

HMI Notification Manager

©2014, QNX Software Systems Limited, a subsidiary of BlackBerry. All rights reserved.

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/>

QNX, QNX CAR, Neutrino, Momentics, Aviage, and Foundry27 are trademarks of BlackBerry Limited that are registered and/or used in certain jurisdictions, and used under license by QNX Software Systems Limited. All other trademarks belong to their respective owners.

Electronic edition published: Thursday, February 20, 2014

Table of Contents

About This Guide	5
Typographical conventions	6
Technical support	8
Chapter 1: HNM Components	9
Chapter 2: Event Processing	11
Chapter 3: Configuration	15
Chapter 4: Plugins	19
The Generic plugin	20
The HandsFreePhone plugin	22
The VirtualMechanic plugin	24
Chapter 5: PPS Objects	29
Chapter 6: API Reference	31
core.h	32
Definitions in core.h	32
Enumerations in core.h	32
Functions in core.h	33
display_event.h	37
Definitions in display_event.h	37
Typedefs in display_event.h	37
Enumerations in display_event.h	40
Functions in display_event.h	41
event.h	49
Definitions in event.h	49
Typedefs in event.h	50
Enumerations in event.h	53
event-source.h	56
Typedefs in event-source.h	56
Functions in event-source.h	59
messaging.h	60
Definitions in messaging.h	60
Typedefs in messaging.h	61
Functions in messaging.h	63
pps.h	65

Definitions in pps.h	65
Typedefs in pps.h	66
Functions in pps.h	67
queue.h	73
Typedefs in queue.h	73
Functions in queue.h	75
status.h	80
Definitions in status.h	80
Typedefs in status.h	81
Functions in status.h	82

About This Guide

This guide describes the HMI Notification Manager (HNM), a multimodal subsystem for managing asynchronous, multimodal events based on predefined priorities. The HNM appraises incoming events, applies appropriate rules, and then notifies all subscribers via PPS.

This guide is intended for developers who will be creating and deploying apps for the QNX CAR platform.

The following table may help you find information quickly:

To find out about:	Go to:
How the HNM is structured	HNM Components (p. 9)
Command-line options for the HNM service	HNM Components (p. 9)
How events are handled	Event Processing (p. 11)
Specifying event priorities	Configuration (p. 15)
The format for configuration files	Configuration (p. 15)
Plugins for event sources	Plugins (p. 19)
PPS objects used by the HNM	PPS Objects (p. 29)
Functions, data types, structures, etc.	API Reference (p. 31)

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	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<i>PATH</i>
File and pathnames	<code>/dev/null</code>
Function names	<code>exit()</code>
Keyboard chords	Ctrl –Alt –Delete
Keyboard input	Username
Keyboard keys	Enter
Program output	login:
Variable names	<i>stdin</i>
Parameters	<i>parm1</i>
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 use a forward slash (/) as a delimiter in all 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

HNM Components

Overview

The HMI Notification Manager (HNM) is responsible for managing asynchronous, multimodal events based on predefined priorities. Like a window manager, the HNM decides when and how events get processed, based on their priority, and determines whether or not to notify the user via the HMI. But unlike a window manager, the HNM also responds to low-level system services using various input modalities and can manage various outputs in addition to a video display (e.g., you could use the HNM to manage audio streams).

Structure of the HNM

The HNM consists of three components:

hmi-notification-mgr

A daemon that implements the notification service.

hmi-notification-core.so

A core library that provides all the common functionality used by the HNM and its associated event-source [plugins](#) (p. 19).

policy.cfg

A [configuration](#) (p. 15) file that specifies the event sources that the HNM should deal with and how it should manage events.

Command-line options for the **hmi-notification-mgr** daemon

-c filename

Specify the policy configuration file (default location is `/etc/system/hmi-notification/policy.cfg`). If no policy file is found, the daemon will run without loading any event-source plugins.

-f

Run in the foreground, rather than in daemon mode (useful for debugging).

-p pps_base

Specify the base URI of the PPS objects managed by the HNM (default is `/pps/services/hmi-notification`). Once the service is running, it

creates Status and Messaging *PPS objects* (p. 29) that provide the external interface to the service.

-v

Set the verbosity level of the log output.

Chapter 2

Event Processing

Overview

Interaction events are the mechanism that the HNM uses to notify clients about changes in the system. The HMI can subscribe to the HNM to receive these events, which can then be dispatched to the appropriate applications.

Interaction events live through two stages:

- **event generation**—applications or services (including the HMI) create events for the HNM to manage
- **event processing**—the HNM examines the *priority* of events to determine which ones to act upon

The process works essentially as follows:

1. An application decides how a certain event should be handled. For example, if the app wants an event to trigger a notice to appear on the display, it will set up a *display event* for the HNM to handle.
2. The HNM maps the received event to its policy configuration to determine the event's priority.
3. The HNM then decides whether to preempt other events, based on their priority, and to activate the newly received event.
4. To notify subscribers of any changes to its state, the HNM updates the PPS Status object (`/pps/services/hmi-notification/Status`).

Priorities

Applications can specify priorities in their policy configuration files by using integer values in the range 0 (lowest) to 7 (highest). The format for the configuration file's relevant section (called “`event-priorities`”) looks like this:


```
event-priorities {
    EventName = <priority_value>
    EventNameSpace {
        # This event's name will be prepended by the
        # 'EventNameSpace::' string, making it
        # distinct from the previous definition of
        # 'EventName'.
        EventName = <priority_value>
    }
}
```

For example, here's an excerpt from the `event-priorities` section of the configuration file for the `VirtualMechanic` plugin module:

```
event-priorities {
  Caution {
    fuelLevel = 2
    washerFluidLevel = 1
    transmissionFluidLevel = 2
    coolantLevel = 2
    brakeFluidLevel = 2
    tirePressure = 1
    tireWear = 1
    brakePadWear = 1
    brakeAbs = 1
    engineOilPressure = 2
    engineOilLevel = 2
    rpm = 0
    #temperature = 2
    #clutch_wear = 2
    lightHead = 2
    lightTail = 2
  }
}
```

For more information on configuration files, see “[Configuration](#) (p. 15)” in this guide.

If a priority configuration isn't specified for a particular event in the configuration file, the HNM will assign a default value. The default priority will depend on the [window type](#) (p. 17) that's requested. Here are the window types and their default priorities:

Window type	Default priority	Description
Fullscreen	0	The notification takes up the whole window.
Growl	0	A subtle, transient notification.
Hidden	0	The notification isn't displayed.  The Hidden window type is defined only for the sake of completeness—it shouldn't be used in practice.
Notification	0	The notification remains on the screen (e.g., a status bar).
Overlay	0	The notification appears in a popup overlay.

Priority scenarios

The HNM must handle three distinct priority scenarios when an application asks for an event to be processed:

1. The event has a *lower priority* than the currently active event.
2. The event has the *same priority* as the currently active event.
3. The event has a *higher priority* than the currently active event.

These scenarios can be extended to accommodate the case where multiple asynchronous events can arrive simultaneously. Although queues are the ideal data structure to accommodate this workflow, not all interactions can be delayed. Therefore, events are handled according to the following rules:

- If multiple events are received with varying priorities, the HNM will process each event, queuing the events with lower priority and handling only the highest-priority event.
- If multiple events are received with the same priority, the HNM will queue those that can be queued and will process one of the remaining events, dropping the rest.

Event types

Events are categorized into classes that correspond to the interaction types (e.g., display, audio). A class of interaction events can have subtypes. For example, the *display* class has these subtypes:

display-start

An app creates this event to ask the HMI to display a window. If the event's priority value is greater than that of all active events, then the HNM will service the event. Otherwise, a *fallback* window type will be requested, causing the appraisal to repeat until no more fallback types are left to try. For example:

```
event->window_type = HNM_WINDOW_NOTIFICATION ;
event->fallback_types[ 0 ] = HNM_WINDOW_GROWL ;
event->fallback_types[ 1 ] = HNM_WINDOW_HIDDEN ;
event->fallback_types[ 2 ] = HNM_WINDOW_HIDDEN ;
event->fallback_types[ 3 ] = HNM_WINDOW_HIDDEN ;
```

display-end

An app creates this event to ask the HMI to hide a window.

Sharing the display

Some apps may be willing to share the display with others. Although the HNM doesn't mandate how the display is physically shared, it must be aware of which applications can share the display (and under which circumstances) so it can decide whether to allow an app to display itself.

Apps can specify whether windows can share the display by setting one of the following display-control flags:

HNM_DISPLAY_SHARED

Indicates that the window type can share the display regardless of what is currently being displayed.

HNM_DISPLAY_EXCLUSIVE

Indicates that the window type can't share the display with any other exclusive window type.

HNM_DISPLAY_SEMI_EXCLUSIVE

Indicates that the window type can be shared with a predefined number of semi-exclusive windows, which depends on the number of available display slots (defined in the policy configuration).

These controls are specified in the `window-types` section of the `policy.cfg` file. For details, see “[Window types](#) (p. 17)” in the “Configuration” chapter.

Chapter 3

Configuration

Overview

The HNM relies on a *policy configuration file* to determine which event sources to manage and how to manage them, based on their priorities. The default configuration file is located here:

```
/etc/system/hmi-notification/policy.cfg
```

Configuration file format

The format is a simple text file that defines a hierarchy (tree) of different *sections*. Each line of the configuration file can hold at most 1024 characters (additional characters will be truncated from the line when the file is processed). Configuration items are listed within these sections as name-value pairs separated by an equals sign (=).

Each name of a section or subsection is followed by an open brace ({). Names themselves can contain any characters except { and space. A closing brace (}) ends each section or subsection. Any text between a number sign (#) and the end of the line is a comment.

Here's the default `policy.cfg` file:

```
hpm-cfg {
  modules {
    Generic {
      dll = /lib/dll/hmi-notification/event-source-generic.so
    }
    HandsFreePhone {
      dll = /lib/dll/hmi-notification/event-source-handsfree.so
      event-priorities {
        HFP_INITIALIZED = 0
        HFP_CONNECTED_IDLE = 0
        HFP_CALL_OUTGOING_DIALING = 1
        HFP_CALL_INCOMING = 1
      }
    }
    VirtualMechanic {
      dll = /lib/dll/hmi-notification/event-source-vm.so
      event-priorities {
        Caution {
          fuelLevel = 2
          washerFluidLevel = 1
          transmissionFluidLevel = 2
          coolantLevel = 2
          brakeFluidLevel = 2
          tirePressure = 1
          tireWear = 1
          brakePadWear = 1
          brakeAbs = 1
        }
      }
    }
  }
}
```

```

        engineOilPressure = 2
        engineOilLevel = 2
        rpm = 0
        #temperature = 2
        #clutch_wear = 2
        lightHead = 2
        lightTail = 2
    }
    Alert {
        fuelLevel = 3
        washerFluidLevel = 1
        transmissionFluidLevel = 3
        coolantLevel = 3
        brakeFluidLevel = 3
        tirePressure = 2
        tireWear = 2
        brakePadWear = 2
        brakeAbs = 2
        engineOilPressure = 3
        engineOilLevel = 3
        rpm = 0
        #temperature = 3
        #clutch_wear = 3
        lightHead = 3
        lightTail = 3
    }
}
}
}

window-types {
    Fullscreen {
        DisplayControl = 0 # Exclusive
        DefaultPriority = 0
    }
    Overlay {
        DisplayControl = 0 # Exclusive
        DefaultPriority = 0
    }
    Notification {
        DisplayControl = 1 # Semi-Exclusive with 1 display slots.
        DefaultPriority = 0
    }
    Growl {
        DisplayControl = -1 # Shared
        DefaultPriority = 0
    }
}
}
}

```

Sections

Although you can set up your own sections, the HNM relies on the following predefined sections in the configuration file:

modules

Informs the HNM about any pluggable event-source modules that need to be loaded by the system. The `all` child item specifies the path of the shared object (`.so` file) for the event-source plugin. For more information about event-source plugins, see “[Plugins](#) (p. 19)” in this guide.

window-types

Specifies how notices should appear on the display.

Modules

The `modules` section contains the names of the event-source plugins to be loaded (e.g., `VirtualMechanic`). Each plugin can contain these subsections:

dll

Path to the `.so` file (e.g., `dll = /lib/dll/hmi-notification/event-source-vm.so`).

event-priorities

Priority mappings specified via a list of name-value pairs (e.g., `fuelLevel = 2`).

The HNM uses the names in the `event-priorities` section to form the event names that appear, for example, in the

`/pps/services/hmi-notification/Status` object:



```
@Status
display:json:[{"name": "Home", "type": "Modal"},
{"name": "Alert", "type": "Notification"}]
```

Window types

The `window-types` section lists the available window types, along with their `DisplayControl` and `DefaultPriority` values. The window types are as follows:

- `Fullscreen`—an application's fullscreen view associated with an event. Although other window types may require *exclusive* access to the display, fullscreen windows are unique in that they can't be closed. They can be replaced only by another fullscreen view.
- `Growl`—transient notification types. These have *nonexclusive* display semantics, so any number of growl notifications can be displayed alongside exclusive or semi-exclusive window types.
- `Notification`—these represent nontransient graphical hints (e.g., graphical status bar). Notifications have *semi-exclusive* display semantics, so they can typically share the display with other exclusive window types (e.g., fullscreen or overlay windows). But note that there may be a limit to the number of nontransient notifications that can be displayed at any given time.
- `Overlay`—can have exclusive or nonexclusive access to the display. Exclusive overlays are essentially like fullscreen windows except that overlay windows can

be closed. When an overlay is closed, the views that were previously obscured should be redisplayed.

DisplayControl values

For each window type, the `DisplayControl` item specifies how the window will share the display:

Value	Flag	Description
0	<code>HNM_DISPLAY_EXCLUSIVE</code>	Window type can't share the display with any other exclusive window type.
1	<code>HNM_DISPLAY_SEMI_EXCLUSIVE</code>	The value 1 indicates the number of display slots. In this case, the window type can share the display with an exclusive display and with one or more semi-exclusive displays.
-1	<code>HNM_DISPLAY_SHARED</code>	Window type can share the display regardless of what is currently being displayed.

Chapter 4

Plugins

Overview

Display events can originate from many different source types. Since the HNM may not know the form of those sources in advance, it exposes a *plugin* framework to allow new event sources to be added.

Plugins are DLLs (shared objects) that must register with the HNM subsystem via the [hnm_register_module\(\)](#) (p. 59) function. Plugins are loaded via the `policy.cfg` file (see “[Configuration](#) (p. 15)” for details.)

You'll find the following plugins on your system under the `/lib/dll/hmi-notification/` directory:

Plugin	Filename	Description
<i>Generic</i>	<code>event-source-generic.so</code>	Provides a generic PPS interface that allows applications to use the HNM policy-management facilities without implementing a custom plugin. This plugin can also be used for automated testing.
<i>HandsFreePhone</i>	<code>event-source-handsfree.so</code>	Generates display events to notify subscribing HMI applications so they can handle incoming handsfree phone calls.
<i>VirtualMechanic</i>	<code>event-source-vm.so</code>	Notifies the HNM of caution and alert conditions for relevant vehicle systems.

The following sections in this chapter cover each of these plugins.

The Generic plugin

Overview

The Generic event-source plugin relies on a PPS *server object* (`/pps/services/hmi-notification/control`) that clients can use to issue asynchronous events to the HNM. Clients send an event command with the appropriate parameters and values, while the HNM responds via the `/pps/services/hmi-notification/Status` and `/pps/services/hmi-notification/Messaging` objects. For details on those objects, see their entries in the *PPS Objects Reference*.

Callback functions

The Generic plugin uses these callback functions:

open()

The plugin subscribes to the `/pps/services/hmi-notification/control` PPS object, listening for clients to publish events. The plugin creates this PPS object when the *open()* function is called.

close()

The counterpart to the *open()* callback, this callback is responsible for closing its connection to the `control` object.

read_event()

This function is responsible for decoding changes made to the `control` object and for constructing events that will be passed to the HNM policy subsystem.

For more information, see “[API Reference](#) (p. 31)” in this guide.

Loading the plugin

To load the Generic plugin, make sure the following lines are in the `modules` section of the HNM policy configuration file (`policy.cfg`):

```
Generic {  
    dll = /lib/dll/hmi-notification/event-source-generic.so  
}
```

Note that the Generic plugin doesn't have an associated event-priority map. Event names for the plugin are specified at run time only through PPS `event` messages.

Reporting errors

The Generic plugin will write messages to the system log periodically. The verbosity is configurable, but the level currently can be set only at compile time. To do this, set the `SLOG_VERBOSITY` preprocessor symbol to a positive number when compiling; the greater this value, the more information will be written to the system log. The default verbosity level is `_SLOG_ERROR` (i.e., print only critical and error messages).

You can configure the plugin to print log messages to the standard output stream. To do this, define the `LOG_TO_STDOUT` preprocessor symbol at compile time. You can also have the plugin generate additional debugging information (e.g., source file, function name, line number) by defining the `DEBUG` flag at compile time.

The HandsFreePhone plugin

Overview

The HandsFreePhone event-source plugin represents the Bluetooth HFP phone service. This plugin is responsible for notifying the HNM of incoming phone call events. The plugin generates display events so that subscribing HMI applications will know how to handle incoming handsfree phone calls.

This plugin subscribes to the `/pps/services/bluetooth/handsfree/status` object, which the Bluetooth Manager uses to publish the results of commands sent to the `/pps/services/bluetooth/handsfree/control` object. For details on these objects, see their entries in the *PPS Objects Reference*.

HFP states

The plugin generates *display* events in response to state changes in the Bluetooth HFP service. When the plugin detects a state change in the HFP `status` object, one or more events will be generated to open and close display windows appropriately. New events are added to the tail of the queue following HFP state changes.

Although the HFP `status` object can report several *state* values (`HFP_CALL_ACTIVE`, `HFP_CALL_ACTIVE_HELD`, etc.), the HandsFreePhone plugin considers only a small set of HFP states:

HFP state	Plugin behavior
<code>HFP_INITIALIZED</code>	When the HFP system is first initialized, there should be no active call. When the HFP <code>status</code> object gets to this state, a <code>display-end</code> event is issued to ensure that no displays associated with handsfree calls are currently displayed. Note that the initialized state occurs when a call ends as well as when the system is first initialized. No priority is explicitly assigned to the initialized state—this state issues only <code>display-end</code> requests, which do not depend on priority.
<code>HFP_CONNECTED_IDLE</code>	The HFP service is connected and ready for call activity.
<code>HFP_CALL_OUTGOING_DIALING</code>	A remote party's number is being dialed.
<code>HFP_CALL_INCOMING</code>	When an incoming call is received, the plugin will issue a <code>display-start</code> event requesting an <code>Overlay</code> window type. The name of the event is simply the state's name (i.e., <code>HFP_CALL_INCOMING</code>). The HMI view responsible for handling the specific display request is the Communication app. If the incoming call can't be displayed by a popup overlay, then the fallback display type is a <code>growl</code> notification. This <code>growl</code> will require that the user activate

HFP state	Plugin behavior
	<p>the Communication app (e.g., by tapping the growl notification window) to accept or reject the incoming call.</p> <p>Whether a communication dialog can be displayed depends on the priority of the currently display application and on the priority of the incoming call event. An incoming call event has default priority unless a custom priority is specified in the policy configuration. You can set a custom priority by adding the line <code>HFP_CALL_INCOMING = 0</code> to the <code>HandsFreePhone</code> module definition in the policy configuration file as shown:</p> <pre> modules { ... HandsFreePhone { dll = /lib/dll/hpm/event-source-handsfree.so event-priorities { ... HFP_CALL_INCOMING = 0 # Set custom priority ... } } ... } </pre>

Callback functions

Since the HandsFreePhone plugin wraps the Bluetooth HFP service's PPS `status` object, the `open()` callback must subscribe to this PPS object so that the plugin will be notified of any state changes by the HFP subsystem.

The `close()` callback closes the PPS object that this plugin subscribes to.

The `read_event` callback is the mechanism that the HNM uses to get event information from this plugin. If there's a pending event from the Bluetooth HFP service, this callback will return the event data in the specified event buffer.

For more information, see “[API Reference](#) (p. 31)” in this guide.

Loading the plugin

The following lines in the `modules` section of the HNM policy configuration file (`policy.cfg`) will load the HandsFreePhone plugin. You can adjust the HFP state priorities as mentioned above:

```

HandsFreePhone {
    dll = /lib/dll/hmi-notification/event-source-handsfree.so
    event-priorities {
        HFP_INITIALIZED = 0
        HFP_CONNECTED_IDLE = 0
        HFP_CALL_OUTGOING_DIALING = 1
        HFP_CALL_INCOMING = 1
    }
}

```

The VirtualMechanic plugin

Overview

The VirtualMechanic (VM) event-source plugin represents the low-level service that reports vehicle status to the system. The VM plugin will notify the HNM subsystem of *caution* and *alert* status conditions for relevant components in these categories:

- Fluid
- Traction
- Braking
- Powertrain
- Electrical

The **Virtual Mechanic** app allows the user to view the status of these components in the platform's HMI:



The VM plugin subscribes to the `/pps/qnxcar/sensors/status` object, which the Virtual Mechanic app uses to report status values for all of its components.

Event types

The VM plugin generates two event types:

- Caution
- Alert

The caution status condition initially has the default priority, whereas the alert status condition has a priority that is one greater than the default. Note that the plugin monitors several categories of data, so each of these can potentially have different caution and alert priorities.

Here's the VM event-priorities section from the default `policy.cfg` file:

```
event-priorities {
  Caution {
    fuelLevel = 2
    washerFluidLevel = 1
    transmissionFluidLevel = 2
    coolantLevel = 2
    brakeFluidLevel = 2
    tirePressure = 1
    tireWear = 1
    brakePadWear = 1
    brakeAbs = 1
    engineOilPressure = 2
    engineOilLevel = 2
    rpm = 0
    #temperature = 2
    #clutch_wear = 2
    lightHead = 2
    lightTail = 2
  }
  Alert {
    fuelLevel = 3
    washerFluidLevel = 1
    transmissionFluidLevel = 3
    coolantLevel = 3
    brakeFluidLevel = 3
    tirePressure = 2
    tireWear = 2
    brakePadWear = 2
    brakeAbs = 2
    engineOilPressure = 3
    engineOilLevel = 3
    rpm = 0
    #temperature = 3
    #clutch_wear = 3
    lightHead = 3
    lightTail = 3
  }
}
```

The nested event names correspond to the names of the attributes published in the `/pps/qnxcar/sensors/status` object.

The following table shows the caution and alert status conditions for each of the attributes. Note that for the ABS system, only the individual wheel sensors can trigger caution status conditions. For attributes with Boolean data types, a `false` value will trigger a caution status condition.

Attribute	Caution status condition	Alert status condition
<i>brakeAbsEnabled</i>	n/a	n/a
<i>brakeAbsFrontLeft</i>	false	n/a
<i>brakeAbsFrontRight</i>	false	n/a
<i>brakeAbsRearLeft</i>	false	n/a
<i>brakeAbsRearRight</i>	false	n/a

Attribute	Caution status condition	Alert status condition
<i>brakeFluidLevel</i>	<=80%	<=70%
<i>brakePadWearFrontLeft</i>	<=40%	<=20%
<i>brakePadWearFrontRight</i>	<=40%	<=20%
<i>brakePadWearRearLeft</i>	<=40%	<=20%
<i>brakePadWearRearRight</i>	<=40%	<=20%
<i>cameraRearviewActive</i>	n/a	n/a
<i>coolantLevel</i>	<=80%	<=70%
<i>engineOilLevel</i>	<=85%	<=75%
<i>engineOilPressure</i>	<=85%	<=75%
<i>fuelLevel</i>	<=25%	<=10%
<i>lightHeadLeft</i>	false	n/a
<i>lightHeadRight</i>	false	n/a
<i>lightTailLeft</i>	false	n/a
<i>lightTailRight</i>	false	n/a
<i>rpm</i>	>=6250	>=7000
<i>speed</i>	n/a	n/a
<i>tirePressureFrontLeft</i>	<=26 PSI, >=36 PSI	<=24 PSI, >=38 PSI
<i>tirePressureFrontRight</i>	<=26 PSI, >=36 PSI	<=24 PSI, >=38 PSI
<i>tirePressureRearLeft</i>	<=26 PSI, >=36 PSI	<=24 PSI, >=38 PSI
<i>tirePressureRearRight</i>	<=26 PSI, >=36 PSI	<=24 PSI, >=38 PSI
<i>tireWearFrontLeft</i>	<=30%	<=20%
<i>tireWearFrontRight</i>	<=30%	<=20%
<i>tireWearRearLeft</i>	<=30%	<=20%
<i>tireWearRearRight</i>	<=30%	<=20%
<i>transmissionClutchWear</i>	<=60%	<=40%
<i>transmissionFluidLevel</i>	<=80%	<=70%
<i>transmissionFluidTemperature</i>	>=215 (degrees F)	>=240 (degrees F)
<i>transmissionGear</i>	n/a	n/a
<i>washerFluidLevel</i>	<=20%	<=10%

Callback functions

Since the VM plugin wraps the `/pps/qnxcar/sensors/status` object, the `open()` callback must subscribe to this PPS object so that the plugin will be notified of any changes published by the Virtual Mechanic app.

The `close()` callback closes the PPS object that the VM plugin subscribes to.

The `read_event` callback will return the event data from the VM plugin. This function will interpret any changes to the PPS data from the `/pps/qnxcar/sensors/status` object and will construct the appropriate event structure, returning this to the caller. If no data is available from PPS, this function returns `false`.

For more information, see “[API Reference](#) (p. 31)” in this guide.

Loading the plugin

The `modules` section of the HNM policy configuration file (`policy.cfg`) contains a subsection for the VM plugin, giving the location of the `.so` file to load:

```
VirtualMechanic {  
    dll = /lib/dll/hmi-notification/event-source-vm.so
```


Chapter 5

PPS Objects

Overview

The HNM uses these PPS objects for communicating with subscribed clients:

`/pps/services/hmi-notification/Messaging`

Server object that is used to send transient notifications.

`/pps/services/hmi-notification/Status`

Contains the status of the various output modalities.

`/pps/services/hmi-notification/control`

Control object for the **generic** event-source plugin.

For more information, see the *PPS Objects Reference*.

Chapter 6

API Reference

Summary

The following table summarizes the header files that provide the HNM API:

Header file	Description
<code>core.h</code>	Provides an interface to the host system's logging facilities.
<code>display_event.h</code>	The types, structures, and functions that comprise the <i>display event</i> type.
<code>event.h</code>	Declaration of the HNM event structure and its associated functions.
<code>event-source.h</code>	Structures and the register function for event-source plugins.
<code>messaging.h</code>	Declaration of the <code>Messaging</code> PPS object used by the HNM.
<code>pps.h</code>	Declaration of a generic PPS object structure.
<code>queue.h</code>	Declaration of a generic queue data structure.
<code>status.h</code>	Declaration of the <code>Status</code> PPS object used by the HNM.

core.h

HNM core library declarations.

The core HNM library provides an interface to the host system's logging facilities.

Definitions in *core.h*

Preprocessor macro definitions for the core.h header file in the libhnm library.

Definitions:

```
#define hnm_dbg hnm_log( _SLOG_DEBUG1, fmt, ##__VA_ARGS__ )
```

This macro implements a log function for debug information.

```
#define hnm_err hnm_log( _SLOG_ERROR, fmt, ##__VA_ARGS__ )
```

This macro implements a log function for errors.

```
#define hnm_info hnm_log( _SLOG_INFO, fmt, ##__VA_ARGS__ )
```

This macro implements a log function for general information.

```
#define DEFAULT_VERBOSITY _SLOG_ERROR
```

This literal specifies the default verbosity level used by the log function.

Library:

libhnm

Enumerations in *core.h*

hnm_LogBufferId

Alias for the log buffer ID type enumeration

Synopsis:

```
#include <hnm/core.h>
```

```
typedef enum hnm_log_buffer_id hnm_LogBufferId;
```

Library:

libhnm

Description:

This type is an alias for the log buffer ID type enumeration, [hnm_log_buffer_id](#) (p. 33).

hnm_log_buffer_id

Enumeration of log buffer IDs.

Synopsis:

```
#include <hnm/core.h>

typedef enum hnm_log_buffer_id{
    HNM_LOG_SLOG == 0
    HNM_LOG_STDOUT
} hnm_LogBufferId;
```

Data:***HNM_LOG_SLOG***

Sets the system log as the target buffer.

HNM_LOG_STDOUT

Sets the standard output stream as the target buffer.

Library:

libhnm

Description:

The HNM_LOG_SLOG and HNM_LOG_STDOUT flags specify the target buffer for logging.

Functions in *core.h****hnm_log()***

Private helper function used to generate log messages.

Synopsis:

```
#include <hnm/core.h>

void hnm_log(int severity, const char *fmt,...)
```

Arguments:***severity***

The severity of the condition that triggered the message. For more information on severity levels, see *slogf()* in the *QNX C Library Reference*. Valid values include:

- `_SLOG_INFO`
- `_SLOG_WARN`
- `_SLOG_ERROR`
- `_SLOG_CRITICAL`

fmt

The format string to print to the log buffer. This may include tokens that to be replaced by values of variable arguments appended to the end of the call. The max length of an expanded log message is 1024 characters (this includes all format substitutions and the null terminator).

Library:

`libhnm`

Description:

NOTE: The *hnm_log()* function is flagged "private" because using it directly is discouraged. Instead, use the macros that follow its declaration, which add debugging data when the `DEBUG` macro is defined. However, you may need to log at different verbosity levels for which macros haven't yet been defined, so this function may be useful in such cases.

The *hnm_log()* function sends debugging information with an associated severity to the appropriate log. The log where the data is actually sent is specified by the global variable `log_stdout`. If this variable is nonzero, output generated by this function is printed to the system log.

Log messages are written to the log buffer only if their severity is less than or equal to the current verbosity setting.

NOTE: If the severity of the log message is critical, the program is aborted. If the severity of the log message is `_SLOG_ERROR`, the program exits with a failure status.

Returns:

Nothing.

hnm_set_log_buffer()

Specify the log buffer for the hnm_log() function to use.

Synopsis:

```
#include <hnm/core.h>

void hnm_set_log_buffer(hnm_LogBufferId log_buffer)
```

Arguments:***log_buffer***

The ID representing the log buffer that *hnm_log()* uses for output.

Library:

libhnm

Description:

The *hnm_set_log_buffer()* function sets the internal flag that specifies the target logging buffer for the log data emitted by *hnm_log()*.

Returns:

Nothing.

hnm_setLogVerbosity()

Set the verbosity level.

Synopsis:

```
#include <hnm/core.h>

void hnm_setLogVerbosity(unsigned verbosity)
```

Arguments:***verbosity***

The verbosity level.

Library:

libhnm

Description:

The *hnm_setLogVerbosity()* function sets the verbosity level for log output generated by the HNM.

Returns:

Nothing.

display_event.h

Definition of the types, structures, and functions that comprise the *display event* type.

Definitions in *display_event.h*

Preprocessor macro definitions for the display_event.h header file in the libhnm library.

Definitions:

```
#define hnm_DisplayEvent_narrow ( hnm_Event_typeof( event, HNM_EVENT_DISPLAY
) ? \
    ( hnm_DisplayEvent* )event : \
    NULL )
```

Macro that lowers an `hnm_Event` instance to an `hnm_DisplayEvent`.

Library:

libhnm

Typedefs in *display_event.h*

hnm_DisplayEvent

Alias for hnm_display_event.

Synopsis:

```
#include <hnm/display_event.h>

typedef struct hnm_display_event hnm_DisplayEvent;
```

Library:

libhnm

Description:

This is an alias for the `hnm_display_event` structure.

hnm_display_event

Structure encapsulating display event specific data.

Synopsis:

```
struct hnm_display_event hnm_DisplayEvent {
    EVENT_BASE ;
    hnm_WindowTypeID window_type ;
    hnm_WindowTypeID fallback_types [HNM_WINDOW_HIDDEN];
    char view [256];
};
```

Data:***EVENT_BASE***

Extend the base Event structure.

hnm_WindowTypeID window_type

An identifier of the requested window type. This is the preferred window type requested by an application. An alternate window type may be specified by the HNM if the event has a lower priority than the currently displayed application. This allows the information to be presented on the display without needing to change which application controls the display.

hnm_WindowTypeID fallback_types[HNM_WINDOW_HIDDEN]

An array of supported window types that can be used as a fallback in case the requested window type cannot be used for displaying application information. This list must be ordered by preference from most to least preferred.

char view[256]

A string describing the name of the view that is associated with the interaction request. This is used to identify the name of the application responsible for fulfilling the display request and to inform the application of the relevant subview.

Library:

libhnm

Description:

The `hnm_DisplayEvent` type is a specialization of an [hnm_Event](#), so it can be passed around as a pointer to an `hnm_Event`, which can be narrowed to an [hnm_DisplayEvent](#) (p. 37).

hnm_window_type

Structure representing a window type.

Synopsis:

```
typedef struct hnm_window_type {
    const char * name ;
    unsigned int default_priority ;
    hnm_DisplayControl exclusive ;
    unsigned short max_display_slots ;
    unsigned short num_display_slots ;
    hnm_DisplayEvent ** display_slots ;
}hnm_WindowType;
```

Data:

const char * name

A pointer to a string literal that expresses the window type as a string. This structure member is immutable; it should be set only at initialization.

unsigned int default_priority

The default priority used for events of the associated window type.

hnm_DisplayControl exclusive

A flag indicating whether the window type requires exclusive access to the display. This is a tri-state value that is evaluated in the decision tree used to determine if a display request can be accepted. A window type may require exclusive, semi-exclusive, or shared access to the display.

unsigned short max_display_slots

The maximum number of display slots available for the current window type. If this type isn't semi-exclusive, this value should be zero.

unsigned short num_display_slots

The number of available display slots for the current window type. If the window type isn't semi-exclusive, the value of this member should be zero.

hnm_DisplayEvent ** display_slots

An array of display slots. Each slot is a pointer to an active event. Available display slots contain NULL pointers.

Library:

libhnm

Description:

This structure is used to group window type configuration data including name strings as well as exclusivity.

Enumerations in *display_event.h****hnm_WindowTypeID***

Alias for the window display type enumeration

Synopsis:

```
#include <hnm/display_event.h>

typedef enum hnm_window_type_e hnm_WindowTypeID;
```

Library:

libhnm

Description:

This type is an alias for the window display type enumeration, [hnm_window_type_e](#) (p. 40).

hnm_window_type_e

Enumeration of window display types.

Synopsis:

```
#include <hnm/display_event.h>

typedef enum hnm_window_type_e{
    HNM_WINDOW_FULLSCREEN
    HNM_WINDOW_OVERLAY
    HNM_WINDOW_NOTIFICATION
    HNM_WINDOW_GROWL
    HNM_WINDOW_HIDDEN
    HNM_WINDOW_NUM_TYPES
} hnm_WindowTypeID;
```

Data:

HNM_WINDOW_FULLSCREEN

Full screen window.

HNM_WINDOW_OVERLAY

Popup overlay.

HNM_WINDOW_NOTIFICATION

Persistent notification.

HNM_WINDOW_GROWL

Transient notification.

HNM_WINDOW_HIDDEN

Do not display.

HNM_WINDOW_NUM_TYPES**Library:**

libhnm

Description:

The available window types are Fullscreen, Growl, Notification, and Overlay. Note that the Hidden window type is defined only for the sake of completeness; it shouldn't be used in practice.

Functions in *display_event.h****display_event_factory_get_next_event()***

Get the next event.

Synopsis:

```
#include <hnm/display_event.h>
```

```
hnm_DisplayEvent* display_event_factory_get_next_event()
```

Arguments:**Library:**

```
libhnm
```

Description:

This function returns the next-highest priority display event to pass to the HNM.

Returns:

The next-highest priority event from the internal queues. Returns NULL if there are no queued events.

display_event_window_type_id()

Get the window type ID that corresponds to the specified string.

Synopsis:

```
#include <hnm/display_event.h>
```

```
hnm_WindowTypeID display_event_window_type_id(const char *type_string)
```

Arguments:

type_string

The JSON string used to represent the window type.

Library:

```
libhnm
```

Description:

This function has worst-case complexity of $O(m*n)$ where m is the average length of the window type string and n is the number of window types defined.

Returns:

An `hnm_WindowTypeID` that corresponds to the specified window type string. If no corresponding window type is found, `HNM_WINDOW_HIDDEN` is returned.

display_event_window_type_name()

Get the window type name that corresponds to the specified ID.

Synopsis:

```
#include <hnm/display_event.h>

const char* display_event_window_type_name(hnm_WindowTypeID type_id)
```

Arguments:

type_id

The WindowTypeID whose literal string representation is being sought.

Library:

libhnm

Description:

This function obtains the string literal that corresponds to the specified WindowTypeID in constant $O(1)$ time.

Returns:

The string literal the corresponds with the specified type ID.

hnm_display_event_appraise()

Appraise a display interaction request to determine if it should be serviced.

Synopsis:

```
#include <hnm/display_event.h>

bool hnm_display_event_appraise(hnm_Event *event, void *hnm_data)
```

Arguments:

event

The interaction event structure that encapsulates the display event.

hnm_data

The HNM data structure that provides access to the active event list and to the HNM policy configuration.

Library:

libhnm

Description:

This function analyzes the specified event to determine whether:

- the request corresponds to a display event

OR:

- given the current status of the HNM, the request should be serviced as is or using a fallback window type.

Returns:

A flag indicating whether the current display interaction request should be accepted (`true`) or rejected (`false`). If the event is not a display event, it will be rejected (`false`).

hnm_DisplayEvent_create()

Create a new DisplayEvent instance.

Synopsis:

```
#include <hnm/display_event.h>

hnm_DisplayEvent* hnm_DisplayEvent_create()
```

Arguments:**Library:**

libhnm

Description:

This function allocates a new `hnm_DisplayEvent` structure and initializes all its members to zero with the exception of the *appraise* and *service* callbacks, which will have the correct functions assigned to them.

Returns:

A pointer to a new `hnm_DisplayEvent` structure. The memory returned by this function is transferred to the calling context responsible for deleting it.

hnm_DisplayEventFactory_findEvent()

Search for a named event.

Synopsis:

```
#include <hnm/display_event.h>

hnm_DisplayEvent* hnm_DisplayEventFactory_findEvent(const char *event_name)
```

Arguments:

event_name

Name of the event.

Library:

libhnm

Description:

This function searches for the event specified. If multiple events have the same name, the first one encountered with the highest priority is returned.

Returns:

A queued event with the specified event name. Returns NULL if none are found.

hnm_DisplayEventFactory_getDefaultEvent()

Obtain a pointer to the default event instance.

Synopsis:

```
#include <hnm/display_event.h>

hnm_DisplayEvent* hnm_DisplayEventFactory_getDefaultEvent()
```

Arguments:**Library:**

libhnm

Description:

This function returns the *default event*, which is used when no other event is specified.

Returns:

The default event.

hnm_DisplayEventFactory_init()

Initialize the display event "factory".

Synopsis:

```
#include <hnm/display_event.h>

void hnm_DisplayEventFactory_init(struct hnm_config_node *config_tree)
```

Arguments:

config_tree

Configuration tree used to initialize the state of the internal factory.

Library:

libhnm

Description:

Display events are generated by a "factory" responsible for allocating and initializing new `DisplayEvent` instances. Although the internal state of the factory isn't visible to other compilation units, a factory API is defined to expose the factory itself as a black box.

Before the factory can be used, it must be initialized via the `hnm_DisplayEventFactory_init()` function.

NOTE: If the factory isn't initialized before it's used, the default policy for window types will be followed.

Returns:

Nothing.

hnm_DisplayEventFactory_InitWindowTypes()

Initialize the window types.

Synopsis:

```
#include <hnm/display_event.h>

void hnm_DisplayEventFactory_InitWindowTypes(struct hnm_config_node
*config_tree)
```

Arguments:

config_tree

Configuration tree searched for the window type semantic information.

Library:

libhnm

Description:

This function initializes the window type semantics from the policy configuration.

Given a configuration tree, this function finds the `window-types` section in the tree and then extracts the window type semantic information. The semantics are expressed by the internal HNM state machine. If the configuration tree doesn't contain a `window-types` section, then a default policy is used.

Returns:

Nothing.

hnm_display_event_service()

Service a display interaction request.

Synopsis:

```
#include <hnm/display_event.h>

void hnm_display_event_service(hnm_Event *event, void *hnm_data)
```

Arguments:

event

The interaction event structure that encapsulates the display event.

hnm_data

The HNM data structure that provides access to the active event list and to the HNM policy configuration.

Library:

libhnm

Description:

This function does the specialized processing required to service the specified display interaction event. The nature of the processing required may vary depending on the type of event received (i.e. `display-start` or `display-end`).

Returns:

Nothing.

event.h

Declaration of the HNM Event structure and its associated functions.

Definitions in *event.h*

Preprocessor macro definitions for the event.h header file in the libhnm library.

Definitions:

```
#define hnm_EventClassShift 8
```

Number of bits to shift; used in `hnm_EventTypeID` and `hnm_EventClassID`

```
#define hnm_EventClassMask (0xffff << hnm_EventClassShift)
```

Mask to extract class ID; used by `hnm_EventTypeID`, `hnm_EventClassID`, and `hnm_Event_typeof`

```
#define hnm_EventTypeMask ~hnm_EventClassMask
```

```
#define hnm_EventTypeID ( hnm_EventType )( class << hnm_EventClassShift | ( type & hnm_EventTypeMask ) )
```

Aggregate the event class and subtype IDs into a single event-type ID.

```
#define hnm_EventClassID ( hnm_EventClass )( event_id >> hnm_EventClassShift )
```

Extract the event class ID from the event-type ID.

```
#define hnm_Event_typeof ( event && hnm_EventClassID( event->type ) == class )
```

Macro used to perform runtime type-checking of events.

This macro evaluates to `true` if the specified `event` corresponds to the specified `event class`.

```
#define EVENT_BASE queue_Element          queue_elem ;          \
        char                name[ 256 ] ;          \
        hnm_Priority        priority ;           \
        hnm_EventType       type ;              \
        /* Callbacks associated with the Event structure. */          \
        bool                ( *appraise )( hnm_Event* self, void* data ) ; \
        void                ( *service )( hnm_Event* self, void* data )
```

`EVENT_BASE` defines the base structure for events.

`EVENT_BASE` contains the following:

- `queue_elem` The queue member of the Event structure. This member must be the first one defined in the structure to allow it to be used with the generic Queue data structure and its associated functions.

- `name[256]` The name of the event. If this string corresponds to an event name in the policy configuration, the associated priority will be assigned to events with that name.
- `priority` The priority of the event. The minimum value is `HNM_DEFAULT_PRIORITY`. The maximum value is `HNM_MAX_PRIORITY`.
- `type` The event type (e.g. `display_start` or `display_end`). A `display_start` event signifies that an application or service wishes to display some information in a window of a specified type. A `display_end` event may occur when an application no longer has information to display (e.g. if a handsfree phone call is terminated on the remote end).

The following callbacks are associated with the Event structure:

- `(*appraise)()` This callback is called to appraise the current event. This function takes a pointer to the HNM data structure and the event instance as arguments. It returns a Boolean flag to indicate whether to service the event.
- `(*service)()` This callback is called to service the current event.

Library:

`libhnm`

Typedefs in *event.h****hnm_event***

An abstract multimodal event structure.

Synopsis:

```
struct hnm_event hnm_Event {  
    EVENT_BASE ;  
};
```

Data:***EVENT_BASE***

Defines the base structure for events.

Library:

`libhnm`

Description:

The `hnm_Event` structure is the base type for asynchronous system events that are handled by the HNM. See [EVENT_BASE](#) (p. 49) for details.

hnm_EventClass

Enumeration of HNM event class IDs.

Synopsis:

```
#include <hnm/event.h>

typedef enum hnm_event_class hnm_EventClass;
```

Library:

libhnm

Description:

Events are categorized into classes that correspond to the interaction types (e.g., display, audio). The HNM subsystem defines an enumeration of event class IDs that are used to distinguish the different classes of events.

hnm_event_class

Enumeration of HNM event class IDs.

Synopsis:

```
#include <hnm/event.h>

typedef enum hnm_event_class{
    HNM_EVENT_NONE == 0
    HNM_EVENT_DISPLAY
    HNM_EVENT_UNKNOWN
} hnm_EventClass;
```

Data:

HNM_EVENT_NONE

HNM_EVENT_DISPLAY

HNM_EVENT_UNKNOWN

Add new event types before this entry.

Library:

libhnm

Description:

Events are categorized into classes that correspond to the interaction types (e.g., display, audio). The HNM subsystem defines an enumeration of event class IDs that are used to distinguish the different classes of events.

hnm_EventPriorityMap

This is an alias for the `hnm_event_priority_map` structure.

Synopsis:

```
#include <hnm/event.h>

typedef struct hnm_event_priority_map hnm_EventPriorityMap;
```

Library:

libhnm

Description:***hnm_event_priority_map***

Structure representing a single mapping of an event name to a priority value.

Synopsis:

```
struct hnm_event_priority_map hnm_EventPriorityMap {
    char event_name [256];
    hnm_Priority priority ;
    hnm_EventPriorityMap * next ;
};
```

Data:

char event_name[256]

The name of an event. The name can be at most 255 characters in length (plus a null terminator).

hnm_Priority priority

The priority that has been assigned to the named event.

hnm_EventPriorityMap * next

A pointer to the next event priority map in the global map list.

Library:`libhnm`**Description:*****hnm_EventType****Event Type ID.***Synopsis:**

```
#include <hnm/event.h>

typedef unsigned short hnm_EventType;
```

Library:`libhnm`**Description:*****hnm_Priority****A type representing the priority level.***Synopsis:**

```
#include <hnm/event.h>

typedef unsigned int hnm_Priority;
```

Library:`libhnm`**Description:****Enumerations in *event.h******hnm_EventClass****Alias for the event class ID type enumeration***Synopsis:**

```
#include <hnm/event.h>

typedef hnm_event_class hnm_EventClass;
```

Library:`libhnm`

Description:

This type is an alias for the event class ID type enumeration, [hnm_event_class](#) (p. 51).

hnm_event_class

Enumeration of HNM event class IDs.

Synopsis:

```
#include <hnm/event.h>

typedef enum hnm_event_class{
    HNM_EVENT_NONE == 0
    HNM_EVENT_DISPLAY
    HNM_EVENT_UNKNOWN
} hnm_EventClass;
```

Data:

HNM_EVENT_NONE

HNM_EVENT_DISPLAY

HNM_EVENT_UNKNOWN

Add new event types before this entry.

Library:

libhnm

Description:

Events are categorized into classes that correspond to the interaction types (e.g., display, audio). The HNM subsystem defines an enumeration of event class IDs that are used to distinguish the different classes of events.

hnm_priority_e

Enumeration of priority levels defined for the HNM subsystem.

Synopsis:

```
#include <hnm/event.h>

enum hnm_priority_e{
    HNM_MIN_PRIORITY == 0
    HNM_MAX_PRIORITY == 7
    HNM_NUM_PRIORITY_LEVELS
```

```
        HNM_DEFAULT_PRIORITY == HNM_MIN_PRIORITY  
};
```

Data:

HNM_MIN_PRIORITY

Lowest priority.

HNM_MAX_PRIORITY

Highest priority.

HNM_NUM_PRIORITY_LEVELS

HNM_DEFAULT_PRIORITY

Library:

libhnm

Description:

event-source.h

Structures and the register function for event-source plugins.

To register with the HNM, plugins must use the [*hnm_register_module\(\)*](#) (p. 59) function.

Typedefs in *event-source.h*

hnm_EventSource

This is an alias for the hnm_event_source structure.

Synopsis:

```
#include <hnm/event-source.h>

typedef struct hnm_event_source hnm_EventSource;
```

Library:

libhnm

Description:

hnm_event_source

Structure that defines the form of event-source plugins for the HNM.

Synopsis:

```
struct hnm_event_source hnm_EventSource {
    char * name ;
    int connection_id ;
    void * data ;
    hnm_EventPriorityMap * priority_map ;
    int(* open )(hnm_EventSource *event_source, int channel_id);
    void(* close )(hnm_EventSource *event_source);
    hnm_Event *(* read_event )(hnm_EventSource *event_source);
};
```

Data:

char * name

The name of the application that is associated with the event source (e.g. HFP for the Bluetooth handsfree phone system).

int connection_id

The connection ID (i.e. file descriptor or connection ID) used to communicate with the plugin. This member must be initialized in the `(*open)()` callback and is used to poll the plugin for input.

void * data

The private data associated with the specialized per-module `hnm_EventSource`.

hnm_EventPriorityMap * priority_map

The global mapping of event names to priorities for the current event-source instance.

int(* open)(hnm_EventSource *event_source, int channel_id)

Called by the HNM subsystem when a plugin is initially loaded via `dlopen()` to open the event source. Returns a file descriptor associated with the event source, if successful; otherwise, returns a negative value.

void(* close)(hnm_EventSource *event_source)

Called by the HNM subsystem when a plugin is being released via `dlclose()`.

hnm_Event *(* read_event)(hnm_EventSource *event_source)

Read the event parameters from the event source.

Library:

`libhnm`

Description:

This structure declares the interface used for dynamically loading code into an existing HNM runtime environment. Once this code is loaded, the known layout of objects can be relied upon to enable the module.

hnm_Module

This is an alias for the hnm_module structure.

Synopsis:

```
#include <hnm/event-source.h>

typedef struct hnm_module hnm_Module;
```

Library:

libhnm

Description:***hnm_module***

Structure that represents an event-source module.

Synopsis:

```
struct hnm_module hnm_Module {
    struct pollfd * poll_entry ;
    hnm_EventSource event_source ;
    hnm_Module * next ;
};
```

Data:***struct pollfd * poll_entry***

The entry in a poll list that is used to poll events on the module.

hnm_EventSource event_source

The event source that is plugged into the HNM via the current module instance.

hnm_Module * next

The next module in the list.

Library:

libhnm

Description:

Event-source modules are represented by a module type that encloses a definition of the specific event-source data and callbacks. This data structure also provides a mechanism to chain the modules in a list.

Functions in *event-source.h****hnm_register_module()***

Register the specified event source as a module of the HNM subsystem.

Synopsis:

```
#include <hnm/event-source.h>

hnm_Module* hnm_register_module(hnm_EventSource *event_source, void
*private_data)
```

Arguments:***event_source***

The event source being registered as a module with the HNM subsystem.

private_data

The private data to associate with the event source.

Library:

libhnm

Description:

Modules that wish to register with the HNM subsystem must call the *hnm_register_module()* function. This function adds the module to the list of managed event sources associated with the HNM subsystem. The new module is added to the front of the list of registered modules. This function is defined in the HNM subsystem's scope.

Returns:

A pointer to the *hnm_module* structure that is created to manage the specified event source.

messaging.h

Declaration of the Messaging PPS object used by the HNM.

Definitions in *messaging.h*

Preprocessor macro definitions for the messaging.h header file in the libhnm library.

Definitions:

```
#define HNM_PPS_MESSAGING_OBJECT_PATH "Messaging?server,nopersist"
```

Definition of the PPS path for the Messaging object.

The Messaging object is created as a *server object* with persistence disabled. Clients connect to the Messaging server to receive messages informing them that a transient notification is ready to be displayed.

```
#define HNM_PPS_MESSAGING_INITIALIZE {                                     \
    .type = HNM_PPS_OBJECT_MESSAGING,                                  \
    .fd = -1,                                                           \
    .path = HNM_PPS_MESSAGING_OBJECT_PATH,                             \
    .pollfd = NULL,                                                    \
    .object_data = NULL,                                              \
    .pps_handler = hnm_Messaging_ppsHandler,                          \
    .pps_update = NULL,                                              \
};
```

HNM_PPS_MESSAGING_INITIALIZE defines the static initializer for the Messaging PPS object. This initializes a static declaration of an `hnm_Messaging` object.

HNM_PPS_MESSAGING_INITIALIZE sets the specified structure members:

- `.type` PPS object ID used by the Messaging object.
- `.fd` File descriptor (-1) for the Messaging object.
- `.path` Path to the Messaging object (e.g., `/pps/services/hmi-notification/Messaging`).
- `.pollfd` List entry used to poll for events on the associated PPS object.
- `.object_data` Object-specific data.
- `.pps_handler` Pointer to the `hnm_Messaging_ppsHandler()` function.
- `.pps_update` Internal use only.

Library:

libhnm

Typedefs in *messaging.h*

hnm_Messaging_Client

Alias for hnm_messaging_client_s.

Synopsis:

```
#include <hnm/messaging.h>

typedef struct hnm_messaging_client_s hnm_Messaging_Client;
```

Library:

libhnm

Description:

This is an alias for the `hnm_messaging_client_s` structure.

hnm_messaging_client_s

Structure used to represent a Messaging client.

Synopsis:

```
struct hnm_messaging_client_s hnm_Messaging_Client {
    char * id ;
};
```

Data:

char * id

The PPS ID of the subscribed client.

Library:

libhnm

Description:

The [*hnm_messaging_client_s*](#) (p. 61) structure encapsulates data about clients that subscribe to the Messaging PPS object.

hnm_Messaging

Alias for hnm_messaging_s.

Synopsis:

```
#include <hnm/messaging.h>

typedef struct hnm_messaging_s hnm_Messaging;
```

Library:

libhnm

Description:

This is an alias for the hnm_messaging_s structure.

hnm_messaging_s

A structure representing the Messaging PPS object.

Synopsis:

```
struct hnm_messaging_s hnm_Messaging {
    PPS_OBJECT_BASE ;
    hnm_Messaging_Client * clients ;
    int num_clients ;
};
```

Data:

PPS_OBJECT_BASE

Defines the base structure for PPS objects.

hnm_Messaging_Client * clients

The list of clients that have subscribed to the Messaging PPS object.

int num_clients

The number of clients that have subscribed to the Messaging PPS object.

Library:

libhnm

Description:

This object is a specialization of the *pps_Object* (p. 66) structure that provides a mechanism that a notification manager can use to send transient notifications to subscribed clients. This structure can be used directly with `pps_Object` methods.

Functions in *messaging.h****hnm_Messaging_ppsHandler()***

Handle PPS messages to the Messaging object.

Synopsis:

```
#include <hnm/messaging.h>

void hnm_Messaging_ppsHandler(pps_Object *pps_object)
```

Arguments:

pps_object

A pointer to the location in memory of a PPS Object. This object provides the private data used by this call to handle the request.

Library:

libhnm

Description:

The *hnm_Messaging_ppsHandler()* function parses an incoming PPS message and determines whether to connect or disconnect the client from the `Messaging` object.

Returns:

Nothing.

hnm_Messaging_sendTransient()

Send a transient notification request to all connected clients.

Synopsis:

```
#include <hnm/messaging.h>

void hnm_Messaging_sendTransient(hnm_Messaging *messaging_obj, const hnm_Event
*event)
```

Arguments:

messaging_obj

A pointer to the messaging object that manages the connected client list. The clients of this server object are notified of the transient message specified by *event*.

event

The event structure used to construct the transient notification. The ownership of this event is retained by the calling context responsible for deleting it.

Library:

libhnm

Description:

The *hnm_Messaging_sendTransient()* function sets up a PPS object describing a transient notification that must be shown by the HMI. This message is transmitted to each client that is connected to the `Messaging` object. Presumably, each connected client represents a different HMI.

Returns:

Nothing.

pps.h

Declaration of a generic PPS object structure.

Definitions in pps.h

Preprocessor macro definitions for the pps.h header file in the libhnm library.

Definitions:

```
#define PPS_OBJECT_BASE pps_ObjectId    type ;
    \
    int          fd ;                      \
    char         path[ 517 ] ;             \
    struct pollfd* pollfd ;                \
    void*        object_data ;             \
    void         (*pps_handler)( pps_Object* object ) ; \
    void         (*pps_update)( pps_Object* object ) ; \
    int          (*open)( pps_Object* self, const char* basename ) ; \
    \
    void         (*close)( pps_Object* self ) ; \
    int          (*read)( pps_Object* self, char** buffer ) ; \
    int          (*write)( pps_Object* self, const char* pps_data,
unsigned pps_data_size ) ; \
    int          (*addToPollList)( pps_Object* self, struct pollfd
poll_list[], unsigned poll_list_size )
```

PPS_OBJECT_BASE defines the base structure for PPS objects.

PPS_OBJECT_BASE contains the following:

- `type` The object ID that identifies the derived object type.
- `fd` The file descriptor of the PPS object. This can be passed to a `poll()` system call to wait for input on the associated PPS object. If `fd` is less than zero, then the PPS object is currently closed.
- `path` The path of the PPS object. The length of the path can be a maximum of 517 characters. This path is relative to the base PPS URI assigned to the PPS Object subsystem.
- `pollfd` A pointer to a `pollfd` list entry used to poll for events on the associated PPS object.
- `object_data` A pointer to object-specific data. This mechanism provides some rudimentary polymorphism by associating object-specific member data with the high-level PPS object.
- `pps_handler` A pointer to a handler callback function that's called by the HNM system whenever a PPS message is received from a connecting client.
- `pps_update` Callback to push changes to the Status object to subscribers of the PPS interface.

NOTE: For the functions contained here, see [Functions in pps.h](#) (p. 67).

Library:

libhnm

Typedefs in *pps.h*

pps_Object

Alias for pps_object.

Synopsis:

```
#include <hnm/pps.h>

typedef struct pps_object pps_Object;
```

Library:

libhnm

Description:

This is an alias for the `pps_object` structure.

pps_object

PPS Object structure data.

Synopsis:

```
struct pps_object pps_Object {
    PPS_OBJECT_BASE ;
};
```

Data:

PPS_OBJECT_BASE

Defines the base structure for PPS objects.

Library:

libhnm

Description:

The [pps_object](#) (p. 66) structure provides a uniform interface for all PPS object definitions. To create a derived `pps_Object` type, declare `PPS_OBJECT_BASE` as the first member and then declare any specialized members. This allows the specialized object to be processed via the various PPS functions defined here.

pps_ObjectId

Type representing PPS object IDs.

Synopsis:

```
#include <hnm/pps.h>

typedef unsigned pps_ObjectId;
```

Library:

libhnm

Description:**Functions in pps.h****pps_encoder_strerror()**

Get the error message that corresponds to the encoder error code.

Synopsis:

```
#include <hnm/pps.h>

const char* pps_encoder_strerror(pps_encoder_error_t code)
```

Arguments:

code

A `pps_encoder` error code for which a matching error message is being requested.

Library:

libhnm

Description:

Use the `pps_encoder_strerror()` function to determine the cause of a failure reported by a PPS *encoder* object.

Returns:

A string containing an error message that corresponds to the specified error `code`.

pps_decoder_strerror()

Get the error message that corresponds to the decoder error code.

Synopsis:

```
#include <hnm/pps.h>

const char* pps_decoder_strerror(pps_decoder_error_t code)
```

Arguments:***code***

A `pps_decoder` error code for which a matching error message is being requested.

Library:

libhnm

Description:

Use the `pps_decoder_strerror()` function to determine the cause of a failure reported by a PPS *decoder* object.

Returns:

A string containing an error message that corresponds to the specified error code.

pps_Object_addToPollList()

Add an open PPS object to the specified poll list.

Synopsis:

```
#include <hnm/pps.h>

int pps_Object_addToPollList(pps_Object *pps_object, struct pollfd poll_list[],
    unsigned poll_list_size)
```

Arguments:***pps_object***

A pointer to the structure that represents the PPS object's file descriptor to be added to the specified poll list.

poll_list

An array that represents the list of file descriptors to poll. The specified PPS object will be added to this list if possible.

poll_list_size

The size of the provided poll list. This represents the maximum number of file descriptors that can be added to the list.

Library:

libhnm

Description:

The *pps_Object_addToPollList()* function adds an open PPS object to the specified poll list and then updates the PPS object's `poll_fd` member to point to the poll entry that corresponds to the object. This allows the revents mask associated with the PPS object to be accessed directly via the object's structure.

Returns:

The index of the current structure in the poll list.

pps_Object_close()

Close the specified PPS object.

Synopsis:

```
#include <hnm/pps.h>

void pps_Object_close(pps_Object *pps_object)
```

Arguments:***pps_object***

A pointer to the structure that represents the PPS object to be closed by this call. If the object isn't open, this call has no effect.

Library:

libhnm

Description:

The *pps_Object_close()* function closes the object that was previously opened by *pps_Object_open()* (or by any other means). If the specified object doesn't correspond to an open PPS object, this function has no effect.

Returns:

Nothing.

pps_Object_open()

Open or create the PPS object specified by the provided pps_Object instance.

Synopsis:

```
#include <hnm/pps.h>

int pps_Object_open(pps_Object *pps_object, const char *basename)
```

Arguments:***pps_object***

The *pps_Object* structure that represents the PPS object that is being opened by this call.

basename

The base path where the PPS object can be found in the filesystem (usually under */pps*).

Library:

libhnm

Description:**Returns:**

The file descriptor of the opened PPS object file.

pps_Object_read()

Stream data from the PPS object.

Synopsis:

```
#include <hnm/pps.h>
```

```
int pps_Object_read(pps_Object *pps_object, char **buffer)
```

Arguments:***pps_object***

A pointer to the structure that represents the PPS object to read data from. This object must be open for the read function to complete successfully.

buffer

An output argument that provides a pointer to the buffer that was created to hold data read from the PPS object. The ownership of the memory referenced by this argument is transferred to the calling context responsible for deleting it.

Library:

```
libhnm
```

Description:

Use the *pps_Object_read()* function to stream data from the PPS object one line at a time. You must call this function consecutively as many times as required to read complete PPS messages that may span more than one line.

Returns:

The number of bytes read from the PPS object.

pps_Object_write()

Send a stream of PPS data to a PPS object.

Synopsis:

```
#include <hnm/pps.h>
```

```
int pps_Object_write(pps_Object *pps_object, const char *pps_data, unsigned pps_data_size)
```

Arguments:***pps_object***

A pointer to the structure that represents the PPS object to write data to. This object must be open for the write function to complete successfully.

pps_data

A buffer containing the PPS string data to write to the PPS object. This data must be NUL-terminated character data that represents valid PPS messages.

pps_data_size

The number of bytes of PPS data contained in the data buffer. The maximum PPS message size is 1024 bytes.

Library:

libhnm

Description:**Returns:**

The number of bytes that were actually written to the PPS object.

queue.h

Declaration of a generic queue data structure.

Typedefs in *queue.h*

queue_Element

Alias for queue_element.

Synopsis:

```
#include <hnm/queue.h>

typedef struct queue_element queue_Element;
```

Library:

libhnm

Description:

This is an alias for the queue_element structure.

queue_element

A data structure that represents a single element in a queue.

Synopsis:

```
struct queue_element queue_Element {
    queue_Element * prev ;
    queue_Element * next ;
    queue_Queue * queue ;
};
```

Data:

queue_Element * prev

The previous element in the queue. If this member is NULL, then the element is likely the first item in the queue.

queue_Element * next

The next element in the queue. If this member is NULL, the element is likely the last in the queue.

queue_Queue * queue

The queue that the element belongs to. If this member is NULL, the element hasn't been added to a queue.

Library:

libhnm

Description:***queue_Queue***

Alias for queue_queue.

Synopsis:

```
#include <hnm/queue.h>

typedef struct queue_queue queue_Queue;
```

Library:

libhnm

Description:

This is an alias for the queue_queue structure.

queue_queue

A generic Queue data structure.

Synopsis:

```
struct queue_queue queue_Queue {
    queue_Element * head ;
    queue_Element * tail ;
};
```

Data:***queue_Element * head***

The first element or head of the queue. If head is NULL, the queue is necessarily empty.

queue_Element * tail

The last element or tail of the queue. If `tail` is NULL, the queue is necessarily empty.

Library:

libhnm

Description:**Functions in *queue.h******queue_delete()***

Delete an arbitrary element from a queue.

Synopsis:

```
#include <hnm/queue.h>

void queue_delete(void *element)
```

Arguments:***element***

A pointer to the element to remove from its queue.

Library:

libhnm

Description:

The *queue_delete()* function deletes an arbitrary event *element* from its queue.

Returns:

Nothing.

queue_get_head()

Get a pointer to the first element in the specified queue.

Synopsis:

```
#include <hnm/queue.h>

void* queue_get_head(queue_Queue queue)
```

Arguments:

queue

The queue structure whose head element will be retrieved.

Library:

libhnm

Description:

The *queue_get_head()* function returns a void pointer to the first element in the specified queue. This pointer can be cast by the calling context to the expected storage type of the queue.

Returns:

A pointer to the head element in *queue*. If the queue is empty, the returned value will be NULL.

queue_has_element()

Test whether an element is part of a queue.

Synopsis:

```
#include <hnm/queue.h>

bool queue_has_element(const queue_Queue *queue, const void *element)
```

Arguments:

queue

The queue for which the element is being tested for membership.

element

A pointer to a specialized structure that contains a `queue_Element` structure as its first member. This element is tested for membership in the specified *queue*.

Library:

libhnm

Description:

Given a `queue_element` data structure, the `queue_has_element()` function determines whether it belongs to the specified *queue*. This test is performed in constant $O(1)$ time. Both the queue and element data remain unchanged following a call to this function.

Returns:

A Boolean flag indicating whether *element* is (`true`) or is not (`false`) a member of *queue*.

queue_is_empty()

Test whether the specified queue is empty.

Synopsis:

```
#include <hnm/queue.h>

bool queue_is_empty(const queue_Queue *queue)
```

Arguments:

queue

The queue whose cardinality will be evaluated.

Library:

libhnm

Description:

The `queue_is_empty()` function checks the cardinality of the specified queue and reports success if it evaluates to zero.

Returns:

`true` if there are no elements in the specified queue; `false` otherwise.

queue_next_element()

Move to the next element following the specified one.

Synopsis:

```
#include <hnm/queue.h>

void* queue_next_element(void *element)
```

Arguments:***element***

A pointer to an element structure that represents the queue element that is currently being visited.

Library:`libhnm`**Description:**

The *queue_next_element()* function can be used to iterate over queued elements from the head toward the tail.

Returns:

A pointer to the containing structure of the next element or NULL if no such element exists.

queue_pop()

Remove the head element from the specified queue.

Synopsis:

```
#include <hnm/queue.h>

queue_Element* queue_pop(queue_Queue *queue)
```

Arguments:***queue***

A pointer to a queue data structure whose head element is to be removed.

Library:`libhnm`**Description:**

The *queue_pop()* function will remove the head `queue_Element` instance from the specified queue. The function is robust to instances where the provided queue is empty; the queue remains unchanged in this scenario.

Returns:

A pointer to the `queue_Element` that was removed from the queue structure. If the `queue_Element` is the first element of a containing structure, you should be able to cast this pointer to the enclosing structure's type to access the specialized structure's members.

queue_push()

Add an element to the tail of the queue.

Synopsis:

```
#include <hnm/queue.h>

bool queue_push(queue_Queue *queue, void *element)
```

Arguments:***queue***

A pointer to the queue structure to which the element is being added.

element

A pointer to a specialized structure that contains a `queue_Element` structure as its first element. This element is added to the specified *queue* and its `queue_Element` members are updated.

Library:

libhnm

Description:

Given a `queue_Element` data structure, the *queue_push()* function adds the enclosing data structure to the specified *queue*. This operation is performed in constant $O(1)$ time. The `queue_Element` structure that adorns the enclosing structure as well as the queue structure itself are updated accordingly.

Note that an element can only ever be a member of a *single* queue. If the supplied element is already a member of another queue, the push operation will fail.

Returns:

A flag indicating whether the element was (`true`) or was not (`false`) successfully added to the queue.

status.h

Declaration of the PPS Status object used by the HNM.

Definitions in *status.h*

Preprocessor macro definitions for the status.h header file in the libhnm library.

Definitions:

```
#define EXTERN extern
```

```
#define STATUS_PPS_OBJECT_ID 0x0
```

Definition of the PPS object ID used by the Status object.

```
#define hnm_Status_Narrow ( obj && obj->type == STATUS_PPS_OBJECT_ID ) ? ( hnm_Status* )obj : NULL
```

Narrow a pps_Object to an hnm_Status structure whenever possible.

```
#define HNM_PPS_STATUS_OBJECT_PATH "Status"
```

Path to the PPS Status object.

```
#define HNM_PPS_STATUS_INITIALIZE {
    .type = STATUS_PPS_OBJECT_ID,
    .fd = -1,
    .path = HNM_PPS_STATUS_OBJECT_PATH,
    .pollfd = NULL,
    .object_data = NULL,
    .pps_handler = hnm_Status_ppsHandler,
    .pps_update = hnm_Status_update,
    .displayList = hnm_Status_displayList,
    .getDisplayList = hnm_Status_getDisplayList,
    .findDisplayEvent = hnm_Status_findDisplayEvent,
}
```

HNM_PPS_STATUS_INITIALIZE defines the static initializer for the Status PPS object. This initializes a static declaration of an hnm_Status object, allowing applications that use the Status object to assign custom type IDs that are unique to the application scope.

HNM_PPS_STATUS_INITIALIZE sets the specified structure members:

- `.type` PPS object ID used by the Status object.
- `.fd` File descriptor (-1) for the Status object.
- `.path` Path to the Status object (e.g., `/pps/services/hmi-notification/Status`).
- `.pollfd` List entry used to poll for events on the associated PPS object.
- `.object_data` Object-specific data.
- `.pps_handler` Pointer to the `hnm_Status_ppsHandler()` function.
- `.pps_update` Pointer to the `hnm_Status_update()` function.
- `.displayList` Pointer to the `hnm_Status_displayList()` function.
- `.getDisplayList` Pointer to the `hnm_Status_getDisplayList()` function.

- `.findDisplayEvent` Pointer to the `hnm_Status_findDisplayEvent()` function.

Library:

libhnm

Typedefs in *status.h****hnm_Status***

This is an alias for the `hnm_status_s` structure.

Synopsis:

```
#include <hnm/status.h>

typedef struct hnm_status_s hnm_Status;
```

Library:

libhnm

Description:***hnm_status_s***

Data structure for the PPS Status object.

Synopsis:

```
struct hnm_status_s hnm_Status {
    PPS_OBJECT_BASE ;
    queue_Queue display_list ;
    bool update_display_list ;
    const queue_Queue *(* displayList )(hnm_Status *self);
    queue_Queue *(* getDisplayList )(hnm_Status *self);
    hnm_DisplayEvent *(* findDisplayEvent )(hnm_Status *self, const char
*event_name);
};
```

Data:***PPS_OBJECT_BASE***

Defines the base structure for PPS objects.

queue_Queue display_list

A list of events that affect the display modality of a subscribed HMI.

bool update_display_list

A flag that specifies whether the `display_list` has been changed since the last update of the `Status` object. This is set automatically by some of the accessor functions to ensure that changes to the display list are propagated to clients that subscribe to the `Status` object.

const queue_Queue *(*displayList)(hnm_Status *self)

Callback referencing the function that provides read-only access to the display list.

queue_Queue *(*getDisplayList)(hnm_Status *self)

Callback that provides read/write access to the display list.

hnm_DisplayEvent *(*findDisplayEvent)(hnm_Status *self, const char *event_name)

Callback that provides an accessor to find a named display event in the display list associated with the current `Status` object.

Library:

`libhnm`

Description:

The `hnm_Status` structure is a specialization of the [pps_Object](#) (p. 66) structure and therefore has an instance of that structure as its first member. This specialization enhances the `pps_Object` with data that is specific to the PPS `Status` object.

Functions in *status.h*

hnm_Status_displayList()

Get the display event list of a PPS Status object.

Synopsis:

```
#include <hnm/status.h>

const queue_Queue* hnm_Status_displayList(hnm_Status *self)
```

Arguments:

self

A pointer to a pps_Object instance that is also an hnm_Status structure. This object is the recipient of the `displayList` message.

Library:

libhnm

Description:

Given a pps_Object, this function:

- verifies that this object represents an hnm_Status type
- returns the display list associated with that object.

Returns:

A pointer to the display list associated with the `status` object. If the object isn't of the correct type, NULL is returned. The ownership of the memory referenced by the pointer is retained by the called context and must not be deleted by the calling context.

hnm_Status_findDisplayEvent()

Find the named event in the display list.

Synopsis:

```
#include <hnm/status.h>
```

```
hnm_DisplayEvent* hnm_Status_findDisplayEvent(hnm_Status *self, const char
*event_name)
```

Arguments:***self***

A pointer to the hnm_Status structure that represents the recipient object of the method invocation.

event_name

The name of the event being sought in the display list.

Library:

libhnm

Description:

The `hnm_Status_findDisplayEvent()` function walks the display list associated with the specified `Status` object until either an event with the specified name is found or the end of the list is reached.

Returns:

A pointer to an `hnm_DisplayEvent` whose name matches `event_name`. If no such event is found, this function returns `NULL`.

hnm_Status_getDisplayList()

Get the display list of a PPS status object and set the update flag.

Synopsis:

```
#include <hnm/status.h>

queue_Queue* hnm_Status_getDisplayList(hnm_Status *self)
```

Arguments:

self

A pointer to the `hnm_Status` instance that receives the `getDisplayList` message.

Library:

`libhnm`

Description:

Given a `pps_Object`, this function:

- verifies that this object represents an `hnm_Status` type
- returns the display list associated with that structure.

This function also sets the update flag associated with the display list to ensure that changes are propagated to subscribers of the PPS object.

Returns:

A pointer to the display list associated with the `Status` object. If the object isn't of the correct type, `NULL` is returned. The ownership of the memory referenced by the pointer is retained by the called context and must not be deleted by the calling context.

hnm_Status_ppsHandler()

Handle PPS I/O.

Synopsis:

```
#include <hnm/status.h>

void hnm_Status_ppsHandler(pps_Object *object)
```

Arguments:

object

The Object that received a message via PPS.

Library:

libhnm

Description:

This reads PPS messages from the PPS interface of the *Status* object and then handles messages appropriately.

Returns:

Nothing.

hnm_Status_update()

Update the PPS object to reflect the current status.

Synopsis:

```
#include <hnm/status.h>

void hnm_Status_update(pps_Object *self)
```

Arguments:

self

A pointer to an *hnm_Status* object whose updates are pushed to subscribers of the PPS interface. If the argument doesn't correspond to an actual status object, no update takes place.

Library:

`libhnm`

Description:

The *hnm_Status_update()* function updates the `Status` object to ensure that subscribers are notified of changes.

Returns:

Nothing.

Index

C

- configuration file (HNM) 15, 16
 - comments in 15
 - predefined sections in 16

D

- display 13
 - sharing (HNM) 13

E

- event-source plugins (HNM) 19, 20, 22, 24
 - Generic 20
 - HandsFreePhone 22
 - VirtualMechanic 24
- event-source-generic.so 19
- event-source-handsfree.so 19
- event-source-vm.so 19
- events (HNM) 11

G

- Generic plugin (HNM) 20

H

- HandsFreePhone plugin (HNM) 22
- HFP states (HNM) 22
- HMI Notification Manager, See HNM

- HNM 9, 11, 12, 13, 15, 17, 19, 29
 - apps can share the display 13
 - command-line options 9
 - components 9
 - configuration file 15
 - display-control flags 13
 - event-source plugins 19
 - events 11
 - PPS objects 29
 - priorities 11
 - window types 12, 17

P

- policy.cfg 15
- PPS objects (HNM) 29
- priorities (HNM) 11

T

- Technical support 8
- Typographical conventions 6

V

- VirtualMechanic plugin (HNM) 24

W

- window types (HNM) 12, 17

