices2 maintains the following statically loaded object:

shutdown

Shut down and reboot the system. See shutdown.

In the QNX CAR platform, coreServices2 doesn't use any dynamically loaded modules.

PPS objects

The coreServices2 service publishes or subscribes to the following PPS object:

- /pps/qnxcar/system/info

Configuration file

A configuration file specifies the core services that will be used. The name and location of the file is specified by the -c. In the QNX CAR platform, the configuration file is located at /etc/system/config/coreServices2.json.

The file is a plain-text file built using the following syntax:

```
{
"static_modules" : comma separated string of static module names
"dynamic_modules" : comma separated string of dynamic module names
"disable_procmon" :  Boolean: true|false
"disable_hwid" : Boolean: true|false
}
```

For example, the configuration file at the time of writing includes only the shutdown service, which is a static module:

```
{
"static_modules" : "shutdown",
"dynamic_modules" : "",
"disable_procmon" : true,
"disable_hwid" : true
}
```

Chapter 15
Software Updates

The QNX CAR platform supports software updates by using Red Bend's vRapid Mobile® FOTA software (v8.0.1.29). The reference HMI includes functionality for applying software updates but the platform provides a library and resource manager that you can use to develop your own update application.

As explained in the *User's Guide*, you can apply updates from the HMI or from the command line. To apply an update, you'll need a *delta file*, which describes the filesystem changes needed to upgrade your current system to a new version. You can obtain a delta file from your system provider or *generate your own delta file* (p. 241). If you update your system through the HMI, your update package must also include a *manifest file*, which you must write. For details, see "*Manifest file* (p. 139)" in this guide.

You can also write your own software update application based on the software update library (`swu-core`) and resource manager (`swud`) shipped with the platform. The library has an extensive API for defining updates (to represent software changes from a base version to a target version) and update targets (to represent systems with updatable software), and for assigning callback functions that carry out update tasks such as installation and verification.

Software update core library

The software update core library (`swu-core`) applies software updates to target systems. The library manages the update process by coordinating with modules that perform specific tasks, such as discovering manifest files, displaying pending updates in the HMI, and initiating the installation of updates onto target systems.

The core library runs as part of the software update daemon (*swud* (p. 235)), which is launched automatically during startup. You can instruct the daemon to load all the modules needed by the library to support software updates.

Architecture of swu-core library

The `swu-core` library maintains the data structures containing update information and the state of the update process, but you must write your own modules to specify how to carry out individual update tasks. This design lets you customize all the steps involved in applying software updates.

The following diagram shows the `swu-core` library and the different modules involved in the software update process:

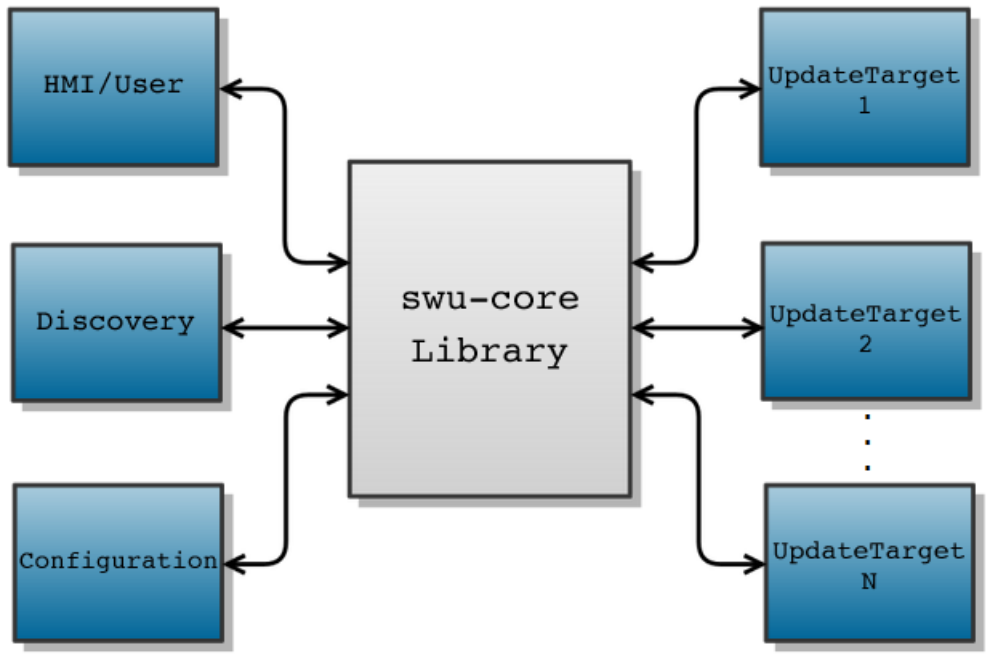


Figure 9: Interaction between swu-core library and modules used for software updates

In *Figure 9: Interaction between swu-core library and modules used for software updates* (p. 124), the `swu-core` library is shown in the center and is surrounded by the modules that developers must write to build a complete software update mechanism. The modules are as follows:

`HMI/User`

Includes code to view the available software updates and to display the progress of any updates currently being installed. This module also handles requests, either automated or issued through a GUI, to accept or decline an update.

`Discovery`

Locates software update manifest files and generates `Update` objects based on these files. For example, this module could search attached USB devices to discover manifest files and then invoke the library API to create objects based on the manifest files found.

`Configuration`

Reads and sets the configuration options of the `swu-core` library. This module must also implement any saving and restoring of configuration options.

`UpdateTarget`

Represents a system that has updatable software. The library can support many `UpdateTarget` objects concurrently.

Library components

The following diagram shows the internal components of the `swu-core` library:

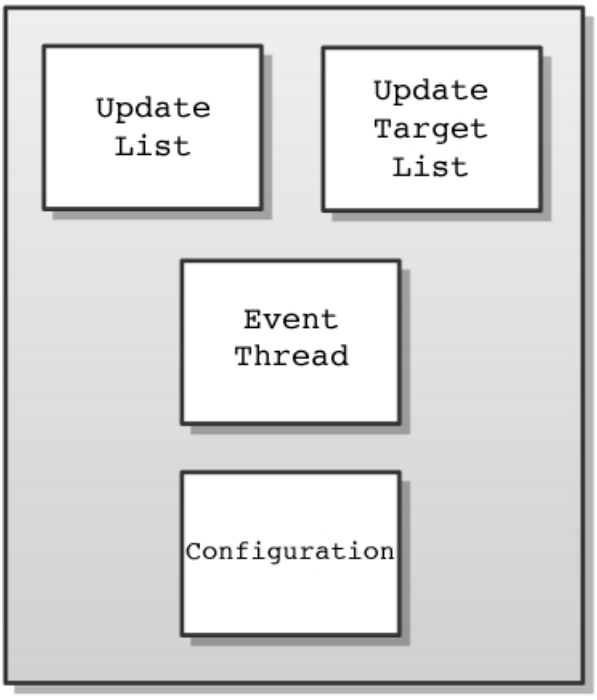


Figure 10: Components in swu-core library

As *Figure 10: Components in swu-core library* (p. 125) shows, the library is made up of these parts:

Update List

A list of `Update` objects known to the library. Each object represents an available update of a target system to a newer version.

UpdateTarget List

A list of `UpdateTarget` objects registered with the library. Each object represents an accessible target system.

Event Thread

The main event thread. All callbacks are executed in the context of this thread, so the client code should quickly exit from any callbacks to ensure good performance of the library.

Configuration

Configuration options for the library. Note that the configuration isn't persisted across reboots by the library, so the `Configuration` module should save and restore the configuration.

Key concepts of the library

The `swu-core` library stores objects that represent pending software updates and accessible target systems. It automates many aspects of interacting with these objects, including memory management, data storage, and state transitions throughout the update lifecycle.

Object handles

The library encapsulates the update data by hiding the implementation of objects. Instead of offering direct access to individual data fields, the library provides a high-level API that uses handles to refer to objects. For example, instead of working with C-strings in an `Update` object, the caller must use the *swu_update_get_name()* API function with an `swu_update_t` handle to access the update name stored in that object.

Reference counting

Many objects in the `swu-core` library are reference-counted, including:

- `swu_string_t`
- `swu_uri_t`
- `swu_client_id_t`
- `swu_update_t`

- `swu_target_t`

When working with these handle types, you should call *swu_object_retain()* to increment the reference count. Most API functions that return these handle types must call this function before exiting. When you're finished with a handle, you should call *swu_object_release()* to decrement the reference count, to indicate that you no longer need the object associated with that handle. When the reference count reaches zero, the library frees the associated object's memory. Calling the release function when you no longer need a handle is important to prevent memory leaks.

Update object

An `Update` object represents a software update that can be installed by a registered `UpdateTarget`. All configuration information for the update and all actions to be performed as part of it are specified in API calls to the `Update` object.

An `Update` object is meant for only one `UpdateTarget` (i.e., target system). You'll need to create an `Update` object for each `UpdateTarget`.

Each `Update` object contains a state machine that drives the software update process. The machine's state-transition diagram looks like this:

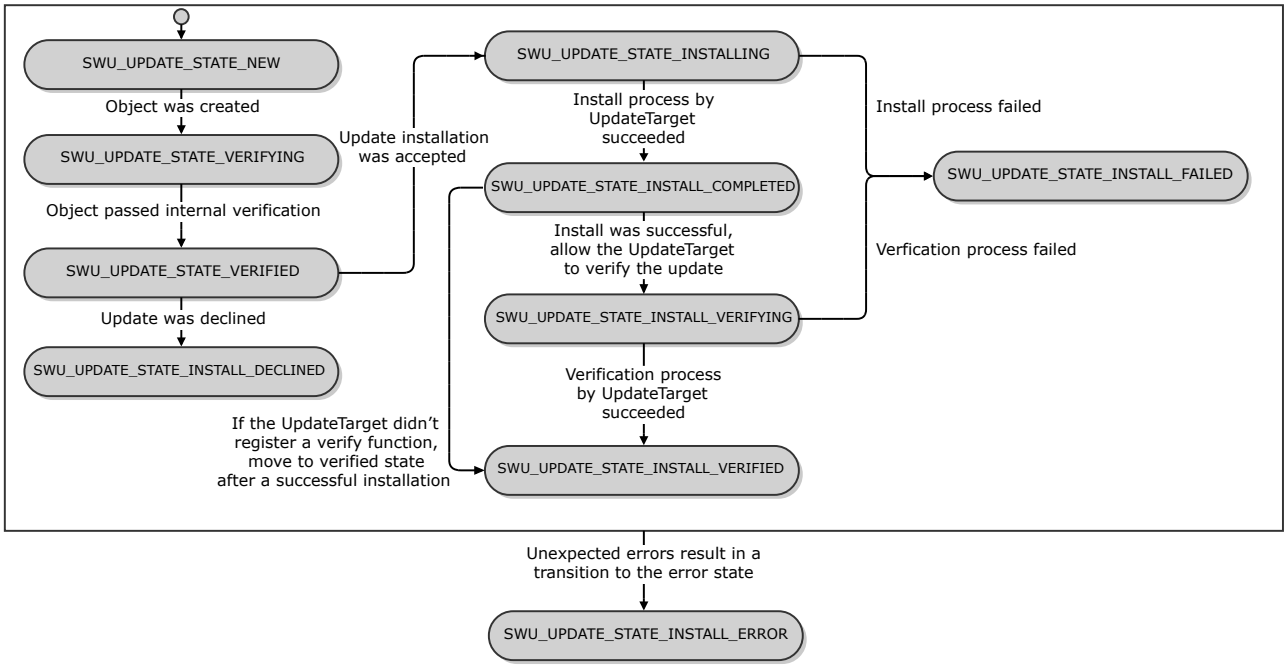


Figure 11: State-transition diagram for Update objects

The states of the `Update` object are:

`SWU_UPDATE_STATE_NEW`

The `Update` object has been created.

SWU_UPDATE_STATE_VERIFYING

The library is verifying the validity of the new object.

SWU_UPDATE_STATE_VERIFIED

The object's validity has been verified and the user or an application can now accept or decline the update.

SWU_UPDATE_STATE_INSTALLING

The *swu_update_accept_install()* function was called to start the update and the associated `UpdateTarget` object was found in the list of registered targets. While the update is in this state, the library calls the *prepare_to_install()* and *install()* functions in the registered interface of the `UpdateTarget` object.

SWU_UPDATE_STATE_INSTALL_COMPLETED

The `UpdateTarget` successfully installed the update.

SWU_UPDATE_STATE_INSTALL_FAILED

The `UpdateTarget` encountered an error during the installation or verification phase.

SWU_UPDATE_STATE_INSTALL_VERIFYING

The update was successfully installed and the `UpdateTarget` is verifying that the installation is correct, in response to the library's call to the *verify_update()* function. If the `UpdateTarget` doesn't register this function, this state is skipped and the `Update` object transitions directly to the `SWU_UPDATE_STATE_INSTALL_VERIFIED` state.

SWU_UPDATE_STATE_INSTALL_VERIFIED

The `UpdateTarget` successfully verified the update installation.

SWU_UPDATE_STATE_ERROR

An unexpected error occurred somewhere in the update lifecycle. The client should log an error message by calling the registered *swu_logging_callback_t()* function.

SWU_UPDATE_STATE_DECLINED

The update has been marked as declined, following a call to *swu_update_decline_install()*. The update remains in this state and can't be installed.

UpdateTarget object

An `UpdateTarget` represents a system with updatable software. This object type is generic so the `swu-core` library can support many different software update models. You can register multiple `UpdateTarget` objects with the library, each of which has its own implementation. For example, you could register multiple target objects to support updating different micro-controllers on a car's vehicle network.

How software update applications integrate with swu-core

The `swu-core` API provides a comprehensive interface for integrating the modules that implement parts of a software update application with the core update library.

Discovery

The `Discovery` module creates `Update` objects by loading manifest files in a call to *swu_client_create_updates()*. To find manifest files, the discovery code could monitor attached USB devices. Or, it could communicate with a server to request manifest files. Regardless of how it finds manifest files (and their associated update packages), the `Discovery` module must call *swu_client_create_updates()* to process them.

The following sample code creates `Update` objects based on a manifest file:

```
/* Function is called when a manifest file was located.
   The manifest ID is returned so that the updates can later be
   released with a call to swu_client_release_updates(). */

swu_manifest_id_t manifest_file_found ( const char* path )
{
    swu_result_t result;
    swu_manifest_id_t id;

    /* Create the updates for the given path */
    result = swu_client_create_updates( path, &id );

    if ( result == SWU_RESULT_SUCCESS )
    {
        /* Updates are created and should be accessible
            through the list of installed updates (accessed
            with swu_client_get_install_update_list()) */
        return id;
    }
    else
    {
        return SWU_INVALID_MANIFEST_ID;
    }
}
```

HMI

A software update application may need to perform many HMI-related tasks to provide the user with sufficient information and control over the update process. The following tasks can be completed in the `HMI` module by using the `swu-core` library:

1. **Displaying the list of available updates**

Typically, an HMI needs to display the list of available software updates. The user can navigate this list, display information about the updates, and then accept or decline each update. To support this interaction, the library provides the `UpdateList` data type, which allows you to hold a list of `Update` objects. There are several functions for working with `UpdateList` objects:

swu_client_get_install_update_list()

Returns a handle to the list of software updates ready to be installed. You can then use this handle in the remaining functions listed to interact with the list.

swu_update_list_get_length()

Gets the length of the list of software updates.

swu_update_list_iterate()

Iterates over the `Update` objects in the list.

swu_update_list_register_notification()

Registers a `swu_update_list_notification_t` structure, which stores a callback in its *change_notifier* field, to receive notifications from an `UpdateList`.

swu_update_list_unregister_notification()

Unregisters a structure from receiving notifications.

After getting the handle to the list of available updates by calling *swu_client_get_install_update_list()*, your client application can register for notifications by calling *swu_update_list_register_notification()*. When the `UpdateList` object changes (e.g., because `Update` objects are added to or removed from the list), the library calls the callback in the registered `swu_update_list_notification_t` structure to notify the client of the list update. This callback can refresh the HMI to display the latest list contents by iterating through the list and processing its `Update` objects.

The *swu_update_list_iterate()* function is the only way to access the contents of an `UpdateList` object. When calling this function, you must pass in a function pointer of the `swu_update_list_iterator_t` type to reference the code that processes list items. The library calls this iterator function on each `Update` object found in the list, and then one final time with a NULL value for the `Update` object to indicate the list's end. The callback function can stop the list iteration at any time by returning `false` instead of `true`, which tells the library to stop iterating through the list.

The following diagram shows how the iteration work is shared between the software update application and the `swu-core` library:

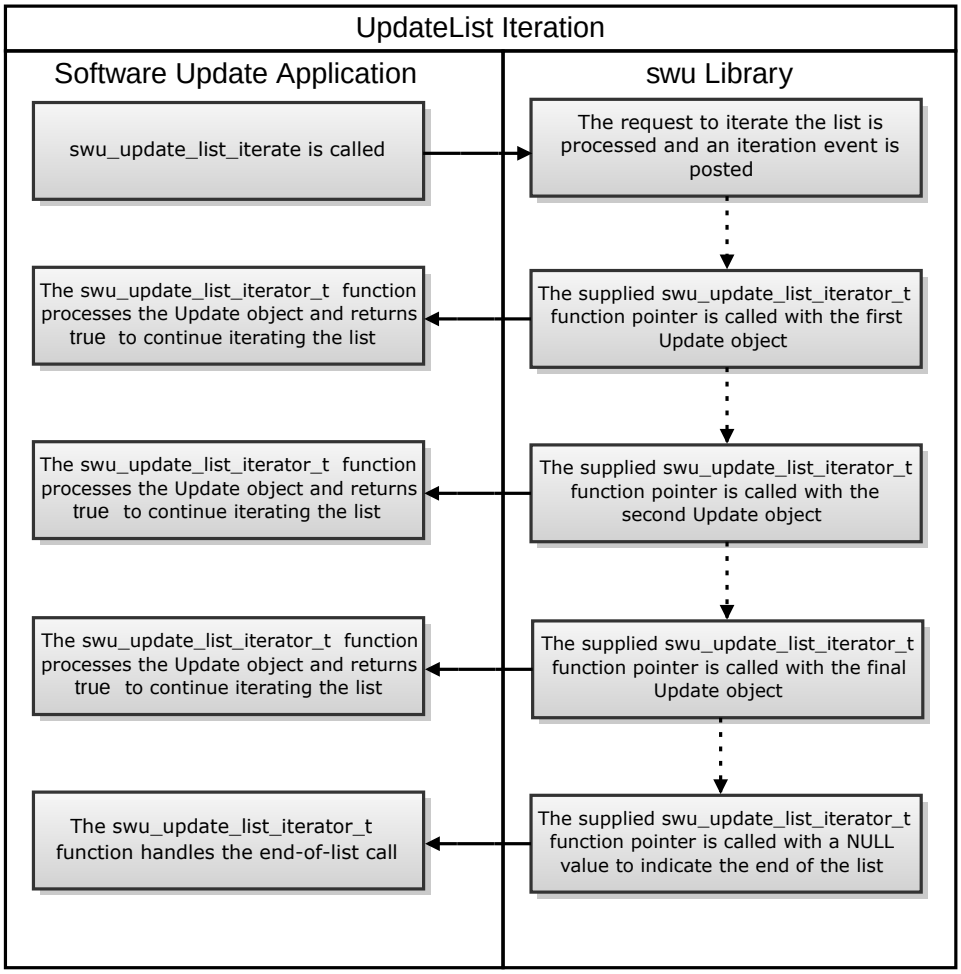


Figure 12: Interaction between the application and the swu-core library during UpdateList iteration

This code sample shows how to process the `UpdateList` after it has changed:

```
/* The function used as the swu_update_list_iterator_t
   callback */
bool iterate_list( swu_update_t update,
                   void *context )
{
    /* When iterating the list, the library calls this
        function a final time with a NULL object reference to
        indicate the end of the list */
    if ( update != NULL )
    {
        /* Call the necessary APIs on the Update object */
        ....
    }

    /* Return true to continue iterating through the list */
    return true;
}
```

```
/* The function for reacting to a list update notification */
void updatelist_changed( swu_update_list_t list,
                         void *context )
{
    /* Our list has changed--refresh the display by processing
        the current list items, using the specified callback
        to iterate through the list */
    swu_update_list_iterate( list, iterate_list, NULL );
}
```

2. **Displaying information about an update**

   The `Update` object provides an access function for each of its properties. You can retrieve these properties and then display update information to the user. The property functions include:

   - *swu_update_get_name()*
   - *swu_update_get_description()*
   - *swu_update_get_version()*
   - *swu_update_get_priorty()*

   Further information on these and other `Update` functions can be found in the *SWU library API* (p. 142).

   The following sample prints the name and version of an `Update` object to `stdout`. Because these functions all return handles to `swu_string_t` data items, the strings contained in these items must be released with a call to *swu_object_release()* after the caller is done with them.

```
void print_update_object( swu_update_t update )
{
    swu_string_t name;
    swu_string_t version;

    if ( swu_update_get_name( update, &name ) ==
                                      SWU_RESULT_SUCCESS )
    {
        printf("Object name is %s\n", name);
        swu_object_release(name);
    }

    if ( swu_update_get_version( update, &version ) ==
                                      SWU_RESULT_SUCCESS )
    {
        printf("Object version is %s\n", version);
        swu_object_release(version);
    }
}
```

3. **Acting on an update**

   To inform the library of the user's decision whether or not to install an update, the `HMI` module must call one of these two API functions:

   swu_update_accept_install()

Requests installation of the software update to begin on the corresponding target.

swu_update_decline_install()

Declines installation of the software update.

Typically, you call these functions after iterating through the list of updates and finding a particular `Update` object to take the action on.

4. **Installing an update**

After you accept the installation for an `Update` object, the library notifies the associated `UpdateTarget` object to begin the installation. To monitor the installation status, your software update application should call *swu_update_register_notifications()* to register callbacks so it will be notified when the state or progress of the installation changes. The `swu_update_notifications_t` structure contains two notification callbacks:

state_changed

Called when the `Update` object changes state, as shown in the *State-transition diagram for* `Update` *objects* (p. 127).

progress

Called when the `UpdateTarget` indicates a change in progress while the `Update` object is in the `SWU_UPDATE_STATE_INSTALLING` or `SWU_UPDATE_STATE_VERIFYING` state.

These notifications can be used to refresh progress bars in the HMI or process a change in the installation status. The change-of-status processing could involve displaying errors when necessary. For example, suppose the `Update` object transitions to the `SWU_UPDATE_STATE_INSTALL_FAILED` state. The client could then call *swu_update_get_failure_info()* to retrieve the failure code, and then display that code and other error information in the HMI or write this data to an error log.

The following sample calls *swu_update_accept_install()* and then registers and monitors these callbacks:

```
void handle_state_change( swu_update_t update,
                          swu_update_state_t state,
                          void *state_changed_context )
{
    if ( state == SWU_UPDATE_STATE_INSTALL_FAILED )
    {
        swu_failure_info_t info;
        if ( swu_update_get_failure_info ( update, &info )
                                        == SWU_RESULT_SUCCESS )
        {
            /* Tell the user about the failure */
            ....
        }
```

```
    }
    else
    {
       /* Handle other state changes */
       ....
    }
}

void handle_progress_change( swu_update_t update,
                             swu_progress_t percent,
                             void *progress_context )
{
    /* Refresh the progress percentage in the HMI */
    update_progress_display( update, percent );
}

static swu_update_notifications_t notifications =
    { handle_progress_change, NULL,
        handle_state_change, NULL };

void accept_install( swu_update_t update )
{
    if ( swu_update_accept_install( update ) ==
                                SWU_RESULT_SUCCESS )
    {
         swu_update_register_notifications(
                        update, &notifications );
    }
}
```

UpdateTarget

Each `UpdateTarget` object supports installing a software update on a target system. You can register multiple `UpdateTarget` objects with the library, but each `Update` object that you create can work with only one `UpdateTarget`.

An update application must do these tasks for an `UpdateTarget`:

1. **Register with the library**

   Before it can install a software update, an `UpdateTarget` must be registered with the library. Through the *swu_target_register()* function, you can provide the library with the target's vendor ID and hardware ID (for identifying it) and with a pointer to an `swu_target_interface_t` structure (for communicating with it). Multiple `UpdateTarget` objects may share the same `swu_target_interface_t`, but each of these registered objects must have a unique combination of vendor ID and hardware ID to distinguish it from other targets.

   When the registration succeeds, the library returns an `swu_target_id_t` handle, which you must use to refer to that same `UpdateTarget` in subsequent API calls.

   The following sample shows a simple `UpdateTarget` registration:

```
swu_target_id_t register_target( const char *vendor_id,
                                 const char *hardware_id )
{
    /* We define only the get_info() and install() function
        pointers in this interface */
    swu_target_interface_t interface = {get_target_info, NULL,
```

```
                                                NULL, NULL,
                                                install_update, NULL,
                                                NULL, NULL,
                                                NULL, NULL,
                                                NULL, NULL};

    /* Store the ID assigned by the swu-core library.
        The ID is needed in future API calls. */
    swu_target_id_t assigned_id;

    if ( swu_target_register( vendor_id,
                              hardware_id,
                              &interface,
                              &assigned_id ) ==
                                  SWU_RESULT_SUCCESS )
    {
        return assigned_id;
    }
    else
    {
        return SWU_INVALID_TARGET_ID;
    }
}
```

In this sample, most function pointers in the `swu_target_interface_t` structure are left undefined. The `swu-core` library considers any function that has a reference pointer of NULL to be unsupported. In this case, the library skips that step in the software update process and tries to continue the installation at the next step.

2. **Retrieve information about the target**

   The function defined by the *get_info()* pointer in the `UpdateTarget` interface retrieves the properties describing the current version of software installed on the target. The library calls this interface function when handling a call to the *swu_target_get_info()* API function and after an update was successfully verified by an `UpdateTarget`. In the latter case, the *get_info* call ensures that the library has up-to-date target information following a successful update, which might have changed some aspect of the `UpdateTarget`.

3. **Implement software update functions**

   To carry out the update process, the library calls the functions referenced by the pointers in the `swu_target_interface_t` structure. In each of these functions, the `UpdateTarget` must call a success or failure function at some point to tell the library whether to continue with the update process.

   The interaction between the `UpdateTarget` object and the library proceeds like this:

   a. When a call from the `HMI` module to *swu_object_accept_install()* completes successfully, the library calls the *prepare_to_install()* function defined for the `UpdateTarget`. In this function, the `UpdateTarget` examines its state to determine whether it can install the update. If so, the `UpdateTarget` calls

swu_target_ready_to_install(); if not, it calls *swu_target_not_ready_to_install()*. If no *prepare_to_install()* function is defined, the library skips this step.

b. Next, the library calls the provided *install()* function and the `UpdateTarget` begins installing the software update. During the installation, the `UpdateTarget` can call *swu_target_install_progress()* to inform the `swu-core` library (and any registered listeners of the associated `Update` object) of the installation progress.

If the installation succeeds, the `UpdateTarget` calls *swu_target_install_successful()*; otherwise, it calls *swu_target_install_failed()*.

c. When the installation completes, the library calls the provided *verify_update()* function so the `UpdateTarget` can verify the installation. During the verification, the `UpdateTarget` can call *swu_target_verification_progress()* to inform the `swu-core` library (and any registered `Update` listeners) of the verification progress.

If the verification succeeds, the `UpdateTarget` calls *swu_target_verification_successful()*; otherwise, it calls *swu_target_verification_failed()*.

If the `UpdateTarget` doesn't support verification of update installations, which means its *verify_update()* function pointer is set to NULL, the library skips the verification step.

A successful update installation involving all three phases—preparation, installation, and verification—is shown in the diagram that follows. The diagram lists the `Update` object state changes and follow-up actions taken by the library in response to notifications it receives from the `UpdateTarget`:

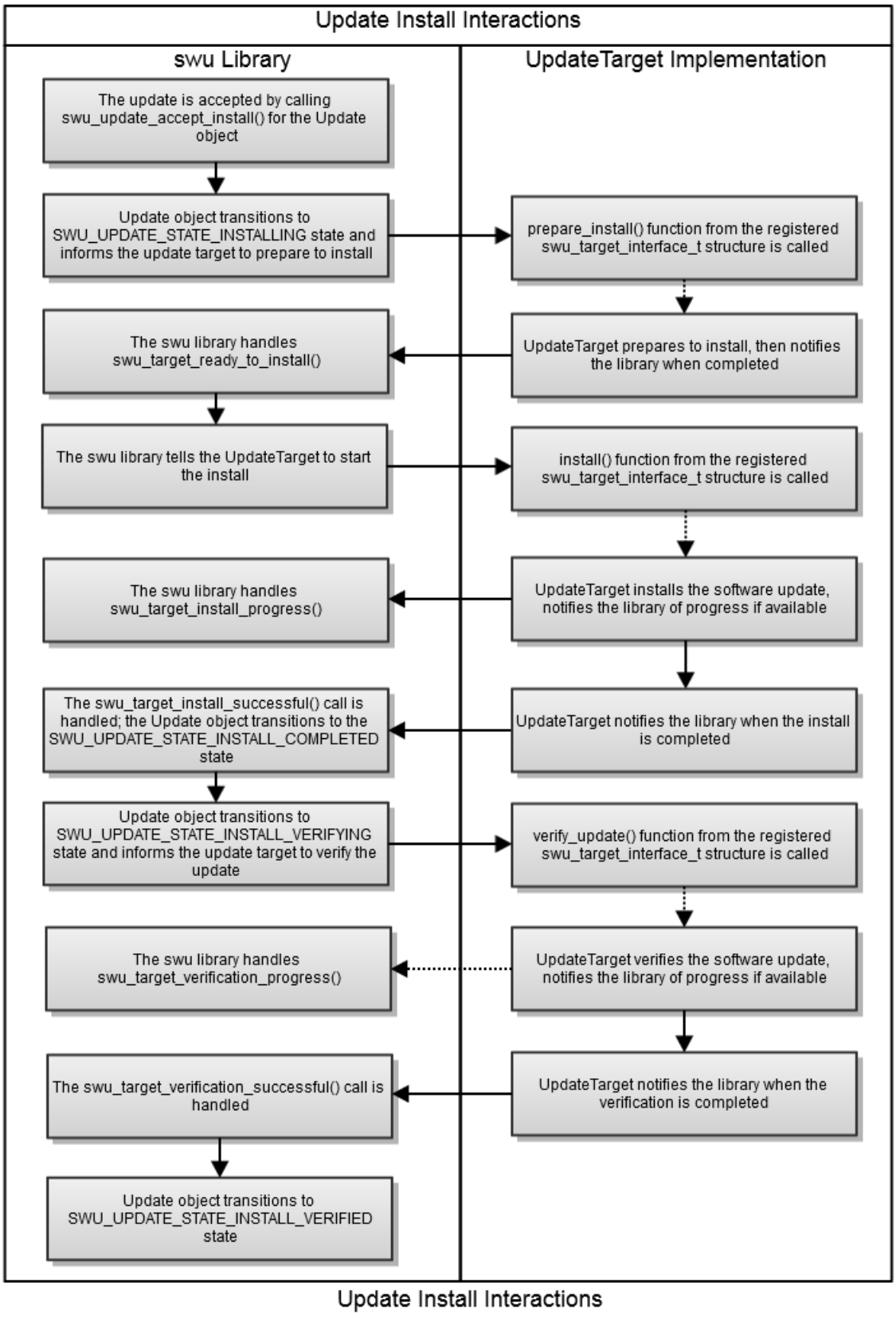


Figure 13: Interaction between an UpdateTarget object and the swu-core library during an update installation

Configuration

You can write a `Configuration` module that sets configuration attributes that apply to all updates.

The `UpdateClient` data type stores and exposes, through the `swu-core` library API, these attributes:

Client ID

read-only

An ID field that uniquely identifies an `UpdateClient`. This field is set during library initialization.

Local updates

read/write

Enables or disables local (e.g., USB or flash) software updates for the `UpdateClient`.

Update grace period

read/write

Grace period for accepting software updates, which indicates how long (in seconds) an update will be available for installation. This attribute setting is used for software updates that don't define their own grace period.

Maximum update retries

read/write

Maximum number of retries allowed for a software update. Note that this attribute is currently unused.

The following code demonstrates how to get and set the update grace period:

```
/* Get the update grace period */
swu_result_t client_config_get_update_grace_period(
                                        swu_timestamp_t *period )
{

    swu_result_t result =
      swu_client_configuration_get_update_grace_period( period );

    if ( result != SWU_RESULT_SUCCESS )
    {
        printf(
          "Error getting update grace period from swu client\n");
    }

    return result;
}

/* Set the update grace period */
```

```
swu_result_t client_config_set_update_grace_period(
                                        swu_timestamp_t period )
{
    swu_result_t result =
      swu_client_configuration_set_update_grace_period( period );

    if ( result != SWU_RESULT_SUCCESS )
    {
        printf(
          "Error setting update grace period on swu client\n");
    }

    return result;
}
```

Manifest file

A *manifest file* stores software update information, including the vendor and hardware IDs of the target, timestamps, version numbers, and the size and path of the delta file. You must provide a manifest file as part of an update package.

Structure

The manifest file follows the `.ini` file format, which has these characteristics:

- Each line is expected to be a key-value pair with an equal sign (=) separating the two tokens.
- Comment lines start with a semicolon character (`;`).
- The first valid line (i.e., the line isn't a blank or a comment) of the file must indicate the format version by using the `format_version` key.

You can describe multiple updates in one manifest file. Each set of fields describing an update must begin with the `id` field wrapped in square brackets:
`[id="UPDATE_ID"]`

The lines that follow are treated as attributes of the same update until another update is specified or the end of the file is reached. All updates require a certain minimum set of fields:

Key	Description	Required	Example
`action`	Comma-seperated list of actions available for this update: • SKIP_PROMPT_INSTALL • CAN_DECLINE_INSTALL • CAN_DEFER_INSTALL	No	`action=CAN_DECLINE_IN STALL,CAN_DEFER_INSTALL`
`base_version`	Expected version of the software currently installed on the target. The `UpdateTarget` can use this field to	No	`base_version="00.00.00"`

Key	Description	Required	Example
	determine if it can accept the update.		
`format_version`	Format version of the file. This must be the first field set in the file. Currently, the `swu-core` library supports version `20130918`.	Yes	`format_version=20130918`
`grace_period`	Grace period for the update. The value is treated as a signed 64-bit unixtime.	No	`grace_period=60000`
`hardware_id`	Hardware ID, used with the value in `vendor_id` to uniquely identify an `UpdateTarget` when it's matched with an `Update`.	Yes	`hardware_id="CAR2.1"`
`id`	Unique ID used by the update. This field starts a new update record in the manifest file.	Yes	`[id="UPDATE_001"]`
`long`	Long description of the update contents. This field has no length limit.	No	`long="The following bugs have been addressed in this release\n\t-Bug 77\n\t-Bug 88..."`
`max_defer_period`	Maximum deferral period for the update. The value is treated as a signed 64-bit value representing Unix time.	No	`max_defer_period=3600000`
`name`	Name of the update.	Yes	`name="Security Release 242"`
`path`	Path of the software update file. This can be either an absolute path or a path relative to the manifest file.	Yes	`path=./update_file/the_file_to_use.bin`
`post_install_command`	Command string to be run by the `UpdateTarget` after the install completes.	No	`post_install_command="/scripts/post_update.sh"`
`pre_install_command`	Command string to be run by the `UpdateTarget` before the install starts.	No	`pre_install_command="/scripts/prepare_for_update.sh"`
`priority`	Priority level of the update. This integer value must be between 1 and	No	`priority=3`

Key	Description	Required	Example
	`20`, where `1` is the highest priority and `20` is the lowest.		
`short`	Short description of the update contents. This field has no length limit.	No	`short="Contains necessary bug fixes"`
`size`	Size of the software update file (in bytes). This is an unsigned 32-bit integer value.	No	`size=22000000`
`timestamp`	Release timestamp for the update. The value is treated as a signed 64-bit value representing Unix time.	No	`timestamp=1375119694`
`vendor_id`	Vendor ID, used with the value in `hardware_id` to uniquely identify an `UpdateTarget` when it's matched with an `Update`.	Yes	`vendor_id="QNX"`
`version`	Version of the software update.	Yes	`version="00.00.01"`

Sample file

```
; This is a sample manifest file.
; All manifest files must start with the format_version key
format_version=20130918

; A comment about this first update
[id="UPDATE001"]
vendor_id="QNX"
hardware_id="CAR2"
timestamp=1375116694
short="A simple update to test some things"
long="This is a simple test of the SWU system.\n\n\
       \tTabbed text on a new line"
version="00.00.01"
base_version="00.00.00"
name="Test Update"
action=CAN_DECLINE_INSTALL,CAN_DEFER_INSTALL
grace_period=600
size=2147483700
; This update points to a delta file in the same directory
path=mydelta.mld
pre_install_command="/base/scripts/prepare_self_update.sh"

; A comment about this second update
[id="UPDATE002"]
name="Another Test Update"
vendor_id="QNX"
hardware_id="CAR2"
long="This text description can be as long as need be,\
      provided you escape any line breaks in the text string"
version="00.00.01"
short="YATU (Yet another Test Update)"
```

```
timestamp=1375119694
; This update points to an absolute path for a file elsewhere
; in the filesystem.
path=/dos/mydelta.mld
```

SWU library API

The SWU library API is exposed in six header files that define the constants, variable types, data structures, and functions that your update application can use to manage the update process. The API allows you to generate updates based on a manifest file, read information on individual updates, and accept or decline an update. It can also report progress and failures of active updates.

The first API function you must call is *swu_client_initialize()*, which sets up the library. Next, you can call *swu_client_create_updates()* and pass in the path of a manifest file, which specifies one or more updates. The library will generate `Update` and `UpdateTarget` lists based on the manifest file contents.

Before you can apply updates to a target, you must register it with the library by calling *swu_target_register()*, passing in a reference to an `swu_target_interface_t` structure. This structure defines the callback functions that carry out the various update tasks, including installation and verification. To receive progress information while an update is being applied, you can call *swu_update_register_notifications()* and register an `swu_update_notifications_t` structure, which defines callbacks for handling progress notifications.

You can call the *swu_update_get_** functions (in `Update.h`) to get useful information such as the manifest file used to create the update, the base (i.e., current) version of the software on the target, and a referent to the target. For the target, you read its information with *swu_target_get_info()*. You can use the update and target information to decide whether to proceed with the update and then call either *swu_update_accept_install()* or *swu_update_decline_install()*.

While applying an update, the library calls the `swu_target_interface_t` functions (to drive the update process) and the `swu_update_notifications_t` functions (to report update progress). The actions done by your application within any of these callbacks depend on your update policy, but you must report the outcomes of update operations when appropriate by calling the functions in `UpdateTargetInterface.h` (e.g., *swu_target_install_successful()*, *swu_target_install_failed()*).

If an update installation or verification fails, your application can try to reapply the update but only as many times as indicated by the retry limit given in the configuration settings. To access these settings, use the functions in `ClientConfiguration.h`.

When you're finished using the SWU library, you must call *swu_client_uninitialize()* to release its memory.

ClientConfiguration.h

The `ClientConfiguration.h` header file defines functions for reading and writing software update configuration settings.

These functions manage the following settings:

- Flag setting for enabling or disabling local updates
- Default grace period for installing updates
- Maximum number of retries allowed for a software update

Functions in ClientConfiguration.h

Functions defined in `ClientConfiguration.h` for retrieving the ID of the `UpdateClient` and for getting and setting the status of local software updates, the default grace period for installing updates, and the maximum number of retries allowed per update.

swu_client_configuration_disable_local_updates()

Disable local software updates for the `UpdateClient`

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_disable_local_updates(
                                                        void )
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Disable local software updates for the `UpdateClient`. Local software updates are updates whose information is read off USB or flash devices. Note that the flag set by this function doesn't affect how the library operates. Instead, the flag is intended to let library users inform other processes or modules of the state of local updates.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_NOT_INITIALIZED`

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_configuration_enable_local_updates()

Enable local software updates for the `UpdateClient`

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_enable_local_updates(
                                                        void )
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Enable local software updates for the `UpdateClient`. Local software updates are updates whose information is read off USB or flash devices. Note that the flag set by this function doesn't affect how the library operates. Instead, the flag is intended to let library users inform other processes or modules of the state of local updates.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_configuration_get_id()

Get the unique ID of the `UpdateClient`

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_get_id(
                                          swu_client_id_t *id )
```

Arguments:

id

On output, the client ID.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the unique ID of the `UpdateClient`. The `UpdateClient` is identified by a unique ID set by the library during initialization. Currently, the library doesn't use this ID internally. However, the ID could be included to distinguish the client from other clients when it reports installation details to a server.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument (e.g., a bad pointer) was given.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_configuration_get_local_updates_enabled()

Determine whether local software updates are enabled

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_get_local_updates_enabled(
                                                    bool *enabled )
```

Arguments:

enabled

On output, `true` if local updates are enabled and `false` if they're disabled.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Determine whether local software updates are enabled for the `UpdateClient`. Local software updates are updates whose information is read off USB or flash devices.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument (e.g., a bad pointer) was given.

`SWU_RESULT_NOT_INITIALIZED`

The library hasn't been initialized yet.

`SWU_RESULT_ERROR`

Another error occurred.

swu_client_configuration_get_max_update_retries()

Get the maximum number of retries allowed per software update

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_get_max_update_retries(
                                            uint8_t *max_retries )
```

Arguments:

max_retries

On output, the maximum number of retries allowed for an update.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the maximum number of retries allowed per software update. If an `Update` object fails to install an update on its target, the `Update` will retry the installation until either it succeeds or has attempted the installation as many times as specified by this configuration setting. Note that the library currently doesn't use the retry value in the installation process.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument (e.g., a bad pointer) was given.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_configuration_get_update_grace_period()

Get the default grace period for accepting software updates

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_get_update_grace_period(
                                        swu_timestamp_t *period )
```

Arguments:

period

On output, the grace period (in seconds).

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the default grace period for accepting software updates. This default setting applies to any `Update` that didn't have its own grace period defined when it was created. To learn the grace period for an individual `Update`, call *swu_update_get_grace_period()* (p. 179).

In this release, the grace period has no impact on how the SWU library handles updates. The grace period is just kept as metadata describing an update (because this value can be set in the manifest file).

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument (e.g., a bad pointer) was given.

`SWU_RESULT_NOT_INITIALIZED`

The library hasn't been initialized yet.

`SWU_RESULT_ERROR`

Another error occurred.

swu_client_configuration_set_max_update_retries()

Set the maximum number of retries allowed per software update

Synopsis:

```
#include <swu/ClientConfiguration.h>

swu_result_t swu_client_configuration_set_max_update_retries(
                                                uint8_t max_retries )
```

Arguments:

max_retries

Maximum number of retries allowed for an update.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Set the maximum number of retries allowed per software update. If an `Update` object fails to install an update on its target, the `Update` can retry the installation as many times as specified by this configuration setting. Note that the library currently doesn't use the retry value in the installation process.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_configuration_set_update_grace_period()

Set the default grace period for accepting software updates

Synopsis:

```
#include <swu/ClientConfiguration.h>
swu_result_t swu_client_configuration_set_update_grace_period(
                                        swu_timestamp_t period )
```

Arguments:

period

Grace period (in seconds).

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Set the default grace period for accepting software updates. This new default setting will override any grace period defined in the manifest file as well as the library's own default value of seven days. The new setting will apply to any `Update` that doesn't have its own grace period defined when it's created. To learn the grace period for an individual `Update`, call *swu_update_get_grace_period()* (p. 179).

In this release, the grace period has no impact on how the SWU library handles updates. The grace period is just kept as metadata describing an update (because this value can be set in the manifest file).

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_NOT_INITIALIZED`

The library hasn't been initialized yet.

`SWU_RESULT_ERROR`

Another error occurred.

Common.h

The `Common.h` header file defines constants, enumerations, data types, and functions used by all modules in the SWU library API.

This module exposes the following useful features:

- Constants for field length limits and invalid values
- Enumerations to represent update states and outcomes of API calls and of update installations
- Data structures for holding information about update targets and installation failures
- Utility functions for reference counting of library objects and for converting enumerated constants to strings

Constants in Common.h

Constants defined in `Common.h` to specify maximum lengths for *UpdateTarget* fields, represent invalid field values, and refer to all update states.

Definitions in Common.h

Preprocessor macro definitions in `Common.h`

Definitions:

```
#define SWU_UPDATE_TARGET_VENDOR_ID_LEN 100
```

Maximum length of the vendor ID on an `UpdateTarget` (not including the null-terminator).

```
#define SWU_UPDATE_TARGET_HARDWARE_ID_LEN 100
```

Maximum length of the hardware ID on an `UpdateTarget` (not including the null-terminator).

```
#define SWU_UPDATE_TARGET_SERIAL_NUM_LEN 100
```

Maximum length of a serial number on an `UpdateTarget` (not including the null-terminator).

```
#define SWU_UPDATE_TARGET_BOM_VERSION_LEN 100
```

Maximum length of the software version of an `UpdateTarget` (not including the null-terminator).

```
#define SWU_INVALID_TARGET_ID ((swu_target_id_t)0)
```

Represents an invalid value for an *swu_target_id_t*.

```
#define SWU_INVALID_GRACE_PERIOD ((swu_timestamp_t) INT64_MAX)
```

Represents an invalid grace period value.

```
#define SWU_INVALID_MANIFEST_ID ((swu_manifest_id_t)0)
```

Represents an invalid manifest ID.

```
#define SWU_UPDATE_ALL_STATES \
        (SWU_UPDATE_STATE_NEW | \
         SWU_UPDATE_STATE_VERIFYING | \
         SWU_UPDATE_STATE_VERIFIED | \
         SWU_UPDATE_STATE_INSTALLING | \
         SWU_UPDATE_STATE_INSTALL_COMPLETED | \
         SWU_UPDATE_STATE_INSTALL_FAILED | \
         SWU_UPDATE_STATE_INSTALL_CANCELLING | \
         SWU_UPDATE_STATE_INSTALL_CANCELLED | \
         SWU_UPDATE_STATE_INSTALL_VERIFYING | \
         SWU_UPDATE_STATE_INSTALL_VERIFIED | \
         SWU_UPDATE_STATE_ROLLING_BACK | \
         SWU_UPDATE_STATE_ROLLBACK_COMPLETED | \
         SWU_UPDATE_STATE_ROLLBACK_FAILED | \
         SWU_UPDATE_STATE_ERROR | \
         SWU_UPDATE_STATE_DECLINED)
```

Represents all possible `Update` states. This constant is useful for setting a notification for all states.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Enumerations in Common.h

Enumerations defined in `Common.h` to represent result codes of API calls, reasons for update failures, states and priorities of updates, and levels of logging.

swu_failure_reason_t

Possible failure reasons for update installations

Synopsis:

```
#include <swu/Common.h>

typedef enum {
      SWU_FAILURE_REASON_UPDATE_NOT_SUPPORTED,
      SWU_FAILURE_REASON_NOT_READY_FOR_UPDATE,
      SWU_FAILURE_REASON_INVALID_CONDITIONS,
      SWU_FAILURE_REASON_INSTALL_FAILED,
      SWU_FAILURE_REASON_INSTALL_VERIFICATION_FAILED
} swu_failure_reason_t;
```

Data:

SWU_FAILURE_REASON_UPDATE_NOT_SUPPORTED

The target system doesn't support updates.

SWU_FAILURE_REASON_NOT_READY_FOR_UPDATE

The target system isn't ready for updates.

SWU_FAILURE_REASON_INVALID_CONDITIONS

The target system couldn't install the update (e.g., because a bad base version of the software was specified).

SWU_FAILURE_REASON_INSTALL_FAILED

A failure occurred during the update installation.

SWU_FAILURE_REASON_INSTALL_VERIFICATION_FAILED

A failure occured during the verification of the update installation.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_failure_reason_t* enumeration defines the possible reasons that an update installation can fail.

swu_result_t

Possible outcomes for an API call

Synopsis:

```
#include <swu/Common.h>

typedef enum {
      SWU_RESULT_SUCCESS,
      SWU_RESULT_ERROR,
      SWU_RESULT_EMPTY,
      SWU_RESULT_DUPLICATE_ENTRY,
      SWU_RESULT_NOT_FOUND,
      SWU_RESULT_INVALID_ARGUMENT,
      SWU_RESULT_OUT_OF_MEMORY,
      SWU_RESULT_API_NOT_AVAILABLE,
      SWU_RESULT_UPDATE_TARGET_BUSY,
      SWU_RESULT_NOT_INITIALIZED,
      SWU_RESULT_CONDITIONS_NOT_VALID_TO_INSTALL
} swu_result_t;
```

Data:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_ERROR

An unknown error ocurred (or the error doesn't fit into the other categories).

SWU_RESULT_EMPTY

The library couldn't find the information needed for the update.

SWU_RESULT_DUPLICATE_ENTRY

A library object with the same identifier values given in the arguments has already been registered.

SWU_RESULT_NOT_FOUND

A library object or callback structure referenced in the arguments couldn't be found (e.g., because an invalid ID was specified).

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_OUT_OF_MEMORY

The system ran out of memory.

SWU_RESULT_API_NOT_AVAILABLE

The library doesn't support this operation.

SWU_RESULT_UPDATE_TARGET_BUSY

The target is busy and can't start an update installation or verification.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_CONDITIONS_NOT_VALID_TO_INSTALL

The target system currently can't perform update installations.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_result_t* enumeration defines the codes indicating if an API call succeeded or if not, the reason it failed. Most functions in the SWU library API return an *swu_result_t* to indicate the success or failure of the call. Typically, the caller simply checks whether the call returned `SWU_RESULT_SUCCESS`. The failure result codes are useful for logging and for debugging.

The *swu_result_to_string()* (p. 166) function returns the string representation of an *swu_result_t* code, which is helpful for logging.

swu_update_priority_t

Possible priority levels for a software update

Synopsis:

```
#include <swu/Common.h>

typedef enum {
      SWU_UPDATE_PRIORITY_CRITICAL = 1,
      SWU_UPDATE_PRIORITY_NORMAL = 10,
      SWU_UPDATE_PRIORITY_USEFUL = 20
} swu_update_priority_t;
```

Data:

SWU_UPDATE_PRIORITY_CRITICAL

Highest priority level.

SWU_UPDATE_PRIORITY_NORMAL

Default priority level.

SWU_UPDATE_PRIORITY_USEFUL

Lowest priority level.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_priority_t` enumeration defines possible priority levels for a software update.

swu_update_state_t

Possible states of a software update

Synopsis:

```
#include <swu/Common.h>

typedef enum {
      SWU_UPDATE_STATE_NEW = 0x00000001,
      SWU_UPDATE_STATE_VERIFYING = 0x00000080,
      SWU_UPDATE_STATE_VERIFIED = 0x00000100,
      SWU_UPDATE_STATE_INSTALLING = 0x00000200,
      SWU_UPDATE_STATE_INSTALL_COMPLETED = 0x00000400,
      SWU_UPDATE_STATE_INSTALL_FAILED = 0x00000800,
      SWU_UPDATE_STATE_INSTALL_CANCELLING = 0x00001000,
      SWU_UPDATE_STATE_INSTALL_CANCELLED = 0x00002000,
      SWU_UPDATE_STATE_INSTALL_VERIFYING = 0x00004000,
      SWU_UPDATE_STATE_INSTALL_VERIFIED = 0x00008000,
      SWU_UPDATE_STATE_ROLLING_BACK = 0x00010000,
      SWU_UPDATE_STATE_ROLLBACK_COMPLETED = 0x00020000,
      SWU_UPDATE_STATE_ROLLBACK_FAILED = 0x00040000,
      SWU_UPDATE_STATE_ERROR = 0x00080000,
      SWU_UPDATE_STATE_DECLINED = 0x00100000
} swu_update_state_t;
```

Data:

SWU_UPDATE_STATE_NEW

The `Update` object has been created.

SWU_UPDATE_STATE_VERIFYING

The update is being verified.

SWU_UPDATE_STATE_VERIFIED

The update has passed internal verification.

SWU_UPDATE_STATE_INSTALLING

The user accepted the update and the system has started installing it.

SWU_UPDATE_STATE_INSTALL_COMPLETED

The update installation successfully completed.

SWU_UPDATE_STATE_INSTALL_FAILED

The update installation or the verification of the installation failed.

SWU_UPDATE_STATE_INSTALL_CANCELLING

The update is being cancelled by the user. This state is currently unused.

SWU_UPDATE_STATE_INSTALL_CANCELLED

The update was cancelled by the user. This state is currently unused.

SWU_UPDATE_STATE_INSTALL_VERIFYING

The `UpdateTarget` is verifying an update after successfully installing it.

SWU_UPDATE_STATE_INSTALL_VERIFIED

The update installation was successfully verified.

SWU_UPDATE_STATE_ROLLING_BACK

The update is being rolled back by the user. This state is currently unused.

SWU_UPDATE_STATE_ROLLBACK_COMPLETED

The update was rolled back by the user. This state is currently unused.

SWU_UPDATE_STATE_ROLLBACK_FAILED

The update rollback failed. This state is currently unused.

SWU_UPDATE_STATE_ERROR

An unexpected error occurred.

SWU_UPDATE_STATE_DECLINED

The user didn't accept the update.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_update_state_t* enumeration defines the possible states of a software update. These enumeration constants are defined in a way that allows one or many constants to be used in a bitmask when defining notifications.

The *swu_update_state_to_string()* (p. 167) function returns the string representation of an *swu_update_state_t* code, which is helpful for logging.

swu_log_level_t

Severity levels for messages logged with swu_logging_callback_t function

Synopsis:

```
#include <swu/Common.h>

typedef enum swu_log_level {
      SWU_LOG_SHUTDOWN,
      SWU_LOG_CRITICAL,
      SWU_LOG_ERROR,
      SWU_LOG_WARNING,
      SWU_LOG_NOTICE,
      SWU_LOG_INFO
} swu_log_level_t;
```

Data:

SWU_LOG_SHUTDOWN

A critical error has occurred and the system must be shut down. This value is currently unused.

SWU_LOG_CRITICAL

A critical error has occured and the library must be shut down. This value is currently unused.

SWU_LOG_ERROR

A serious error has occurred, preventing the library from completing the update process.

SWU_LOG_WARNING

An issue has been found by the library but it can continue with the update process.

SWU_LOG_NOTICE

The library is reporting a significant event related to the update process.

SWU_LOG_INFO

The library is reporting information on the update process.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_log_level_t* enumeration specifies the possible severity levels for messages logged with the *swu_logging_callback_t* (p. 200) function.

Data types in Common.h

Data types defined in `Common.h` for storing IDs and handles of library objects and for representing timestamps, strings, and *UpdateTarget* information.

swu_client_id_t

Unique ID of an `UpdateClient`

Synopsis:

```
#include <swu/Common.h>

typedef swu_string_t swu_client_id_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_client_id_t` data type stores the unique ID of an `UpdateClient`.

swu_failure_code_t

Customer-specific code indicating why an installation failed

Synopsis:

```
#include <swu/Common.h>

typedef uint32_t swu_failure_code_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_failure_code_t` data type stores a customer-specific code indicating why an installation failed.

swu_failure_info_t

Describes the failure experienced by an `UpdateTarget`

Synopsis:

```
typedef struct {
    swu_failure_reason_t reason;
    swu_failure_code_t code;
} swu_failure_info_t;
```

Data:

swu_failure_reason_t reason

An `swu_failure_reason_t` constant indicating the type of failure.

swu_failure_code_t code

A customer-specific code providing more details about the failure.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_failure_info_t` structure describes the failure experienced by an `UpdateTarget`. The structure stores an `swu_failure_reason_t` (p. 152) constant, which specifies a library-defined failure category, as well as an `swu_failure_code_t` (p. 158), which contains a customer-specific failure code.

swu_manifest_id_t

Unique ID of a manifest successfully parsed by library

Synopsis:

```
#include <swu/Common.h>

typedef uint32_t swu_manifest_id_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_manifest_id_t` data type stores the unique ID of a manifest successfully parsed by the library.

swu_progress_t

Percentage-based progress indicator

Synopsis:

```
#include <swu/Common.h>

typedef uint8_t swu_progress_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_progress_t` data type represents operation progress as an integer between `0` and `100` that indicates percentage of completion. Operations that can be measured this way include update installations and verifications.

swu_string_t

Reference-counted string stored in library

Synopsis:

```
#include <swu/Common.h>

typedef char* swu_string_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_string_t` data type represents a referenced-counted string stored in the library. When working with an `swu_string_t` returned by an API call, you must call *swu_object_retain()* (p. 165) (unless otherwise noted) to increase the reference count and maintain access to the string variable. Any string that has its reference count increased this way must be released later with *swu_object_release()* (p. 165) when no longer needed.

swu_target_t

Handle of an `UpdateTarget`

Synopsis:

```
#include <swu/Common.h>

typedef void* swu_target_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_target_t` data type stores the handle of an `UpdateTarget`, which is a type of `swu_object`. In the SWU library API, `UpdateTarget` objects are represented by `swu_target_t` handles. Because each `UpdateTarget` object is reference-counted, you must call *swu_object_retain()* (p. 165) to increase the reference count and maintain access to an `UpdateTarget`. Any `swu_target_t` that has its reference count increased this way must be released later with *swu_object_release()* (p. 165) when no longer needed.

swu_target_id_t

Unique ID of an `UpdateTarget`

Synopsis:

```
#include <swu/Common.h>

typedef uint32_t swu_target_id_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_target_id_t` data type stores the unique ID of an `UpdateTarget`. This ID is unique among all `UpdateTarget` objects registered with a given `UpdateClient`.

The `UpdateTarget` ID isn't unique across power cycles.

swu_target_sw_information_t

Stores information about an `UpdateTarget`

Synopsis:

```
typedef struct {
    size_t size;
    char vendor_id[SWU_UPDATE_TARGET_VENDOR_ID_LEN + 1];
    char hardware_id[SWU_UPDATE_TARGET_HARDWARE_ID_LEN + 1];
    char serial_number[SWU_UPDATE_TARGET_SERIAL_NUM_LEN + 1];
    char bom_version[SWU_UPDATE_TARGET_BOM_VERSION_LEN + 1];
} swu_target_sw_information_t;
```

Data:

size_t size

Size of the structure. This field should be set by calling `sizeof(swu_target_sw_information_t)` before using the structure in an SWU library API call.

char vendor_id[SWU_UPDATE_TARGET_VENDOR_ID_LEN + 1]

Vendor ID, used with the value in *hardware_id* to uniquely identify the `UpdateTarget` when it's matched with an `Update`.

char hardware_id[SWU_UPDATE_TARGET_HARDWARE_ID_LEN + 1]

Hardware ID, used with the value in *vendor_id* to uniquely identify the `UpdateTarget` when it's matched with an `Update`.

char serial_number[SWU_UPDATE_TARGET_SERIAL_NUM_LEN + 1]

Serial number of the `UpdateTarget`. The library integrator can decide how to use this field.

char bom_version[SWU_UPDATE_TARGET_BOM_VERSION_LEN + 1]

Version of the `UpdateTarget`. The library integrator can decide how to use this field.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_target_sw_information_t` structure stores information describing the software update contained in an `UpdateTarget`. The *vendor_id* and *hardware_id*

fields identify the `UpdateTarget` that an `Update` is meant for, so the combination of values in these fields must be unique among all `UpdateTarget` objects registered with the library.

swu_timestamp_t

UNIX timestamp

Synopsis:

```
#include <swu/Common.h>

typedef int64_t swu_timestamp_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_timestamp_t` data type stores a UNIX timestamp, which indicates the number of seconds since the start of 1970 (i.e., `1/1/1970 UTC`).

swu_update_t

Handle of an `Update`

Synopsis:

```
#include <swu/Common.h>

typedef void* swu_update_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_t` data type stores the handle of an `Update`, which is a type of `swu_object`. In the SWU library API, `Update` objects are represented by `swu_update_t` handles. Because each `Update` object is reference-counted, you must call *swu_object_retain()* (p. 165) to increase the reference count and maintain access to an `Update`. Any `swu_update_t` that has its reference count increased this way must be released later with *swu_object_release()* (p. 165) when no longer needed.

swu_update_id_t

Unique ID of an `Update`

Synopsis:

```
#include <swu/Common.h>

typedef swu_string_t swu_update_id_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_id_t` data type stores the unique ID of an `Update`. This ID is unique among all `Update` objects.

swu_update_list_t

Handle of an `UpdateList`

Synopsis:

```
#include <swu/Common.h>

typedef void* swu_update_list_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_list_t` data type stores a handle to an `UpdateList`, which represents a list of `Update` objects.

swu_uri_t

Standard URI string

Synopsis:

```
#include <swu/Common.h>

typedef swu_string_t swu_uri_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_uri_t` data type represents a standard URI string.

Functions in Common.h

Functions defined in `Common.h` for adjusting object reference counts and for returning string versions of enumerated constants.

swu_object_release()

Release an swu_object previously returned by another API call

Synopsis:

```
#include <swu/Common.h>

void swu_object_release( void *object )
```

Arguments:

object

An *swu_object* returned from an earlier API call.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Release an *swu_object* previously returned by another API call. This function decrements the reference counts of objects returned in calls to the SWU library API. Such objects include *swu_string_t*, *swu_update_t*, *swu_target_t*, and others.

To maintain access to an `swu-core` object, you must call *swu_object_retain()* (p. 165) to increase the object's reference count. Any object that has its reference count increased this way must be released later with *swu_object_release()* when no longer needed.

swu_object_retain()

Retain an swu_object previously returned by another API call

Synopsis:

```
#include <swu/Common.h>

void swu_object_retain( void *object )
```

Arguments:

object

An *swu_object* returned from an earlier API call.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Retain an *swu_object* previously returned by another API call. This function increments the reference counts of objects returned in calls to the SWU library API. Such objects include *swu_string_t*, *swu_update_t*, *swu_target_t*, and others.

You must call this function to increase an object's reference count if you want to maintain access to that object. Any object that has its reference count increased this way must be released later with *swu_object_release()* (p. 165) when no longer needed.

swu_result_to_string()

Return a string representation of an swu_result_t constant

Synopsis:

```
#include <swu/Common.h>
const char* swu_result_to_string( swu_result_t result )
```

Arguments:

result

The *swu_result_t* code whose string representation is requested by the caller.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Return a null-terminated string representation of the specified *swu_result_t* (p. 153) enumeration constant. This constant refers to one of many possible outcomes from an API call.

Returns:

A string version of the specified API call outcome.

swu_update_state_to_string()

Return a string representation of an swu_update_state_t constant

Synopsis:

```
#include <swu/Common.h>

const char* swu_update_state_to_string(
                          swu_update_state_t state )
```

Arguments:

state

The *swu_update_state_t* code whose string representation is requested by the caller.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Return a null-terminated string representation of the specified *swu_update_state_t* (p. 155) enumeration constant. This constant refers to one of many possible states for a software update.

Returns:

A string version of the specified update state.

Update.h

The `Update.h` header file defines data types for storing update notification callbacks and update state bitmasks and defines functions for registering those callbacks, reading update information and progress, and accepting or declining updates.

The `swu_update_notifications_t` structure stores callback functions for handling update state changes and progress reports. The functions exposed in the same header file allow you to:

- Register and unregister a set of notification callbacks for a specific update
- Read the ID, name, target software version, and other information fields from an update
- Accept, defer, or decline updates

- Retrieve an update's installation and verification progress and failure information (if present)

Data types in Update.h

Data types defined in `Update.h` for specifying update state masks related to notifications and for assigning notification callback functions and data pointers to pass to these functions.

swu_update_notifications_t

Structure containing notification callbacks and their context pointers

Synopsis:

```
typedef struct {

    void (*progress)
                ( swu_update_t update,
                  swu_progress_t percent,
                  void *progress_context );

    void *progress_context;

    void (*state_changed)
                ( swu_update_t update,
                  swu_update_state_t state,
                  void *state_changed_context );

    swu_update_state_mask_t state_mask;

    void *state_changed_context;

} swu_update_notifications_t;
```

Data:

void (*progress) (swu_update_t update, swu_progress_t percent, void *progress_context)

Callback function for processing notifications of progress in the update installation or verification.

void *progress_context

Pointer to user-supplied data that will be passed to the *progress* function.

void (*state_changed) (swu_update_t update, swu_update_state_t state, void *state_changed_context)

Callback function for processing notifications of update state changes.

The *state_changed* function is called for the first time when the notifications structure is registered in a call to *swu_update_register_notifications()* (p. 195).

swu_update_state_mask_t state_mask

Mask specifying which update states to be notified of when the update transitions to one of those states.

void *state_changed_context

Pointer to user-supplied data that will be passed to the *state_changed* function.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_notifications_t` structure defines the notification callbacks for handling progress updates or update state changes as well as the context pointers to pass to those callbacks. Setting a notification callback pointer to NULL prevents that notification from being sent. Setting a context pointer to NULL means no user-supplied data will be passed to the corresponding callback function.

You must register the callbacks specified in the `swu_update_notification_t` structure by calling *swu_update_register_notifications()* (p. 195), to ensure they will run in response to progress updates or update state changes. When you no longer need to run the callbacks, you should unregister them by calling *swu_update_unregister_notifications()* (p. 197).

swu_update_state_mask_t

Mask for selecting update states for which to send notifications

Synopsis:

```
#include <swu/Update.h>

typedef swu_update_state_t swu_update_state_mask_t;
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_state_mask_t` data type stores a mask that specifies the exact set of update states for which notifications will be sent. You can assign one or many `swu_update_t` (p. 155) constants to the mask to request notifications when an `Update` transitions into one of the corresponding states.

Functions in Update.h

Functions defined in `Update.h` for retrieving information about update configurations and targets, checking update progress, registering and unregistering notifications, and accepting and declining updates.

swu_update_accept_install()

Accept the installation of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_accept_install(
                                    swu_update_t update )
```

Arguments:

update

Handle of the `Update`.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Accept the installation of the software update referenced in *update*. If the library successfully processes this request, the `Update` state transitions to `SWU_UPDATE_STATE_INSTALLING`.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_compare_to_id()

Compare an ID with the ID of an `Update`

Synopsis:

```
#include <swu/Update.h>

int32_t swu_update_compare_to_id(
                              swu_update_t update,
                              swu_update_id_t id )
```

Arguments:

update

Handle of the `Update`.

id

ID to compare with the `Update` ID.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Compare the ID specified in *id* with the ID of the `Update` specified in *update*. This function is useful when searching the `UpdateList` for a specific `Update` or in any other case where simply comparing `swu_update_t` handles is inadequate.

Returns:

A signed integer value indicating how the IDs compare:

>0

The specified ID is greater than the ID of the `Update`.

0

The specified ID equals the ID of the `Update`.

<0

The specified ID is less than the ID of the `Update`.

swu_update_decline_install()

Decline the installation of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_decline_install(
                                    swu_update_t update )
```

Arguments:

update

Handle of the `Update`.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Decline the installation of the software update referenced in *update* (if the update allows it). If the library successfully processes this request, the `Update` state transitions to `SWU_UPDATE_STATE_DECLINED`.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

`SWU_RESULT_ERROR`

Another error occurred.

swu_update_defer_install()

Defer the installation of a software update for a specified amount of time

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_defer_install(
                                    swu_update_t update,
```

```
                                        swu_timestamp_t defer_period
)
```

Arguments:

update

Handle of the `Update`.

defer_period

Time period, in seconds, to defer the update installation.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Defer the installation of the software update referenced in *update* (if the update allows it). This function defers the update for the amount of time specified in *defer_period*.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_get_base_version()

Get the base version of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_base_version(
                                        swu_update_t update,
                                        swu_string_t *version )
```

Arguments:

update

Handle of the `Update`.

version

Pointer to the string to store the version.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the base version of the software update referenced in *update*. This function retrieves the base version specified when the `Update` was created and stores it in *version*. Note that the version can't be changed after the `Update` has been created. An `UpdateTarget` should call this function to validate its update before attempting to install it.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_can_be_declined()

Get flag indicating whether an update can be declined

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_can_be_declined(
                                    swu_update_t update,
                                    bool *can_be_declined )
```

Arguments:

update

Handle of the `Update`.

can_be_declined

Pointer to a Boolean to store whether the update can be declined.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the flag indicating whether the HMI allows the user to decline installation of the software update specified in *update*. This function stores the flag setting in *can_be_declined*. This flag was set when the `Update` object was created and can't be changed afterwards.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_can_be_deferred()

Get flag indicating whether an update can be deferred

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_can_be_deferred(
                                    swu_update_t update,
                                    bool *can_be_deferred )
```

Arguments:

update

Handle of the `Update`.

can_be_deferred

Pointer to a Boolean to store whether the update can be deferred.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the flag indicating whether the HMI allows the user to defer installation of the software update specified in *update*. This function stores the flag setting in *can_be_deferred*. This flag was set when the `Update` object was created and can't be changed afterwards.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_defer_period()

Get the maximum deferral period for accepting a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_defer_period(
                                    swu_update_t update,
                                    swu_timestamp_t *period )
```

Arguments:

update

Handle of the `Update`.

period

Pointer to the `swu_timestamp_t` to store the deferral period.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the maximum deferral period for accepting the software update specified in *update*. This function retrieves the deferral period specified when the `Update` was created and stores it in *period*. Note that the deferral period can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_description()

Get the full description of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_description(
                                    swu_update_t update,
                                    swu_string_t *description )
```

Arguments:

update

Handle of the `Update`.

description

Pointer to the string to store the long description.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the full description of the software update referenced in *update*. This function retrieves the long description specified when the `Update` was created and stores it in *description*. Note that the long description can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_failure_info()

Get failure information for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_failure_info(
                                    swu_update_t update,
                                    swu_failure_info_t *info )
```

Arguments:

update

Handle of the `Update`.

info

Pointer to the `swu_failure_info_t` to store the latest failure information for the update. The library fills in the fields in this structure.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the latest failure information for the software update referenced in *update*. This function stores this information in the structure referred to by *info*. The failure information fields aren't filled in until the `Update` is in the `SWU_UPDATE_STATE_INSTALL_FAILED` state.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_get_grace_period()

Get the grace period for accepting a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_grace_period(
                                    swu_update_t update,
                                    swu_timestamp_t *period )
```

Arguments:

update

Handle of the `Update`.

period

Pointer to the `swu_timestamp_t` to store the version.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the grace period for accepting the software update referenced in *update*. This function retrieves the grace period specified when the `Update` was created and stores it in *period*. Note that the grace period can't be changed after the `Update` has been created.

If you didn't set a grace period for this specific `Update`, the default value will be used. The library defines a default grace period of seven days, but this can be overridden in the manifest file. Furthermore, you can call *swu_client_configuration_set_update_grace_period()* (p. 149) to change the default grace period after the library is initialized. The value specified with this client configuration function will then become the default for all updates, overriding any value set by the library or manifest file.

In this release, the grace period has no impact on how the SWU library handles updates. The grace period is just kept as metadata describing an update (because this value can be set in the manifest file).

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_id()

Get the unique identifier of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_id(
                              swu_update_t update,
                              swu_update_id_t *id )
```

Arguments:

update

Handle of the `Update`.

id

Pointer to the `swu_update_id_t` to store the ID.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the unique identifier of the software update referenced in *update*. This function retrieves the unique ID assigned when the `Update` was created and stores it in *id*. The ID never changes during the same power cycle but can change across power cycles.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_install_source_location()

Get the location from which a software update will be installed

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_install_source_location(
                                        swu_update_t update,
                                        swu_uri_t *location )
```

Arguments:

update

Handle of the `Update`.

location

Pointer to the `swu_uri_t` to store the location (i.e., software update path).

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the location from which the software update referenced in *update* will be installed. This function retrieves the software update path specified when the `Update` was created and stores it in *location*. Note that the path can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_install_percent_completed()

Get the percentage of the installation completed for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_install_percent_completed(
                        swu_update_t update,
                        swu_progress_t *percent_completed )
```

Arguments:

update

Handle of the `Update`.

percent_completed

Pointer to the `swu_progress_t` to store the percentage of the update currently installed.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the percentage of the installation completed for the software update referenced in *update*. This function stores this percentage in *percent_completed*. Note that when the `Update` isn't in the `SWU_UPDATE_STATE_INSTALLING` state, the value retrieved by this function is undetermined.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_get_manifest_id()

Get the manifest ID associated with an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_manifest_id(
                              swu_update_t update,
                              swu_manifest_id_t *manifest_id )
```

Arguments:

update

Handle of the `Update`.

manifest_id

Pointer to the `swu_manifest_id_t` to store the manifest ID.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the manifest ID associated with the `Update` specified in *update*. This function stores this ID in *manifest_id*. You can then read this field to determine which manifest was used to create the `Update`.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_name()

Get the name of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_name(
                                 swu_update_t update,
                                 swu_string_t *name )
```

Arguments:

update

Handle of the `Update`.

name

Pointer to the string to store the name.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the name of the software update referenced in *update*. This function retrieves the name specified when the `Update` was created and stores it in *name*. Note that the name can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_post_install_command()

Get the post-installation command for an update

Synopsis:

```
#include <swu/Update.h>
```

```
swu_result_t swu_update_get_post_install_command(
                                    swu_update_t update,
                                    swu_string_t *command )
```

Arguments:

update

Handle of the `Update`.

command

Pointer to the string to store the command that the `UpdateTarget` is meant to run after installing the update.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the post-installation command for the software update referenced in *update*. This command is meant to be run by the `UpdateTarget` after it has installed the update. The function stores this command in *command*. If no such command was provided in the manifest file, the function returns `SWU_RESULT_SUCCESS` but stores an empty string in *command*.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_priority()

Get the priority of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_priority(
                                swu_update_t update,
                                swu_update_priority_t *priority )
```

Arguments:

update

Handle of the `Update`.

priority

Pointer to the `swu_update_priority_t` to store the priority.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the priority of the software update referenced in *update*. This function retrieves the priority specified when the `Update` was created and stores it in *priority*. Note that the priority can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_pre_install_command()

Get the pre-installation command for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_pre_install_command(
                                    swu_update_t update,
                                    swu_string_t *command )
```

Arguments:

update

Handle of the `Update`.

command

Pointer to the string to store the command that the `UpdateTarget` is meant to run before installing the update (i.e., during the update preparation phase).

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the pre-installation command for the software update referenced in *update*. This command is meant to be run by the `UpdateTarget` before it installs the update, while it prepares for installation. The function stores this command in *command*. If no such command was provided in the manifest file, the function returns `SWU_RESULT_SUCCESS` but stores an empty string in *command*.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_prompt_to_install()

Get flag indicating whether the user should be prompted to install an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_prompt_to_install(
                                    swu_update_t update,
                                    bool *prompt )
```

Arguments:

update

Handle of the `Update`.

prompt

Pointer to Boolean to store whether the user should be prompted for this update.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the flag indicating whether the HMI should prompt the user to accept the installation of the software update referenced in *update*. This function stores the flag setting in *prompt*. This flag was set when the `Update` object was created and can't be changed afterwards.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_short_description()

Get the descriptive summary of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_short_description(
                                    swu_update_t update,
                                    swu_string_t *description )
```

Arguments:

update

Handle of the `Update`.

description

Pointer to the string to store the short description.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the descriptive summary of the software update referenced in *update*. This function retrieves the short description specified when the `Update` was created and stores it in *description*. Note that this description can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_state()

Get the current state of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_state(
                              swu_update_t update,
                              swu_update_state_t *state )
```

Arguments:

update

Handle of the `Update`.

state

Pointer to the `swu_update_state_t` to store the state.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the current state of the software update referenced in *update*. This function stores the update state in *state*.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_version()

Get the version of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_version(
                                    swu_update_t update,
                                    swu_string_t *version )
```

Arguments:

update

Handle of the `Update`.

version

Pointer to the string to store the version.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the version of the software update referenced in *update*. This function retrieves the version specified when the `Update` was created and stores it in *version*. Note that the version can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_release_timestamp()

Get the date and time that a software update was released

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_release_timestamp(
                                    swu_update_t update,
                                    swu_timestamp_t *timestamp )
```

Arguments:

update

Handle of the `Update`.

timestamp

Pointer to the `swu_timestamp_t` to store the release timestamp.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the date and time that the software update referenced in *update* was released. This function retrieves the release timestamp specified when the `Update` was created and stores it in *timestamp*. Note that the timestamp can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

swu_update_get_size()

Get the total size, in bytes, of a software update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_size(
                                swu_update_t update,
                                size_t *size )
```

Arguments:

update

Handle of the `Update`.

size

Pointer to the `size_t` to store the update size.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the total size, in bytes, of the software update referenced in *update*. This function retrieves the update size determined when the `Update` was created and stores it in *size*. Note that the size can't be changed after the `Update` has been created.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

swu_update_get_target()

Get the handle of the `UpdateTarget` *that will install a software update*

Synopsis:

```
#include <swu/Update.h>
```

```
swu_result_t swu_update_get_target(
                                swu_update_t update,
                                swu_target_t *target )
```

Arguments:

update

Handle of the `Update`.

target

Pointer to the handle of the associated `UpdateTarget`.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the handle of the `UpdateTarget` that will install the software update referenced in *update*. This function stores the handle in *target*. To find the associated `UpdateTarget`, the library looks through its list of registered `UpdateTarget` objects and finds the one whose hardware ID and vendor ID values match those contained in the `Update`. These ID values were specified when the `Update` was created.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_FOUND

The associated `UpdateTarget` can't be found.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_get_verification_percent_completed()

Get the percentage of the verification completed for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_get_verification_percent_completed(
                            swu_update_t update,
                            swu_progress_t *percent_completed )
```

Arguments:

update

Handle of the `Update`.

percent_completed

Pointer to the `swu_progress_t` to store the percentage of the update currently verified.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the percentage of the verification completed for the software update referenced in *update*. This function stores this percentage in *percent_completed*. Note that when the `Update` isn't in the `SWU_UPDATE_STATE_VERIFYING` state, the value retrieved by this function is undetermined.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

`SWU_RESULT_ERROR`

Another error occurred.

swu_update_register_notifications()

Register a set of notification callbacks for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_register_notifications(
            swu_update_t update,
            const swu_update_notifications_t *notifications )
```

Arguments:

update

Handle of the `Update`.

notifications

A pointer to the `swu_update_notifications_t` structure containing the notification callbacks to register.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Register a set of notification callbacks for the `Update` specified in *update*. After this function returns `SWU_RESULT_SUCCESS`, the library calls the callback functions referred to in *notifications* whenever the `Update` state changes or the associated `UpdateTarget` reports progress.

The notification data isn't copied, allowing the owner of the notifications to change the data at runtime, which is the intended design. If this function is called multiple times with different pointers to notification structures, then multiple notification sets are registered. If the same pointer is used in different calls to this function, only the last set of notifications registered will remain registered.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

SWU_RESULT_ERROR

Another error occurred.

swu_update_to_string()

Create a string representation of an `Update`

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_to_string(
                              swu_update_t update,
                              char *output,
                              size_t len )
```

Arguments:

update

Handle of the `Update`.

output

Buffer to store the outputted string.

len

Length of the buffer.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Create a string representation of the `Update` specified in *update*. This function writes as much information from the `Update` as possible into a null-terminated string, which it stores in the buffer pointed to by *output*. The number of bytes written to the buffer is at most *len*. The data is written in fields, with each field on its own line and in the following format:

```
ID: UPDATE_ID
Name: This Software Update
Version: 00.00.01
```

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

`SWU_RESULT_ERROR`

Another error occurred.

swu_update_unregister_notifications()

Unregister a set of notification callbacks for an update

Synopsis:

```
#include <swu/Update.h>

swu_result_t swu_update_unregister_notifications(
            swu_update_t update,
            const swu_update_notifications_t *notifications )
```

Arguments:

update

Handle of the `Update`.

notifications

A pointer to the `swu_update_notifications_t` structure containing the notification callbacks to unregister.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Unregister a set of notification callbacks for the update referenced in *update*. After this function returns `SWU_RESULT_SUCCESS`, the caller can safely release the memory used by the *notifications* structure (because the library won't call the callbacks in this structure anymore).

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_NOT_FOUND`

The specified `swu_update_notifications_t` structure can't be found.

`SWU_RESULT_INVALID_ARGUMENT`

One of the arguments is invalid.

`SWU_RESULT_ERROR`

Another error occurred.

UpdateClient.h

The `UpdateClient.h` header file defines prototypes of functions that interact with SWU library objects and specifies functions that set up the library, prepare software updates, and manage change notifications related to library objects.

You can use the functions exposed by this module to:

- Initialize the `swu-core` library
- Create `Update` objects based on the contents of an update package
- Register and unregister listeners for changes to `UpdateList` or `TargetList` objects
- Iterate through these lists and assign callbacks to handle changes in their contents
- Log messages to `sloginfo`

Data types in UpdateClient.h

Data types defined in `UpdateClient.h` for specifying prototypes of callback functions that iterate through lists, log messages, and handle change notifications from lists.

swu_client_target_iterator_t

Callback function for iterating through the `UpdateTarget` *list*

Synopsis:

```
#include <swu/UpdateClient.h>

typedef bool( *swu_client_target_iterator_t )
                ( swu_target_t target, void *context );
```

Arguments:

target

Handle of an `UpdateTarget` object. When NULL, it indicates the end of the list.

context

A pointer set by the call to *swu_client_iterate_targets()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_client_target_iterator_t* data type defines the prototype for a callback function that iterates over the contents of the `UpdateTarget` list. When *swu_client_iterate_targets()* (p. 207) is called, this callback function gets called for each list item and then a final time with a *target* value of NULL to indicate the end of the list.

swu_client_target_notification_t

Defines a notification callback for changes to `UpdateTarget` *list*

Synopsis:

```
typedef struct {
    void (*change_notifier)(void *context);
    void *context ;
} swu_client_target_notification_t;
```

Data:

void (*change_notifier)(void *context)

Callback function that runs in response to changes to the `UpdateTarget` list. When this structure is first registered with the library, the callback function is called to inform the client of the current list contents.

void *context

Context pointer used when the *change_notifier* function is called.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_client_target_notification_t` structure stores the pointer to a callback function that processes notification changes related to the `UpdateTarget` list as well as a context pointer that stores additional information.

swu_logging_callback_t

Callback function to log messages generated by the library

Synopsis:

```
#include <swu/UpdateClient.h>

typedef void( *swu_logging_callback_t )
               ( swu_log_level_t level,
                 const char *message,
                 va_list msg_args );
```

Arguments:

level

Severity level of the message.

message

A null-terminated string containing the message to log.

msg_args

Variable argument list for formatting values within the message text.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_logging_callback_t* data type defines the prototype for a callback function that must be registered with *swu_client_set_logging_callback()* to receive log messages from the library. These messages may be errors, warnings, or debugging information. The callback function can set the *level* parameter to filter the messages by severity.

swu_update_list_iterator_t

Callback function for iterating through an `UpdateList`

Synopsis:

```
#include <swu/UpdateClient.h>
```

```
typedef bool( *swu_update_list_iterator_t )
                ( swu_update_t update, void *context );
```

Arguments:

update

Handle of the current `Update` object in the list. When NULL, it indicates the end of the list.

context

A pointer set by the call to *swu_update_list_iterate()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The *swu_update_list_iterator_t* data type defines the prototype for a callback function that iterates over the contents of an `UpdateList`. When *swu_update_list_iterate()* (p. 213) is called, this callback function gets called for each `UpdateList` item and then a final time with an *update* value of NULL to indicate the end of the list.

swu_update_list_notification_t

Defines a notification callback for changes to an `UpdateList`

Synopsis:

```
typedef struct {
    void (*change_notifier)
            (swu_update_list_t list, void *context);
    void *context ;
} swu_update_list_notification_t;
```

Data:

void (*change_notifier)(swu_update_list_t list, void *context)

Callback function that runs in response to changes to an `UpdateList`. When the structure is first registered with the library, the callback function is called to inform the client of the current list contents.

void *context

Context pointer used when the *change_notifier* function is called.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

The `swu_update_list_notification_t` structure stores the pointer to a callback function that processes notification changes related to an `UpdateList` as well as a context pointer that stores additional information.

Functions in UpdateClient.h

Functions defined in `UpdateClient.h` for initializing the SWU library, creating and managing `Update` and `UpdateList` objects, informing the client if the current system can be updated, and registering and unregistering listeners for list change notifications.

swu_client_create_updates()

Create `Update` *objects by reading a manifest file*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_create_updates(
                                const char *path,
                                swu_manifest_id_t *id )
```

Arguments:

path

Path of the update package.

id

Unique ID to identify the manifest file. This ID can be used later to release the `Update` objects created with *swu_client_release_updates()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Create `Update` objects based on the manifest file found at the location in *path*. The manifest file stores software update information in the `.ini` file format (for details, see the *Manifest file* (p. 139) section). The `Update` objects created are associated with the manifest file ID specified in *id*.

If the library fails to parse any updates listed in the manifest file, the overall operation is considered to have failed and the function returns an SWU_RESULT_ERROR code.

Returns:

One of the following swu_result_t values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

One of the arguments is invalid.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_conditions_invalid_for_installs()

Inform the UpdateClient *that updates currently can't be installed*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_conditions_invalid_for_installs(
                                                    void )
```

Library:

libswu-core

Use the -l swu-core option with qcc to link against the SWU library. This library is usually included automatically.

Description:

Inform the UpdateClient that the current system conditions are invalid for installing updates. Use this function to inform the library that the system currently can't accept update installations. For example, you could call this function to disable updates when the device is running on battery power.

Returns:

One of the following swu_result_t values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_conditions_valid_for_installs()

Inform the `UpdateClient` *that updates can be installed*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_conditions_valid_for_installs(
                                                    void )
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Inform the `UpdateClient` that the current system conditions allow for installing updates. Use this function to inform the library that the system can accept update installations. For example, you could call this function to enable updates when the device is no longer running on battery power.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_get_install_update_list()

Get a handle to the list of updates available to install or being installed

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_get_install_update_list(
                                    swu_update_list_t *list )
```

Arguments:

list

Pointer to an `UpdateList` handle. This pointer is set by the library.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get a handle to the list of updates available to install or being installed.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_NOT_INITIALIZED`

The library hasn't been initialized yet.

`SWU_RESULT_ERROR`

Another error occurred.

swu_client_get_target_list_length()

Get the number of items in the `UpdateTarget` *list*

Synopsis:

```
#include <swu/UpdateClient.h>
```

```
swu_result_t swu_client_get_target_list_length(
                                                size_t *length )
```

Arguments:

length

On output, the length of the list.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the number of items in the `UpdateTarget` list. The function stores the list length (i.e., the number of targets) in *length*.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_client_initialize()

Initialize the library

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_initialize( const char *client_id )
```

Arguments:

client_id

Unique ID for identifying the client.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Initialize the library. This function must be called before any other SWU library API function to set up the `swu-core` library. The initialization function also assigns a unique ID for the `UpdateClient`.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The library was successfully initialized.

SWU_RESULT_ERROR

The library couldn't be initialized.

swu_client_iterate_targets()

Iterate over the `UpdateTarget` *list contents*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_iterate_targets(
                swu_client_target_iterator_t iterator,
                void *context )
```

Arguments:

iterator

Callback function to use for iterating over the list contents.

context

Context pointer that will be passed to the callback function.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Iterate over the list of registered update targets, which are represented as `UpdateTarget` objects. After a call to this function completes successfully, the library calls the callback function specified in *iterator* once for each list item and then a final time with an `swu_target_t` value of NULL to indicate the end of the list.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_client_register_target_list_notification()

Register a listener to receive notifications about changes to the `UpdateTarget` *list*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_register_target_list_notification(
      const swu_client_target_notification_t *notification )
```

Arguments:

notification

Pointer to an `swu_client_target_notification_t` (p. 199) structure that defines how to notify a listener of changes to the `UpdateTarget` list.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Register a listener to receive notifications about changes to the `UpdateTarget` list. This function informs the `UpdateClient` about a new listener interested in the

`UpdateTarget` list contents. The function is called when a new `UpdateTarget` is registered with the library or when an existing one is unregistered.

The notification structure isn't copied, so it's expected that the caller maintains this structure. After the notification is successfully registered, the callback function referred to in *notification* gets called.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_client_release_updates()

Release any updates associated with a certain manifest ID

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_release_updates(
                                swu_manifest_id_t id )
```

Arguments:

id

Manifest ID set in an earlier call to *swu_client_create_updates()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Release any updates associated with the manifest ID specified in *id*.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_INITIALIZED

The library hasn't been initialized yet.

SWU_RESULT_ERROR

Another error occurred.

swu_client_set_logging_callback()

Enable logging of messages generated by the library

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_set_logging_callback(
                                swu_logging_callback_t log_func )
```

Arguments:

log_func

The logging function to use. A NULL value causes the library to log information on serious errors to `stderr`.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Enable logging of messages generated by the library. If no logging function is specified (i.e., *log_func* is NULL), all messages with a severity level of at least `SWU_LOG_WARNING` are logged to `stderr`.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_ERROR

An error occurred.

swu_client_uninitialize()

Release the memory held by the library

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_uninitialize()
```

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Release the memory held by the library. This function must be called to free the environment when the library is no longer being used.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_ERROR

An error occurred.

swu_client_unregister_target_list_notification()

Unregister a listener from receiving notifications about changes to the `UpdateTarget` *list*

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_client_unregister_target_list_notification(
      const swu_client_target_notification_t *notification )
```

Arguments:

notification

Pointer to the `swu_client_target_notification_t` (p. 199) structure used in the earlier call to *swu_client_register_target_list_notification()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Unregister a listener from receiving notifications about changes to the `UpdateTarget` list. This function informs the `UpdateClient` that the listener is no longer interested in the `UpdateTarget` list contents.

You must pass in a pointer to the same `swu_client_target_notification_t` structure that you used to register for notifications when calling *swu_client_register_target_list_notification()*. At this point, the notification interface registered in this earlier call is no longer needed.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_FOUND

The structure referred to in *notification* couldn't be found.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_update_list_get_length()

Get the number of `Update` *objects in an* `UpdateList`

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_update_list_get_length(
                              swu_update_list_t list,
                              size_t *length )
```

Arguments:

list

Handle of an `UpdateList` whose length is requested by the caller.

length

On output, the length of the list.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the number of `Update` objects in the `UpdateList` specified in *list*. The function stores the list length (i.e., the number of objects) in *length*.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_update_list_iterate()

Iterate over the contents of an `UpdateList`

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_update_list_iterate(
                        swu_update_list_t list,
                        swu_update_list_iterator_t iterator,
                        void *context)
```

Arguments:

list

Handle of an `UpdateList` to iterate through.

iterator

Callback function to use for iterating over the list contents.

context

Context pointer that will be passed to the callback function.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Iterate over the contents of the `UpdateList` specified in *list*. After a call to this function completes successfully, the library calls the callback function specified in *iterator* once for each list item (i.e., `Update` object) and then a final time with an `swu_update_t` value of NULL to indicate the end of the list.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_update_list_register_notification()

Register a listener to receive notifications about changes to an `UpdateList`

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_update_list_register_notification(
        swu_update_list_t list,
        const swu_update_list_notification_t *notification )
```

Arguments:

list

Handle of an `UpdateList` for which the caller wants to receive notifications of content changes.

notification

Pointer to an swu_update_list_notification_t (p. 201) structure that defines how to notify a listener of changes to the UpdateList.

Library:

libswu-core

Use the -l swu-core option with qcc to link against the SWU library. This library is usually included automatically.

Description:

Register a listener to receive notifications about changes to an UpdateList. This function informs the UpdateClient about a new listener interested in the contents of the UpdateList referenced in *list*.

The notification structure isn't copied, so the caller is expected to maintain this structure. After the notification is successfully registered, the callback function referred to in *notification* gets called.

Returns:

One of the following swu_result_t values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_update_list_unregister_notification()

Unregister a listener from receiving notifications about changes to an UpdateList

Synopsis:

```
#include <swu/UpdateClient.h>

swu_result_t swu_update_list_unregister_notification(
        swu_update_list_t list,
        const swu_update_list_notification_t *notification )
```

Arguments:

list

Handle of the `UpdateList` for which the caller no longer needs to receive notifications.

notification

Pointer to the `swu_update_list_notification_t` (p. 201) structure used in the earlier call to *swu_update_list_register_notification()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Unregister a listener from receiving notifications about changes to an `UpdateList`. This function informs the `UpdateClient` that the listener is no longer interested in content changes to the `UpdateList` referred to in *list*.

You must pass in a pointer to the same `swu_update_list_notification_t` structure that you used to register for notifications when calling *swu_update_list_register_notification()*. At this point, the notification interface registered in this earlier call is no longer needed.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_NOT_FOUND`

The structure referred to in *notification* couldn't be found.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

UpdateTarget.h

The `UpdateTarget.h` header file defines functions that read `UpdateTarget` information.

Functions in UpdateTarget.h

Functions defined in `UpdateTarget.h` for reading IDs and other `UpdateTarget` information.

swu_target_get_id()

Get the ID of an `UpdateTarget`

Synopsis:

```
#include <swu/UpdateTarget.h>

swu_result_t swu_target_get_id( swu_target_t target,
                                swu_target_id_t *id )
```

Arguments:

target

Handle of an `UpdateTarget` whose ID is requested by the caller.

id

A valid target ID on success.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get the ID assigned to the `UpdateTarget` specified in *target*. The ID is assigned to the `UpdateTarget` when it calls *swu_target_register()* (p. 228). Each `UpdateTarget` is given a different ID by the library, but this ID isn't unique across power cycles.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

The target ID was invalid.

swu_target_get_info()

Get information about an `UpdateTarget`

Synopsis:

```
#include <swu/UpdateTarget.h>

swu_result_t swu_target_get_info(
                          swu_target_t target,
                          swu_target_sw_information_t *info )
```

Arguments:

target

Handle of an `UpdateTarget` whose information is requested by the caller.

info

Pointer to an `swu_target_sw_information_t` (p. 162) structure that will store the target information.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Get information about the `UpdateTarget` referred to by *target*. This function copies the target's information into the structure whose address is given in *info*. The information includes the ID values of the target, the version of the software installed on it, and more.

When *swu_target_get_info()* is called, the library calls the function referenced in the *get_info* field of the `swu_target_interface_t` (p. 219) structure associated with the `UpdateTarget` and then passes this function's return value back to the caller.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_INVALID_ARGUMENT`

The target ID was invalid.

Return code from `get_info` function

When a valid ID is given, the function returns the result code it receives from the `get_info` function supplied by the `UpdateTarget`.

UpdateTargetInterface.h

The `UpdateTargetInterface.h` header file specifies the interface for the `UpdateClient` to talk an `UpdateTarget` and defines functions that allow an `UpdateTarget` to register itself with the library and to report the progress and outcomes of update installations and verifications.

Data types in UpdateTargetInterface.h

The `UpdateTargetInterface.h` header file defines, in a structure, the `UpdateTarget` interface used by the `UpdateClient` to request specific actions in the update process.

swu_target_interface_t

Interface used by the `UpdateClient` *to make requests of an* `UpdateTarget`

Synopsis:

```
typedef struct {

    swu_result_t (*get_info)
                    ( swu_target_id_t id,
                      swu_target_sw_information_t *info,
                      void *get_info_context );

    void *get_info_context;

    swu_result_t (*prepare_to_install)
                    ( swu_target_id_t id,
                      swu_update_t update,
                      void *prepare_to_install_context );

    void *prepare_to_install_context;

    swu_result_t (*install)
                    ( swu_target_id_t id,
                      swu_update_t update,
                      void *install_context );

    void *install_context;

    swu_result_t (*cancel_install)
                    ( swu_target_id_t id,
                      swu_update_t update,
                      void *cancel_install_context );

    void *cancel_install_context;

    swu_result_t (*verify_update)
                    ( swu_target_id_t id,
                      swu_update_t update,
                      void *verify_update_context );

    void *verify_update_context;
```

```
    swu_result_t (*rollback_update)
                    ( swu_target_id_t id,
                      swu_update_t update,
                      void *rollback_update_context );

    void *rollback_update_context;

} swu_target_interface_t;
```

Data:

swu_result_t (*get_info) (swu_target_id_t id, swu_target_sw_information_t *info, void *get_info_context)

Pointer to a function that the `UpdateClient` can call to retrieve ID and software version information from an `UpdateTarget`.

This function is primarily used when the application calls *swu_target_get_info()* (p. 218), which always calls the *get_info* function. However, the `UpdateClient` may call this latter function at other times in the software update process.

void *get_info_context

Context pointer to use as a parameter when the *get_info* function is called.

swu_result_t (*prepare_to_install) (swu_target_id_t id, swu_update_t update, void *prepare_to_install_context)

Pointer to a function that the `UpdateClient` can call to inform an `UpdateTarget` that an update is available and that it should prepare for the update.

The `UpdateClient` calls this function after the user accepts an update that's ready for installation. The *prepare_to_install* function allows the `UpdateTarget` to determine if it's in a state suitable for updates and, if so, to prepare for an update installation.

If the `UpdateTarget` is ready, it should respond by calling *swu_target_ready_to_install()* (p. 227) from its own context. If it's not ready, it should call *swu_target_not_ready_to_install()* (p. 226).

void *prepare_to_install_context

Context pointer to use as a parameter when the *prepare_to_install* function is called.

swu_result_t (*install) (swu_target_id_t id, swu_update_t update, void *install_context)

Pointer to a function that the `UpdateClient` can call to tell an `UpdateTarget` to start installing an update.

As soon as the `UpdateTarget` returns from this function, it can begin the installation. As the installation progresses, the `UpdateTarget` can call *swu_target_install_progress()* (p. 224) to indicate how far the installation has progressed, but this reporting is optional. The decision to track and report installation progress (and how often to do this) is an implementation detail.

void *install_context

Context pointer to use as a parameter when the *install* function is called.

swu_result_t (*cancel_install) (swu_target_id_t id, swu_update_t update, void *cancel_install_context)

(Currently unused)

Pointer to a function that the `UpdateClient` can call to request an `UpdateTarget` to cancel an ongoing update installation.

void *cancel_install_context

(Currently unused)

Context pointer to use as a parameter when the *cancel_install* function is called.

swu_result_t (*verify_update) (swu_target_id_t id, swu_update_t update, void *verify_update_context)

Pointer to a function that the `UpdateClient` can call to tell an `UpdateTarget` to verify that an update was installed correctly.

The library calls this function after an update finishes installing. The `UpdateTarget` can then perform any post-installation verification steps defined for it. As with the other function pointers in this structure, *verify_update* should be set to NULL if the `UpdateTarget` doesn't support or require verification.

After the verification completes successfully, the `UpdateTarget` must call *swu_target_verification_successful()* (p. 233) to indicate the success of this operation to the library. If a problem occurs during the verification, the `UpdateTarget` should call *swu_target_verification_failed()* (p. 231) and provide the reason why the verification failed.

void *verify_update_context

Context pointer to use as a parameter when the *verify_update* function is called.

swu_result_t (*rollback_update) (swu_target_id_t id, swu_update_t update, void *rollback_update_context)

(Currently unused)

Pointer to a function that the `UpdateClient` can call to request an `UpdateTarget` to try to rollback a previously installed update.

void *rollback_update_context

(Currently unused)

Context pointer to use as a parameter when the *rollback_update* function is called.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Interface used by the `UpdateClient` to make requests of an `UpdateTarget`. Each `UpdateTarget` must register an `swu_target_interface_t` structure with the `UpdateClient` so this latter component knows how to talk to the `UpdateTarget` when requesting software update actions. Any functions not supported by the `UpdateTarget` should have their function pointers set to NULL.

Because the functions referenced in this structure are executed in the context of the library, it's expected that the `UpdateTarget` will quickly return from these functions. The `UpdateTarget` should use its own context to perform any long-running operations.

Functions in UpdateTargetInterface.h

Functions defined in `UpdateTargetInterface.h` for registering and unregistering an `UpdateTarget`, informing the library whether a target can currently be updated, and reporting the progress and outcomes of update installations and verifications.

swu_target_install_failed()

Report an update installation failure

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_install_failed(
                                    swu_target_id_t id,
```

```
                                        swu_update_t update,
                                        swu_failure_reason_t reason,
                                        swu_failure_code_t code )
```

Arguments:

id

ID of the `UpdateTarget` that failed to install the update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update that couldn't be installed on the target.

reason

Constant defined by the `swu_failure_reason_t` (p. 152) enumeration for indicating an installation failure (`SWU_FAILURE_REASON_INSTALL_FAILED`).

code

User-defined error code to help determine the issue that caused the update installation to fail. This value is only passed through and isn't used by the library, but logging this value can be handy for debugging.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report an update installation failure. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` that an error was encountered while installing the update specified in *update*. It's expected that the `UpdateTarget` call this function after returning from the `install` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_target_install_progress()

Report progress in an ongoing update installation

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_install_progress(
                                    swu_target_id_t id,
                                    swu_update_t update,
                                    swu_progress_t progress )
```

Arguments:

id

ID of the `UpdateTarget` that's reporting installation progress. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update currently being installed.

progress

Installation progress, as a percentage of the installation that's completed.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report progress in an ongoing update installation. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` about installation progress for the update specified in *update*. Note that this progress reporting is optional. If it chooses to report progress, an `UpdateTarget` should call this function only after successfully returning from the `install` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_install_successful()

Report a successful update installation

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_install_successful(
                                    swu_target_id_t id,
                                    swu_update_t update )
```

Arguments:

id

ID of the `UpdateTarget` that successfully installed the update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update that was installed on the target.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report a successful update installation. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` that the update specified in *update* was successfully installed. It's expected that the `UpdateTarget` call this function after returning from the `install` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_not_ready_to_install()

Inform the `UpdateClient` *that a target isn't ready to install an update*

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_not_ready_to_install(
                                    swu_target_id_t id,
                                    swu_update_t update,
                                    swu_failure_reason_t reason,
                                    swu_failure_code_t code )
```

Arguments:

id

ID of the `UpdateTarget` that's not ready to install an update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object containing an update that the client is trying to install.

reason

Constant defined by the `swu_failure_reason_t` (p. 152) enumeration for indicating that the target is not ready for an update (`SWU_FAILURE_REASON_NOT_READY_FOR_UPDATE`).

code

User-defined error code to help determine the issue preventing the target from installing the update. This value is only passed through and isn't used by the library, but logging this value can be handy for debugging.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Inform the `UpdateClient` that a target isn't ready to install an update. The `UpdateTarget` indicated in *id* calls this function after determining that the target is *not* ready to install the update specified in *update*. It's expected that the `UpdateTarget` call this function after returning from the `prepare_to_install` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_ready_to_install()

Inform the `UpdateClient` *that a target is ready to install an update*

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_ready_to_install(
                                    swu_target_id_t id,
                                    swu_update_t update )
```

Arguments:

id

ID of the `UpdateTarget` that's ready to install an update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object containing an update to install on the target.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Inform the `UpdateClient` that a target is ready to install an update. The `UpdateTarget` indicated in *id* calls this function after determining that the target is ready to install the update specified in *update*. It's expected that the `UpdateTarget` call this function after returning from the `prepare_to_install` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_register()

Inform the `UpdateClient` *of a new* `UpdateTarget`

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_register(
                         const char *vendor_id,
                         const char *hardware_id,
                         swu_target_interface_t *interface,
                         swu_target_id_t *id )
```

Arguments:

vendor_id

Constant string for the target's vendor ID, used with the value in *hardware_id* to uniquely identify the `UpdateTarget`. The function makes its own copy of the string.

hardware_id

Constant string for the target's hardware ID, used with the value in *vendor_id* to uniquely identify the `UpdateTarget`. The function makes its own copy of the string.

interface

Handle of the `swu_target_interface_t` (p. 219) structure used by the `UpdateClient` to communicate with a specific `UpdateTarget`.

id

On output, the `UpdateTarget` ID assigned by the `UpdateClient`.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Inform the `UpdateClient` of a new `UpdateTarget`. The client assigns an ID to the target and uses that ID when communicating with the target. The function doesn't copy the `swu_target_interface_t` structure, which is expected to be maintained by the caller.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_DUPLICATE_ENTRY`

An `UpdateTarget` with this Vendor ID and Hardware ID pair has already been registered.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_unregister()

Inform the `UpdateClient` *that an* `UpdateTarget` *is no longer available*

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_unregister( swu_target_id_t id )
```

Arguments:

id

ID of the `UpdateTarget` to unregister. This ID was assigned by the library in the call to *swu_target_register()*.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Inform the `UpdateClient` that an `UpdateTarget` is no longer available. This function removes the `UpdateTarget` specified in *id* from the library. After this function is called, the `UpdateClient` won't call the functions referenced by the pointers in the target's associated `swu_target_interface_t` structure. At this point, this interface structure is no longer needed.

The ID assigned to the unregistered `UpdateTarget` won't be used again during the same power cycle.

Returns:

One of the following `swu_result_t` values:

SWU_RESULT_SUCCESS

The operation succeeded.

SWU_RESULT_NOT_FOUND

An `UpdateTarget` with the specified ID could not be found.

SWU_RESULT_INVALID_ARGUMENT

An invalid argument was given.

SWU_RESULT_ERROR

Another error occurred.

swu_target_verification_failed()

Report an update verification failure

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_verification_failed(
                                    swu_target_id_t id,
                                    swu_update_t update,
                                    swu_failure_reason_t reason,
                                    swu_failure_code_t code )
```

Arguments:

id

ID of the `UpdateTarget` that failed to verify the update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update whose installation couldn't be verified on the target.

reason

Constant defined by the `swu_failure_reason_t` (p. 152) enumeration for indicating a verification failure (`SWU_FAILURE_REASON_INSTALL_VERIFICATION_FAILED`).

code

User-defined error code to help determine the issue that caused the update installation to fail. This value is only passed through and isn't used by the library, but logging this value can be handy for debugging.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report an update verification failure. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` that an error occurred while verifying the installation of the update specified in *update*. It's expected that the

`UpdateTarget` call this function after returning from the `verify_update` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_verification_progress()

Report progress in an ongoing update verification

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_verification_progress(
                                    swu_target_id_t id,
                                    swu_update_t update,
                                    swu_progress_t progress )
```

Arguments:

id

ID of the `UpdateTarget` that's reporting verification progress. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update currently being verified.

progress

Verification progress, as a percentage of the verification that's completed.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report progress in an ongoing update verification. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` about verification progress for the update specified in *update*. Note that this progress reporting is optional. If it chooses to report progress, an `UpdateTarget` should call this function only after successfully returning from the `verify_update` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

swu_target_verification_successful()

Report a successful update verification

Synopsis:

```
#include <swu/UpdateTargetInterface.h>

swu_result_t swu_target_verification_successful(
                                    swu_target_id_t id,
                                    swu_update_t update )
```

Arguments:

id

ID of the `UpdateTarget` that successfully verified the update. This ID was assigned by the library in the call to *swu_target_register()*.

update

Handle of an `Update` object representing the update that was verified on the target.

Library:

`libswu-core`

Use the `-l swu-core` option with `qcc` to link against the SWU library. This library is usually included automatically.

Description:

Report a successful update verification. An `UpdateTarget` (which is referenced in *id*) calls this function to inform the `UpdateClient` that the update specified in *update* was successfully verified. It's expected that the `UpdateTarget` call this function after returning from the `verify_update` function referenced in its associated `swu_target_interface_t` structure.

Returns:

One of the following `swu_result_t` values:

`SWU_RESULT_SUCCESS`

The operation succeeded.

`SWU_RESULT_INVALID_ARGUMENT`

An invalid argument was given.

`SWU_RESULT_ERROR`

Another error occurred.

Software update daemon

The software update daemon (`swud`) contains the `swu-core` library as well as a resource manager that dynamically loads modules that use the `swu-core` API to perform software updates. These modules contain platform-specific software update functionality that extends the library's capabilities.

The `swud` service and the modules that it uses are started automatically by *SLM* (p. 247) during startup but you can explicitly start the service or manually load its modules if you want to. You can also define your own `swud` modules to customize the software update process. The QNX CAR platform comes with several reference modules that provide programming examples of how to process software update packages stored on USB devices and how to define HMI and command-line controls from which users can initiate updates.

swud

Validates update packages on attached devices, notifies HMI of pending updates, and initiates update process when requested by user.

Syntax:

```
swud -i client_id [-d] [-v] [-m module_path[=module_args]]*
```

Options:

`-d`

Run `swud` in the foreground, not as a daemon. This flag is useful for debugging.

`-i` *client_id*

(Required)

The update client ID. This ID is defined by your system integrator and is available as part of the Software Update (SWU) client configuration.

`-m` *module_path*[*=module_args*]

Load the module at *module_path*. The *=module_args* component is an optional, comma-separated list of arguments to be passed to the module.

Here's an example of launching `swud` with the `swud-self-update-hmi` module while specifying the vendor ID and hardware ID:

```
swud -m swud-self-update-hmi.so=QNX,CAR2.1
```

You can provide multiple `-m` options on the command line to load and pass parameters to as many modules as you like.

`-v`

Increase output verbosity (messages are written to `sloginfo`). The `-v` option is cumulative; each additional `v` adds a level of verbosity, up to 7 levels. For example, `swud -vvv` sets a verbosity level of `3`.

The default setting is `0`, meaning verbosity is turned off.

Description:

You should start `swud` with an explicit command only if this daemon process terminates unexpectedly or you've disabled its automatic startup in SLM. Before trying to start `swud` manually, always confirm that the process isn't already running by checking the list of active processes with *pidin* or *ps*.

The `swud` service detects manifest files stored on attached devices. When it finds a manifest file, `swud` validates this file along with the delta file named in it. If these files are valid, `swud` notifies the HMI, which can then display the pending update to the user. If either file is invalid (e.g., because of an incorrect base version), `swud` doesn't process the update further but logs an error to `slogger` and to the HMI.

When the user initiates the update through the HMI, `swud` copies the manifest file and the delta file from the attached device to `/var/swud`. Next, `swud` uses the partial-shutdown utility, `downsize`, to terminate all processes except those needed to apply the update. Finally, `swud` applies the software update based on the delta file contents.

You must provide a client ID when running `swud`, whether through SLM or the command line. The client ID is an alphanumeric string and is found in the SWU client configuration. You can optionally use the `-m` option to load modules and pass arguments to them. Also, you can run the service in the foreground for debugging purposes (by using `-d`), and enable different levels of verbosity by using between 1 and 7 `-v` options.

Loading `swud` modules

You can dynamically load `swud` modules with the `mount` command.

To load modules when `swud` is running:

1. Run the `mount` command like this: `mount -T swud [-o module_options] module_path`

   The *module_path* argument must contain either a path defined in the *LD_LIBRARY_PATH* environment variable or the absolute path of the module. The

optional *module_options* argument can contain a comma-separated list of options to pass into the swud module.

Here's an example that loads the swud-self-update-hmi module while specifying the vendor ID and hardware ID to match the self-update target:

```
mount -T swud -o QNX,CAR2.1 swud-self-update-hmi.so
```

Developing swud modules

You can customize the software update process by developing your own swud modules.

Developing custom swud modules requires implementing the *SWU module API* (p. 244) defined in swud/swu_module.h and compiling the module as a dynamically linked library (DLL). After the module has been loaded and initialized, it can start interacting with the swu-core library through its *core API* (p. 244).

Implementing a swud module

Every swud module is required to implement the *SWU_MODULE_INITIALIZE()* function from the swu_module API. The swud service calls this function when loading a module. This function should only initialize the module, start any needed threads, and then return as quickly as possible so it doesn't block swud.

The initialization function has a generic function signature, accepting an argc-argv argument list. However, these arguments aren't meant to be parsed with *getopt()*, primarily because the mount command (which swud supports) doesn't allow optarg arguments. The recommended approach for parsing arguments is to use comma-separated key-value pairs after the -o option, like this:

```
mount -T swud -o a=value,b=value,c=value
```

In this example, the argument list, as parsed by swud, would be passed into *SWU_MODULE_INITIALIZE()* like this:

```
argc = 3
argv[0] = "a=value"
argv[1] = "b=value"
argv[2] = "c=value"
argv[3] = NULL
```

Implementing *SWU_MODULE_SHUTDOWN()* is optional. Any implementation of this function must be signal handler-safe because it will be called from a signal handler in swud. This means all calls made by the shutdown function must also be signal handler-safe. The intended use for this function is to do any cleanup that must be done before swud exits, such as saving files or any other tasks needed to ensure information persistence.

Reference modules

The swud utility is packaged with a set of reference modules. Together, these components provide a reference implementation, based on the QNX CAR platform, for performing USB-based software updates.

swud-usb.so

The swud-usb.so module monitors USB mass-storage devices for software update packages and notifies the swu-core library when they're discovered. This module has limitations. For example, it loads the first manifest file it finds and looks only in the root directory of the USB device.

The module relies on the mcd service to detect when mass-storage devices are attached. The QNX CAR target image contains the following entry in /etc/mcd.conf to search for manifest files on mass-storage devices:

```
[UPDATE]
Callout         = FNAME_PATTERN
Argument        = depth=1,*.manifest
Match Rule      = INSERTED
Fail Rule       = INSERTED
```

The Argument key in this sample mcd configuration entry contains argument values matching the default arguments given to swud-usb.so. Your system integrator may configure mcd differently; if so, you should ensure that the arguments passed to swud-usb.so match the mcd configuration.

You can load the module with the following optional arguments:

swud-usb.so [*manifest_file_extension insertion_path ejection_path*]

Either all or none of these arguments must be present.

These arguments have the following meanings:

manifest_file_extension

The file extension that the module looks for to recognize manifest files. The default is .manifest.

insertion_path

The path for mcd to write notifications of inserted USB devices containing software update packages. The default is /dev/mcd/UPDATE.

ejection_path

The path for `mcd` to write notifications of ejected USB devices containing software update packages. The default is `/dev/mcd/EJECTED`.

swud-legacy-hmi.so

The `swud-legacy-hmi.so` module provides a PPS bridge between the `swu-core` library and the Settings app in the HMI. The Settings app has some functionality limits, one of which is its ability to display only the first update in the list to the user. The module publishes notifications to the HMI through the `/pps/services/update/status` object and subscribes to HMI-issued commands through the `/pps/services/update/control` object.

You can load the module with the following optional argument:

`swud-legacy-hmi.so [`*`pps_base_path`*`]`

where *pps_base_path* is the root path of the PPS subsystem. The default value is `/pps`. All PPS objects used by this module are located in *`pps_base_path`*`/services/update/`.

swud-client-config.so

The `swud-client-config.so` module allows the following parameters to be configured in the `swu-core` library using PPS:

- `localUpdatesEnabled`
- `maxUpdateRetries`
- `updateGracePeriod`

This module both publishes and subscribes to these parameters in the `/pps/services/update/settings` PPS object.

You can load the module with the following optional argument:

`swud-client-config.so [`*`pps_base_path`*`]`

Here, *pps_base_path* is the root path of the PPS subsystem. The default value is `/pps`. All PPS objects used by this module are located in *`pps_base_path`*`/services/update/`.

swud-self-update-hmi.so

The `swud-self-update-hmi.so` module provides a basic HMI that's displayed when the QNX CAR platform updates itself. The HMI shows a simple progress bar while a software update is being applied.

To specify the target to update, provide the following arguments when the module is loaded:

`swud-self-update-hmi.so` *`vendor_id hardware_id`*

These mandatory arguments have the following meanings:

vendor_id

A vendor-defined string identifying the vendor of the self-update target.

hardware_id

A vendor-defined string identifying the hardware of the self-update target.

swud-simple-self-update.so

The `swud-simple-self-update.so` module reads a manifest file, loads it, and then installs the first update defined in the file. An example use case is when you want the system to resume applying an incomplete update after it gets interrupted, for example, by an unexpected power cycle. This resumed self-update is initiated by launching `swud` with the `swud-simple-self-update.so` module during system startup.

Load this module with the following argument:

`swud-simple-self-update.so` *`manifest_path`*

where *manifest_path* is the path where the manifest file to be processed is located.

rb-self-update.so

The `rb-self-update.so` module applies a software update. You must provide a delta file, which you can obtain from your system provider or *generate using Red Bend tools* (p. 241). To update your system through the HMI, you must also provide a *manifest file* (p. 139).

You can load this module with the following optional arguments:

`rb-self-update.so [delta=`*`delta_path`*`] [temp=`*`temp_path`*`] [pps_target=`*`ppstarget`*`] [persist=`*`persist_file_path`*`]`

These arguments have the following meanings:

delta_path

The path where the delta file will be copied to. The default path is `/dos/mydelta.mld`.

temp_path

The path used by Red Bend when working with the delta file. The default path is `/dos/updAgentTmp`.

ppstarget

The path for storing the target information, which consists of the hardware ID, vendor ID, and a serial number. The default path is `/pps/services/update/target`.

persist

The output path for a persistent manifest file to support resuming the update in case of an unexpected system restart. The default path is `/dos/swu_persist.manifest`.

Generating a delta file

You can generate your own delta file to update your system to a target version.

To generate a delta file, you must have a version of the Red Bend tools with the same major revision (currently v8.x) as the Software Updates app used by the QNX CAR platform. The file contents are based on the source and target version trees. The source tree must match the version of software that's currently running on your system. The target tree is always the version of software that you want to upgrade to.

These instructions also assume that you're running the update-generation process on an Ubuntu Linux system; Windows hosts aren't currently supported.

Before you can generate the delta file, you must first obtain a configuration file to control the delta generator (this process is described in the Red Bend Integrator's Reference manual). Step *1* (p. 241) in the following procedure shows how to create your own configuration file.

To generate a delta file:

1. If necessary, create the main configuration file by copying the following content into a new XML file:

```
<vrm>
    <DeltaType>update</DeltaType>
    <ComponentDeltaFileName>
        QNXCAR2-myname-from_version-to_version.mld
    </ComponentDeltaFileName>
    <RamSize>0x20000000</RamSize>
    <Statistics>stats_from_version-to_version.txt</Statistics>
    <keepinvaliddelta>1</keepinvaliddelta>
    <partition>
        <PartitionName>apps</PartitionName>
        <PartitionType>PT_FS</PartitionType>
        <MountPoint>/</MountPoint>
        <SourceVersion>source</SourceVersion>
        <TargetVersion>target</TargetVersion>
        <ExcludeSourceFilter>filter.xml</ExcludeSourceFilter>
        <ExcludeTargetFilter>filter.xml</ExcludeTargetFilter>
    </partition>
</vrm>
```

The listing above contains the recommended contents for the configuration file. The <ComponentDeltaFileName> tag names the delta file. The required filename format depends on whether you want to apply the update through the HMI or the command line. If you want to use the HMI, your filename must follow the format QNXCAR2-*myname*-*from_version*-*to_version*.mld, where:

myname

An optional substring you can use to make the filename more readable.

from_version

Decimal number of the version of the source tree.

to_version

Decimal number of the version of the target tree.

To find the source version number, look in /etc/os.version (it's the number next to buildNum, for example, 5346). The location of the target version number depends on your system provider. It may be in the name of the archive file that contains the filesystem image of the target version or it may be listed on your system provider's download webpage.

If you want to use the command line to apply the update, your filename must be mydelta.mld; in this case, you don't need to include the source and target version numbers in the filename.

In this example, the statistics file is given a name (in the <Statistics> tag) that contains the source and target version numbers, but you can assign any name to this file.

2. If necessary, create the filter definition file that's used by the main configuration file by copying the following content into a new XML file:

```
<FilterFile version=1.0>
    <ExcludeSourceFilter>dos/qnx-ifs</ExcludeSourceFilter>
    <ExcludeTargetFilter>dos/qnx-ifs</ExcludeTargetFilter>
</FilterFile>
```

In this example, you would name the filter definition file filter.xml so it will be picked up by the main configuration file created in Step *1* (p. 241).

This additional XML file defines the filters that prevent the boot image of the DOS partition on the system's SD card from being overwritten during the update.

3. In the directory where you plan to run the update generator, create two new subdirectories called source and target.

The `source` directory will contain the filesystem image that matches what is currently running on your system. The `target` directory will contain the filesystem image that your system will run after the update.

When unpacking the filesystem image files, make sure to use `sudo` and to include the `-p` option when running `tar`. Since the permissions of some of the files in the image are owned by `root`, it's important not to modify them during this process.

4. In the `source` directory, unpack the filesystem:

```
# cd source
# sudo tar xmzpf ../board.variant.version.tar.gz
            base dos var/pps/qnxcar/system/info
```

where *`board.variant.version`*`.tar.gz` is the name of the source filesystem image.

When listing the files to unpackage in the `tar` command, don't include any files from the user data partition, which is mounted to '`/`', except for `/var/pps/qnxcar/system/info`. You must prevent user data from being overwritten by the update so you can retain your preferred settings for all apps. You can specify a more restricted set of files than what's shown in the sample `tar` command to further narrow the scope of the update.

5. In the `target` directory, unpack the filesystem image of the build that you want to update your device to:

```
# cd ../target
# sudo tar xmzpf ../board.variant.version.tar.gz
            base dos var/pps/qnxcar/system/info
```

where *`board.variant.version`*`.tar.gz` is the name of the target filesystem image.

As in Step *4* (p. 243), don't include any files from the user data partition in the list of files to unpackage (except for `/var/pps/qnxcar/system/info`).

6. Run the update generator, using `sudo` privileges:

```
# sudo ./vRapidMobileCMD-Linux.exe /type=vRM
            /configuration_file=./deltaConfig.xml
```

where *`deltaConfig`*`.xml` is the name of the file you created in Step *1* (p. 241).

An update (delta) file with a name in the format `QNXCAR2-`*`myname-from_version-to_version`*`.mld` will be written to your current directory. You'll use this file to update your system.

SWU module API

The SWU module API must be implemented by any custom swud module that you develop so that swud can load and unload the module.

The API is small, consisting of only an initialization function (*SWU_MODULE_INITIALIZE()*), which is mandatory for all modules to implement, as well as a shutdown function (*SWU_MODULE_SHUTDOWN()*), which is optional to implement.

Constants in swu_module.h

Constants defined in `swu_module.h` to provide short forms of API function names.

Definitions in swu_module.h

Preprocessor macro definitions in `swu_module.h` for aliasing SWU module API functions.

Definitions:

```
#define SWU_MODULE_INITIALIZE swu_module_initialize
#define SWU_MODULE_SHUTDOWN swu_module_uninitialize
```

Functions in swu_module.h

Functions defined in `swu_module.h` for initializing and shutting down modules.

SWU_MODULE_INITIALIZE()

Initialize a module after it has been loaded.

Synopsis:

```
#include <swud/swu_module.h>
swu_result_t SWU_MODULE_INITIALIZE( int argc, char *argv[] )
```

Arguments:

argc

Number of items in *argv*.

argv

An array containing the arguments passed into the module when it was loaded.

Description:

Initialize a module. The swud service calls this function on a module after it's finished loading it. The function should only initialize the module, allocate any necessary resources (e.g., threads), and then return as quickly as possible so it doesn't block swud.

Returns:

An swu_result_t constant indicating if the initialization was successful or what error (if any) occurred.

SWU_MODULE_SHUTDOWN()

Shut down a module before the SWU process exits.

Synopsis:

```
#include <swud/swu_module.h>

swu_result_t SWU_MODULE_SHUTDOWN( void )
```

Description:

Shut down a module. The swud service calls this function on each loaded module, while the service is shutting down. Implementing this function is optional; you can skip its implementation for a module if nothing special needs to be done to shut it down. Any implementation of this function must be signal handler-safe because it's called from within a signal handler in swud. This means that all function calls made within this function must also be signal handler-safe.

Returns:

An swu_result_t constant indicating if the shutdown was successful or what error (if any) occurred.

Chapter 16
System Launch and Monitor (SLM)

The SLM service automates process management.

Overview

SLM is started early in the boot sequence to launch complex applications consisting of many processes that must be started in a specific order.

System Launch and Monitor (SLM) is a utility controlled by a configuration file that specifies the processes to run and their properties, especially any interprocess dependencies. For example, suppose a multimedia application needs the services of the audio subsystem and the database server, which in turn needs the services of PPS. When SLM learns of these one-way dependencies when reading the configuration file, the service internally constructs a *directed acyclic graph (DAG)* representing the workflow of the underlying processes. This DAG is sorted to produce a partial ordering for scheduling the processes so that all control-flow dependencies are respected. In this example, SLM would first check that PPS is running before starting the database server and then check that the database server is up before starting the multimedia app.

For more information about how to use SLM, see `slm` in the *Utilities Reference*.

Chapter 17
Tether Manager (`tetherman`)

Tether portable devices to the QNX CAR platform

Syntax:

```
tetherman [-d] [-t] &
```

Options:

`-d`

Run in debug mode.

`-t`

Run in test mode.

Description:

Tethering allows portable devices, such as smartphones, tablets and laptops, to use the QNX CAR platform to connect to the cloud. The `tetherman` service allows you to tether portable devices to the platform using a Wi-Fi connection.

PPS objects

The tether manager uses the following PPS objects:

- `/pps/services/tethering/control`
- `/pps/services/tethering/status`
- `/pps/system/navigator/status/tethering`

Chapter 18
Wi-Fi configuration (wpa_pps)

PPS interface for Wi-Fi configuration

Syntax:

```
wpa_pps [-A addr:port] [-c file]
[-D library] [-d] [-h path]
-i iface [iface1 [iface2]...]
[-j ap_iface] [-P script]
[-p path] [-r sec] [-S script] [-s] &
```

Options:

-A *addr*:*port*

Proxy to publish when proxy authentication is required.

-c *file*

Specify configuration file (default: `/etc/wpa_pps.conf`).

-D *library*

Specify and load a library with direct access to the driver.

-d

Enable debug messages (to `stdout`).

-h *path*

Specify the path and name of the binary to execute when configured for Access Point. For example: `-h /usr/sbin/hostapd_ti18xx`.

-i *iface* [*iface1* [*iface2*]...]

Specify the Wi-Fi interface to use. This must be the last argument on the command line. Note that you can specify an optional prioritized list of interfaces (e.g., for multi-homed operation, preference of default routes, etc.).

-j *ap_iface*

Specify the AP (access-point) interface to use.

-P *script*

Specify the script to run for updating proxy settings.

-p ***path***

Specify the path for the wpa_supplicant control interface (e.g., /var/run/wpa_supplicant).

-r ***sec***

Specify the number of seconds between updates of the connected network's RSSI (default: 10). To never do any updates, specify -1.

-S ***script***

Specify the script to run for updating the RSSI.

-s

Run in simulation mode; that is, run without a Wi-Fi.

Description:

The wpa_pps service provides an interface for configuring WPA (Wi-Fi Protected Access) connections. For more information about QNX support for networking, see "Networking Architecture" in the *System Architecture Guide*.

For more information about the configuration changes wpa_pps can make, see the relevant PPS object descriptions:

- /pps/services/wifi/control
- /pps/services/wifi/status

Configuration

The wpa_pps service uses the /etc/wpa_pps.conf configuration file. To configure WPA behavior, edit the parameters in this plain-text file.

Index

.qcf configuration files 38

A

AAP, See acoustic processing
acoustic echo cancellation 30, 31, 42, 43, 49, 57, 58, 59, 61, 71, 75
  See also see handsfree telephony
  configuration 31
  events 49
  handsfree calls 42
  latency data 58, 59
  prepare 61
  set up 71
  setting up 42
  start 57
  starting 42
  stop 75
  stopping 43
  troubleshooting 43
  volume 43
    See also see handsfree telephony
acoustic echo cancellation (AEC) 26
acoustic processing 25, 39, 48, 49, 75
  event data 48
  events 49
  remote access to library 39
  stop 75
AEC, See acoustic echo cancellation
album art 14
  retrieving 14
apm-aap-rcs-hf.so 39
artwork 14
  retrieving 14
artwork client 14
attaching 42
  io-acoustic 42
audio 15, 58, 59, 109, 114
  ducking 15, 109, 114
  managing concurrent streams 15, 109, 114
  system latency 58, 59
audio manager 109
Audio Manager 15

B

Bluetooth 25
  handsfree phone calls 25

C

Car Connectivity Consortium (CCC) 95
certificate manager 19
certmgr_pps 19
clock 117
  synchronizes during startup 117
configuration 31, 38, 252
  acoustic echo cancellation 31
  acoustic processing 38
  io-acoustic 31
  WPA 252
core services 120
coreServices, See coreServices2
coreServices2 120

D

delta file 123, 241
  definition 123
  generating 241
description 236
  swud 236
displays utility 93
ducking 15, 109, 114
  audio 15, 109, 114

E

EB street director, See Elektrobit
echo cancellation, See handsfree telephony
Elektrobit 105
  navigation engine 105
events 42, 49, 63, 65
  acoustic processing 49
  handsfree 63, 65
  io-acoustic 42

G

gen-ifs 78, 82

H

handsfree 42, 50, 51, 53, 56, 57, 60, 61, 63, 65, 66, 69, 71, 73, 75
  acoustic echo cancellation 42
  acoustic routing 66
  attachio-acoustic 50
  latency estimate 53
  latency test 73
  mute audio 60
  prepare acoustic processing 61
  read events 63
  register for events 65
  set up acoustic echo cancellation 71
  start acoustic processing 57
  stop acoustic processing 75
  tuning file 51
  volume for echo cancellation 56, 69

handsfree telephony 25, 39
RCS 39
tuning file 39

I

image 78, 80, 82, 84
creating 78, 80, 82, 84
io-acoustic 26, 31, 32, 36, 42, 50
attach for handsfree 50
attaching 42
configuration 31
configuration keys 32, 36
registering for events from 42
set path to configuration file 42
io-audio 26
io-bluetooth 26
IOAP_* 44
ioap_device_t 47
ioap_event_next() 49
ioap_event_t 48
ioap_get_latency_estimate() 43
ioap_hf_attach() 42, 50
ioap_hf_config() 42, 51
IOAP_HF_EVENT_* 46
IOAP_HF_EVENT_STARTED 57
ioap_hf_events 46
ioap_hf_get_latency_estimate() 53
ioap_hf_get_log_level() 43, 54
ioap_hf_get_output_volume() 56
ioap_hf_get_volume_level() 43
ioap_hf_go 57
ioap_hf_go() 42
ioap_hf_latency_estimate_t 53, 58
ioap_hf_latency_test_t 59, 73
ioap_hf_mute() 60
ioap_hf_prepare() 42, 61
ioap_hf_read_events() 42, 63
ioap_hf_register_events() 42, 65
ioap_hf_route() 42, 66
ioap_hf_set_log_level() 43, 68
ioap_hf_set_output_volume() 69
ioap_hf_set_volume_level() 43
ioap_hf_setup() 43, 71
ioap_hf_start_latency_test() 73
ioap_hf_stop() 43, 75
ioap_io_map_t 72, 76
ioap_map_io_t 66
IOAP_MAX_DEVICE_IO 47
IOAP_MAX_DEVICE_PATH 47
ioap_start_latency_test() 43

K

keyboard 91, 93
keyboard-imf 91, 93
running 93

L

latency 53, 58, 59, 73
acoustic 53, 73
estimate 53
estimate data 58
test 73
test configuration 59
libacoustic 26
with handsfree acoustic echo cancellation 26
log 54, 68
verbosity level for acoustic processing 54, 68

M

management 19
certificate 19
manifest file 139
MCBSP, See multichannel buffered serial port
media controllers 16
using with now playing service 16
media players 16
using with now playing service 16
MirrorLink 95, 96, 98, 99, 100, 101, 102
devices for whitelist and blacklist 100
licensing 96
mlink-daemon 98
mlink-rtp 101
mlink-viewer 102
network sandbox 95
phones tested with 95
PPS objects used with 96
rtp PPS object 101
server devices 95
USB device enumeration 99
using SLM with 96
mkimage.py 82
mksysimage 84
mksysimage.py 80, 82, 84
mktar 87
multi-channel buffered serial port 26
mute 60
audio during acoustic processing 60

N

navigation engines 105
net_pps 107, 108
command-line options 107
configuration 108
running 107
net_pps.conf 108
network manager 107, 108
configuration 108
network sandbox (for MirrorLink) 95
noise reduction 26, 30
Now Playing service 16, 109, 114
nowplaying 16, 109, 114
NR, See noise reduction

P

PCM (pulse-code modulation digital audio format) 26
playback 16
    stopping with the now playing service 16
PPS 105
    navigation engines 105

R

radio 115, 116
    reference application 116
Radio App 116
RadioApp 115, 116
RCS 39
    handsfree telephony 39
Red Bend 241
    version of FOTA software required 241
reference signal (acoustic echo cancellation) 29
remote control server, See RCS
routing 66
    handsfree acoustic 66

S

SGID, See supplementary group ID
SLM 96, 100, 101
    using to start a network sandbox for MirrorLink 96
    using to start mlink-daemon 100
    using to start mlink-rtp 101
software update core library, See swu-core
software update daemon, See swud
software updates 123, 127, 139, 241
    applying 123
    configuration file 241
    delta file, See delta file
    manifest file, See swu-core manifest file
    state-transition diagram 127
    supported platforms 241
street director, See Elektrobit
supplementary group ID 30
    io-acoustic 30
swu-core 124, 125, 126, 129, 139, 142, 143, 150, 167, 198, 217, 219
    architecture 124
    concepts 126
    integrating software update modules 129
    internal components 125
    introduction 124
    manifest file 139
    SWU library API 142, 143, 150, 167, 198, 217, 219
        ClientConfiguration.h 143
        Common.h 150
        Update.h 167
        UpdateClient.h 198
        UpdateTarget.h 217
        UpdateTargetInterface.h 219
swud 235, 236, 237, 238, 244
    developing custom modules 237
    introduction 235
    loading modules dynamically 236
    programming sample 238
    reference modules 238
    SWU module API 244
syslink_drv 115
System Launch and Monitor, See SLM

T

tabs 105
    configuring 105
tar 87
    creating 87
Technical support 11
tether manager 249
tethering, See tether manager
tetherman 249
    running 249
TI J5 ECO EVM811x EVM 115
    platform for reference radio application 115
TI Jacinto 5 116
troubleshooting 43
    acoustic echo cancellation 43
tuning 39, 51
    acoustic processing 51
    configuration file for handsfree telephony 39
    handsfree telephony 39
Typographical conventions 9

U

USB device enumeration (for MirrorLink) 99

V

verbosity 54, 68
    acoustic processing log 54, 68
volume 43, 56, 69
    for handsfree echo cancellation 56, 69
    set for acoustic echo cancellation 43

W

Wi-Fi 252
    WPA configuration 252
Wi-Fi Protected Access, See WPA
WPA 251, 252
    configuration 251, 252
wpa_pps 251, 252
    command-line options 251
    configuration 252
    running 251
wpa_pps.conf 252