4S Device Communication Module Collection

Version 0.6-SNAPSHOT

Generated by Doxygen 1.8.11

Contents

Chapter 1

Introduction

The 4S Device Communication module collection (4SDC) is a cross platform library for applications interacting with personal health devices (like blood pressure monitors, oximeters, thermometers, weight scales and so on).

4SDC comprises a cross-platform library written in C++ which is able to communicate through a Platform Abstraction Layer (PAL) with platform-specific modules.

This manual is aimed at developers who are either going to use modules from the 4SDC module collection or need a general technical overview of the 4SDC architecture. The content of this manual will therefore not cover implementation details.

This manual is also available as interactive web pages. Please refer to www.4s-online.dk/4SDC/↩ Documentation for the relevant documentation.

Note

> Developers already familiar with the architecture, who are planning to contribute to the 4SDC modules, are encouraged to head over to the *Full Documentation*, which contains all the hairy implementation details.
> The latest version of the *Full Documentation* can be found here:
> www.4s-online.dk/4SDC/Documentation/latest/full/

1.1 Documentation Overview

This manual is organised with a number of introductory chapters followed by chapters with detailed information about each of the modules, classes and files. The detailed information in the last (huge) chapters may be easier to navigate on the interactive web pages version of the manual.

If you are new to the world of 4SDC, we recommend that you read the Getting Started chapter on page 3, and continue through the following chapters about the 4SDC architecture.

1.2 Other Sources of Information

The documentation found in this document is generated from the source code of a certain version of the 4SDC library, and is therefore rather static in nature. However, there are a few other sources of 4SDC information, you should be aware of – all of them are more dynamic, and of course you are invited to contribute as well:

Wiki The 4SDC Wiki page at 4s-online.dk/wiki/doku.php?id=4sdc: is used first of all as a general overview over the 4SDC module collection and secondly as the tutorial on how to get the demo application up and running – on various platforms.

Issuetracker The issuetracker at issuetracker4s.atlassian.net/projects/SDC/ is used to report bugs and feature requests, and to coordinate the work on the 4SDC modules.

Forum The forum at 4s-online.dk/forum/ is used for Q/A and discussions.

Chapter 2

Getting Started

This chapter will guide you through the steps necessary to download the 4SDC source code and get the HTML/JS demo application successfully up and running. Furthermore, the final section will offer some instructions about how to contact 4S and how to contribute.

2.1 Download the 4SDC Sources and Demo Application

4SDC comprises a cross-platform library written in C++ which is able to communicate through a Platform Abstraction Layer (PAL) with platform-specific modules. We currently have some support for Android, iOS, Linux and OS X.

Furthermore, there is a demo application, `DemoHtmlIntegration`, which will present a web-browser window, and provide access from the JavaScript code executed within this browser window to connected personal health devices. This demo application is based on Qt5 (see `www.qt.io`).

Currently, the 4SDC core libraries and the demo application is kept in the same repository. This is only temporary! Stay tuned, as the core libraries will be moving out to a different repository in the (near?) future.

The 4SDC source code (the library along with the demo application) may either be downloaded as a single Zip package from `bitbucket.org/4s/4sdcdemo/downloads`, or you can use Git to clone the code from `bitbucket.org/4s/4sdcdemo.git`.

2.2 Finding the Documentation

An introduction to the the 4SDC module collection is available at the 4S wiki (`4s-online.dk/wiki/doku.php?id=4sdc:`). **This would be the place to begin for newcomers.**

The technical documentation (which you are currently reading) is embedded in the source code and generated by Doxygen. There are two variants of the technical documentation available: the default, also known as "Public Interfaces", will provide the reader with the information necessary to *use* the 4SDC modules; the other one is known as "Full Developer' documentation", and its contents is aimed at developers working on the internals of 4SDC. You are currently reading the "Public Interfaces" variant.

The technical documentation is available at `4s-online.dk/4SDC/Documentation`, and you can also build them yourself. Please refer to the instructions at the top of `Doxyfile` (found in the repository root folder) for the required tools to do so.

2.3 Executing the Demo Application

Currently, the only supported/documented way to build and run the 4SDC library is together with the demo application. The instructions to do so is available at this link: 4s-online.dk/wiki/doku.php?id=4sdc:demoproject:. Please feel free to contribute to these wiki pages with any issue you come across when testing the demo on *your* platform, in order to help others get up and running as smoothly as possible. The demo should currently run (with various device support) on Android, iOS, Linux and OS X. Only Android and iOS are officially supported, though!

2.4 Contact and How To Contribute

If you encounter bugs or have a feature request, our issue tracker is available at issuetracker4s.atlassian.net/projects/SDC/. Please read the 4S issuetracking guidelines at 4s-online.dk/wiki/doku.php?id=process:overview before using it.

Source code contributions can be submitted as pull requests. Again, please read the 4S source contribution guidelines at 4s-online.dk/wiki/doku.php?id=source:overview before submitting. Furthermore, please contact us at software@4s-online.dk before you start making contributions, in order to coordinate the work.

Improvements to our wiki-guide on building and running the demo application at 4s-online.dk/wiki/doku.php?id=4sdc:demoproject: are also welcome. Take a look at 4s-online.dk/wiki/doku.php?id=participating:wiki for a guide on how to contribute to the wiki.

Finally, if you have any questions whatsoever (regarding 4SDC, that is), feel free to ask it on our forum at 4s-online.dk/forum/!

Chapter 3

Architecture

The 4SDC module collection is born out of – and closely linked to – the OpenTele3 architectural design efforts, which you can read more about here: `4s-online.dk/wiki/doku.php?id=opentele3:`. The General Principles of the OpenTele3 architecture are all fundamental in the architecture of 4SDC as well. Furthermore, the 4SDC project is currently used by the 4S organisation to develop the governance models needed to support certification according to the Medical Devices Directive – specifically the IEC 62304 [1] software life cycle processes.

The OpenTele3 architectural design relates to the Danish *Reference Architecture for Collecting Health Data From Citizens* ([2] and [3]). This reference architecture in turn points toward the *Continua Design Guidelines* (CDG) [4] as a technical framework for collecting the data from personal health devices. As this is the core purpose of 4SDC, a basic understanding of the CDG would be a good place to begin. A brief introduction to the CDG can be found in this *Fundamentals of Data Exchange* white paper [5].

The purpose of this chapter is to describe the 4SDC overall architecture, as well as the reasoning behind it. To set the stage, the following sections will present the scope of this module collection and explain some of the key terms and concepts. In section 3.3 on page 8 the basic requirements and their implications on the overall architecture are presented. This will then lead to the presentation of the (goal) architecture for the 4SDC module collection in section 3.4 on page 8. Finally, section 3.5 on page 8 will outline the status of the current 4SDC library, and how we plan to complete the implementation of this architecture.

3.1 Scope

The 4SDC library is not intended to be just another implementation of communication standards, rather it is designed to be a framework adapting to the users' needs, drawing on the extensive experience from OpenTele. To better understand what the 4SDC is designed to accomplish, consider the analogy from Figure 3.1.

To complete this analogy, think of OpenTele – or any other application using 4SDC – as the "application" and a personal health device as the "printer". The standards mandated by the CDG (such as Bluetooth HDP [6] and IEEE 11073 PHD [7]) or any proprietary device communication will play the role as "communication protocol" and "document format", and finally 4SDC itself will be the "printing system". As this analogy illustrates, the purpose of the "printer drivers" of 4SDC is not merely to provide communication between the application and the device, but their purpose is also to focus on the users' needs, providing in-context user instructions and help/guides.

So while the CDG have a significant influence on the architecture of 4SDC, many important features in 4SDC are completely out of their scope. Furthermore, 4SDC must support any type of device, and cannot be limited to only Continua compliant devices. Focus is on the needs of OpenTele, and in OpenTele a number of non-Continua-compliant devices are currently in use. Most likely, this will always be the case, as there will always be a need for using new and experimental (or proprietary) devices (at least in research projects). Furthermore, as a new device type is introduced to the market, it will take a couple of years before a standard way of communicating with this device is ratified, and in the meantime only non-standard communication protocols can be available.

You are about to print a hardcopy of a document on a printer. In order to do so, your application must communicate with the printer, so there must be a *communication protocol* in place (for instance communication over a local network). Furthermore, your application must provide the printer with a description of what the printed page should look like, so the application and printer must share a *document format* (PostScript is a commonly used format for this). This is all you will need to get the document hardcopy – if all goes well, that is. For what will happen if your printer is out of paper or toner, or perhaps experiences a paper jam? Or what if you want to print on both sides of the paper or want to use the built-in stapler? You would want your application to help you with that, right?
Now, if you have 10 different applications and 4 types of printers, and each application should be able to handle all printers, you would need 40 integration efforts to make that possible!
Introducing the operating system's *printing system* with *printer drivers*: rather than having each application communicating directly with the printer, the application will deliver its document to the OS printing system which will use one of the just 4 different printer drivers to interact with your printer. The printer drivers will offer access to the advanced settings of the printer (such as duplex printing or stapling), and help you troubleshoot problems – typically with images or even animated guides instructing you to fix paper jams or change the toner cartridge.

Figure 3.1 A story with an analogy from the world of printing

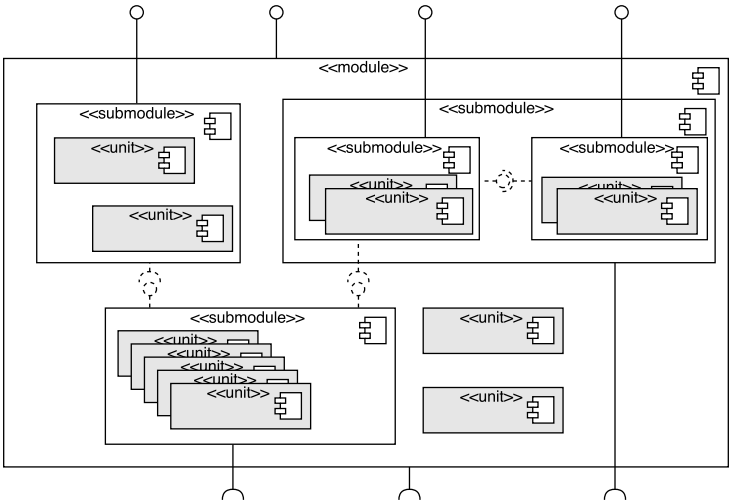


Figure 3.2 Example of the relation between *modules*, *submodules*, and *units*.

3.2 Terminology

Before we dig into the architecture details, this section will list the terms and definitions used in the remainder of this document.

3.2.1 Components

Software components are classified and grouped using the following terms (cf. Figure 3.2):

Unit This word is used to indicate the smallest / atomic software components – typically a class or a compilation unit (i.e. a source file). This is different from a module (defined below) as a unit need not have any public interfaces, nor does it have to adhere to the module contract. A (very small) module may also be a unit, however, this is probably a rare case.

Module and Module interface A *module* is a software component, which is independently replaceable (and upgradable). It is fully defined by a set of *module interfaces* (which are always public), and other modules exposing the same module interfaces may be used as drop-in replacements (possibly resulting in different functionality). A module must adhere to the *Module Contract* described in section 3.3.3 on page 8.

Submodule Larger modules may be constructed by a number of smaller components, called *submodules*, which in turn may comprise more submodules and so on. Submodules are *not* subject to the *Module Contract* described in section 3.3.3 on page 8, although they may adhere to some or all of the requirements. (Typically submodules will expose private interfaces.) See Figure 3.2 for an example of this.

Module collection This term is used to signify a group of modules sharing *governance*, such as the 4SDC *Module Collection*. The modules of the collection will reside in the same source code repository and share version/revision numbers etc. A module collection may comprise modules that are not normally used together (thus, not belonging to the same *runtime library*). For instance, one subsets of the 4SDC module collection may be used to build a device communication *library* for *Android* and a different subset will build a library for *iOS*.

Layer This is another term used to group related modules based on their function / purpose. Please read section ?? for more information. Notice that two modules may belong to different module collections but be in the same layer or vice versa.

Todo link to architecture.

3.2.2 Terms inherited from CDG

The following terms are inherited from the Continua Design Guidelines (CDG)[4] and all the related standards, these guidelines recommends:

Personal Health Device (PHD) This is the general term used for all the kinds of devices that may be connected to 4SDC. The term comprises medical sensors (such as blood pressure monitors, thermometers, oximeters etc.), medical actuators (e.g. insulin pumps), as well as health and fitness devices (e.g. weight scales, pedometers, cadence sensors, activity monitors).

PAN A personal area network is typically defined as a network with a short range of about 10 meters. As the name suggest, it is typically centered around a *person* and "moves" with her – for instance with a smartphone or laptop as the central hub. In the CDG three technologies are recommended:

- **USB** which is a cabled network with a range of 5 meters.
- **Bluetooth Classic** which is a wireless network with a range of about 10 meters (see below).
- **Bluetooth LE** which is a wireless network with a range of about 10 meters (see below).

LAN (Sensor) A local area network is typically covering a larger area than the PAN and in addition to that, it is normally fixed to a geographic location – for instance permanently installed in a building. CDG currently recommends only one LAN technology:

- **ZigBee** which is a low-power wireless network typically used for home-automation, and especially suited for long-time battery-operated sensors.

TAN A touch area network is like a PAN but with a significantly shorter range – less than 1 meter, and often in the order of a few cm. TAN technologies often (but not always) includes power transfer, so that the sensor device need not have its own power supply, but receives power from the master device during the communication. CDG currently recommends only one TAN technology:

- **NFC** which is a group of standards from the RFID family of technologies. It supports wireless communication within a range of (typically) less than 10 cm along with wireless power transfer to the sensor.

Bluetooth Classic (BC) The original Bluetooth wireless cable-replacement standard, also known as Bluetooth Basic Rate / Extended Data Rate (BR/EDR). The technology behind BC is fundamentally different from BLE (see below) and a BC-only device will not be able to communicate with a BLE-only device. Even the communication protocols used for personal health devices are completely different. BC uses the HDP profile (also described below) along with the IEEE 11073 PHD standards[7] for PHD communication.

Bluetooth LE (BLE) The new Bluetooth wireless cable-replacement standard for low power (and low data throughput) communication (also branded as "Bluetooth Smart"). The BLE technology is fundamentally different from BC (see above) and a BC-only device will not be able to communicate with a BLE-only device. Even the communication protocols used for personal health devices are completely different. BLE uses the GATT profile (see below) along with different specific profiles and transcoding[8] for PHD communication.

Health Device Profile (HDP) The Bluetooth (classic) HDP[6] defines how a BC-based personal health device can be connected to a host and deliver data using the IEEE 11073 PHD[7] protocol.

Generic Attribute Profile (GATT) The parent of all Bluetooth LE profiles. BLE sensors publish one or more *services*, each exposing a number of *characteristics*, containing *attributes* that may be queried using operations from the Generic Attribute Profile.

Personal Healthcare Devices Class (USB-PHDC) The USB PHDC[9] defines how a USB-based personal health device can be connected to a host and deliver data using the IEEE 11073 PHD[7] protocol.

ZigBee Health Care Profile (ZHC) The ZigBee ZHC[10] defines how a ZigBee-based personal health device can be connected to a host and deliver data using the IEEE 11073 PHD[7] protocol.

Personal Health Device Communication (NFC-PHDC) The NFC PHDC[11] specification defines how an NFC-based personal health device can be connected to a host and deliver data using the IEEE 11073 PHD[7] protocol.

AHD The Application hosting Device, also known as the *Personal Health Gateway*, is the point of data collection near the citizen. Typically a smartphone, pc, tablet, or a set-top box. The role of the AHD is to collect the data from the PHDs and forward these data to the WAN device (see below) using the PCD-01 data format (see below).

WAN (device / interface) The WAN (wide area network) device, also known as the *Health and Fitness Service*, is the central point of data collection hosted by the service provider. This is where the AHD will deliver the collected data. The WAN interface (between the AHD and WAN devices) uses the PCD-01 data format (see below).

PCD-01 The data format for WAN interface communication (between the AHD and WAN devices) is defined by [12]. The CDG recommends using either SOAP or REST as the exchange protocol with PCD-01 as the message content.

3.3 Background and Basics

3.3.1 Run-time Environment Requirements

3.3.2 Language

3.3.3 Module Contract

3.3.4 External Dependencies

3.4 The 4SDC Architecture

3.4.1 System Layer and Module Interactions

3.5 Architecture Status and Road-maps

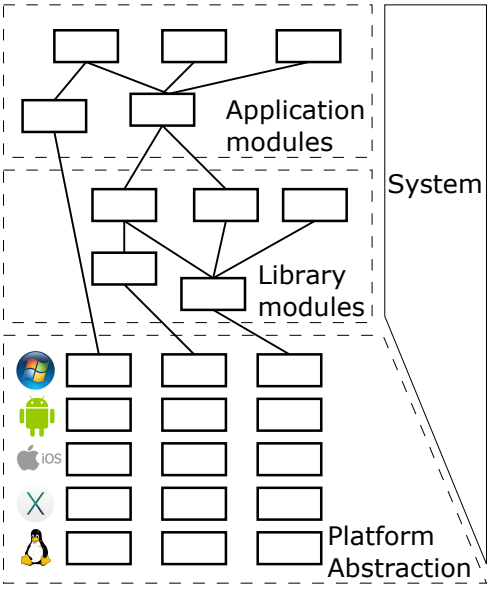


Figure 3.3 Conceptual blabla.

OSI Layer	
7 Application	OpenTele etc.
6 Presentation	User instructions and UI presentation, translate DIM to JSON etc.
5 Session	IEEE 11073 PHD, Bluetooth LE transcoding, proprietary interfaces
4 Transport	Bluetooth (LE) TCP/IP ZigBee RS 232
3 Network	
2 Data link	
1 Physical	

Figure 3.4 Conceptual blabla.

Chapter 4

License

The 4S Device Communication Module Collection is released under the Apache 2.0 license.

A copy of the full license text can be found below and at www.apache.org/licenses/LICENSE-2.0.

Notice, however, that the *demo application* depends on Qt libraries, **hence the demo application will inherit the Qt license terms**. Qt has a dual license – you may choose between the "copy-left" LGPL license or a commercial license, which you will have to buy from The Qt Company. More information here: www.qt.io/FAQ/.

editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and

Chapter 5

Contributors

The following statement lists the messages from 4S Device Communication Module Collection contributors (the `NOTICE` file mentioned in the license).

```
4S Device Communication
Copyright 2014-2015 The 4S Foundation (www.4s-online.dk)
```

Chapter 6

Todo List

Class fsdc::Core::DataAvailable

Fixme: Document me

Class fsdc::Core::RegisterDataType

Fixme: Document me

Class fsdc::Core::UnRegisterDataType

Fixme: Document me

Class FSYS::MsgAddrGenerator< T >

Make a base class of this to handle the MsgAddr

Class PAL::BluetoothModule

Revisit documentation for instructions on starting and terminating the module using the module launcher

Member PAL::BluetoothModule::BluetoothModule ()

Revisit documentation for instructions on starting and terminating the module using the module launcher

Member PAL::PersonalHealthDeviceConnector::disconnectIndication (std::shared_ptr< VirtualPHD > device, errortype error=0) noexcept=0

Fix doc when the actual errortype has been chosen...

Class PAL::PHDBluetoothDevice

Describe how the HDP standard maps to this interface

Class PAL::PHDUSBConnector

This specialization must be defined when the first USB device is integrated in this framework.

Class PAL::PHDUSBDevice

This specialization must be defined when the first USB device is integrated in this framework.

Class PAL::PHDZigBeeConnector

This specialization must be defined when the first ZigBee device is integrated in this framework.

Class PAL::PHDZigBeeDevice

This specialization must be defined when the first ZigBee device is integrated in this framework.

Chapter 7

Module Index

7.1 Modules

Here is a list of all modules:

Chapter 8

Hierarchical Index

8.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 9

Class Index

9.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 10

File Index

10.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 11

Module Documentation

11.1 Presentation Layer

11.2 Session Layer

11.3 Platform Abstraction Layer

11.4 System Layer

Chapter 12

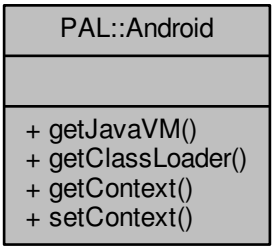
Class Documentation

12.1 PAL::Android Class Reference

Handles to the Java VM running our Android service.

```
#include <android.h>
```

Collaboration diagram for PAL::Android:



Static Public Member Functions

- static JavaVM * getJavaVM ()

  Get a pointer to the Java VM instance used to connect to Android.
- static jobject getClassLoader ()

  Get a global reference to the Java ClassLoader.
- static jobject getContext ()

  Get a global reference to the Android Context.
- static void setContext (jobject contextRef)

  Provide a reference to the Android Context.

12.1.1 Detailed Description

This class bridges the C++ and Java worlds of the Android service by providing the handles into the Android/Java world needed by the C++ components of the PAL layer.

12.1.2 Member Function Documentation

12.1.2.1 static jobject PAL::Android::getClassLoader ( ) [static]

Acquire a global reference to a Java ClassLoader that will be capable of loading PAL-layer classes (the reference will be valid until the application terminates).

Returns

The java.lang.ClassLoader global reference.

12.1.2.2 static jobject PAL::Android::getContext ( ) [static]

Acquire a global reference to the Android Context (the reference will be valid until the application terminates).

See also

setContext()

Returns

The android.content.Context global reference.

12.1.2.3 static JavaVM* PAL::Android::getJavaVM ( ) [static]

This method will return a pointer to the Java VM instance running the Android service.

Returns

A pointer to the Java VM.

12.1.2.4 static void PAL::Android::setContext (jobject *contextRef*) [static]

Provide a reference to the Android Context object which provides the access rights to the system, such as Bluetooth or USB access. The android.content.Context object reference provided may be of any scope, it will be copied into a global reference, which can be accessed later using getContext().

This function must be called exactly once during initialization of the system *before* the PAL layer modules are initialized, as the modules in the PAL layer may depend on this reference being available during initialization.

See also

getContext()

Parameters

contextRef	A reference of any scope to an android.content.Context object.

The documentation for this class was generated from the following file:

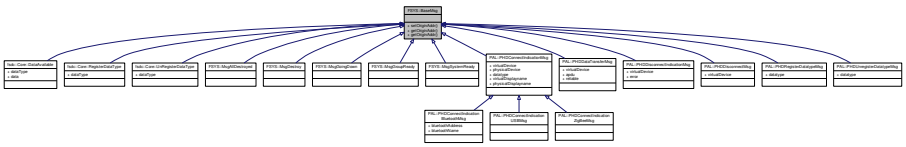
- Interface/PAL/android.h (This file was last changed: 2015-03-02 13:18:32 +0100, by Jacob Andersen)

12.2 FSYS::BaseMsg Class Reference

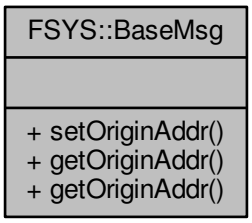
The BaseMsg class is the base class for all messages.

```
#include <basemsg.h>
```

Inheritance diagram for FSYS::BaseMsg:



Collaboration diagram for FSYS::BaseMsg:



Public Member Functions

- void setOriginAddr (MsgAddr &addr)

  Set the origin address of the message.
- MsgAddr & getOriginAddr (void)

  Get/return the origin address of the message.
- const MsgAddr & **getOriginAddr** (void) const

Friends

- class **MsgQueue**

12.2.1 Detailed Description

All messages sent with the message system must inherit from this class, it contains information to the receiver of who sent the message, so replies can be directed back to the sender.

12.2.2 Member Function Documentation

12.2.2.1 MsgAddr& FSYS::BaseMsg::getOriginAddr (void)

The Origin address of a message, is the address of the instance that sent the message orginally.

Returns

The origin address of this message

12.2.2.2 void FSYS::BaseMsg::setOriginAddr (MsgAddr & *addr*)

The Origin address of a message, is the address of the instance that sent the message orginally.

Parameters

addr	The origin address of this object

The documentation for this class was generated from the following file:

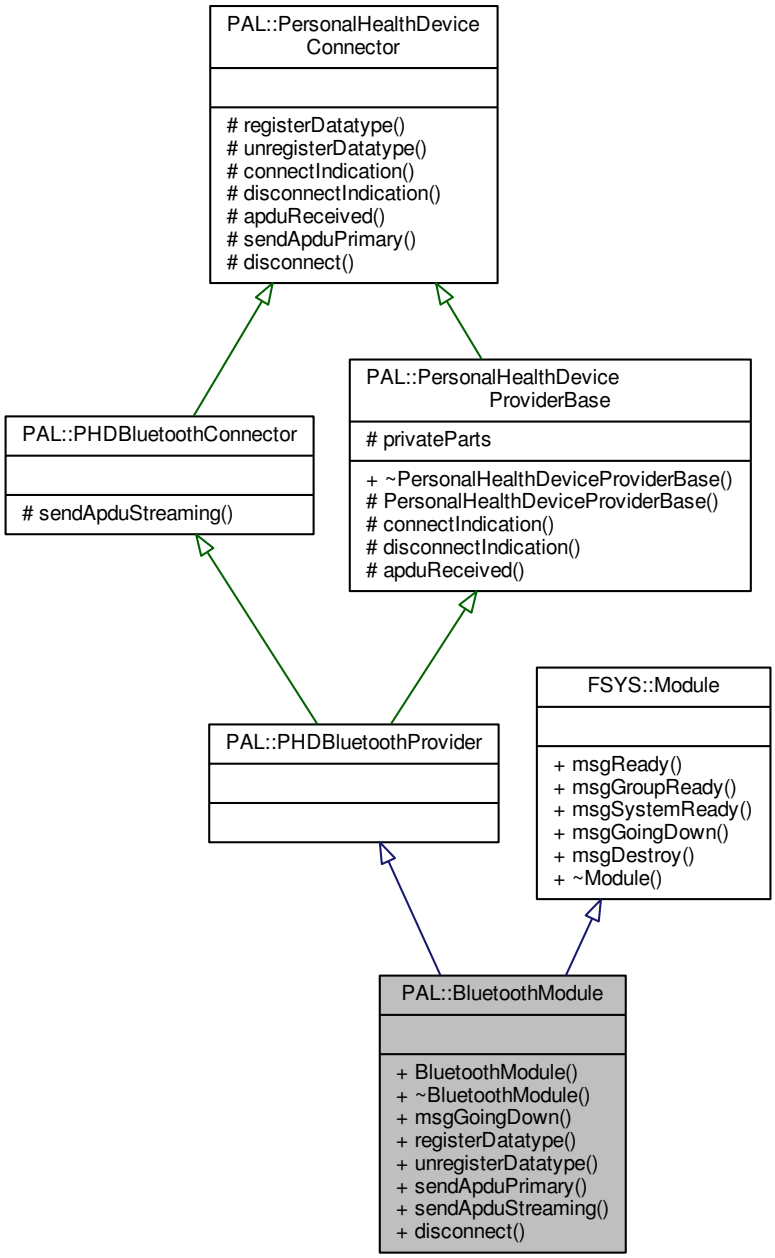
- Interface/FSYS/basemsg.h (This file was last changed: 2015-02-16 13:46:05 +0100, by Jacob Andersen)

12.3 PAL::BluetoothModule Class Reference

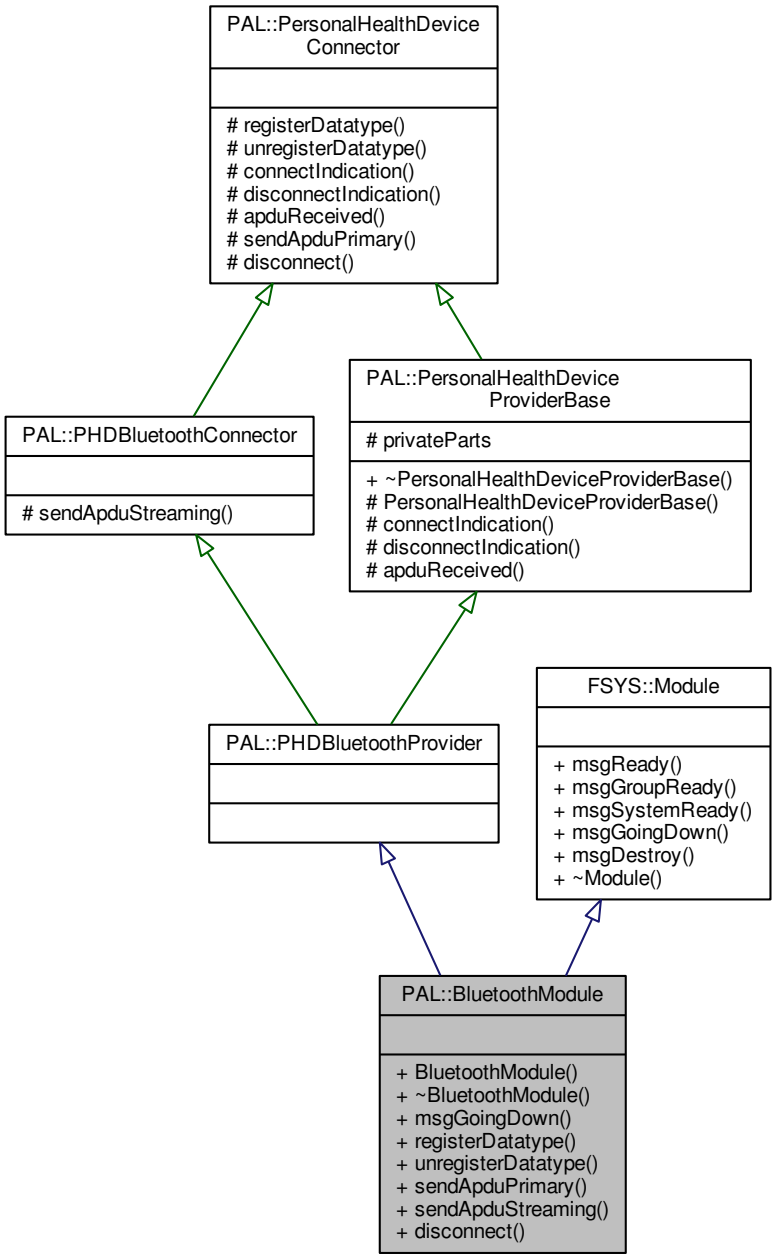
Bluetooth PAL-layer module wrapper class.

```
#include <bluetooth.h>
```

Inheritance diagram for PAL::BluetoothModule:



Collaboration diagram for PAL::BluetoothModule:



Public Member Functions

- BluetoothModule ()

  Launch and initialize the Bluetooth module.
- ~BluetoothModule ()

  Free resources used by the Bluetooth module.
- void msgGoingDown ()

  Will stop the module and (relatively gracefully) disconnect any open connections.

- void registerDatatype (uint16_t datatype) noexcept

  A session-layer component has been registered as handler of a datatype.
- void unregisterDatatype (uint16_t datatype) noexcept

  Session-layer components are no longer registered to handle this datatype.
- void sendApduPrimary (std::shared_ptr< PAL::VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept

  Send a message to this device on the primary channel.
- void sendApduStreaming (std::shared_ptr< PAL::VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept

  Send a message to this device on a streaming channel.
- void disconnect (std::shared_ptr< PAL::VirtualPHD > device) noexcept

  Session-layer component request to disconnect a device.

Friends

- class **PrivateBluetoothModule**

Additional Inherited Members

12.3.1 Detailed Description

This class wraps a module providing connections to classic Bluetooth (Health Device Profile) devices. In the future, support for BLE devices and device management operations should be be added to this class as well.

The class derives from the PAL::PHDBluetoothProvider which handles the HDP communication with other modules.

Todo Revisit documentation for instructions on starting and terminating the module using the module launcher

See also

PAL::PHDBluetoothProvider

12.3.2 Constructor & Destructor Documentation

12.3.2.1 PAL::BluetoothModule::BluetoothModule ( )

The Bluetooth module is initialized.

Todo Revisit documentation for instructions on starting and terminating the module using the module launcher

12.3.3 Member Function Documentation

12.3.3.1 void PAL::BluetoothModule::disconnect (std::shared_ptr< PAL::VirtualPHD > *device*) [noexcept]

The session-layer component handling this device wishes to disconnect. When the device has been disconnected, we must reply with a disconnectIndication().

Parameters

device	The device we must disconnect.

12.3.3.2 void PAL::BluetoothModule::msgGoingDown ( ) [virtual]

This method stops the Bluetooth module. If any connections are open, they will be disconnected with "gentle force". This signal should be delivered to the module just before it is destructed. When the module receives this signal, it will stop accepting new connections and close all existing - in other words: it will be a brick waiting for destruction.

Parameters

signal	The Terminate signal.

Reimplemented from FSYS::Module.

12.3.3.3 void PAL::BluetoothModule::registerDatatype (uint16_t *datatype*) [virtual], [noexcept]

This method is called when a session-layer component is ready to handle the given datatype. We should start accepting incoming connections of this type.

Parameters

datatype	The datatype that will be handled.

Implements PAL::PersonalHealthDeviceConnector.

12.3.3.4 void PAL::BluetoothModule::sendApduPrimary (std::shared_ptr< PAL::VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*) [noexcept]

Send a message to the given device on the reliable primary channel. If no channel is currently open to this device, ignore the call; if the transmission fails (e.g. because the destination went out of range), follow up with a disconnectIndication() with a suitable error code.

Parameters

device	The destination
apdu	The message payload

12.3.3.5 void PAL::BluetoothModule::sendApduStreaming (std::shared_ptr< PAL::VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*) [noexcept]

Attempt to send this message to the given device on an unreliable streaming channel. If no streaming channel is currently open, use the primary channel instead (

See also

sendApduPrimary()).

Parameters

device	The destination
apdu	The message payload

12.3.3.6 void PAL::BluetoothModule::unregisterDatatype (uint16_t *datatype*) [virtual], [noexcept]

This method is called when no more session-layer components are ready to handle the given datatype. We should no longer accept incoming connections of this type.

Parameters

datatype	The datatype that will no longer be accepted.

Implements PAL::PersonalHealthDeviceConnector.

The documentation for this class was generated from the following file:

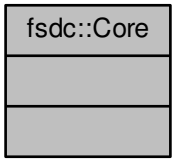
- Interface/PAL/bluetooth.h (This file was last changed: 2015-03-13 09:35:26 +0100, by Jacob Andersen)

12.4 fsdc::Core Class Reference

4SDC Core class

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

Collaboration diagram for fsdc::Core:



Classes

- class DataAvailable
- class RegisterDataType
- class UnRegisterDataType

12.4.1 Detailed Description

The core class of 4SDC, contains the interfaces that should eventually exsist in the presentation layer

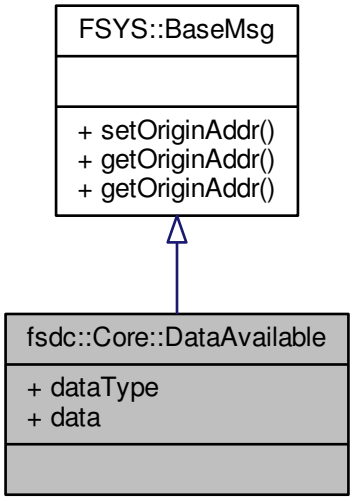
The documentation for this class was generated from the following file:

- Interface/4SDC/core.h (This file was last changed: 2015-03-10 16:58:11 +0100, by Mike Kristoffersen)

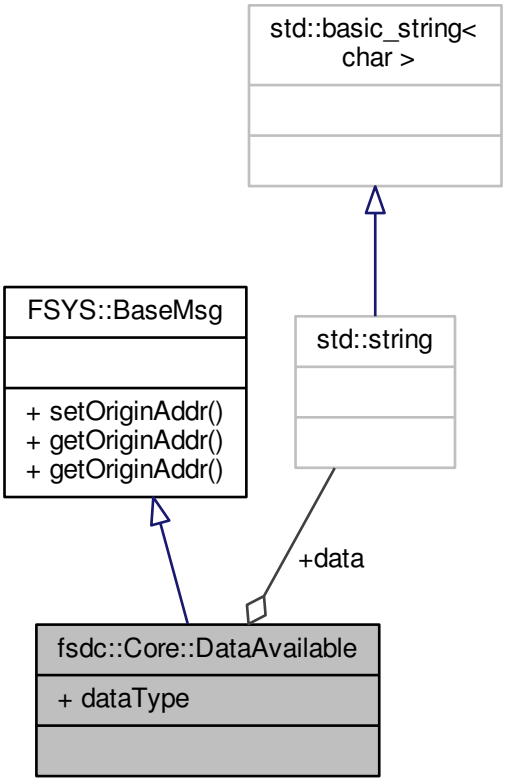
12.5 fsdc::Core::DataAvailable Class Reference

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

Inheritance diagram for fsdc::Core::DataAvailable:



Collaboration diagram for fsdc::Core::DataAvailable:



Public Attributes

- uint16_t **dataType**
- std::string **data**

Additional Inherited Members

12.5.1 Detailed Description

Todo Fixme: Document me

The documentation for this class was generated from the following file:

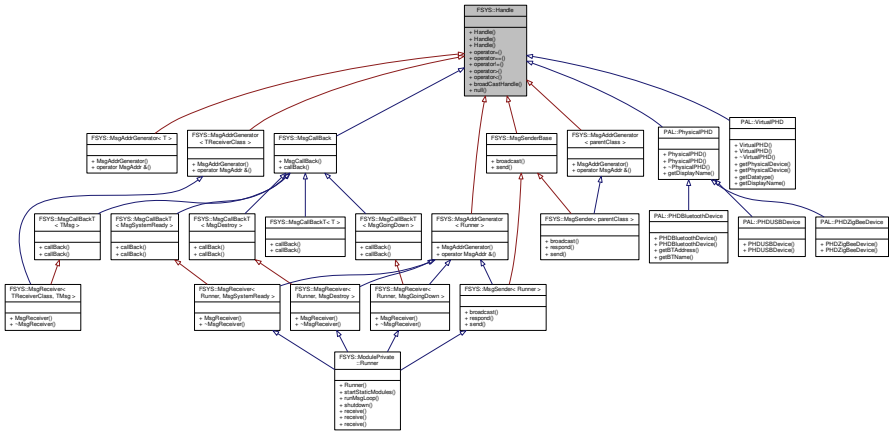
- Interface/4SDC/core.h (This file was last changed: 2015-03-10 16:58:11 +0100, by Mike Kristoffersen)

12.6 FSYS::Handle Class Reference

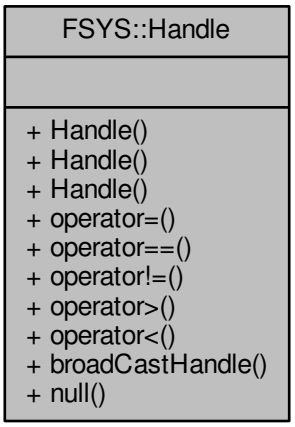
Class that holds and assigns handles.

```
#include <handle.h>
```

Inheritance diagram for FSYS::Handle:



Collaboration diagram for FSYS::Handle:



Public Member Functions

- Handle (void)

  Default constructor.
- Handle (Handle &ref)

  Copy constructor for handle object.

- Handle (const Handle &ref)

  Copy constructor for handle object.
- Handle & **operator=** (Handle const &theOtherOne)
- bool operator== (Handle const &compareTo) const

  Equality comparator.
- bool operator!= (Handle const &compareTo) const

  Not equalify operator.
- bool operator> (Handle const &compareTo) const

  Greater than operator ('>')
- bool operator< (Handle const &compareTo) const

  Less than operator ('<')

Static Public Member Functions

- static Handle & broadCastHandle (void)

  Generic broadcast handle.
- static Handle & null (void)

  The null handle.

12.6.1 Detailed Description

If you instantiate this class with the default constructor then it will represent a unique handle, if you copy the object, the copy will contain the same handle.

It is possible to compare handles with the '!=', '==', '<' and '>' operators, you should not rely on any special meaning of the '<' and '>' operators, as future implemenations might change the order of created handles.

Two special handels are exposed by the class - the "null" and "broadcast" handles - you can not rely on how these two handles are ordered compared to other handles (e.g. null() will not always compare smaller to other handles, nor will null() compare equal to 0.

12.6.2 Constructor & Destructor Documentation

12.6.2.1 FSYS::Handle::Handle (void) [inline]

Using the default constructor will assign a unique handle to this object.

If you do not wish to have new handle assigned, then you can get a more efficient program by initializing the handle to the null handle. This will prevent a global mutex lock to be taken.

```
FSYS::Handle myHandle(FSYS::null());
.
.
.
myHandle = arg.handle;
```

is more efficient than:

```
FSYS::Handle myHandle;
.
.
.
myHandle = arg.handle;
```

The most efficient is thou:

```
FSYS::Handle myHandle(arg.handle)
```

12.6.2.2 FSYS::Handle::Handle (Handle & *ref*) [inline]

Parameters

ref	The handle that should be copied

12.6.2.3 FSYS::Handle::Handle (const Handle & *ref*) [inline]

Parameters

ref	The handle that should be copied

12.6.3 Member Function Documentation

12.6.3.1 static Handle& FSYS::Handle::broadCastHandle (void) [static]

The meaning of this handle is specified by its use, from the handle class's point of view, it is just a unique handle.

Returns

The broadcast handle

12.6.3.2 static Handle& FSYS::Handle::null (void) [static]

The null handle is the handle that doesn't represent a valid handle.

Two null handles will compare equal to each other, and not equal to other handles.

The null handle does not compare equal to the integer value 0, you should consider it equally likely that it compares smaller or greater to any other specific handle, and because it compares smaller to one handle, it doesn't mean that it compares smaller to evey other handle.

The actuall value of the handle, will be dependend on a number of factors both compile time and excecution time wise.

Returns

The null handle

12.6.3.3 bool FSYS::Handle::operator!= (Handle const & *compareTo*) const [inline]

Operator will return true if the two handles are different, e.g. not copies of each other.

Parameters

comparTo	The handle this handle should be compared to

Returns

12.6.3.4 bool FSYS::Handle::operator< (Handle const & *compareTo*) const [inline]

See also

operator>

If the greater than operator ('>') returns true for a relationship between A and B, then this function will return true when the arguments are swapped. So if A > B evaluates to true, then B < A will also evaluate to true.

Parameters

compareTo	The handle that this handle should be compared to

Returns

true or false as described in the description of this function

12.6.3.5 bool FSYS::Handle::operator== (Handle const & *compareTo*) const [inline]

Operator will return true if the two handles are copies of each other

Parameters

compareTo	The handle this handle should be compared to

Returns

True if the two handles are copies, False if not.

12.6.3.6 bool FSYS::Handle::operator> (Handle const & *compareTo*) const [inline]

While handles don't have any meaning relation ship between them, except equality or not equality it is still helpfull for certian algorithms / data structures to be able to order them.

You should not rely on any connection between the creation order of the handles and how they compare to each other.

The only thing you can rely on is that if we you have three handles A, B and C - and A > B evaluates to true and B > C evaluates to true then A > C will also evaluate to true and in that case B > A, C > B and C > A will also all evaluate to false.

Parameters

compareTo	The handle that should be compared to this handle

Returns

true or false as descriped in the description of this function

The documentation for this class was generated from the following file:

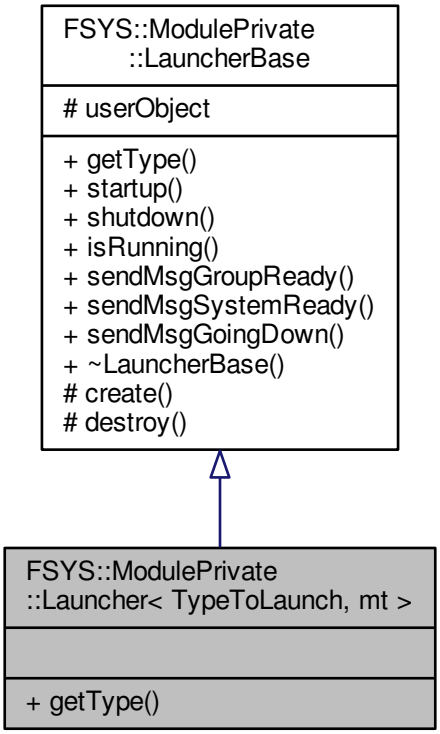
- Interface/FSYS/handle.h (This file was last changed: 2015-02-16 10:35:24 +0100, by Jacob Andersen)

12.7 FSYS::ModulePrivate::Launcher< TypeToLaunch, mt > Class Template Reference

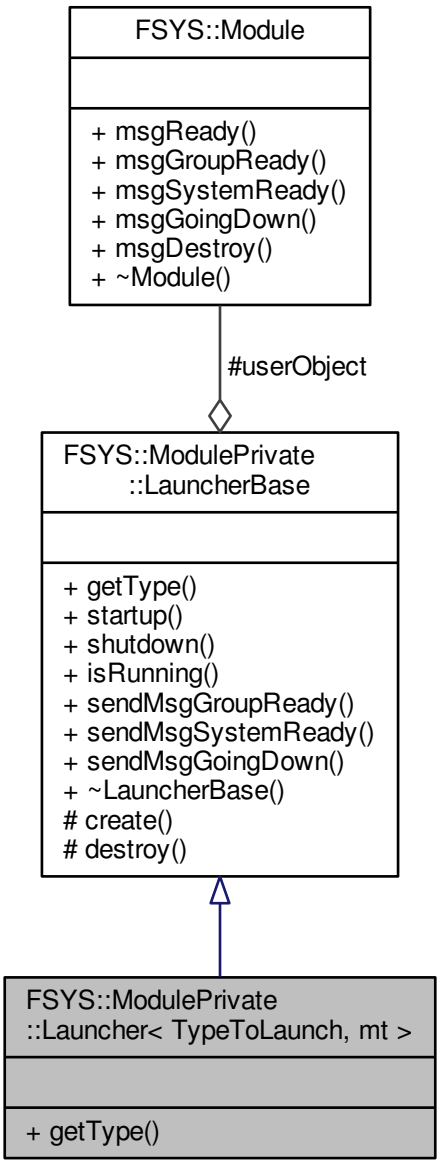
The Launcher class is used for dynamically launching the user clases.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >:



Collaboration diagram for FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >:



Public Member Functions

- ModuleType getType (void) override

  Function to retrieve info about the user object containd.

Additional Inherited Members

12.7.1 Detailed Description

template<class TypeToLaunch, LauncherBase::ModuleType mt = LauncherBase::ModuleType::STATIC>
class FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >

12.7.2 Member Function Documentation

12.7.2.1 template<class TypeToLaunch , LauncherBase::ModuleType mt = LauncherBase::ModuleType::STATIC> ModuleType FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >::getType (void) [inline], [override], [virtual]

This funciton can be used to find the type module the user module contained in this launcher has

Returns

The ModuleType of the contained object

Implements FSYS::ModulePrivate::LauncherBase.

The documentation for this class was generated from the following file:

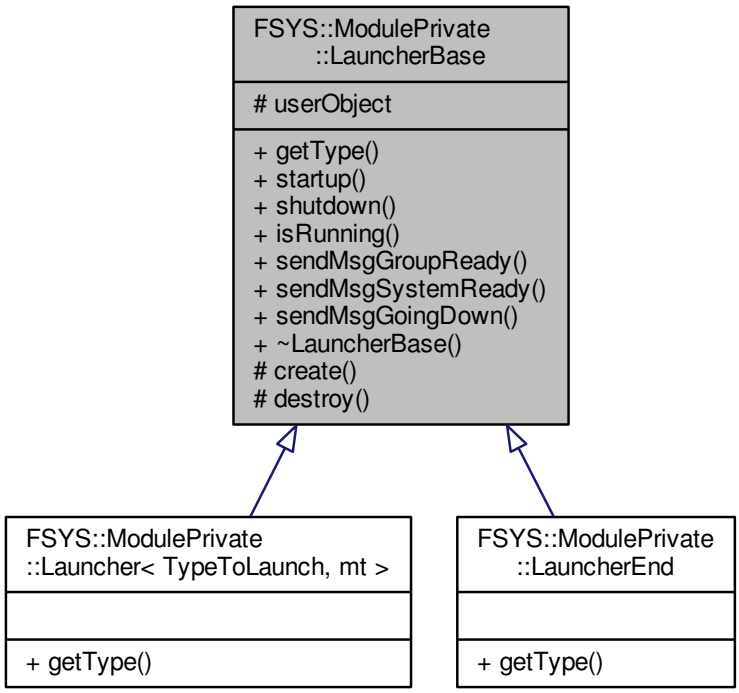
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.8 FSYS::ModulePrivate::LauncherBase Class Reference

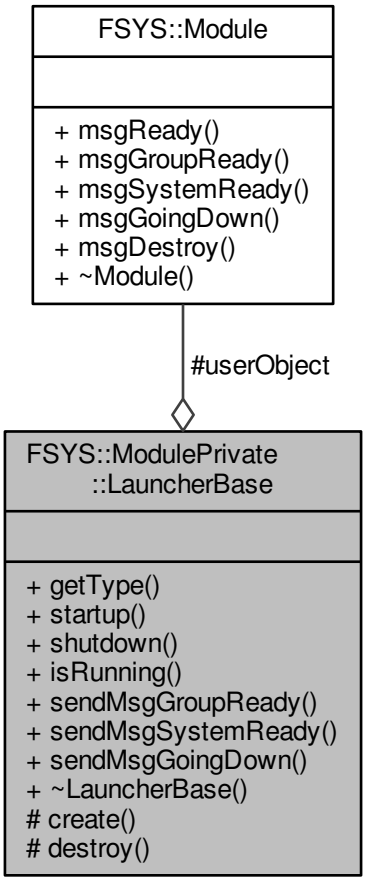
The LauncherBase class is the non-template class for other Launchers.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::ModulePrivate::LauncherBase:



Collaboration diagram for FSYS::ModulePrivate::LauncherBase:



Public Types

- enum **ModuleType** {
  UNKNOWN, **STATIC**, **DELAYED**, **DYNAMIC**,
  DYNAMIC_THREAD }

Public Member Functions

- virtual ModuleType getType (void)=0

  Function to retrieve info about the user object containd.
- void startup (void)

  Function that starts the user class.
- void shutdown (void)

  Function that shuts the user class down.
- bool isRunning (void)

  Boolean function telling if the module is currently running.

- void sendMsgGroupReady (void)

  Calls the msgGroupReady function on the user class.
- void sendMsgSystemReady (void)

  Calls the msgSystemReady function on the user class.
- void sendMsgGoingDown (void)

  Calls the msgGoingDown function on the user class.
- virtual ∼LauncherBase (void)

  LauncherBase destructor.

Protected Member Functions

- virtual void create (void)=0

  Virtual function, called when the user object should be created.
- virtual void destroy (void)=0

  Virtual function, called when the user object should be destroyed.

Protected Attributes

- Module ∗ **userObject** {nullptr}

12.8.1 Detailed Description

The Launcher class is responsible for the user class (a user class is a user written class that is not part of the FSYS framework).

The Launcher class is a member of the Macro class (the class that is generated by Macros in the module group macro definitions).

12.8.2 Constructor & Destructor Documentation

12.8.2.1 virtual FSYS::ModulePrivate::LauncherBase::∼LauncherBase (void) [inline],[virtual]

Ensures the LauncherBase has a virtual destructor

12.8.3 Member Function Documentation

12.8.3.1 virtual void FSYS::ModulePrivate::LauncherBase::create (void) [protected],[pure virtual]

The specialisations to the LaunchBase class needs to implement this function, so this class can call this function to have the object (the user class) instantiated.

12.8.3.2 virtual void FSYS::ModulePrivate::LauncherBase::destroy (void) [protected],[pure virtual]

The specialisations to the LauncherBase class needs to implement this function, so the LauncherBase class can call this function to have the (the user class) object destroyed again.

12.8.3.3 virtual ModuleType FSYS::ModulePrivate::LauncherBase::getType (void) [pure virtual]

This funciton can be used to find the type module the user module contained in this launcher has

Returns

The ModuleType of the contained object

Implemented in FSYS::ModulePrivate::LauncherEnd, and FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >.

12.8.3.4 bool FSYS::ModulePrivate::LauncherBase::isRunning (void) [inline]

Returns

true if the module is up and running, false if not

12.8.3.5 void FSYS::ModulePrivate::LauncherBase::shutdown (void)

This function informs the class that it is being shut down, and shuts it down.

12.8.3.6 void FSYS::ModulePrivate::LauncherBase::startup (void)

This function starts the class, and ensures that the right callbacks are made to it.

The documentation for this class was generated from the following file:

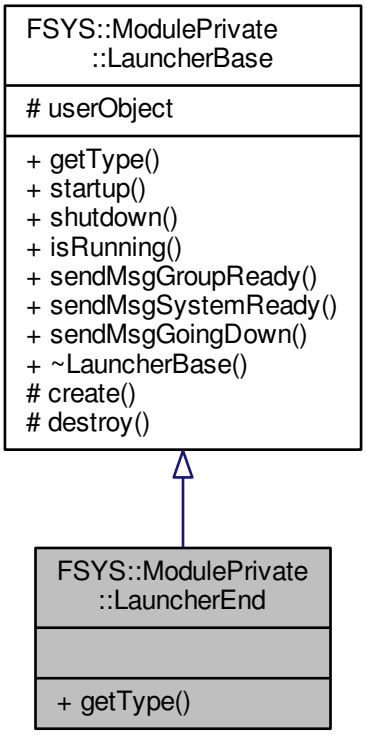
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.9 FSYS::ModulePrivate::LauncherEnd Class Reference

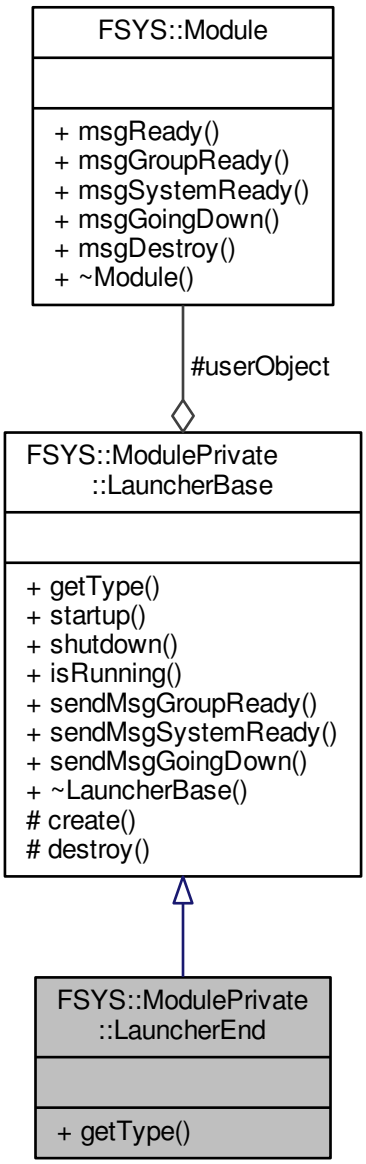
Dummy launcher class to mark the end of the launcher list.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::ModulePrivate::LauncherEnd:



Collaboration diagram for FSYS::ModulePrivate::LauncherEnd:



Public Member Functions

- ModuleType getType (void) override

  Function to retrieve info about the user object containd.

Additional Inherited Members

12.9.1 Detailed Description

12.9.2 Member Function Documentation

12.9.2.1 ModuleType FSYS::ModulePrivate::LauncherEnd::getType (void) [inline],[override],[virtual]

This funciton can be used to find the type module the user module contained in this launcher has

Returns

The ModuleType of the contained object

Implements FSYS::ModulePrivate::LauncherBase.

The documentation for this class was generated from the following file:

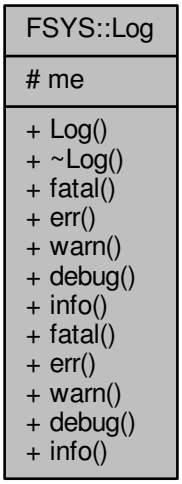
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.10 FSYS::Log Class Reference

Log class used for logging information about program execution.

```
#include <log.h>
```

Collaboration diagram for FSYS::Log:



Public Member Functions

- Log (std::string const &module)

  The Log constructor.
- void **fatal** (std::string const &fatal, std::string const &file, int line)
- void **err** (std::string const &err, std::string const &file, int line)
- void **warn** (std::string const &warn, std::string const &file, int line)
- void **debug** (std::string const &debug, std::string const &file, int line)
- void **info** (std::string const &info, std::string const &file, int line)

Static Public Member Functions

- static void fatal (std::string const &fatal)

  Static function to log messages about fatal incidents in the code.
- static void err (std::string const &err)

  Static function to log messages about error conditions in the code.
- static void warn (std::string const &warn)

  Static function to log warning messages from the code.
- static void debug (std::string const &debug)

  Static function to log debug messages.
- static void info (std::string const &info)

  Static function to log info messages.

Protected Attributes

- void * me

  The me member is available for use for actuall implementations.

12.10.1 Detailed Description

You don't need to instantiate a log object to use the system, you can if you want to, but it is recommended that you use the macro system instead.

To just get a log line out:

```
#include "FSYS/log.h"

...
FSYS::Log::info("This is an info message");
...
```

The above will output something like:

Info: "This is an info message"

How ever, it is recomended to use the macro interface:

```
#include "FSYS/log.h"

DECLARE_LOG_MODULE("MyModuleName");

...

INFO("This is an info message");

...
```

As this will also output the modulename, filename, and line number:

Info: (MyModuleName)"This is an info message"@filepath(line number)

Other macros are:

- FATAL(msg) - For fatal, non-recoverable errors (leads to program crash/closure)
- ERR(msg) - For error messages, that are handled in one way or the other
- WARN(msg) - For warnings/unexpected behavior
- DEBUG(msg) - For debug messages, used during development
- INFO(msg) - For info/convinient messages usefull for knowing the state of the program

Note

It is not possible to have multiple calls to DECLARE_LOG_MODULE in the same source file.

12.10.2 Constructor & Destructor Documentation

12.10.2.1 FSYS::Log::Log (std::string const & *module*)

The Log constructor takes a module name in, the form of a string, as argument, this string will be appended to all of the non-static log strings in the interface.

The same module name can be used across multiple source files, and it is possible to have multiple log objects/module names in the same source file (as long as you don't use the macros).

You don't need to instantiate the class if you only use the static functions in it.

Parameters

module	The

12.10.3 Member Function Documentation

12.10.3.1 static void FSYS::Log::debug (std::string const & *debug*) [static]

```
#include "PAL/Interface/log.h"

...
FSYS::Log::debug("This is a debug message");
...
```

Parameters

debug	String reference to the debug message

12.10.3.2 static void FSYS::Log::err (std::string const & *err*) [static]

```
#include "PAL/Interface/log.h"

...
FSYS::Log::err("This is a message about an error");
...
```

Parameters

err	String reference to the error message that should be logged

12.10.3.3 static void FSYS::Log::fatal (std::string const & *fatal*) [static]

```
#include "PAL/Interface/log.h"

...
FSYS::Log::fatal("This is a message about a fatal event");
...
```

Parameters

fatal	String reference to the message that should be logged as fatal

12.10.3.4 static void FSYS::Log::info (std::string const & *info*) [static]

```
#include "PAL/Interface/log.h"

...
FSYS::Log::debug("This is an info message");
...
```

Parameters

debug	String reference to the info message

12.10.3.5 static void FSYS::Log::warn (std::string const & *warn*) [static]

```
#include "PAL/Interface/log.h"

...
FSYS::Log::warn("This is a warning message");
...
```

Parameters

warn	String reference to the message that should be logged as a warning

12.10.4 Member Data Documentation

12.10.4.1 void* FSYS::Log::me [protected]

The me member is not part of the public interface of the Log class, it has been added in case any of the platform specific implementations of the Log class needs to store some kind of information in the object.

The documentation for this class was generated from the following file:

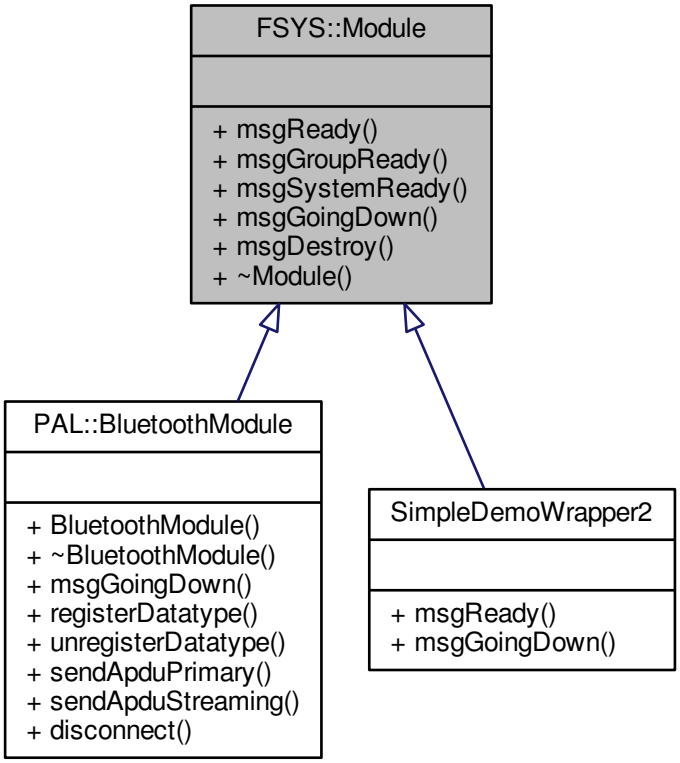
- Interface/FSYS/log.h (This file was last changed: 2015-03-03 02:30:31 +0100, by Mike Kristoffersen)

12.11 FSYS::Module Class Reference

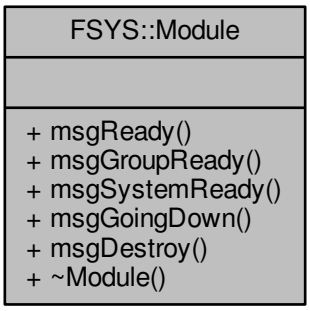
Base class for all modules.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::Module:



Collaboration diagram for FSYS::Module:



Public Member Functions

- virtual void **msgReady** (void)
- virtual void **msgGroupReady** (void)
- virtual void **msgSystemReady** (void)
- virtual void **msgGoingDown** (void)
- virtual void **msgDestroy** (void)

12.11.1 Detailed Description

The Module class is the base class for all modules, except for the ModuleX classes, no other classes should ever need to inherit directly from this class.

The documentation for this class was generated from the following file:

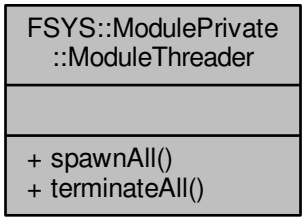
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.12 FSYS::ModulePrivate::ModuleThreader Class Reference

The ModuleThreader class.

```
#include <modulethreader.h>
```

Collaboration diagram for FSYS::ModulePrivate::ModuleThreader:



Public Member Functions

- void spawnAll (void)

  Function that spawns all threads and starts their message loops.
- void terminateAll (void)

  Function that terminates all threads.

12.12.1 Detailed Description

The module threader class is the class that starts all the threads in the system.

The functions in this class must only be called from the main thread

12.12.2 Member Function Documentation

12.12.2.1 void FSYS::ModulePrivate::ModuleThreader::spawnAll (void)

Call this function to start the system.

Note

Calling this function will not start the message loop in the calling thread

12.12.2.2 void FSYS::ModulePrivate::ModuleThreader::terminateAll (void)

Call this function to stop all message loops and terminate the threads.

The function will not return until all threads are terminated.

The message loop of the main thread will not run while the other threads are being terminated.

Alternatively you can first broadcast FSYS::MsgGoingDown wait a set time while you let the main message loop continue, and then broadcast FSYS::MsgDestroy.

In any case when the other threads are terminated, a FSYS::MsgAllDestroyed message will be broadcasted

The documentation for this class was generated from the following file:

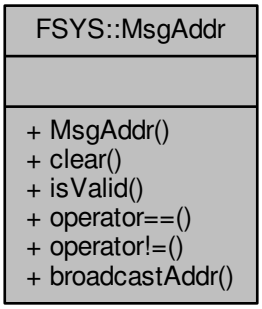
- Interface/FSYS/modulethreader.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.13 FSYS::MsgAddr Class Reference

The MsgAddr class contains a unique address in the message system.

```
#include <msgaddr.h>
```

Collaboration diagram for FSYS::MsgAddr:



Public Member Functions

- MsgAddr (void)

  Default constructor.
- void clear (void)

  Funciton that clears the address stored in the object.
- bool isValid (void)

  Function to determine if this address is valid (!=null)
- bool operator== (MsgAddr const &compareTo) const

  operator ==
- bool operator!= (MsgAddr const &compareTo) const

  operator !=

Static Public Member Functions

- static MsgAddr & broadcastAddr (void)

  Returns the broadcast address.

Friends

- template<class parentClass >
  class **MsgAddrGenerator**
- class **MsgSenderBase**
- class **MsgQueue**

12.13.1 Detailed Description

A MsgAddr identifies a specific instance of a specifc object to the message system.

12.13.2 Constructor & Destructor Documentation

12.13.2.1 FSYS::MsgAddr::MsgAddr (void)

The default constructor doesn't do anything except initializing the private members.

12.13.3 Member Function Documentation

12.13.3.1 static MsgAddr& FSYS::MsgAddr::broadcastAddr (void) [static]

The broadcast address is an address that will match all addresses, and hence can be used for sending to all addresses

Returns

The bradcast address

12.13.3.2 bool FSYS::MsgAddr::isValid (void)

This function verifies if the address is null or not, or if part of the address is null - it does not try to determine if the address is actually a valid address of an existing object.

Returns

true if the address is non-null
false if the addres or part it is null

12.13.3.3 bool FSYS::MsgAddr::operator!= (MsgAddr const & *compareTo*) const [inline]

Compares two addresses and return false if they are equal

Parameters

compareTo	the addres this object should be compared to

Returns

false if the addresses are equal, true if not

12.13.3.4 bool FSYS::MsgAddr::operator== (MsgAddr const & *compareTo*) const [inline]

Compares two addresses and return true if they are equal

Parameters

compareTo	the addres this object should be compared to

Returns

true if the addresses are equal, false if not

The documentation for this class was generated from the following file:

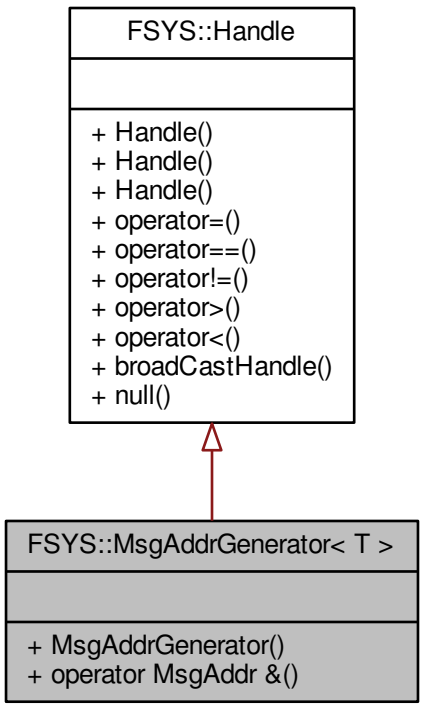
- Interface/FSYS/msgaddr.h (This file was last changed: 2015-02-13 11:45:28 +0100, by Mike Kristoffersen)

12.14 FSYS::MsgAddrGenerator< T > Class Template Reference

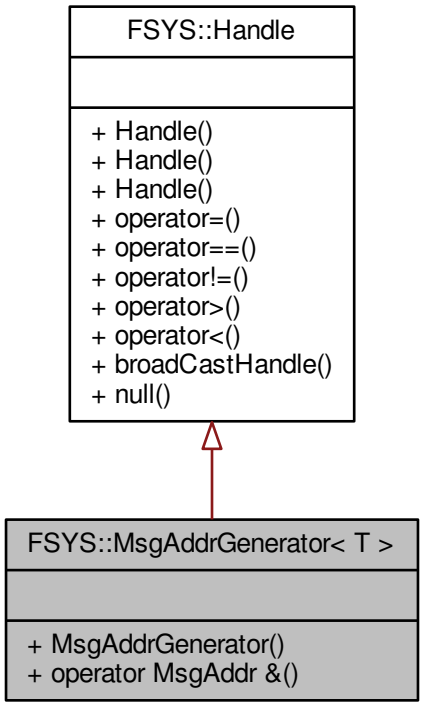
Constructor to MsgAddrGenerator object.

```
#include <msgaddrgenerator.h>
```

Inheritance diagram for FSYS::MsgAddrGenerator< T >:



Collaboration diagram for FSYS::MsgAddrGenerator< T >:



Public Member Functions

- MsgAddrGenerator (void)

  Constructor that creates an address.
- operator MsgAddr & ()

  MsgAddr operator for type conversion of MsgAddrGenerator to a MsgAddr.

12.14.1 Detailed Description

template<class T>
class FSYS::MsgAddrGenerator< T >

The MsgAddrGenerator object generates a unique address for an instance of a class.

Todo Make a base class of this to handle the MsgAddr

12.14.2 Member Function Documentation

12.14.2.1 template<class T> FSYS::MsgAddrGenerator< T >::operator MsgAddr & ( ) [inline]

This operator allows for conversion of a MsgAddrGenerator objects to a MsgAddr object MsgAddr operator for type conversion of MsgAddrGenerator to a MsgAddr

This operator allows for conversion of a MsgAddrGenerator objects to a MsgAddr object

The documentation for this class was generated from the following files:

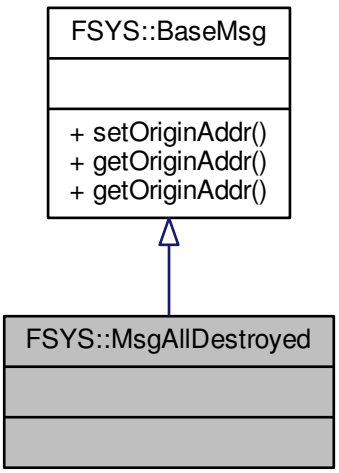
- Interface/FSYS/msgaddr.h (This file was last changed: 2015-02-13 11:45:28 +0100, by Mike Kristoffersen)
- Interface/FSYS/msgaddrgenerator.h (This file was last changed: 2015-02-12 14:59:20 +0100, by Mike Kristoffersen)

12.15 FSYS::MsgAllDestroyed Class Reference

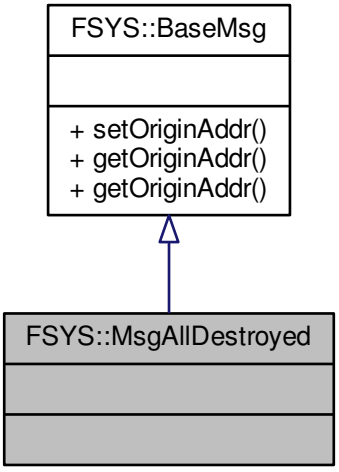
Last message broadcasted in system.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::MsgAllDestroyed:



Collaboration diagram for FSYS::MsgAllDestroyed:



Additional Inherited Members

12.15.1 Detailed Description

When this message is broadcasted, the main threads message loop will terminate.

The documentation for this class was generated from the following file:

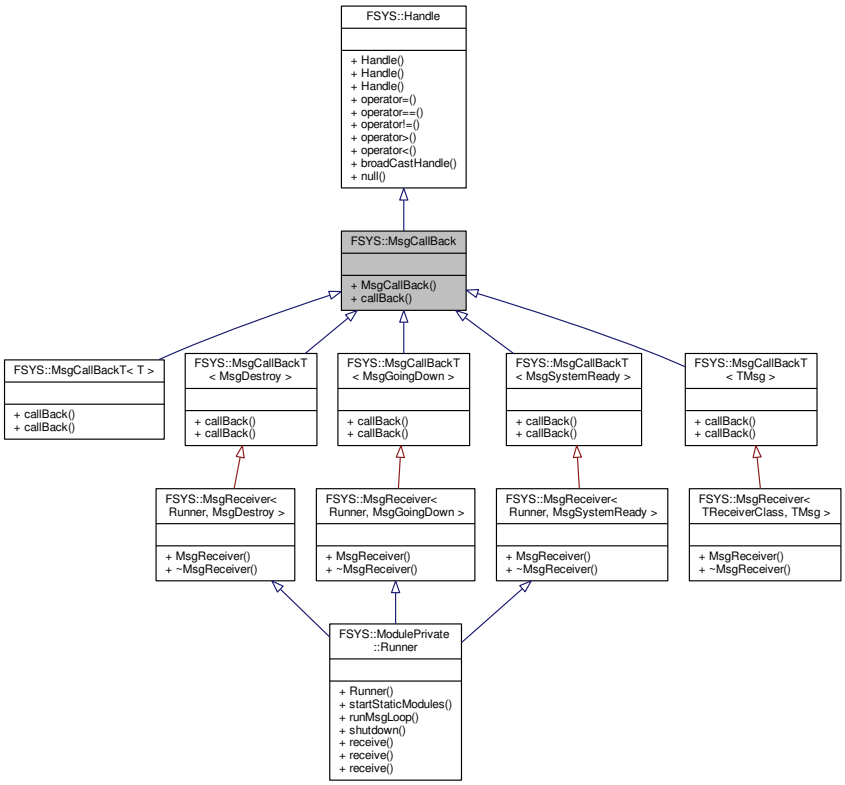
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.16 FSYS::MsgCallBack Class Reference

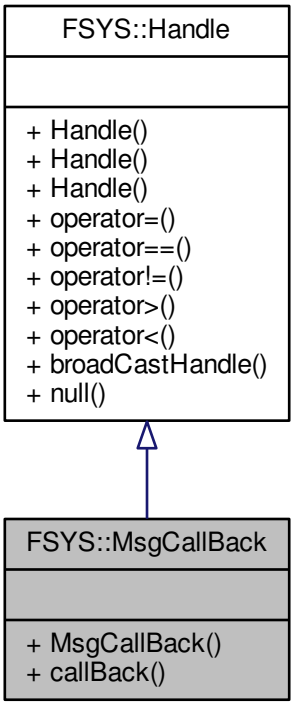
The MsgCallBack class is an internal class to the message system.

```
#include <msgcallback.h>
```

Inheritance diagram for FSYS::MsgCallBack:



Collaboration diagram for FSYS::MsgCallBack:



Public Member Functions

- MsgCallBack (void)

  Default constructor.
- virtual void callBack (std::shared_ptr< BaseMsg > &baseMsg)=0

  Call back function that evantually maps to receive(T)

Additional Inherited Members

12.16.1 Detailed Description

Warning

Do NOT use this class unless you are coding the message system it self. Use the MsgReceiver class instead

See also

MsgReceiver

The MsgCallBack class is the base class that gets registered as listening on a specific message type, it contains a single virtual function which is the one that is called from the queue when the message is taken of the queue.

12.16.2 Member Function Documentation

12.16.2.1 virtual void FSYS::MsgCallBack::callBack (std::shared_ptr< BaseMsg > & *baseMsg*) [pure virtual]

This function is called with a shared pointer to the message that is being processed.

Parameters

baseMsg	The message that should be forwarded

Implemented in FSYS::MsgCallBackT< T >, FSYS::MsgCallBackT< TMsg >, FSYS::MsgCallBackT< MsgDestroy >, FSYS::MsgCallBackT< MsgGoingDown >, and FSYS::MsgCallBackT< MsgSystemReady >.

The documentation for this class was generated from the following file:

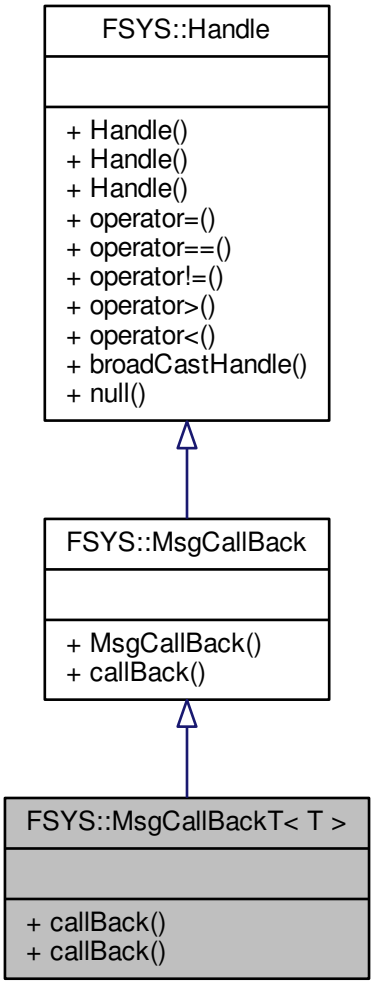
- Interface/FSYS/msgcallback.h (This file was last changed: 2015-02-11 00:49:49 +0100, by Mike Kristoffersen)

12.17 FSYS::MsgCallBackT< T > Class Template Reference

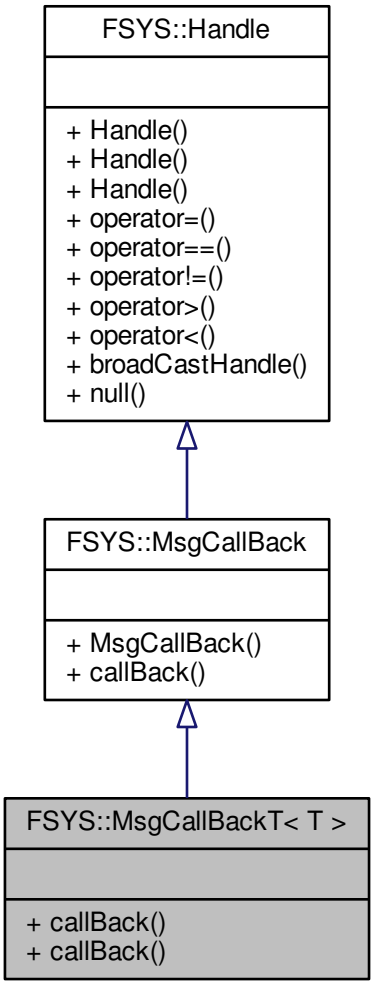
Template class that inherits from MsgCallBack.

```
#include <msgcallback.h>
```

Inheritance diagram for FSYS::MsgCallBackT< T >:



Collaboration diagram for FSYS::MsgCallBackT< T >:



Public Member Functions

- virtual void callBack (T &arg)=0

  Callback function with argument of the correct type.
- void callBack (std::shared_ptr< BaseMsg > &baseMsg)

  callBack function that casts the message to the correct typedef

Additional Inherited Members

12.17.1 Detailed Description

template<class T>
class FSYS::MsgCallBackT< T >

The MsgCallBackT class differs from the MsgCallBack class in that it casts the message sent as argument in the callback to the correct type.

12.17.2 Member Function Documentation

12.17.2.1 template<class T> virtual void FSYS::MsgCallBackT< T >::callBack (T & *arg*) [pure virtual]

This is the callback with the message as the correct type

Parameters

arg	The message that is being sent to this class

12.17.2.2 template<class T> void FSYS::MsgCallBackT< T >::callBack (std::shared_ptr< BaseMsg > & *baseMsg*) [inline],[virtual]

This function does a type cast of the message to the type given in the template parameter.

Parameters

baseMsg	The message that is received by the class

Implements FSYS::MsgCallBack.

The documentation for this class was generated from the following file:

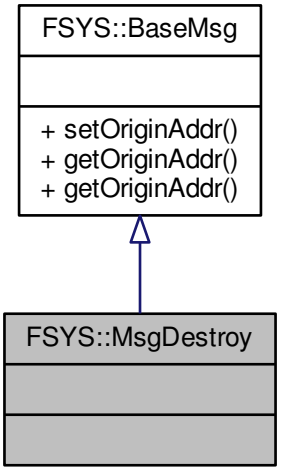
- Interface/FSYS/msgcallback.h (This file was last changed: 2015-02-11 00:49:49 +0100, by Mike Kristoffersen)

12.18 FSYS::MsgDestroy Class Reference

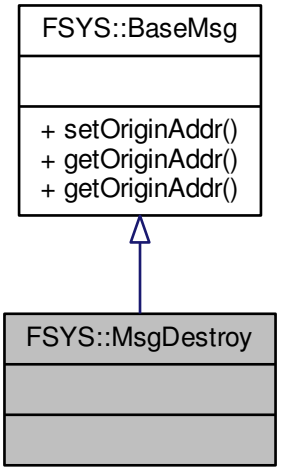
Last message broadcasted to threads.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::MsgDestroy:



Collaboration diagram for FSYS::MsgDestroy:



Additional Inherited Members

12.18.1 Detailed Description

After this message has been received the message loops in the system will terminate and the program shut down

The documentation for this class was generated from the following file:

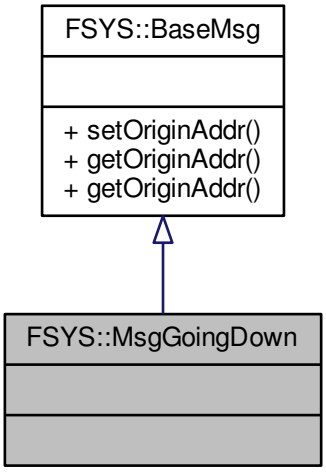
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.19 FSYS::MsgGoingDown Class Reference

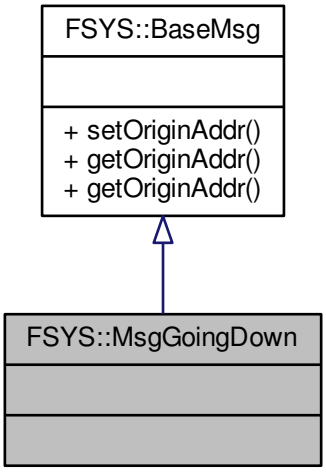
Message broadcasted when the system is being terminated.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::MsgGoingDown:



Collaboration diagram for FSYS::MsgGoingDown:



Additional Inherited Members

12.19.1 Detailed Description

Broadcasted when the system is shutting down. If a module is created after this message is broadcasted, then it won't be able to receive this message, but it will still have its void msgGointDown(void) function called.

The documentation for this class was generated from the following file:

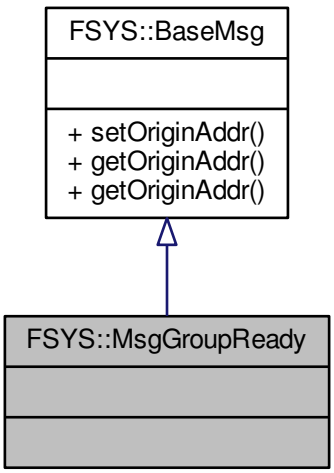
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.20 FSYS::MsgGroupReady Class Reference

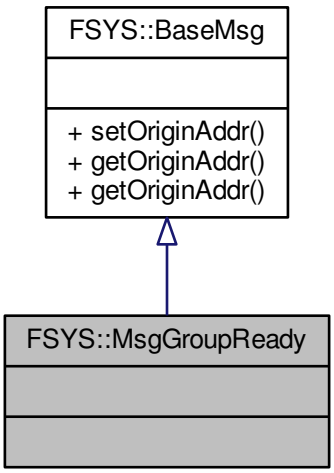
Message broadcasted when all static modules in a group are ready.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::MsgGroupReady:



Collaboration diagram for FSYS::MsgGroupReady:



Additional Inherited Members

12.20.1 Detailed Description

This message is sent by each of the runners when their static modules have started and they are about to enter their message loop

The documentation for this class was generated from the following file:

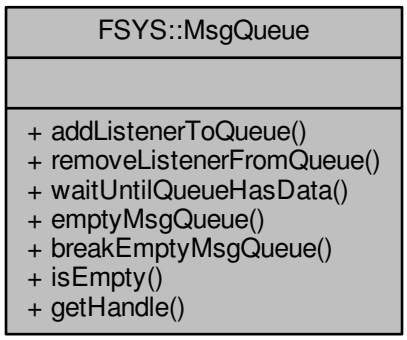
- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.21 FSYS::MsgQueue Class Reference

This is a wrapper class for the class that does all the hard work.

```
#include <msgqueue.h>
```

Collaboration diagram for FSYS::MsgQueue:



Static Public Member Functions

- static void addListenerToQueue (MsgAddr &receiverAddr, const std::type_info &typeOfMsg, MsgCallBack *msgCallBack)

  Function that registers a listener to the message queue.
- static void removeListenerFromQueue (MsgCallBack *msgCallBack)

  Function that unregisters a listener from the message queue.
- static void waitUntilQueueHasData (void)

  Function that halts the current thread execution.
- static void emptyMsgQueue (int maxTimeMs=-1)

  The function will process messages on the message queue.
- static void breakEmptyMsgQueue (void)

  Breaks out of emptyMsgQueue.
- static bool isEmpty (void)

  Function to determine if there is data on the msg queue.
- static Handle getHandle (void)

  Retrieves the handle of the queue for the current thread.

12.21.1 Detailed Description

The MsgQueue class is a wrapper class with the public interface for the message queue. It contains the functions that gives the caller the ability to:

- Register listeners (a listener being the ability for a given instance of a given class to recevie a specific message type).
- Unregister listeners (like right before an instance of a class is destroyed)
- Checking if the message queue has data for the current thread that needs to be processed
- Stopping the current thread until there is a message being sent to it
- Stopping the process of emptying the queue, like for breaking out of the internal message loop, say if a class keeps sending messages to it self
- Emptying the message queue, either it can start a loop that terminates when the queue is empty, after a given time, or after all the messages that were in the queue at the time of the call has been processed

12.21.2 Member Function Documentation

12.21.2.1 static void FSYS::MsgQueue::addListenerToQueue (MsgAddr & *receiverAddr*, const std::type_info & *typeOfMsg*, MsgCallBack * *msgCallBack*) [static]

This function registers a listener (the ability of a class to receive a given type of message) with the queue of the thread that the function is called from.

Parameters

receiverAddr	Address of the receiver
typeOfMsg	
msgCallBack	

12.21.2.2 static void FSYS::MsgQueue::breakEmptyMsgQueue (void) [static]

Function must be called from a message handler (directly or indirectly) and will cause the message loop internal to the emptyMsgQueue function to break.

This function is used in the case where you have a component that as part of its operation sends messages to it self.

Usually the emptyMsgQueue function will only return when the message is queue is empty, but in the case where components on the queue keeps sending to other components in the same thread, the queue will never be empty and hence will never terminate.

Calling this function will break the msg loop.

12.21.2.3 static void FSYS::MsgQueue::emptyMsgQueue (int *maxTimeMs* = -1) [static]

This function is used for processing messages from the message queue of the thread that this function is called from.

The default behavior is for the function to run until there are no more messages on the queue at which point the function will return.

It is possible to give an optional argument to the function limiting the time it will run.

Parameters

maxTimeMs	The maximum time in MS that will pass before the function breaks the loop that empties the queue.

maxTimeMs < 0 => Run until the queue is empty or breakEmptyMsgQueue() is called

maxTimeMS == 0 => Run until all messages on the queue at the time of the call are processed

maxTimeMS > 0 => Run approximately until maxTimeMS have passed

12.21.2.4 static Handle FSYS::MsgQueue::getHandle (void) [static]

This function can be used by threads to identify which queue they belong to.

Returns

The handle of the queue for the current thread.

12.21.2.5 static bool FSYS::MsgQueue::isEmpty (void) [static]

Call this function to determine if there are messages waiting to be processed on the queue.

Returns

true if there is messages waiting to be processd, false if not

12.21.2.6 static void FSYS::MsgQueue::removeListenerFromQueue (MsgCallBack * *msgCallBack*) [static]

This function removes a listener from the internal datastructures of the message queue.

Parameters

msgCallBack	Pointer to the callback that should be removed

12.21.2.7 static void FSYS::MsgQueue::waitUntilQueueHasData (void) [static]

This function halts the current thread until someone sends a message to the thread. Be careful not the halt all of your threads, since in that case no-one will be able to send any messages to wake the system up again.

The documentation for this class was generated from the following file:

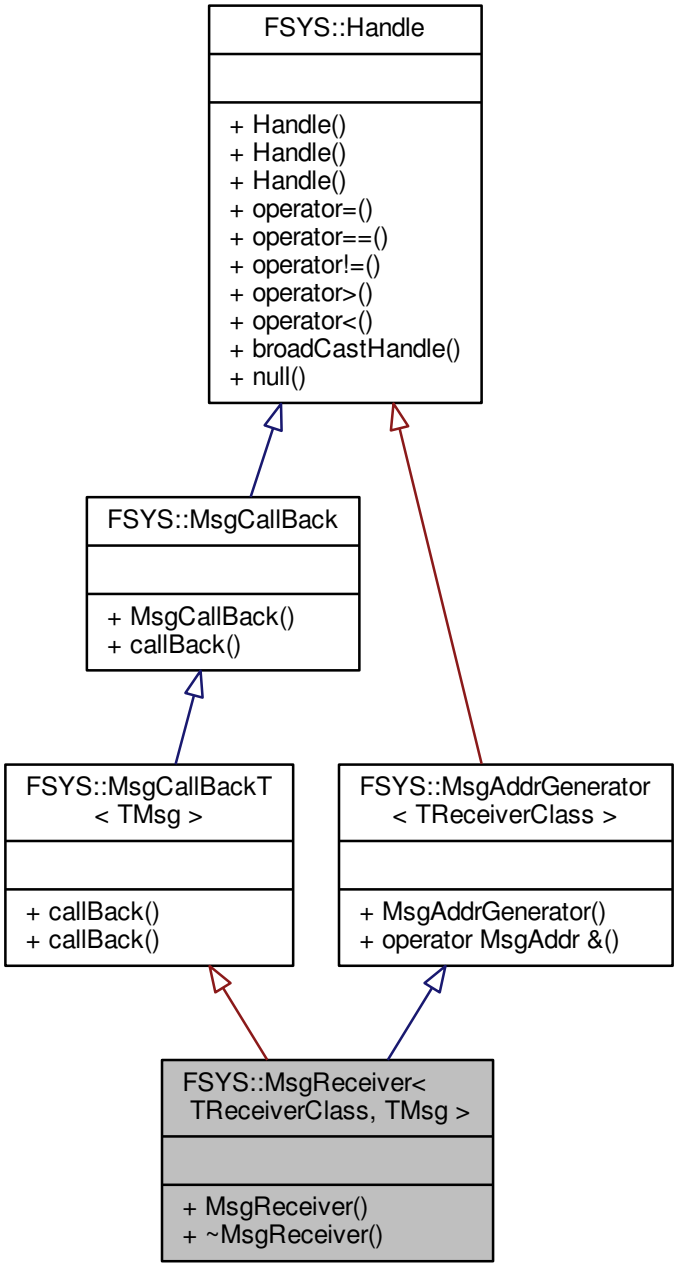
- Interface/FSYS/msgqueue.h (This file was last changed: 2015-02-11 00:49:49 +0100, by Mike Kristoffersen)

12.22 FSYS::MsgReceiver< TReceiverClass, TMsg > Class Template Reference

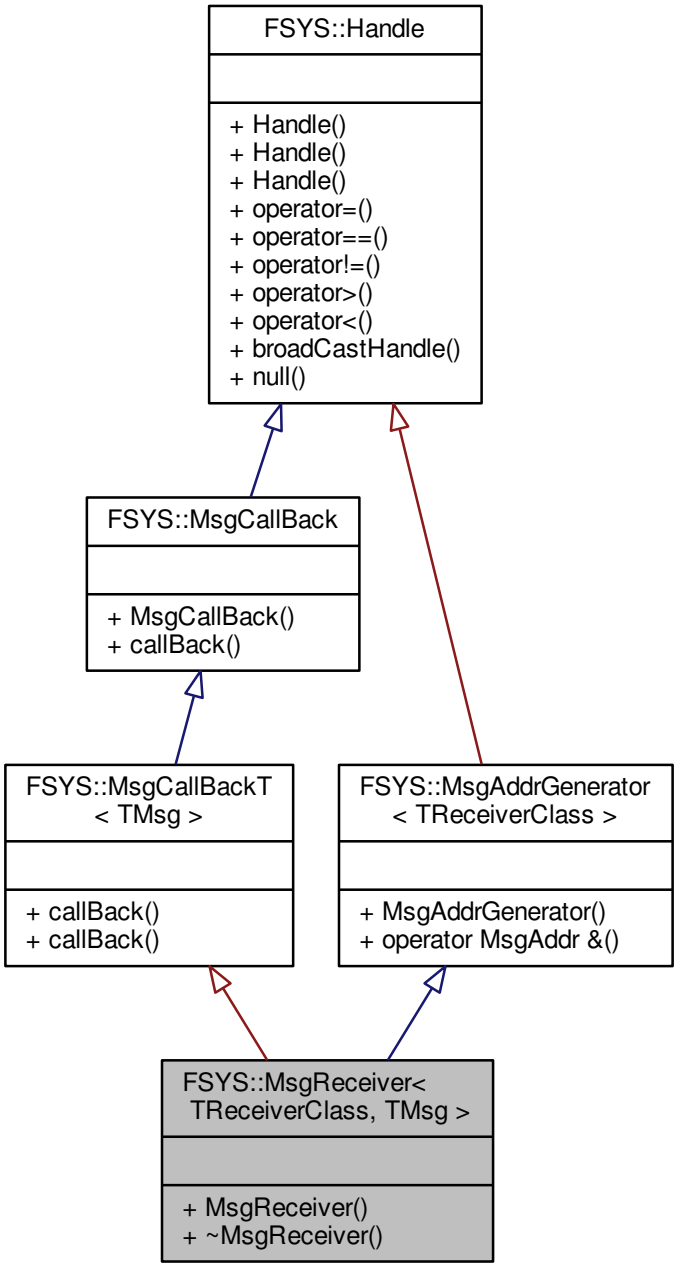
Template class allowing other classes to receive messages.

```
#include <msgreceiver.h>
```

Inheritance diagram for FSYS::MsgReceiver< TReceiverClass, TMsg >:



Collaboration diagram for FSYS::MsgReceiver< TReceiverClass, TMsg >:



Public Member Functions

- MsgReceiver (void)

  Constructor for MsgReceiver class.
- ~MsgReceiver ()

  Destructor for MsgReceiver class.

12.22.1 Detailed Description

template<class TReceiverClass, class TMsg>
class FSYS::MsgReceiver< TReceiverClass, TMsg >

This is the class that other classes can inherit from in order to receive specific messages.

When inheriting from this class you give the parent class and the class of the message that you want to receive as template parameters.

In order to receive a message, the class must inherit from this class and implement a receive function that takes the type of message as parameter.

```
class MyClass : public MsgReceiver<MyClass, MyMessage>
{
public:
 void receive(MyMessage &message);
}
```

Remember to take a copy of message if you want to use its content later, or want to respond to it.

The receive function will be called in the thread context where the object was initialised, and it is not possible to move it to another thead.

Note

> The thread where the object is instantiatet must be a thread that is created in a way that is compatible with the C++11 thread system.
> A class can only count on receiving a message if the class is fully constructed and registered with the queue/message system at the time where the message is sent.

It is possible to declare the receive function as virtual if that is desired.

12.22.2 Constructor & Destructor Documentation

12.22.2.1 template<class TReceiverClass, class TMsg> FSYS::MsgReceiver< TReceiverClass, TMsg >::MsgReceiver (void) [inline]

The constructor will register this class with the message system, so it is possible for it to receive messages of the specific type.

12.22.2.2 template<class TReceiverClass, class TMsg> FSYS::MsgReceiver< TReceiverClass, TMsg >::~MsgReceiver ( ) [inline]

The destructor of the MsgReceiver class unregisters this class from the message system, so it won't attempt to call it again.

The documentation for this class was generated from the following file:

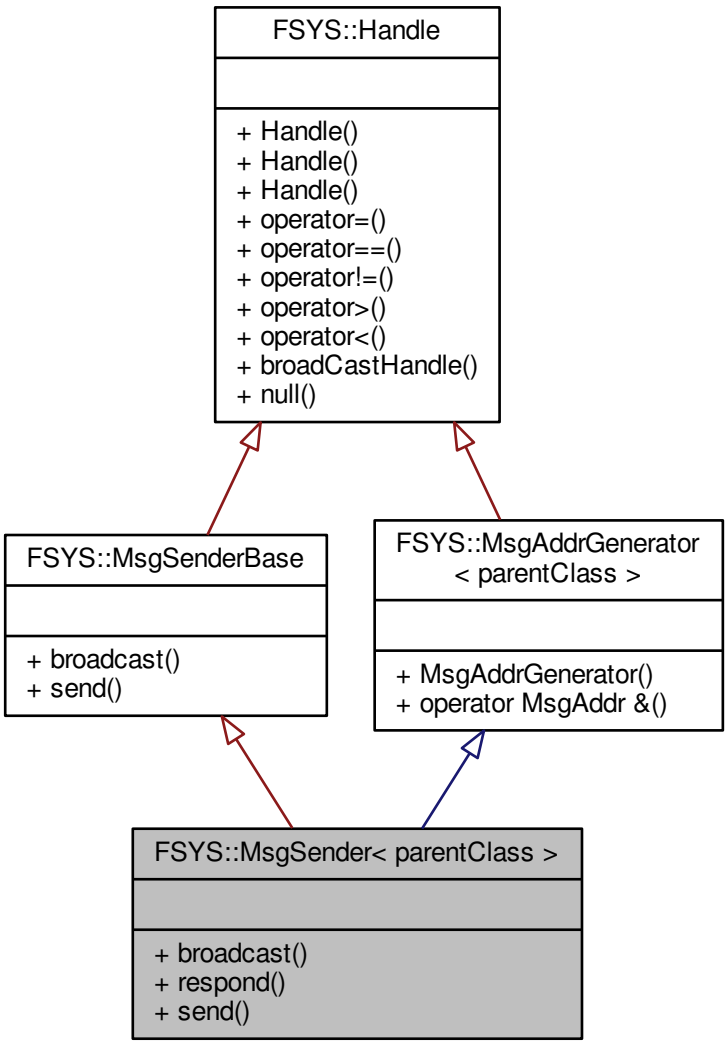
- Interface/FSYS/msgreceiver.h (This file was last changed: 2015-02-11 00:49:49 +0100, by Mike Kristoffersen)

12.23 FSYS::MsgSender< parentClass > Class Template Reference

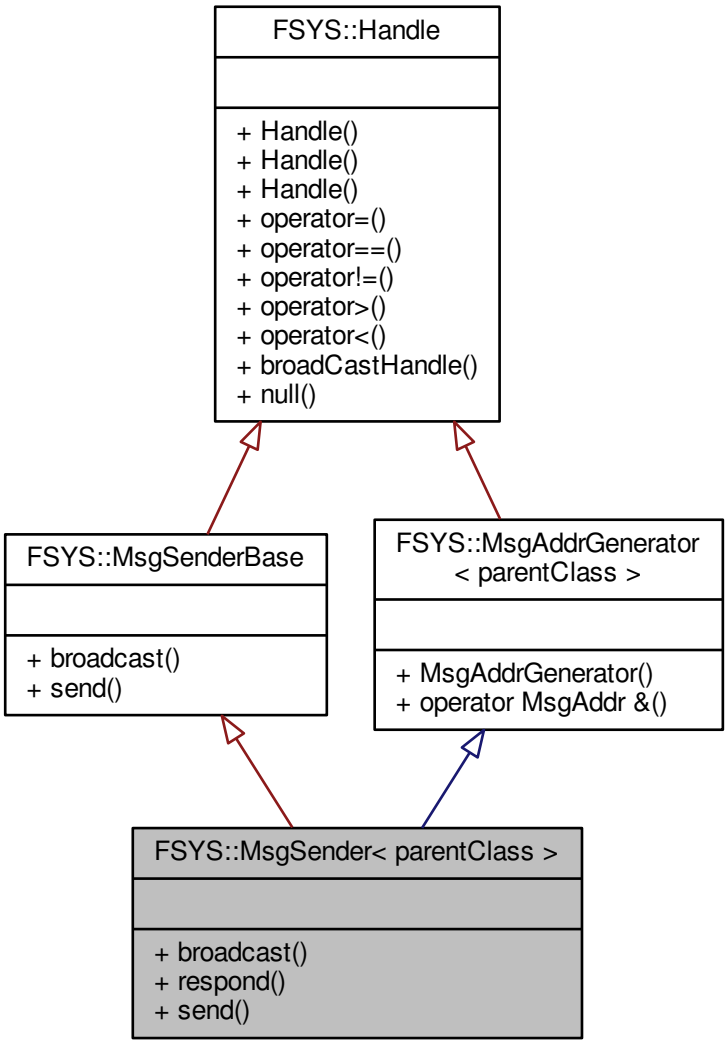
Class enabling other classes to send messages.

```
#include <msgsender.h>
```

Inheritance diagram for FSYS::MsgSender< parentClass >:



Collaboration diagram for FSYS::MsgSender< parentClass >:



Public Member Functions

- template<class T >
  void broadcast (T &msg)

  Template function to broadcast messages.

- template<class T >
  void respond (T &msg, BaseMsg &received)

  Template function to respond to messages.

- template<class T >
  void **send** (T &msg, const MsgAddr &destination)

12.23.1 Detailed Description

template<class parentClass>
class FSYS::MsgSender< parentClass >

When inheriting from this class it becomes possible for the class inheriting to send all types of messages.

There are two ways to send messages, either as broadcasts or as responses to other messages.

Broadcasts will be send to all instances of classes that listen to the specific type of message.

Responses will only go the the instance of the class that sended the message that is given in the argument to the respond function.

For a class to be considered as a receiver of a message, the class needs to be instantiated at the time when the message was sent.

12.23.2 Member Function Documentation

12.23.2.1 template<class parentClass> template<class T > void FSYS::MsgSender< parentClass >::broadcast (T & *msg*) [inline]

When a message is broadcasted it will be received by all classes listening on the specific message type at the time the message was sent.

Warning

It is the type of the reference to the message (T in the template class) that is used by the system to identify the type of the message not the actuall message type. If you have a base class B, and inherit A from it: B <- A and you then instantiate an instance of A, but uses a B reference to hold it (like A a; B &b = a;) then the message that will be sent will be of type B.

```
class B : public BaseMsg {};
class A ; public B {};
A a;
B &b=a;
broadcast(b); // Even this is an A, it will be send as a B, so only
              // classes listening for B will get the message
```

Parameters

msg	The message that should be sent

12.23.2.2 template<class parentClass> template<class T > void FSYS::MsgSender< parentClass >::respond (T & *msg*, BaseMsg & *received*) [inline]

When a message is recevied it is possible for at class to send a message back to the sender and only the sender. To do this you use the respond function.

You can use the respond function and another message to send several messages to the sender of that message.

A common use case for a flow where the respond function is used is:

```
// Client searches for someone to handle a request
ServiceXRequest sXr(<arguments to request>);
broadcast(sXr);
...

// Server receives the request
receive(ServiceXRequest &sXr)
{
   // Server sends ack to the client
   ServiceXAck sXa
   respond(sXa, sXr);
   ...

   // At some later point the server can send data to the client
   ServiceXProgressIndicator sXpi10(10);
   respond(sXpi10, sXr);
   ---

   // It can send multiple data
   ServiceXProgressIndicator xXpi20(20);
   respond(sXpi20, sXr);

// Client don't need the service of the server any more, so it closes
// the connection
receive(ServiceXAck sXa)
{
  ServiceXTerminate sXt;
  respond(sXt, sXa);

// The service the listsn for the ServiceXTerminate signal, and when
// received does what is needed to free internal resources ect.
```

The documentation for this class was generated from the following file:

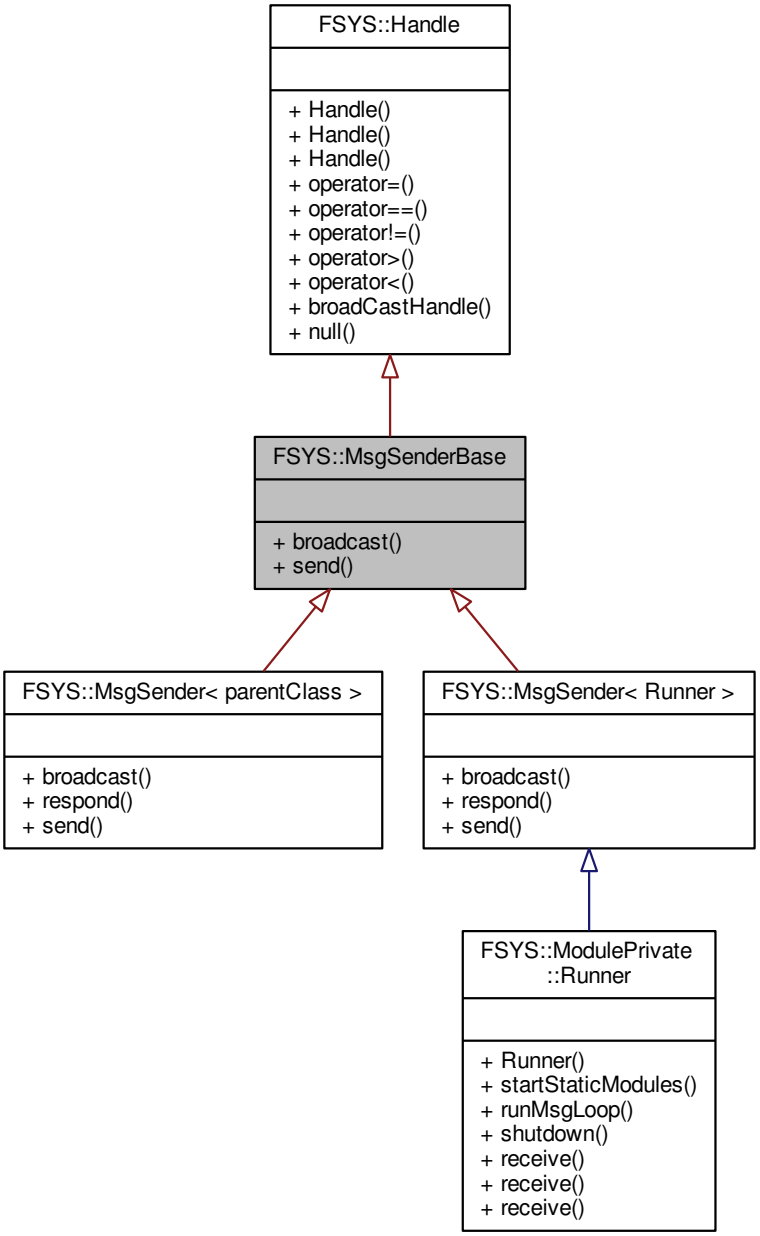
- Interface/FSYS/msgsender.h (This file was last changed: 2015-02-16 12:50:29 +0100, by Jacob Andersen)

12.24 FSYS::MsgSenderBase Class Reference

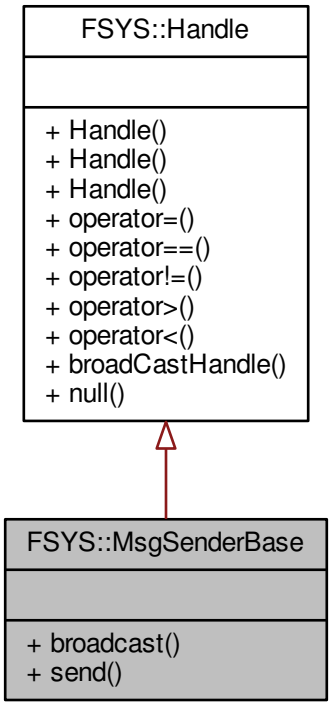
Type neutral message sender class.

```
#include <msgsenderbase.h>
```

Inheritance diagram for FSYS::MsgSenderBase:



Collaboration diagram for FSYS::MsgSenderBase:



Public Member Functions

- void broadcast (std::shared_ptr< BaseMsg > &msg, const std::type_info &typeOfMsg, const MsgAddr &sender)

  Broadcast function, to broadcast messages.
- void send (std::shared_ptr< BaseMsg > &msg, const std::type_info &typeOfMsg, const FSYS::MsgAddr &sender, const MsgAddr &destination)

  Type independent respond function.

12.24.1 Detailed Description

This is the sender base class, the sender class uses the thin template priciple, where the type specific code is concentrated in one class and the type generic code is in a different base class, this is that base class.

12.24.2 Member Function Documentation

12.24.2.1 void FSYS::MsgSenderBase::broadcast (std::shared_ptr< BaseMsg > & *msg*, const std::type_info & *typeOfMsg*, const MsgAddr & *sender*)

This is a type neutral function that broadcasts a message. It takes the type info of the message as parameter as well as the message it self.

Parameters

msg	The message that should be sent.
typeOfMsg	The type (like the typeinfo() type.
sender	Address of the sender of the message

12.24.2.2 void FSYS::MsgSenderBase::send (std::shared_ptr< BaseMsg > & *msg*, const std::type_info & *typeOfMsg*, const FSYS::MsgAddr & *sender*, const MsgAddr & *destination*)

This function is used for sending a message back to an instance of a class that previously had sent the "received" message.

Parameters

msg	The message that should be sent.
typeOfMsg	Type of the message that is being sent
sender	Address of the sender of the message
destination	The destination of the message

The documentation for this class was generated from the following file:

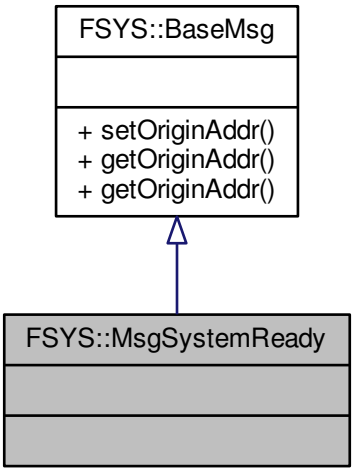
- Interface/FSYS/msgsenderbase.h (This file was last changed: 2015-02-16 12:50:29 +0100, by Jacob Andersen)

12.25 FSYS::MsgSystemReady Class Reference

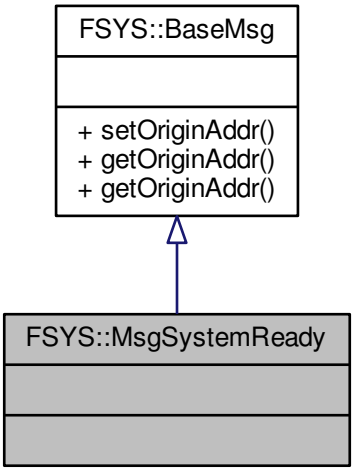
Message broadcasted when the system is initialised.

```
#include <moduledeclare.h>
```

Inheritance diagram for FSYS::MsgSystemReady:



Collaboration diagram for FSYS::MsgSystemReady:



Additional Inherited Members

12.25.1 Detailed Description

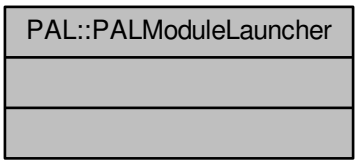
Broadcasted when all static modules are started. If a module is created after this message is broadcasted, then it won't be able to receive this message, but it will still have its void msgSystemReady(void) function called.

The documentation for this class was generated from the following file:

- Interface/FSYS/moduledeclare.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.26 PAL::PALModuleLauncher Class Reference

Collaboration diagram for PAL::PALModuleLauncher:



The documentation for this class was generated from the following file:

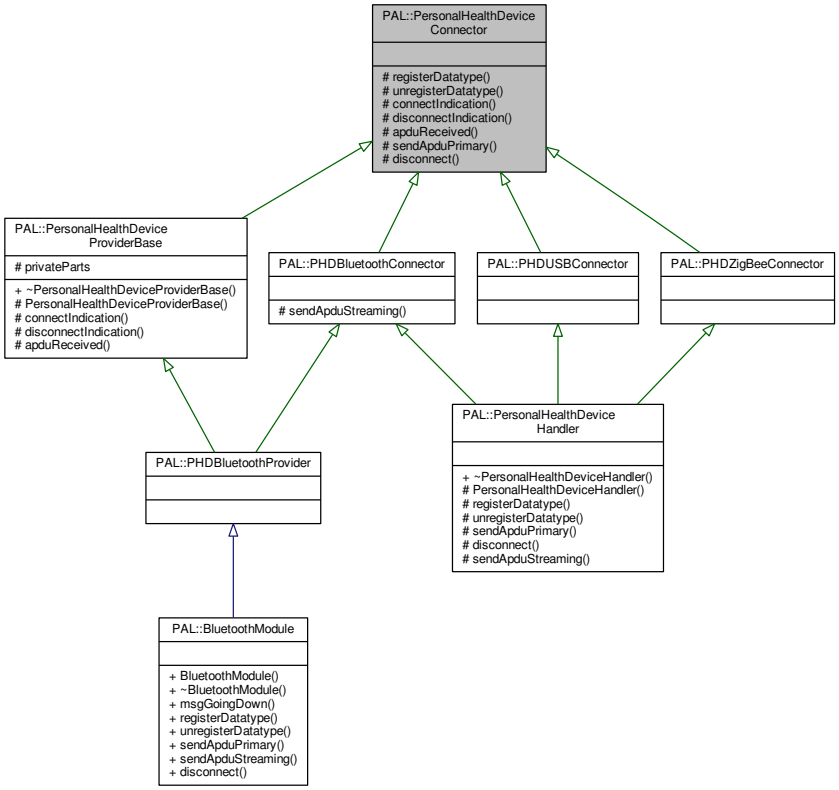
- Interface/PAL/platform.h (This file was last changed: 2015-02-16 10:35:24 +0100, by Jacob Andersen)

12.27 PAL::PersonalHealthDeviceConnector Class Reference

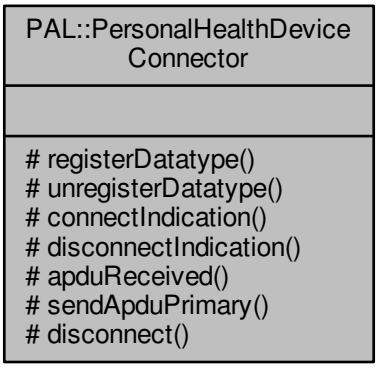
The PAL interface for communication with personal health devices.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PersonalHealthDeviceConnector:



Collaboration diagram for PAL::PersonalHealthDeviceConnector:



Protected Member Functions

- virtual void registerDatatype (uint16_t datatype)=0

Registers a session-layer component as handler of a datatype.

- virtual void unregisterDatatype (uint16_t datatype) noexcept=0

  Unregisters a session-layer component as handler of a datatype.
- virtual void connectIndication (std::shared_ptr< VirtualPHD > device) noexcept=0

  Indicates the connection of a new device.
- virtual void disconnectIndication (std::shared_ptr< VirtualPHD > device, errortype error=0) noexcept=0

  Indicates a device disconnection along with error details.
- virtual void apduReceived (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu, bool reliableTransport) noexcept=0

  Receive a message from this device.
- virtual void sendApduPrimary (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept=0

  Send a message to this device on the primary virtual channel.
- virtual void disconnect (std::shared_ptr< VirtualPHD > device) noexcept=0

  Disconnect the device.

12.27.1 Detailed Description

The communication between transport components in the PAL layer and session-layer components is defined by this class.

All possible signals communicated across the PAL interface are realized as methods of this class. Some are implemented by the session layer and called from the PAL layer, while others travels in the opposite direction. Two abstract subclasses of this class, PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler manages the magic of wrapping these method calls in messages and sending them back and forth between the modules. The PersonalHealthDeviceHandler will be extended by a class in the session layer, while the PersonalHealthDeviceProviderBase is extended by a class in the PAL layer (a component of the appropriate transport).

The PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler classes will keep track of which PAL- and session-layer components handles which PersonalHealthDevice, and make sure the messages are forwarded appropriately. A few messages does not relate to particular devices, and these messages will instead be broadcast to all possible peers.

See also

PersonalHealthDevice
PersonalHealthDeviceProviderBase
PersonalHealthDeviceHandler

12.27.2 Member Function Documentation

12.27.2.1 virtual void PAL::PersonalHealthDeviceConnector::apduReceived (std::shared_ptr< VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*, bool *reliableTransport*) [protected], [pure virtual], [noexcept]

A PAL-to-protocol message. A message arrived from a connected device. This device was previously announced by the connectIndication() signal and has not yet been disconnected.

Parameters

device	The source of the message.
apdu	The buffer of bytes received from the device.
reliableTransport	The message arrived on a reliable transport channel.

Implemented in PAL::PersonalHealthDeviceProviderBase.

12.27.2.2 virtual void PAL::PersonalHealthDeviceConnector::connectIndication (std::shared_ptr< VirtualPHD > *device*) [protected],[pure virtual],[noexcept]

A PAL-to-protocol message. A new device has been connected at the PAL layer and is now available for the session layer.

The message magic implemented by PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler will choose a session-layer component which has previously registered itself for the datatype matching this device using registerDatatype() and establish a one-to-one connection. If no session-layer component is currently registered, the message will be dropped.

Parameters

device	The device that was just connected.

Implemented in PAL::PersonalHealthDeviceProviderBase.

12.27.2.3 virtual void PAL::PersonalHealthDeviceConnector::disconnect (std::shared_ptr< VirtualPHD > *device*) [protected],[pure virtual],[noexcept]

A protocol-to-PAL message. If the device is currently connected, it will be disconnected immediately (and a disconnectIndication() will be returned from the PAL shortly), otherwise nothing will happen.

Parameters

device	The device, we wish to disconnect.

Implemented in PAL::PersonalHealthDeviceHandler.

12.27.2.4 virtual void PAL::PersonalHealthDeviceConnector::disconnectIndication (std::shared_ptr< VirtualPHD > *device*, errortype *error* = 0) [protected],[pure virtual],[noexcept]

A PAL-to-protocol message. A device (which was previously connected using connectIndication()) has been disconnected. It is no longer available for the session layer. No more messageReceived() signals will ever be emitted for this object, and sendMessage() attempts will be discarded. The session layer should forget all about this object and delete all references.

Notice that this message also cuts the link between the PAL and session layer components, so no more messages regarding this device can be communicated until the PAL layer emits a new connectionIndication() (which, by-the-way, may choose to connect the device to a different protocol component, if more are available).

The optional error argument carries detailed information of an error in the PAL-layer component that caused this disconnection. The usage of this argument is completely optional for the PAL component – it is always allowed to use the non-error variant instead. The session-layer component may also choose to ignore the error argument; however, it is recommended that the session layer should forward the error message to the application layer in some way, to allow the error message to be presented to the user, if relevant.

Todo Fix doc when the actual errortype has been chosen...

Parameters

device	The device object, which will no longer be in use.
error	An error code (optional - defaults to "no error").

Implemented in PAL::PersonalHealthDeviceProviderBase.

12.27.2.5 virtual void PAL::PersonalHealthDeviceConnector::registerDatatype (uint16_t *datatype*) [protected], [pure virtual]

A protocol-to-PAL message. This message is issued by the session-layer component, when it is ready to handle a given datatype. When the component no longer handles the datatype, the unregisterDatatype() is used to stop new connections of that particular datatype.

The datatype must be one of the IEEE `MDC_DEV_SPEC_PROFILE_` codes belonging to the `MDC_PART_INFRA` partition.

The PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler classes will keep track of which session-layer components handles which datatypes, and will broadcast the registerDatatype() and unregister↩Datatype() messages to all PAL-layer components when a previously unhandled datatype is registered for the first time and when a datatype is no longer being handled.

Except in the unlikely event of memory exhaustion, this method should never throw any exceptions.

See also

unregisterDatatype()

Parameters

datatype	The datatype handled by this component.

Exceptions

std::bad_alloc	in case of memory exhaustion.

Implemented in PAL::PersonalHealthDeviceHandler, and PAL::BluetoothModule.

12.27.2.6 virtual void PAL::PersonalHealthDeviceConnector::sendApduPrimary (std::shared_ptr< VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*) [protected], [pure virtual], [noexcept]

A protocol-to-PAL message. If the device is currently connected, this method will (attempt to) send a message to the device on the primary virtual channel (which by definition is a reliable transport service). There will be no indication of whether this attempt was succesful or not. In particular, if this device object is currently not connected, the method invocation will be ignored.

Parameters

device	The device, we are sending to.
apdu	The buffer of bytes to transmit to the device.

Implemented in PAL::PersonalHealthDeviceHandler.

12.27.2.7 virtual void PAL::PersonalHealthDeviceConnector::unregisterDatatype (uint16_t *datatype*) [protected], [pure virtual], [noexcept]

A protocol-to-PAL message. This message is issued by the session-layer component, when it no longer wishes to handle the given datatype. For more details, see the documentation of the registerDatatype() method.

See also

registerDatatype()

Parameters

datatype	The datatype no longer handled by this component.

Implemented in PAL::PersonalHealthDeviceHandler, and PAL::BluetoothModule.

The documentation for this class was generated from the following file:

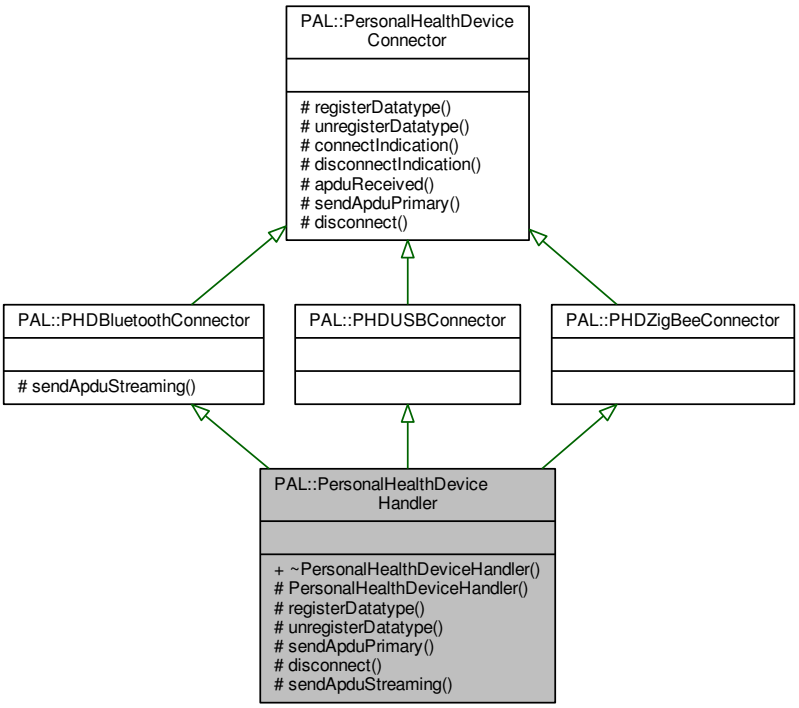
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.28 PAL::PersonalHealthDeviceHandler Class Reference

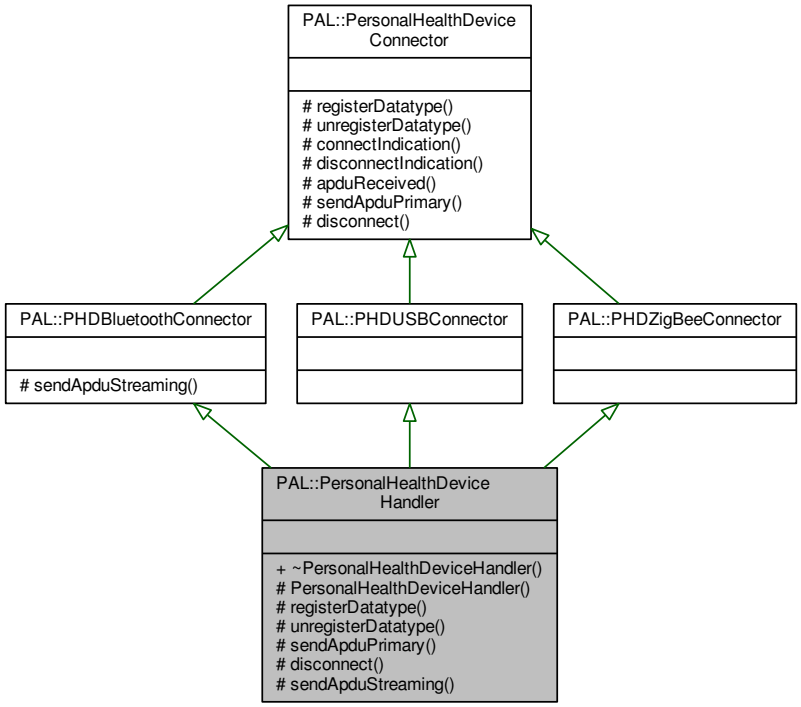
The abstract class, session-layer components must implement to handle communication with personal health devices from any transport class.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PersonalHealthDeviceHandler:



Collaboration diagram for PAL::PersonalHealthDeviceHandler:



Public Member Functions

- virtual ~PersonalHealthDeviceHandler ()

  Cleanup and make sure all devices are disconnected.

Protected Member Functions

- void registerDatatype (uint16_t datatype)

  Registers a session-layer component as handler of a datatype.
- void unregisterDatatype (uint16_t datatype) noexcept

  Unregisters a session-layer component as handler of a datatype.
- void sendApduPrimary (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept

  Send a message to this device on the primary virtual channel.
- void disconnect (std::shared_ptr< VirtualPHD > device) noexcept

  Disconnect the device.
- void sendApduStreaming (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept

  Send a message to this device on a streaming virtual channel.

12.28.1 Detailed Description

See also

PersonalHealthDevice
PersonalHealthDeviceConnector
PersonalHealthDeviceProviderBase

12.28.2 Constructor & Destructor Documentation

12.28.2.1 virtual PAL::PersonalHealthDeviceHandler::~PersonalHealthDeviceHandler ( ) [virtual]

The destructor will make sure that any connection to the PAL-layer components that was left open will be disconnected, and that any registered datatype that the session-layer component (child class) forgot to unregister will be properly unregistered.

12.28.3 Member Function Documentation

12.28.3.1 void PAL::PersonalHealthDeviceHandler::disconnect (std::shared_ptr< VirtualPHD > *device*) [protected], [virtual], [noexcept]

A protocol-to-PAL message. If the device is currently connected, it will be disconnected immediately (and a disconnectIndication() will be returned from the PAL shortly), otherwise nothing will happen.

Parameters

device	The device, we wish to disconnect.

Implements PAL::PersonalHealthDeviceConnector.

12.28.3.2 void PAL::PersonalHealthDeviceHandler::registerDatatype (uint16_t *datatype*) [protected], [virtual]

A protocol-to-PAL message. This message is issued by the session-layer component, when it is ready to handle a given datatype. When the component no longer handles the datatype, the unregisterDatatype() is used to stop new connections of that particular datatype.

The datatype must be one of the IEEE MDC_DEV_SPEC_PROFILE_ codes belonging to the MDC_PART_INFRA partition.

The PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler classes will keep track of which session-layer components handles which datatypes, and will broadcast the registerDatatype() and unregisterDatatype() messages to all PAL-layer components when a previously unhandled datatype is registered for the first time and when a datatype is no longer being handled.

Except in the unlikely event of memory exhaustion, this method should never throw any exceptions.

See also

unregisterDatatype()

Parameters

datatype	The datatype handled by this component.

Exceptions

std::bad_alloc	in case of memory exhaustion.

Implements PAL::PersonalHealthDeviceConnector.

12.28.3.3 void PAL::PersonalHealthDeviceHandler::sendApduPrimary (std::shared_ptr< VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*) [protected], [virtual], [noexcept]

A protocol-to-PAL message. If the device is currently connected, this method will (attempt to) send a message to the device on the primary virtual channel (which by definition is a reliable transport service). There will be no indication of whether this attempt was succesful or not. In particular, if this device object is currently not connected, the method invocation will be ignored.

Parameters

device	The device, we are sending to.
apdu	The buffer of bytes to transmit to the device.

Implements PAL::PersonalHealthDeviceConnector.

12.28.3.4 void PAL::PersonalHealthDeviceHandler::sendApduStreaming (std::shared_ptr< VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*) [protected], [virtual], [noexcept]

A protocol-to-PAL message. If the device is currently connected, this method will (attempt to) send a message to the device on a streaming virtual channel. If such a channel is not available, it will attempt the primary virtual channel instead. There will be no indication of whether this attempt was succesful or not. In particular, if this device object is currently not connected, the method invocation will simply be ignored.

Parameters

device	The bluetooth device to send to. The shared pointer must point to a PHDBluetoothDevice, otherwise the method will use the primary virtual channel.
apdu	The buffer of bytes to transmit to the device.

Implements PAL::PHDBluetoothConnector.

12.28.3.5 void PAL::PersonalHealthDeviceHandler::unregisterDatatype (uint16_t *datatype*) [protected], [virtual], [noexcept]

A protocol-to-PAL message. This message is issued by the session-layer component, when it no longer wishes to handle the given datatype. For more details, see the documentation of the registerDatatype() method.

See also

registerDatatype()

Parameters

datatype	The datatype no longer handled by this component.

Implements PAL::PersonalHealthDeviceConnector.

The documentation for this class was generated from the following file:

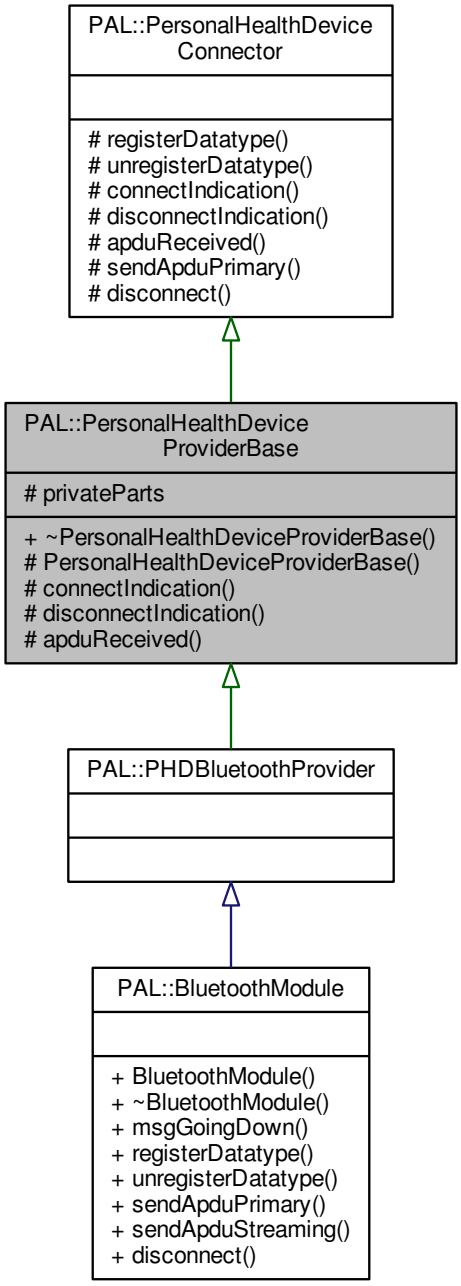
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.29 PAL::PersonalHealthDeviceProviderBase Class Reference

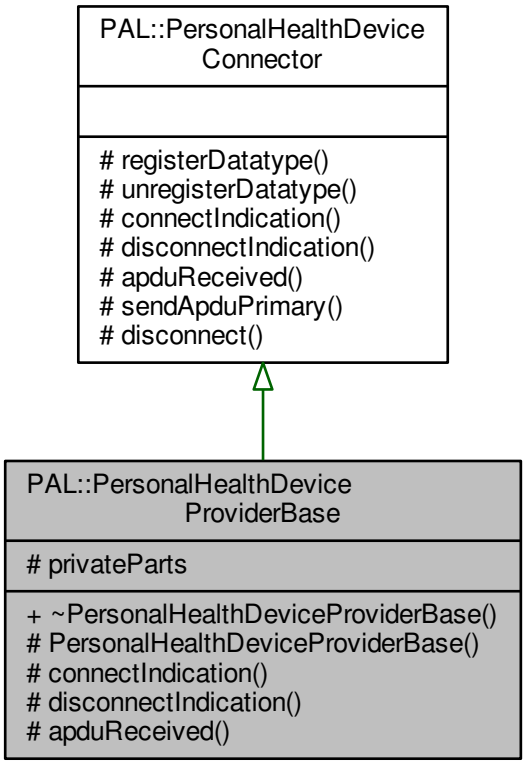
The abstract class, PAL-layer components must implement to provide communication with personal health devices.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PersonalHealthDeviceProviderBase:



Collaboration diagram for PAL::PersonalHealthDeviceProviderBase:



Public Member Functions

- virtual ~PersonalHealthDeviceProviderBase ()

  Cleanup and make sure all devices are disconnected.

Protected Member Functions

- void connectIndication (std::shared_ptr< VirtualPHD > device) noexcept

  Indicates the connection of a new device.
- void disconnectIndication (std::shared_ptr< VirtualPHD > device, errortype error) noexcept

  Indicates a device disconnection along with error details.
- void apduReceived (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu, bool reliableTransport) noexcept

  Receive a message from this device.

Protected Attributes

- PrivatePHDProvider * **privateParts**

12.29.1 Detailed Description

See also

PersonalHealthDevice
PersonalHealthDeviceConnector
PersonalHealthDeviceHandler

12.29.2 Constructor & Destructor Documentation

12.29.2.1 virtual PAL::PersonalHealthDeviceProviderBase::~PersonalHealthDeviceProviderBase ( ) [virtual]

The destructor will make sure that any connection to the session-layer components that was left open will be disconnected.

12.29.3 Member Function Documentation

12.29.3.1 void PAL::PersonalHealthDeviceProviderBase::apduReceived (std::shared_ptr< VirtualPHD > *device*, std::shared_ptr< std::vector< uint8_t > > *apdu*, bool *reliableTransport*) [protected], [virtual], [noexcept]

A PAL-to-protocol message. A message arrived from a connected device. This device was previously announced by the connectIndication() signal and has not yet been disconnected.

Parameters

device	The source of the message.
apdu	The buffer of bytes received from the device.
reliableTransport	The message arrived on a reliable transport channel.

Implements PAL::PersonalHealthDeviceConnector.

12.29.3.2 void PAL::PersonalHealthDeviceProviderBase::connectIndication (std::shared_ptr< VirtualPHD > *device*) [protected], [virtual], [noexcept]

A PAL-to-protocol message. A new device has been connected at the PAL layer and is now available for the session layer.

The message magic implemented by PersonalHealthDeviceProviderBase and PersonalHealthDeviceHandler will choose a session-layer component which has previously registered itself for the datatype matching this device using registerDatatype() and establish a one-to-one connection. If no session-layer component is currently registered, the message will be dropped.

Parameters

device	The device that was just connected.

Implements PAL::PersonalHealthDeviceConnector.

12.29.3.3 void PAL::PersonalHealthDeviceProviderBase::disconnectIndication (std::shared_ptr< VirtualPHD > *device*, errortype *error*) [protected],[virtual],[noexcept]

A PAL-to-protocol message. A device (which was previously connected using connectIndication()) has been disconnected. It is no longer available for the session layer. No more messageReceived() signals will ever be emitted for this object, and sendMessage() attempts will be discarded. The session layer should forget all about this object and delete all references.

Notice that this message also cuts the link between the PAL and session layer components, so no more messages regarding this device can be communicated until the PAL layer emits a new connectionIndication() (which, by-the-way, may choose to connect the device to a different protocol component, if more are available).

The optional error argument carries detailed information of an error in the PAL-layer component that caused this disconnection. The usage of this argument is completely optional for the PAL component – it is always allowed to use the non-error variant instead. The session-layer component may also choose to ignore the error argument; however, it is recommended that the session layer should forward the error message to the application layer in some way, to allow the error message to be presented to the user, if relevant.

Todo Fix doc when the actual errortype has been chosen...

Parameters

device	The device object, which will no longer be in use.
error	An error code (optional - defaults to "no error").

Implements PAL::PersonalHealthDeviceConnector.

The documentation for this class was generated from the following file:

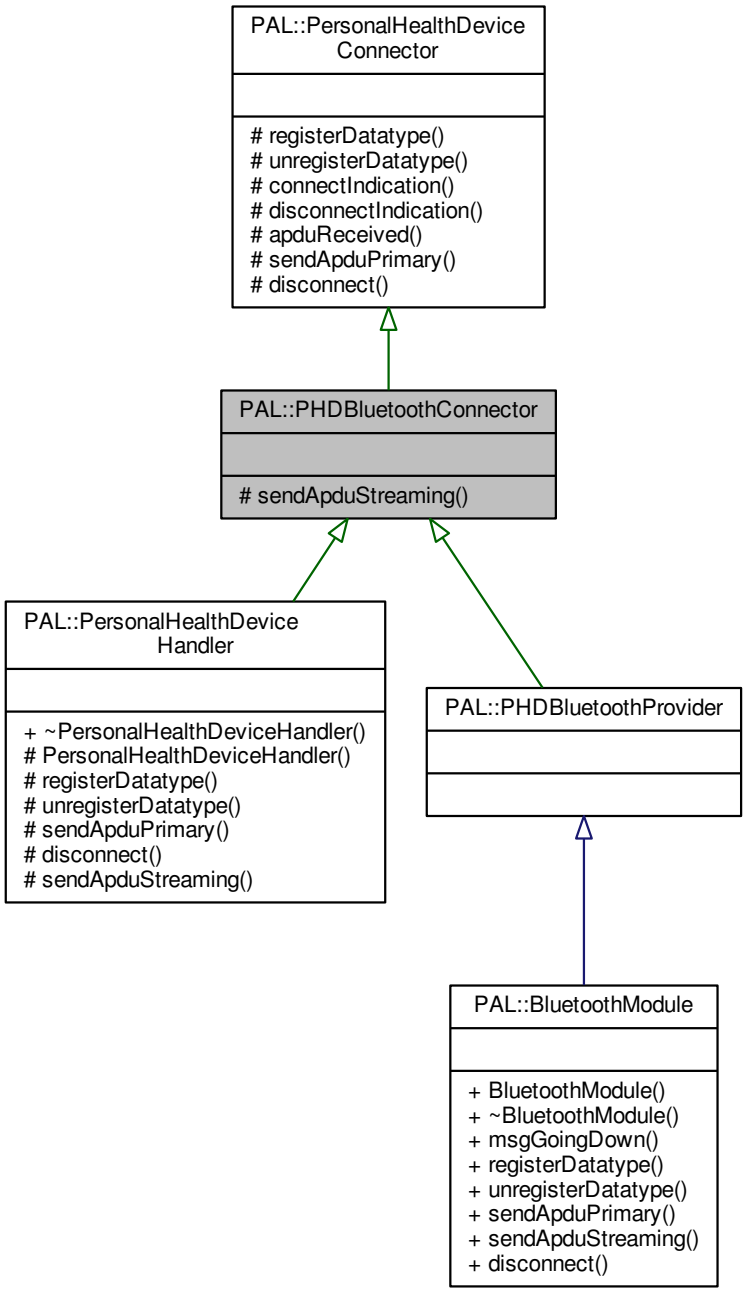
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.30 PAL::PHDBluetoothConnector Class Reference

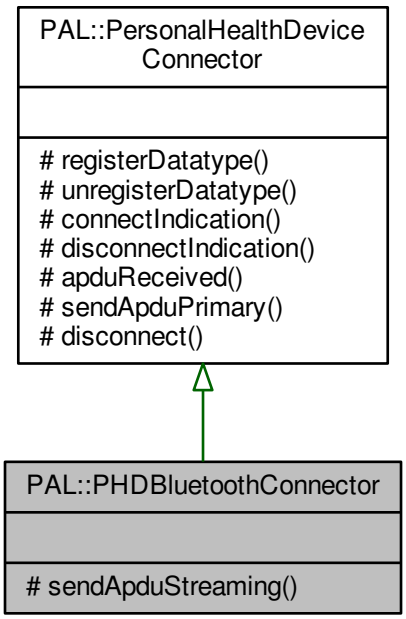
The PersonalHealthDeviceConnector class specialization for communication with bluetooth devices (PHD↩BluetoothDevice).

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDBluetoothConnector:



Collaboration diagram for PAL::PHDBluetoothConnector:



Protected Member Functions

- virtual void sendApduStreaming (std::shared_ptr< VirtualPHD > device, std::shared_ptr< std::vector< uint8_t > > apdu) noexcept=0

  Send a message to this device on a streaming virtual channel.

12.30.1 Detailed Description

This class defines the communication between the bluetooth HDP transport component in the PAL layer and session-layer components.

In addition to the common communication primitives, bluetooth offers a mode of transport more suited for streaming data. In this mode, data will be transferred faster and/or with less energy consumption (best-effort), but with reduced reliability.

See also

PersonalHealthDeviceConnector
PHDBluetoothDevice

12.30.2 Member Function Documentation

12.30.2.1 virtual void PAL::PHDBluetoothConnector::sendApduStreaming (std::shared_ptr< VirtualPHD > *device,* std::shared_ptr< std::vector< uint8_t > > *apdu*) [protected],[pure virtual],[noexcept]

A protocol-to-PAL message. If the device is currently connected, this method will (attempt to) send a message to the device on a streaming virtual channel. If such a channel is not available, it will attempt the primary virtual channel instead. There will be no indication of whether this attempt was succesful or not. In particular, if this device object is currently not connected, the method invocation will simply be ignored.

Parameters

device	The bluetooth device to send to. The shared pointer must point to a PHDBluetoothDevice, otherwise the method will use the primary virtual channel.
apdu	The buffer of bytes to transmit to the device.

Implemented in PAL::PersonalHealthDeviceHandler.

The documentation for this class was generated from the following file:

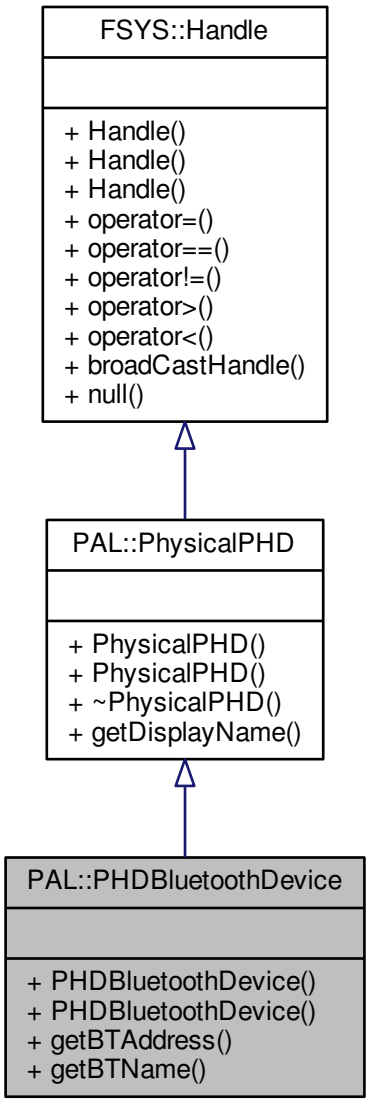
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.31 PAL::PHDBluetoothDevice Class Reference

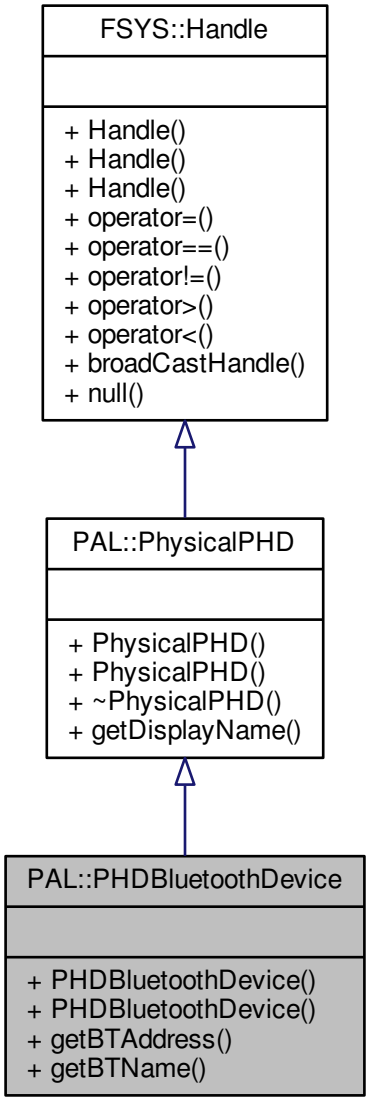
Bluetooth specialization of the PersonalHealthDevice.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDBluetoothDevice:



Collaboration diagram for PAL::PHDBluetoothDevice:



Public Member Functions

- PHDBluetoothDevice ()

  Default constructor.
- PHDBluetoothDevice (FSYS::Handle &h)

  Constructor used to clone the handle.
- virtual std::string getBTAddress () const noexcept=0

  The bluetooth address of this device.
- virtual std::string getBTName () const noexcept=0

  The bluetooth name of this device.

Additional Inherited Members

12.31.1 Detailed Description

This abstract class extends the PersonalHealthDevice with properties of the underlying bluetooth Health Device Profile (HDP) device, namely the bluetooth address and name.

Todo Describe how the HDP standard maps to this interface

See also

PersonalHealthDevice
PHDBluetoothConnector

12.31.2 Constructor & Destructor Documentation

12.31.2.1 PAL::PHDBluetoothDevice::PHDBluetoothDevice ( ) [inline]

The default constructor will create a PersonalHealthDevice object with a new unique FSYS::Handle.

12.31.2.2 PAL::PHDBluetoothDevice::PHDBluetoothDevice (FSYS::Handle & *h*) [inline]

This alternative constructor will create a PersonalHealthDevice object with a cloned FSYS::Handle.

12.31.3 Member Function Documentation

12.31.3.1 virtual std::string PAL::PHDBluetoothDevice::getBTAddress ( ) const [pure virtual],[noexcept]

This getter will return the bluetooth address (a.k.a. MAC address) of this device. The format will be 6 double-digit hex values separated by colons, e.g.: "01:23:45:67:89:AB", so the string will always be 17 characters long.

Returns

The bluetooth address of this device.

12.31.3.2 virtual std::string PAL::PHDBluetoothDevice::getBTName ( ) const [pure virtual],[noexcept]

This getter will return the bluetooth name of this device. This may or may not be identical to the name returned by getDisplayName().

Returns

The bluetooth name of this device.

The documentation for this class was generated from the following file:

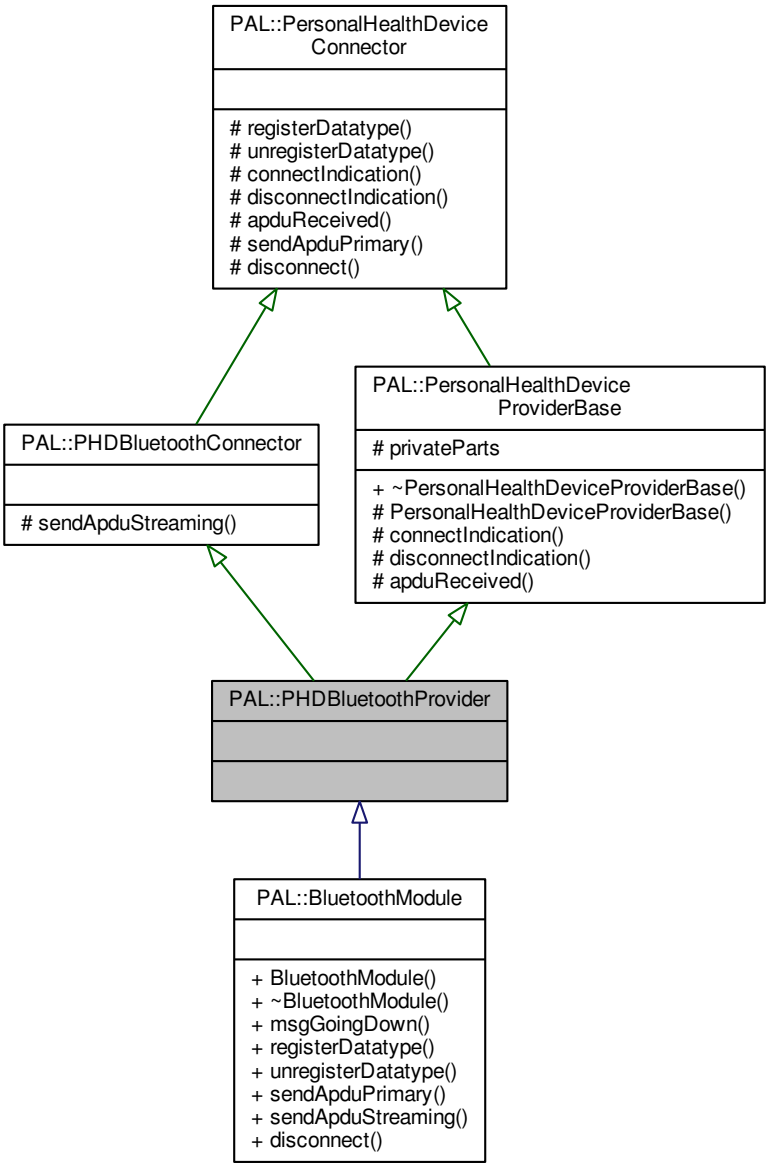
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.32 PAL::PHDBluetoothProvider Class Reference

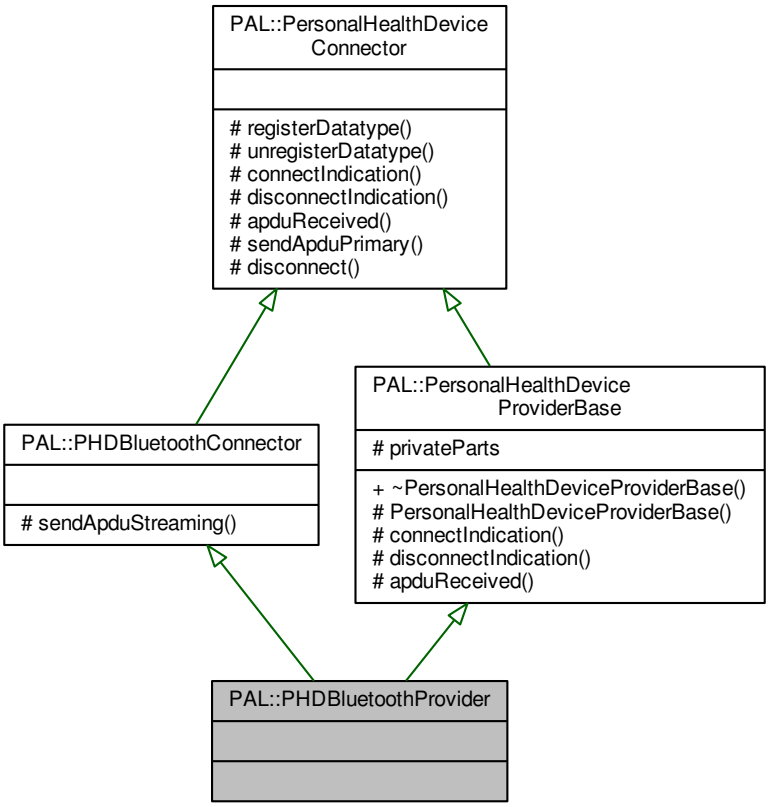
Bluetooth specialisation of the PersonalHealthDeviceProviderBase.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDBluetoothProvider:



Collaboration diagram for PAL::PHDBluetoothProvider:



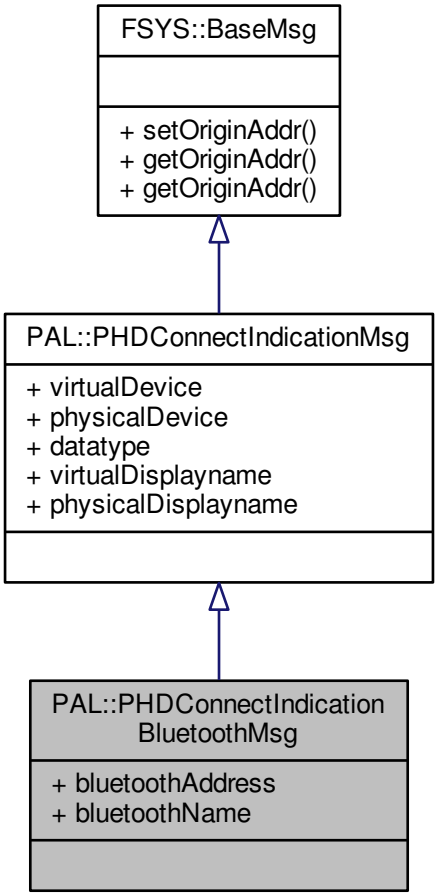
Additional Inherited Members

The documentation for this class was generated from the following file:

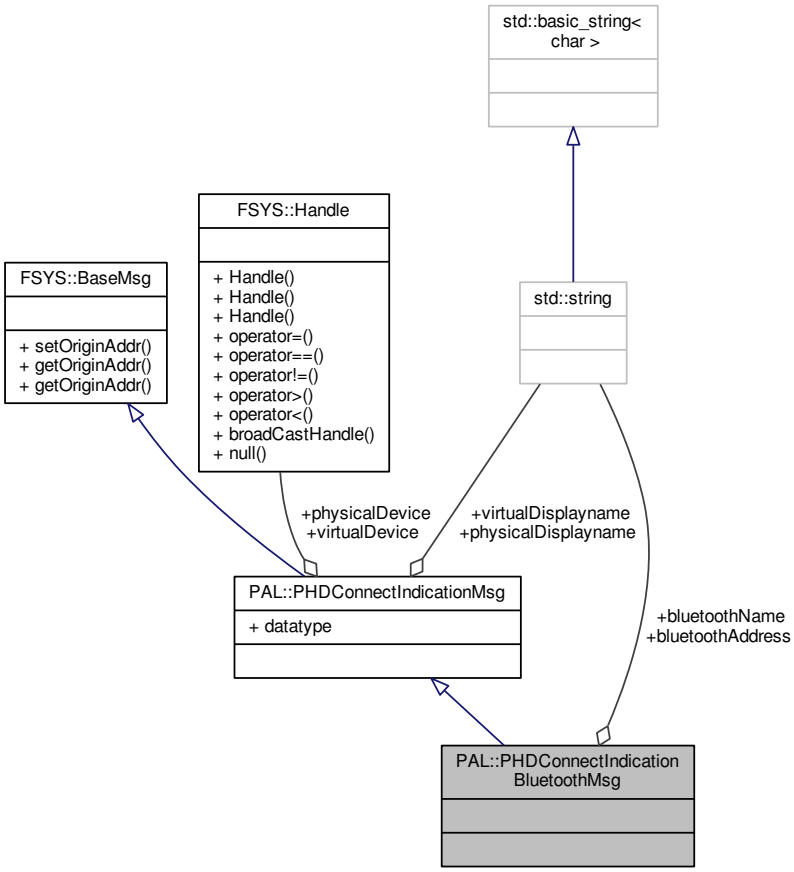
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.33 PAL::PHDConnectIndicationBluetoothMsg Class Reference

Inheritance diagram for PAL::PHDConnectIndicationBluetoothMsg:



Collaboration diagram for PAL::PHDConnectIndicationBluetoothMsg:



Public Attributes

- std::string **bluetoothAddress**
- std::string **bluetoothName**

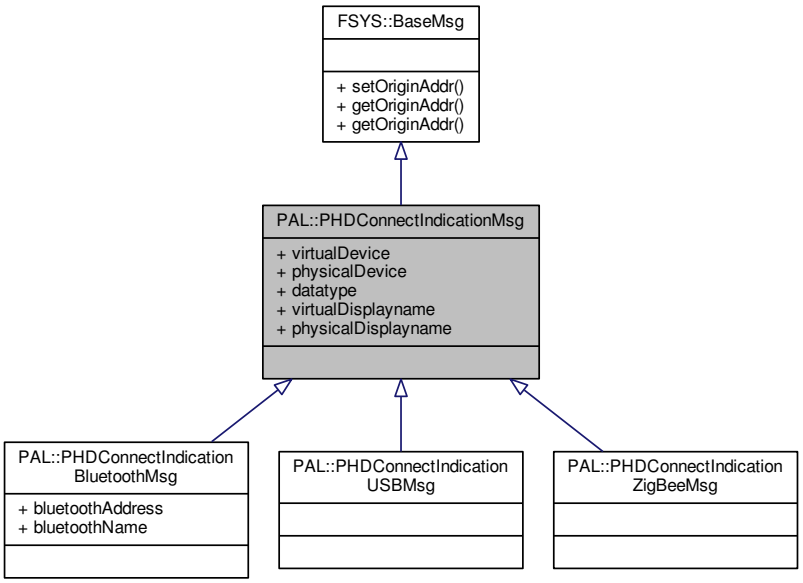
Additional Inherited Members

The documentation for this class was generated from the following file:

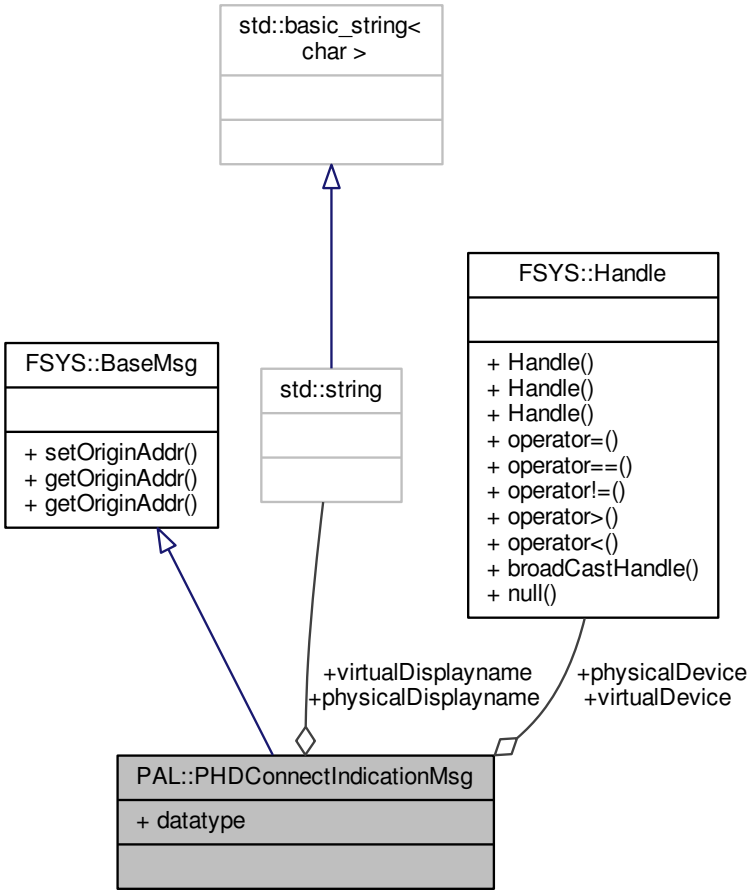
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.34 PAL::PHDConnectIndicationMsg Class Reference

Inheritance diagram for PAL::PHDConnectIndicationMsg:



Collaboration diagram for PAL::PHDConnectIndicationMsg:



Public Attributes

- FSYS::Handle **virtualDevice**
- FSYS::Handle **physicalDevice**
- uint16_t **datatype**
- std::string **virtualDisplayname**
- std::string **physicalDisplayname**

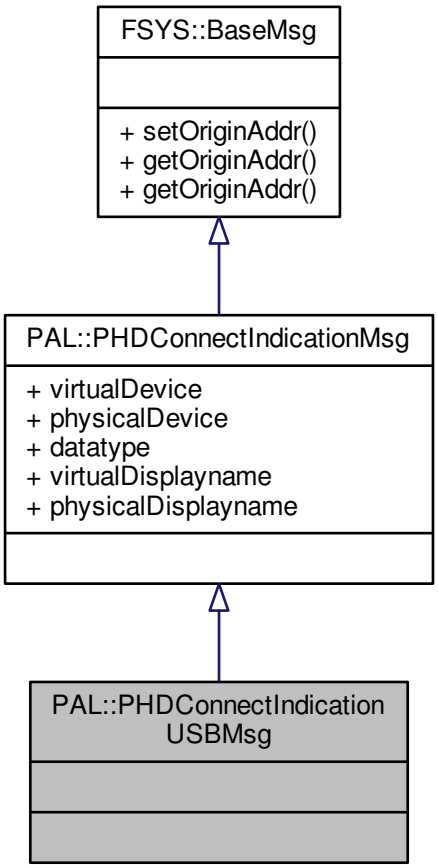
Additional Inherited Members

The documentation for this class was generated from the following file:

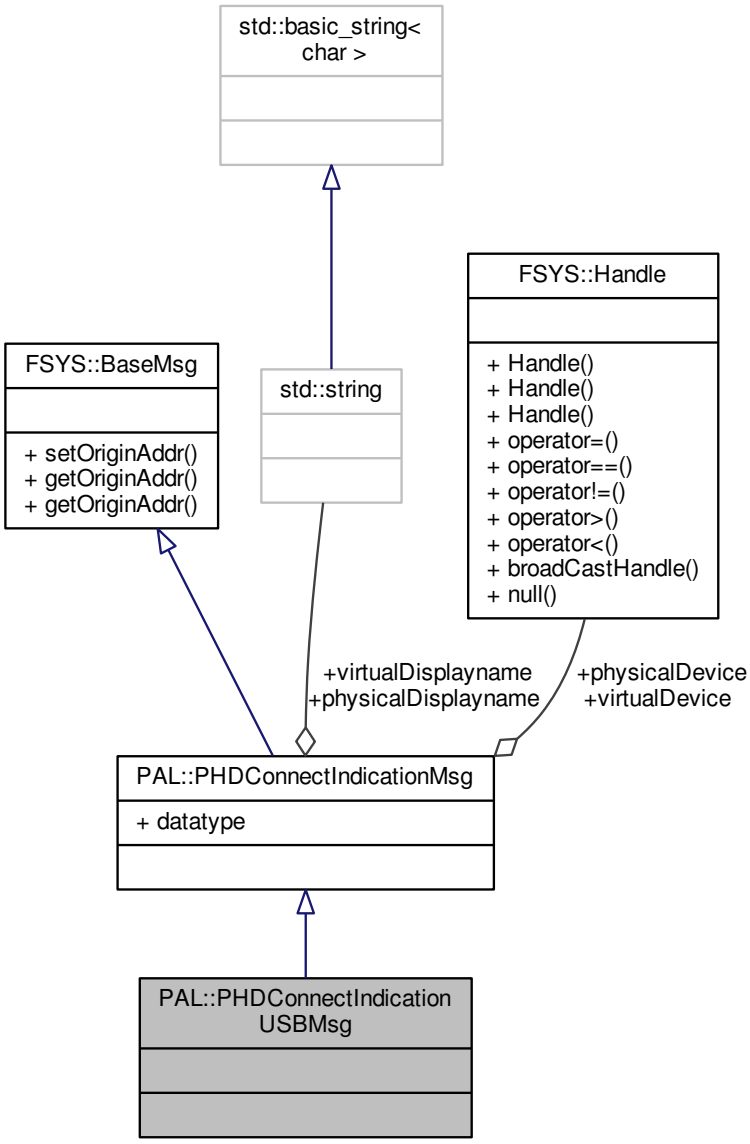
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.35 PAL::PHDConnectIndicationUSBMsg Class Reference

Inheritance diagram for PAL::PHDConnectIndicationUSBMsg:



Collaboration diagram for PAL::PHDConnectIndicationUSBMsg:



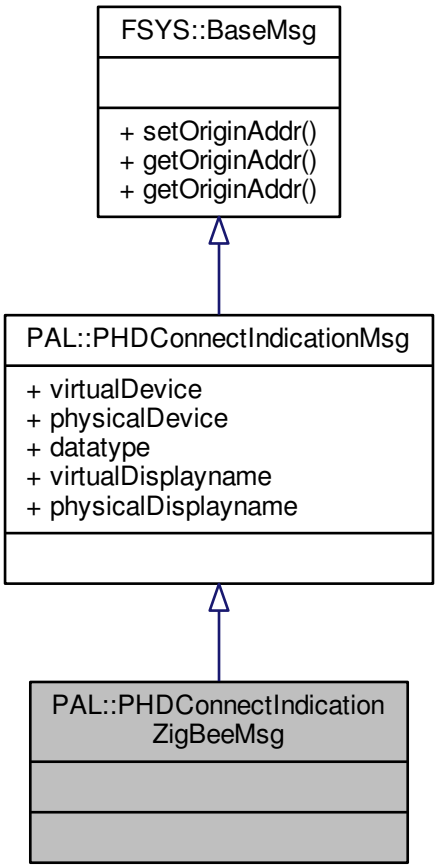
Additional Inherited Members

The documentation for this class was generated from the following file:

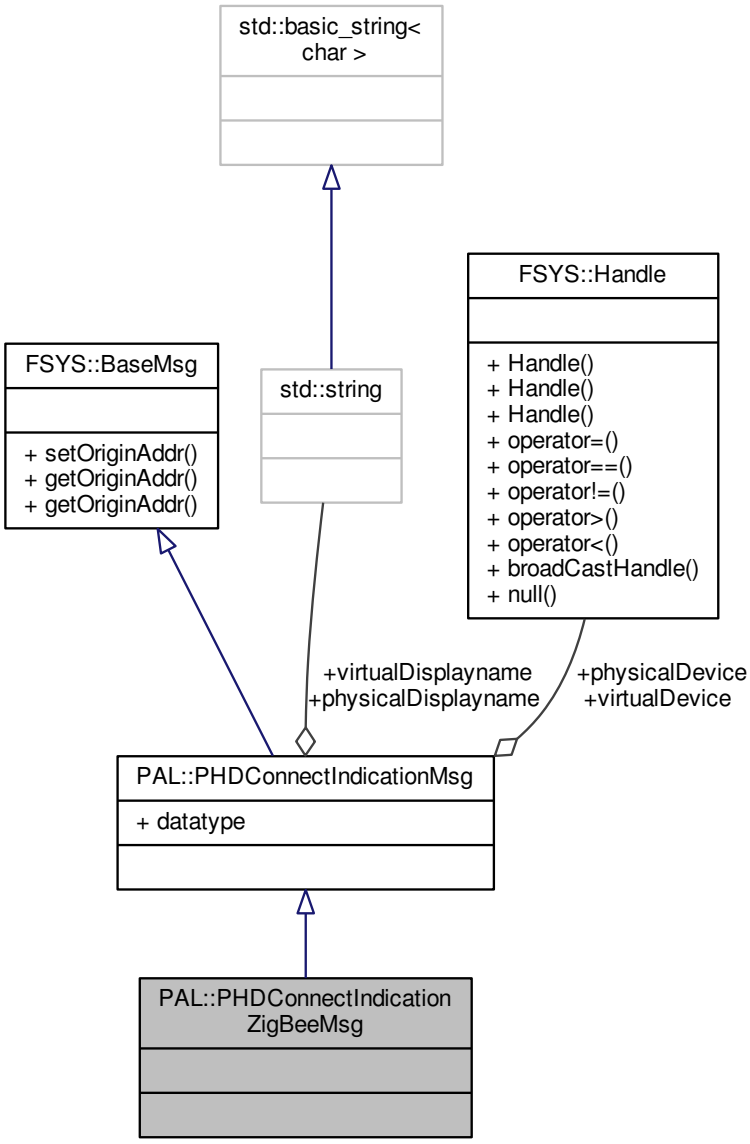
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.36 PAL::PHDConnectIndicationZigBeeMsg Class Reference

Inheritance diagram for PAL::PHDConnectIndicationZigBeeMsg:



Collaboration diagram for PAL::PHDConnectIndicationZigBeeMsg:



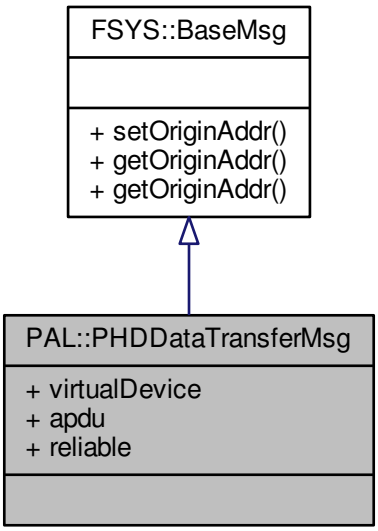
Additional Inherited Members

The documentation for this class was generated from the following file:

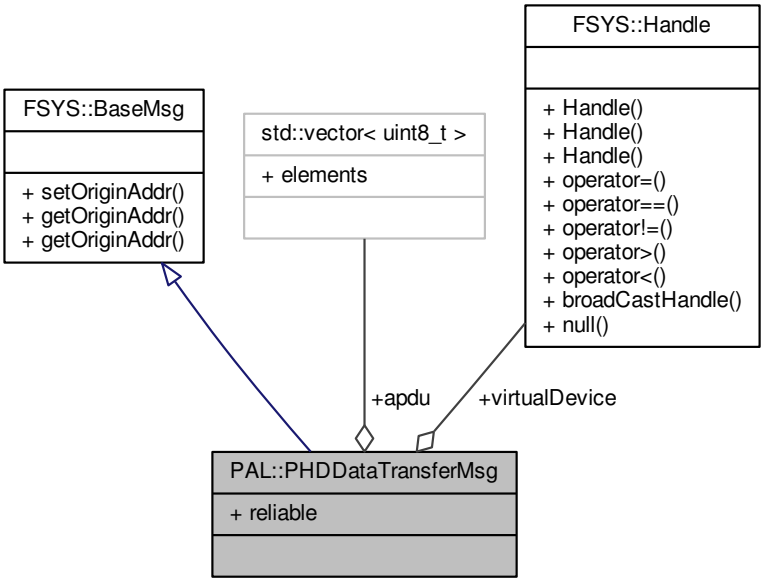
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.37 PAL::PHDDataTransferMsg Class Reference

Inheritance diagram for PAL::PHDDataTransferMsg:



Collaboration diagram for PAL::PHDDataTransferMsg:



Public Attributes

- FSYS::Handle **virtualDevice**
- std::vector< uint8_t > **apdu**
- bool **reliable**

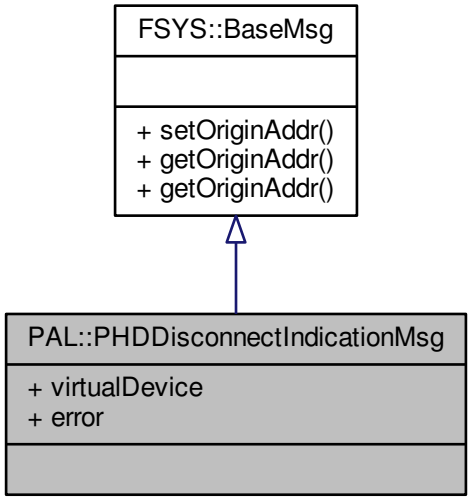
Additional Inherited Members

The documentation for this class was generated from the following file:

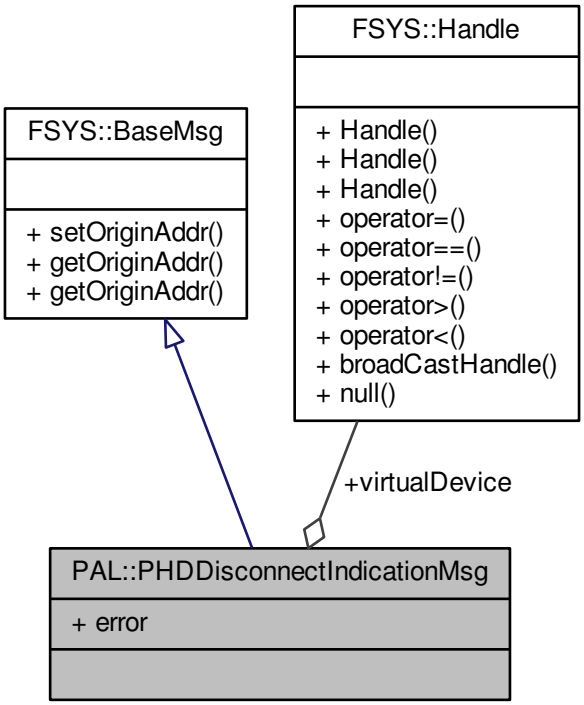
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.38 PAL::PHDDisconnectIndicationMsg Class Reference

Inheritance diagram for PAL::PHDDisconnectIndicationMsg:



Collaboration diagram for PAL::PHDDisconnectIndicationMsg:



Public Attributes

- FSYS::Handle **virtualDevice**
- errortype **error**

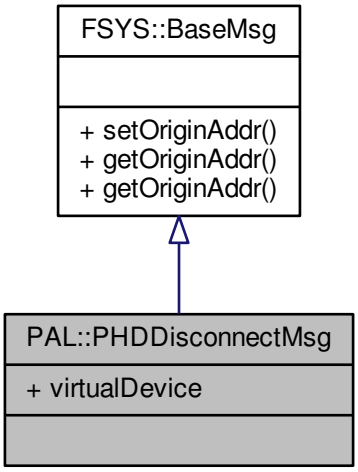
Additional Inherited Members

The documentation for this class was generated from the following file:

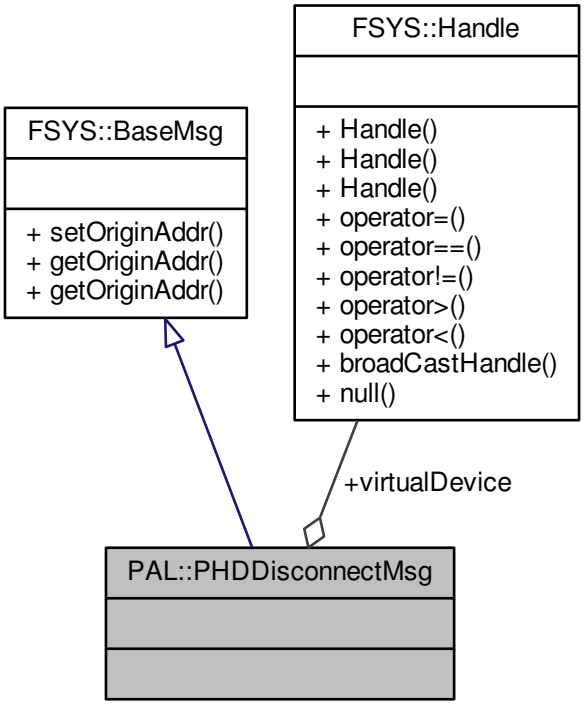
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.39 PAL::PHDDisconnectMsg Class Reference

Inheritance diagram for PAL::PHDDisconnectMsg:



Collaboration diagram for PAL::PHDDisconnectMsg:



Public Attributes

- FSYS::Handle **virtualDevice**

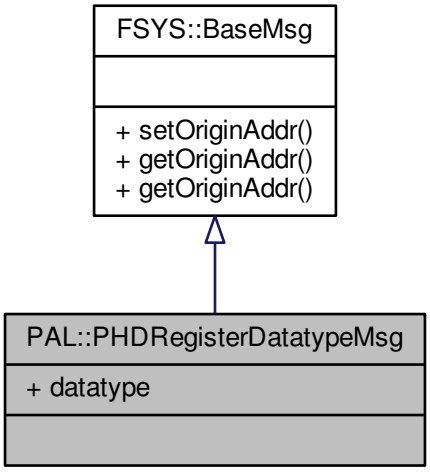
Additional Inherited Members

The documentation for this class was generated from the following file:

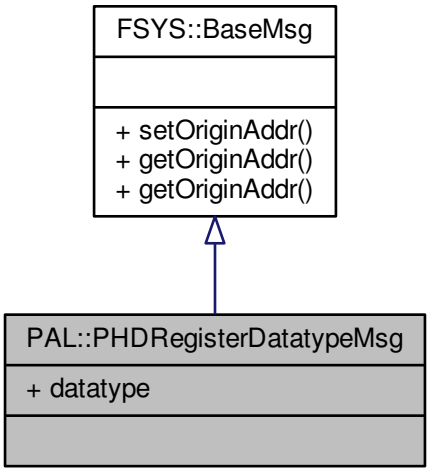
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.40 PAL::PHDRegisterDatatypeMsg Class Reference

Inheritance diagram for PAL::PHDRegisterDatatypeMsg:



Collaboration diagram for PAL::PHDRegisterDatatypeMsg:



Public Attributes

- uint16_t **datatype**

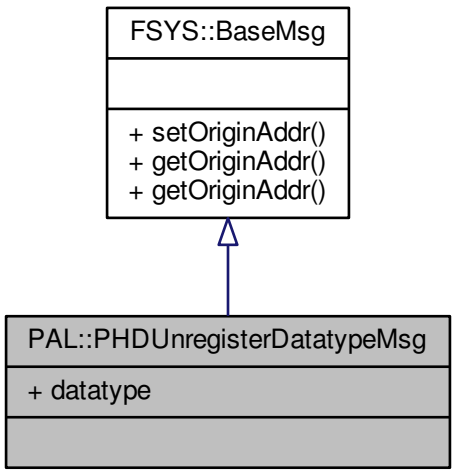
Additional Inherited Members

The documentation for this class was generated from the following file:

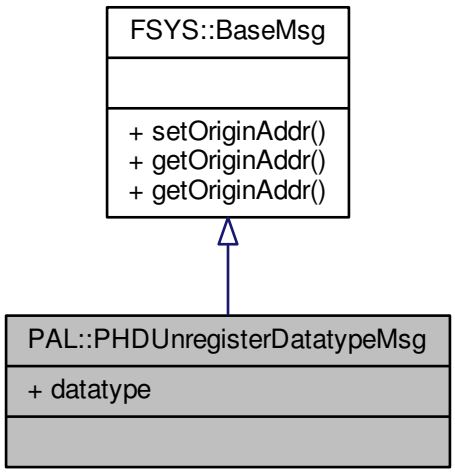
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.41 PAL::PHDUnregisterDatatypeMsg Class Reference

Inheritance diagram for PAL::PHDUnregisterDatatypeMsg:



Collaboration diagram for PAL::PHDUnregisterDatatypeMsg:



Public Attributes

- uint16_t **datatype**

Additional Inherited Members

The documentation for this class was generated from the following file:

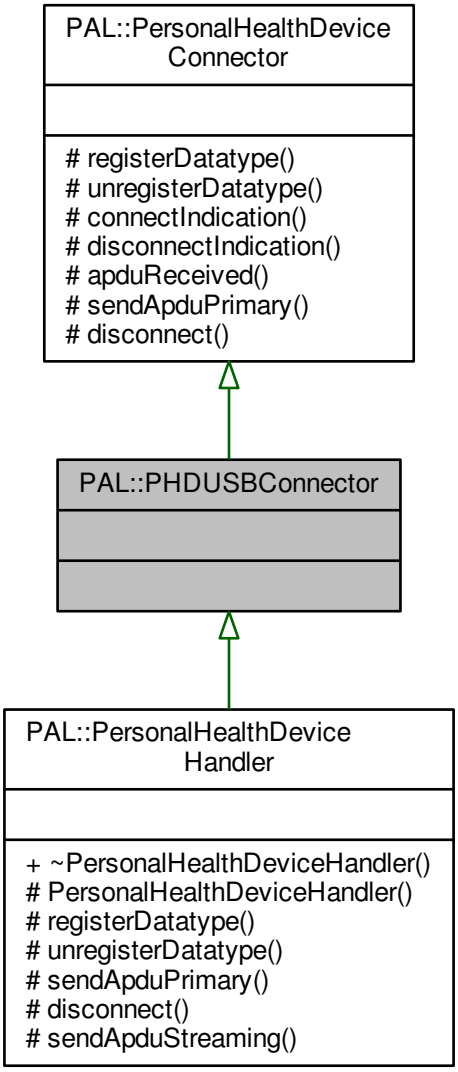
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.42 PAL::PHDUSBConnector Class Reference

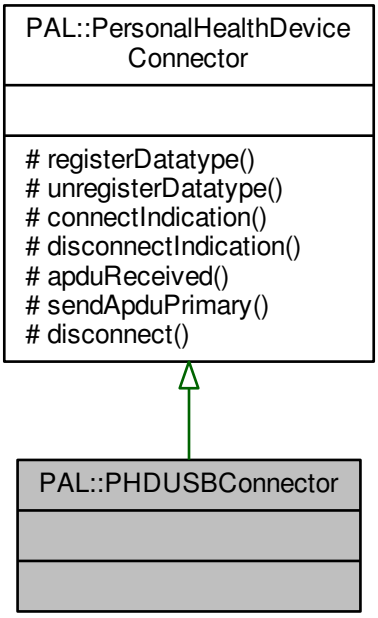
The PersonalHealthDeviceConnector class specialization for communication with USB devices (PHDUSBDevice).

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDUSBConnector:



Collaboration diagram for PAL::PHDUSBConnector:



Additional Inherited Members

12.42.1 Detailed Description

This class defines the communication between the USB Personal Healthcare Device Class (PHDC) component in the PAL layer and session-layer components.

Todo This specialization must be defined when the first USB device is integrated in this framework.

See also

PersonalHealthDeviceConnector
PHDUSBDevice

The documentation for this class was generated from the following file:

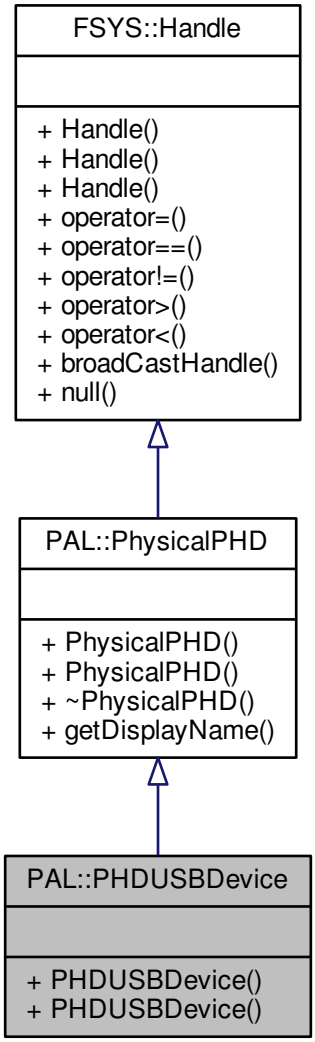
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.43 PAL::PHDUSBDevice Class Reference

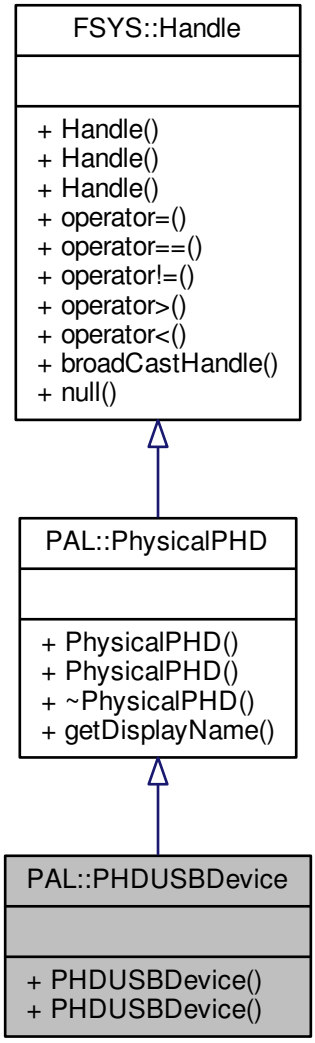
A USB specialization of the PersonalHealthDevice.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDUSBDevice:



Collaboration diagram for PAL::PHDUSBDevice:



Public Member Functions

- **PHDUSBDevice** (FSYS::Handle &h)

Additional Inherited Members

12.43.1 Detailed Description

This abstract class extends the PersonalHealthDevice with properties of the underlying USB Personal Healthcare Device Class (PHDC) device.

Todo This specialization must be defined when the first USB device is integrated in this framework.

The documentation for this class was generated from the following file:

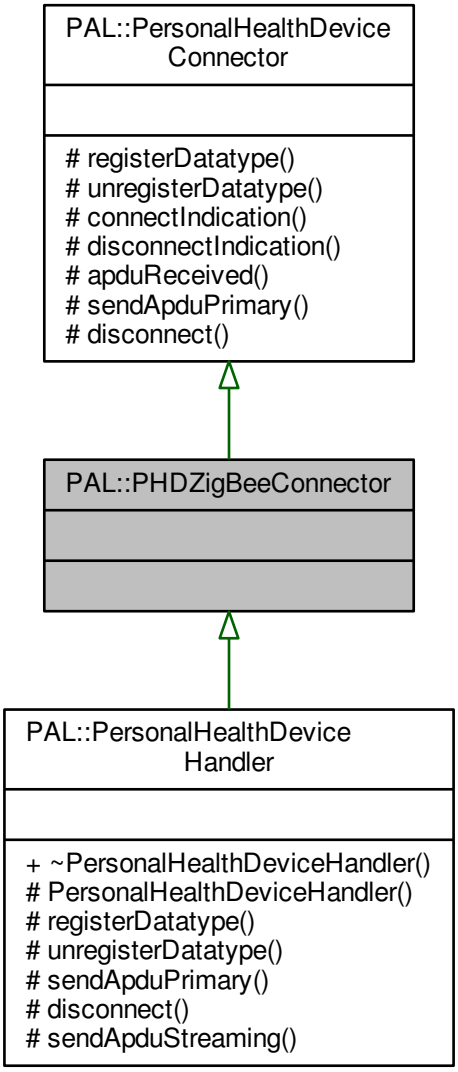
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.44 PAL::PHDZigBeeConnector Class Reference

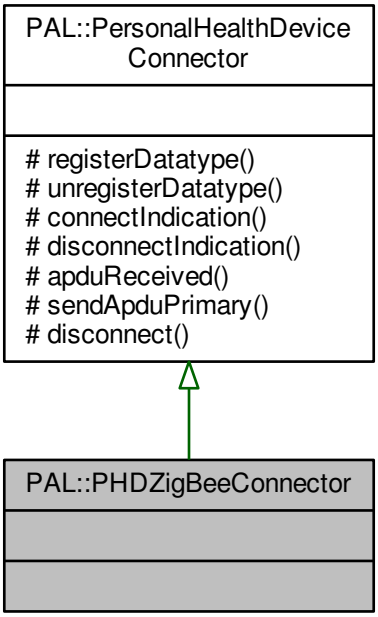
The PersonalHealthDeviceConnector class specialization for communication with ZigBee devices (PHDZigBee↩Device).

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDZigBeeConnector:



Collaboration diagram for PAL::PHDZigBeeConnector:



Additional Inherited Members

12.44.1 Detailed Description

This class defines the communication between the ZigBee Health Care transport component in the PAL layer and session-layer components.

Todo This specialization must be defined when the first ZigBee device is integrated in this framework.

See also

PersonalHealthDeviceConnector
PHDZigBeeDevice

The documentation for this class was generated from the following file:

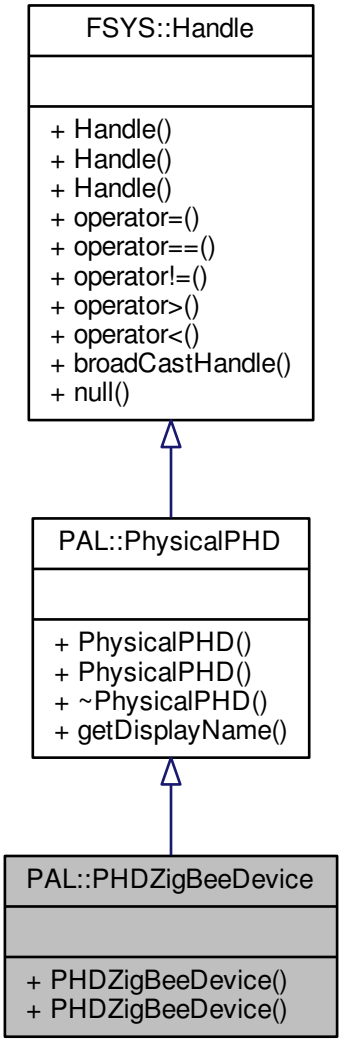
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.45 PAL::PHDZigBeeDevice Class Reference

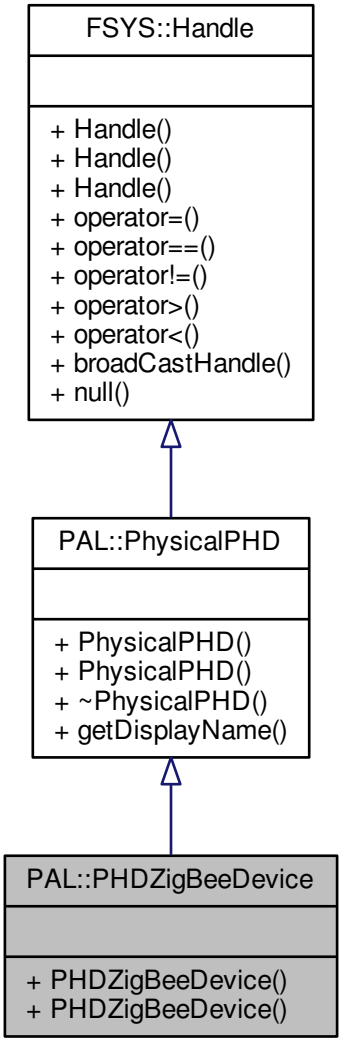
A ZigBee specialization of the PersonalHealthDevice.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::PHDZigBeeDevice:



Collaboration diagram for PAL::PHDZigBeeDevice:



Public Member Functions

- **PHDZigBeeDevice** (FSYS::Handle &h)

Additional Inherited Members

12.45.1 Detailed Description

This abstract class extends the PersonalHealthDevice with properties of the underlying ZigBee Health Care (ZHC) device.

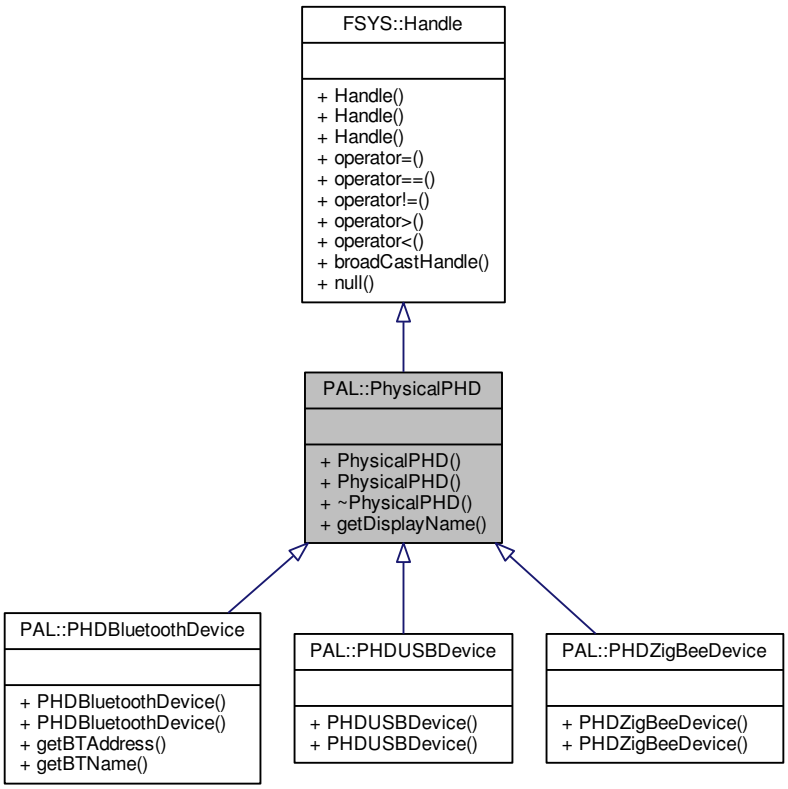
Todo This specialization must be defined when the first ZigBee device is integrated in this framework.

The documentation for this class was generated from the following file:

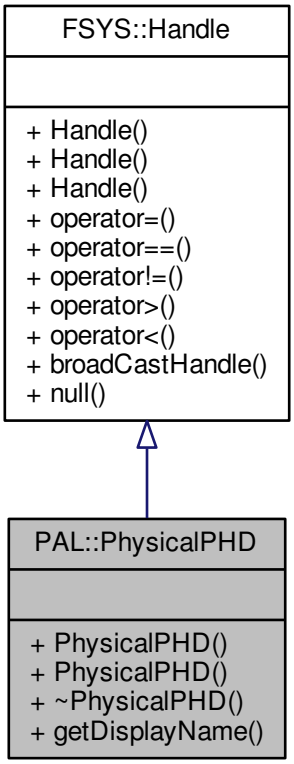
- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

12.46 PAL::PhysicalPHD Class Reference

Inheritance diagram for PAL::PhysicalPHD:



Collaboration diagram for PAL::PhysicalPHD:



Public Member Functions

- PhysicalPHD ()

  Default constructor.
- PhysicalPHD (FSYS::Handle &h)

  Constructor used to clone the handle.
- virtual ~PhysicalPHD ()

  Virtual destructor.
- virtual std::string getDisplayName () const noexcept=0

  A user-friendly name of the device.

Additional Inherited Members

12.46.1 Constructor & Destructor Documentation

12.46.1.1 PAL::PhysicalPHD::PhysicalPHD ( ) [inline]

The default constructor will create a PhysicalPHD object with a new unique FSYS::Handle.

12.46.1.2 PAL::PhysicalPHD::PhysicalPHD (FSYS::Handle & *h*) [inline]

This alternative constructor will create a PhysicalPHD object with a cloned FSYS::Handle.

12.46.2 Member Function Documentation

12.46.2.1 virtual std::string PAL::PhysicalPHD::getDisplayName ( ) const [pure virtual],[noexcept]

This getter will return a user-friendly (or at least user-recognizable) name of the physical device, suitable for use in the user interface. Implementing classes should enforce consistent naming across other PAL layer interfaces (e.g. Bluetooth pairing, ZigBee network monitors and the like).

Bluetooth devices may (should?) grab the *Service Name* field of the SDP record; while USB devices may (should?) grab the name from the device descriptor's *iProduct* field.

Returns

The device name to be used in user interfaces.

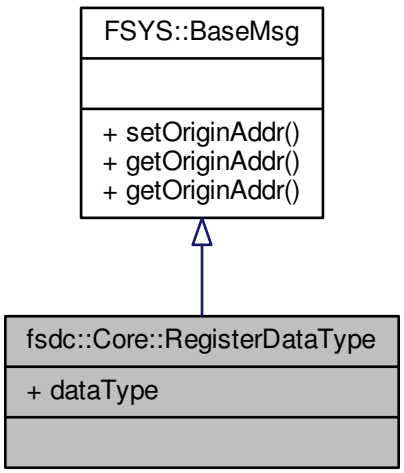
The documentation for this class was generated from the following file:

- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

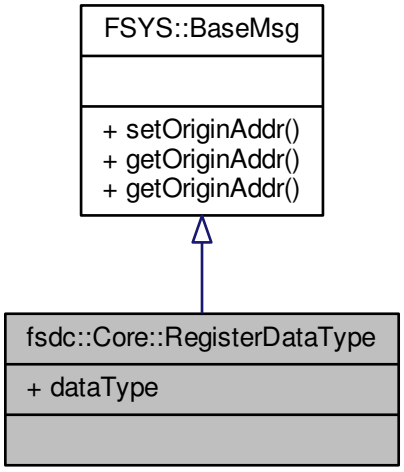
12.47 fsdc::Core::RegisterDataType Class Reference

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

Inheritance diagram for fsdc::Core::RegisterDataType:



Collaboration diagram for fsdc::Core::RegisterDataType:



Public Attributes

- uint16_t **dataType**

Additional Inherited Members

12.47.1 Detailed Description

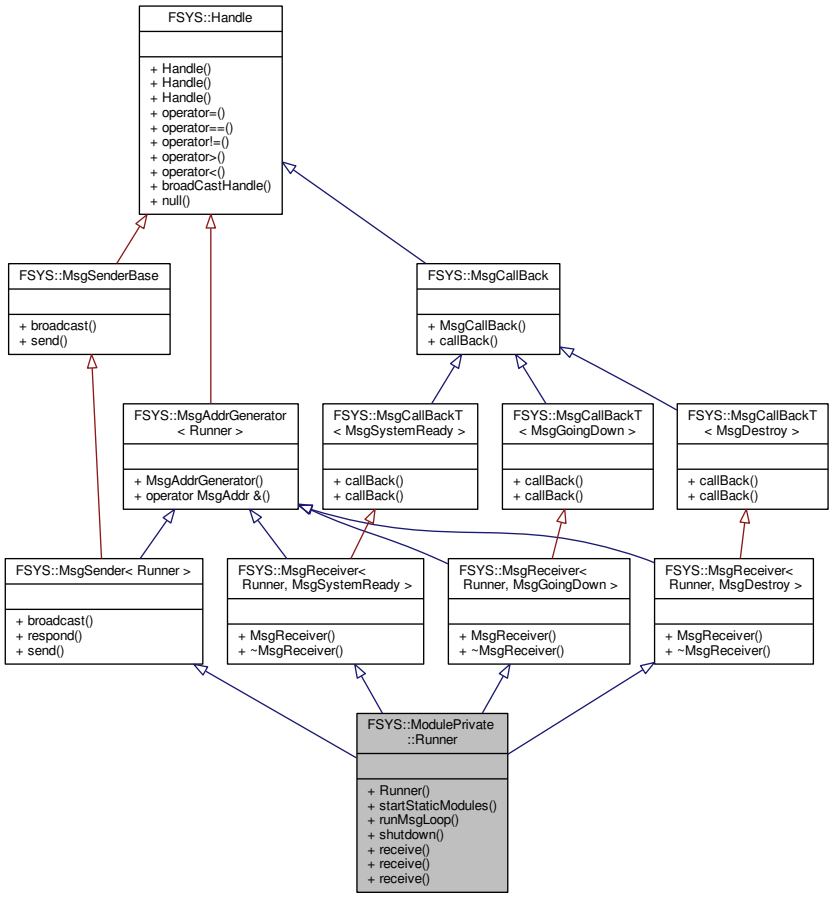
Todo Fixme: Document me

The documentation for this class was generated from the following file:

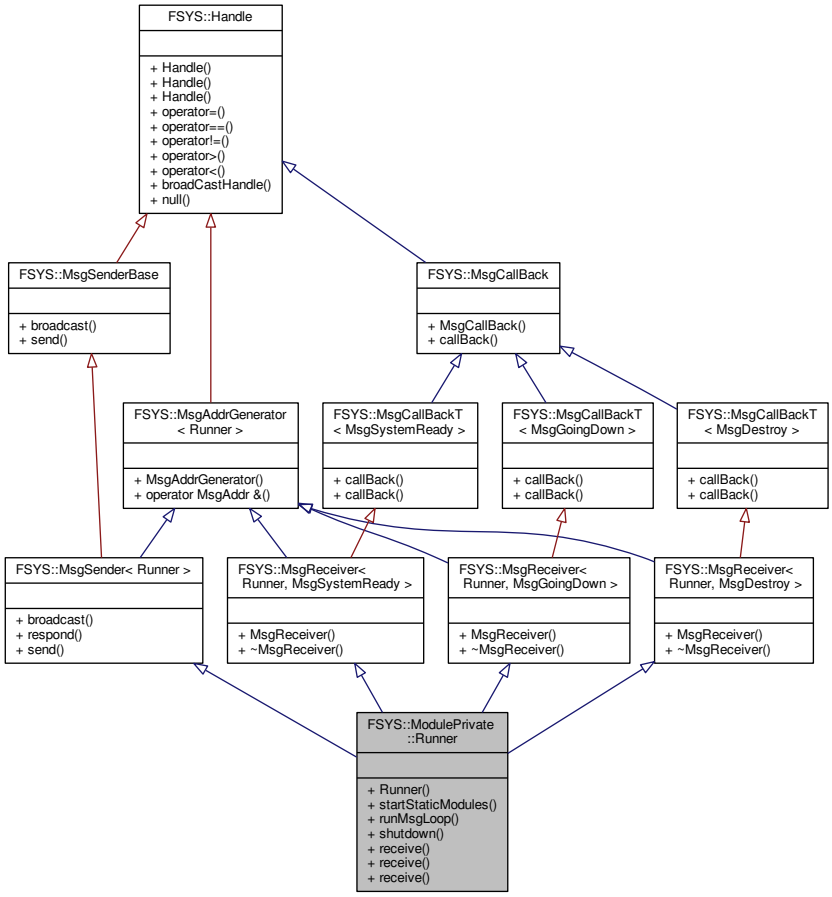
- Interface/4SDC/core.h (This file was last changed: 2015-03-10 16:58:11 +0100, by Mike Kristoffersen)

12.48 FSYS::ModulePrivate::Runner Class Reference

Inheritance diagram for FSYS::ModulePrivate::Runner:



Collaboration diagram for FSYS::ModulePrivate::Runner:



Public Member Functions

- Runner (std::string groupName, std::initializer_list< LauncherBase * > moduleList)

  Constructor for the Runner class.
- void startStaticModules (void)

  Function to start all static modules.
- void runMsgLoop (void)

  Calling this function will run the message loop for this thread.
- void shutdown (void)

  Function that stops and delete all modules owned by this runner.
- void receive (MsgSystemReady)

  Receiver function for the MsgSystemReady broadcast.
- void receive (MsgGoingDown)

  Receiver function for the MsgGoingDown broadcast.
- void receive (MsgDestroy)

  Receiver function for the MsgDestroy broadcast.

12.48.1 Constructor & Destructor Documentation

12.48.1.1 FSYS::ModulePrivate::Runner::Runner (std::string *groupName*, std::initializer_list< LauncherBase * > *moduleList*)

This class keeps track of the user classes in a given group, by keeping a list of launchers for the user classes.

Ownership of the items in the module list is transfered to the Runner class by calling this function, and they will be deleted with delete.

Parameters

groupName	The name for the group that is being initialised.
moduleList	A list of the launchers for modules that belog to the group.

12.48.2 Member Function Documentation

12.48.2.1 void FSYS::ModulePrivate::Runner::receive (MsgSystemReady)

The runner will call the msgSystemReady an all modules when this message is received.

12.48.2.2 void FSYS::ModulePrivate::Runner::receive (MsgGoingDown)

The runner will call msgGoingDown on all modules when this message is received.

12.48.2.3 void FSYS::ModulePrivate::Runner::receive (MsgDestroy)

After this message is received the message loop will be terminated and the runner will destroy all of its modules

12.48.2.4 void FSYS::ModulePrivate::Runner::runMsgLoop (void)

This function will run the message loop for this thread until a MsgDestroy is received by the runner class

12.48.2.5 void FSYS::ModulePrivate::Runner::shutdown (void)

Calling this function will stop and delete all modules from this runner

12.48.2.6 void FSYS::ModulePrivate::Runner::startStaticModules (void)

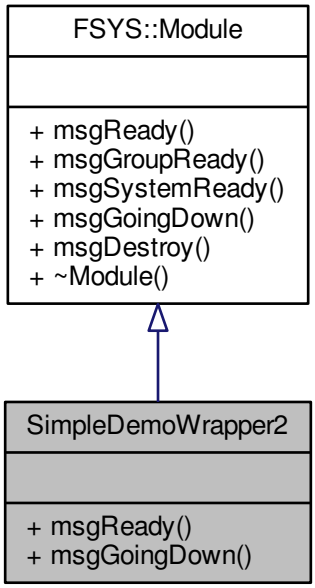
Calling this function will initialize and start all static modules

The documentation for this class was generated from the following file:

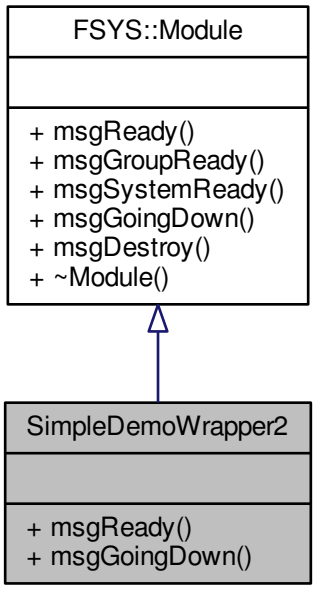
- Interface/FSYS/modulerunner.h (This file was last changed: 2015-03-04 00:14:43 +0100, by Mike Kristoffersen)

12.49 SimpleDemoWrapper2 Class Reference

Inheritance diagram for SimpleDemoWrapper2:



Collaboration diagram for SimpleDemoWrapper2:



Public Member Functions

- void **msgReady** (void) override
- void **msgGoingDown** (void) override

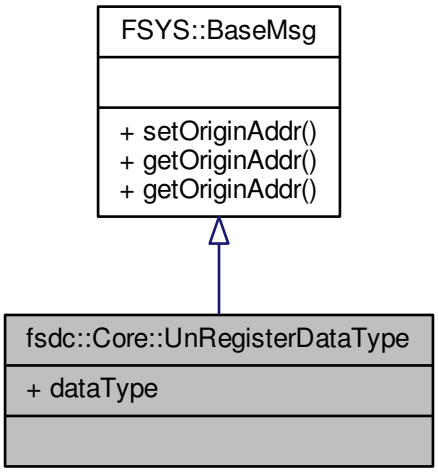
The documentation for this class was generated from the following file:

- Interface/4SDC/simpledemowrapper.h (This file was last changed: 2015-03-10 16:58:11 +0100, by Mike Kristoffersen)

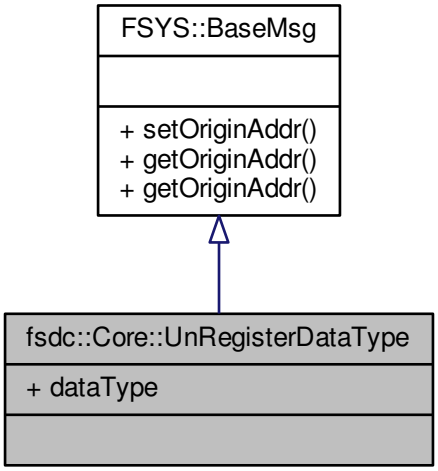
12.50 fsdc::Core::UnRegisterDataType Class Reference

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

Inheritance diagram for fsdc::Core::UnRegisterDataType:



Collaboration diagram for fsdc::Core::UnRegisterDataType:



Public Attributes

- uint16_t **dataType**

Additional Inherited Members

12.50.1 Detailed Description

Todo Fixme: Document me

The documentation for this class was generated from the following file:

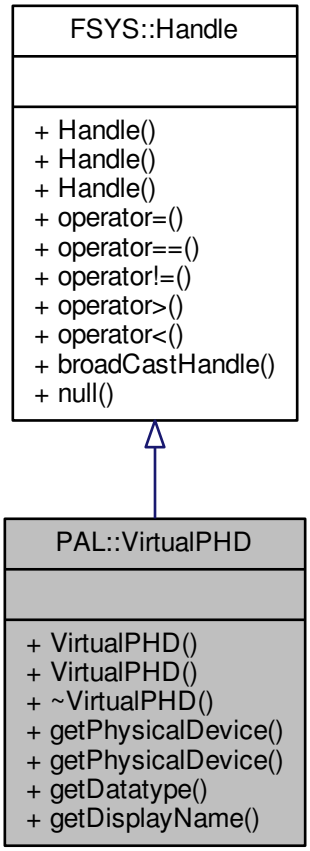
- Interface/4SDC/core.h (This file was last changed: 2015-03-10 16:58:11 +0100, by Mike Kristoffersen)

12.51 PAL::VirtualPHD Class Reference

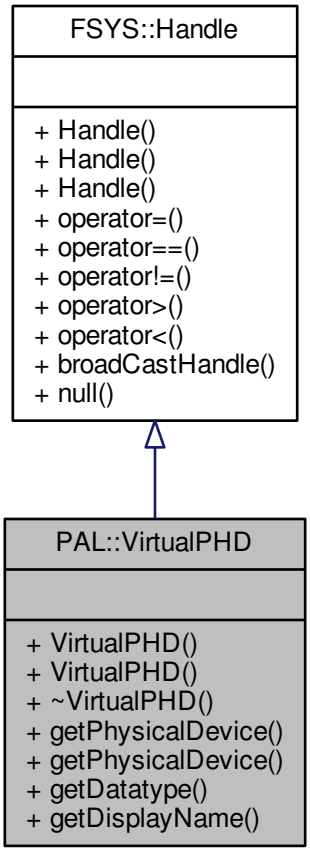
The base interface of a personal health device seen from the perspective of a transport module.

```
#include <personalhealthdevice.h>
```

Inheritance diagram for PAL::VirtualPHD:



Collaboration diagram for PAL::VirtualPHD:



Public Member Functions

- VirtualPHD ()

  Default constructor.
- VirtualPHD (FSYS::Handle &h)

  Constructor used to clone the handle.
- virtual ~VirtualPHD ()

  Virtual destructor.
- virtual PhysicalPHD & **getPhysicalDevice** () noexcept=0
- virtual const PhysicalPHD & **getPhysicalDevice** () const noexcept=0
- virtual uint16_t getDatatype () const noexcept=0

  The IEEE `MDC_DEV_SPEC_PROFILE_` datatype implemented by this device.
- virtual std::string getDisplayName () const noexcept=0

  A user-friendly name of the device.

Additional Inherited Members

12.51.1 Detailed Description

This class is implemented by the transport components in the PAL layer to model a connected device. Each transport type (USB, Bluetooth, ZigBee) implements different subclasses of this class, each adding the characteristics of that particular transport type.

Objects deriving from this class are always created by components in the PAL layer and managed by smart references (std::shared_ptr). The objects are then communicated back and forth between the PAL and session layers using the PersonalHealthDeviceConnector interfaces. Notice that the object appearing at the session layer will be a by-value copy (a proxy) of the original object.

This class derives from FSYS::Handle so that upper layers and applications may compare these device objects (and proxies) with objects/proxies obtained from other PAL layer interfaces (e.g. Bluetooth pairing, ZigBee network monitors and the like). Hence, two (different) proxy-objects representing the same device will always turn out to be equal when compared (==) as they will be clones of the same FSYS::Handle.

12.51.2 Constructor & Destructor Documentation

12.51.2.1 PAL::VirtualPHD::VirtualPHD ( ) [inline]

The default constructor will create a PersonalHealthDevice object with a new unique FSYS::Handle.

12.51.2.2 PAL::VirtualPHD::VirtualPHD (FSYS::Handle & *h*) [inline]

This alternative constructor will create a PersonalHealthDevice object with a cloned FSYS::Handle.

12.51.3 Member Function Documentation

12.51.3.1 virtual uint16_t PAL::VirtualPHD::getDatatype ( ) const [pure virtual],[noexcept]

This getter will return the datatype implemented by this device. The datatype is a 16-bit unsigned integer – one of the `MDC_DEV_SPEC_PROFILE_` codes belonging to the `MDC_PART_INFRA` partition.

Returns

The datatype implemented by this device.

12.51.3.2 virtual std::string PAL::VirtualPHD::getDisplayName ( ) const [pure virtual],[noexcept]

This getter will return a user-friendly (or at least user-recognizable) name of the virtual device, suitable for use in the user interface. Implementing classes should enforce consistent naming across other PAL layer interfaces (e.g. Bluetooth pairing, ZigBee network monitors and the like).

Bluetooth devices may (should?) grab the name from the optional *MDEP Description* field of the SDP record, if available; while USB devices may (should?) grab the name from the interface descriptor's *iInterface* field. If no name is available (including the case when the PAL component is not allowed to read these names), this name should be set to the empty string, and the user interface should be able to generate a default text based on the getDatatype() value.

Returns

The device name to be used in user interfaces, or "".

The documentation for this class was generated from the following file:

- Interface/PAL/personalhealthdevice.h (This file was last changed: 2015-03-09 13:25:53 +0100, by Jacob Andersen)

Chapter 13

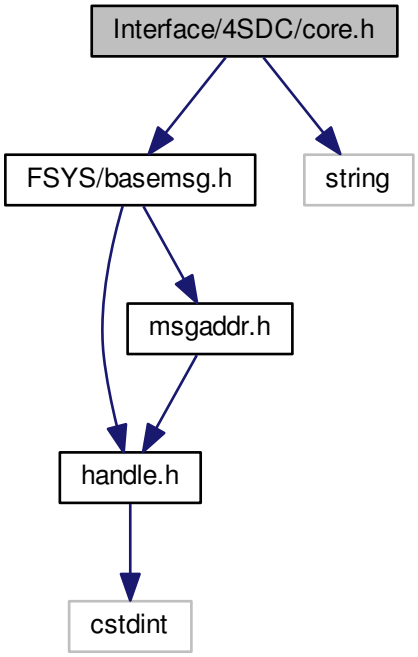
File Documentation

13.1 Interface/4SDC/core.h File Reference

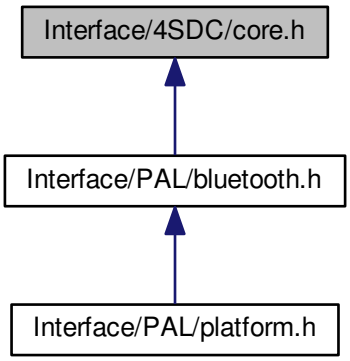
Interface file for the 4SDC.

```
#include "FSYS/basemsg.h"
#include <string>
```

Include dependency graph for core.h:



This graph shows which files directly or indirectly include this file:



Classes

- class fsdc::Core

  4SDC Core class
- class fsdc::Core::RegisterDataType
- class fsdc::Core::UnRegisterDataType
- class fsdc::Core::DataAvailable

13.1.1 Detailed Description

The class contained in this file has the responsibility of controlling the 4SDC stack's main thread, initialising the stack, and running its message loop.

Author

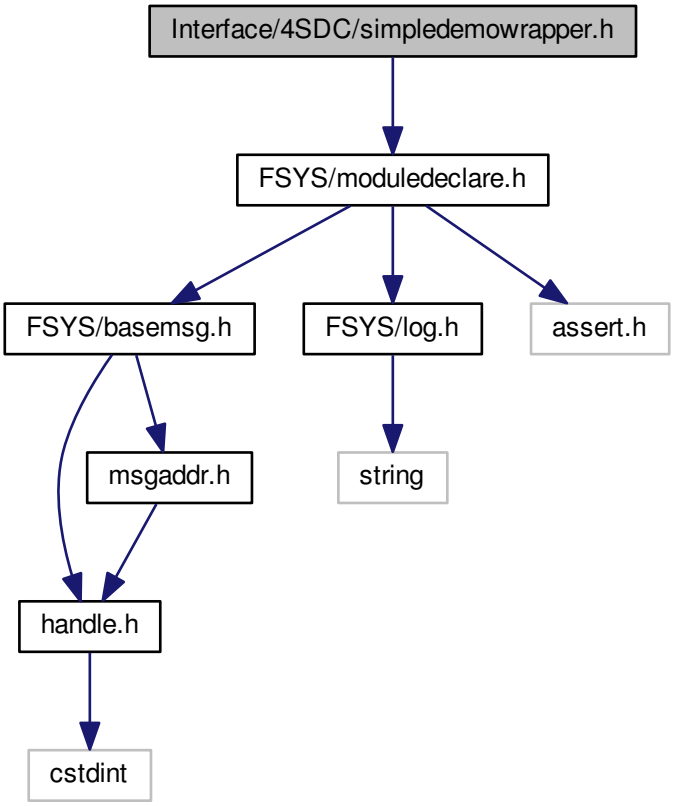
`Mike Kristoffersen`, The Alexandra Institute.

Copyright

13.2 Interface/4SDC/simpledemowrapper.h File Reference

Header file for the wrapper for the demo 4SDC implementation.

`#include "FSYS/moduledeclare.h"`
Include dependency graph for simpledemowrapper.h:



Classes

- class SimpleDemoWrapper2

13.2.1 Detailed Description

Author

Jacob Andersen, The Alexandra Institute.
Mike Kristoffersen, The Alexandra Institute.

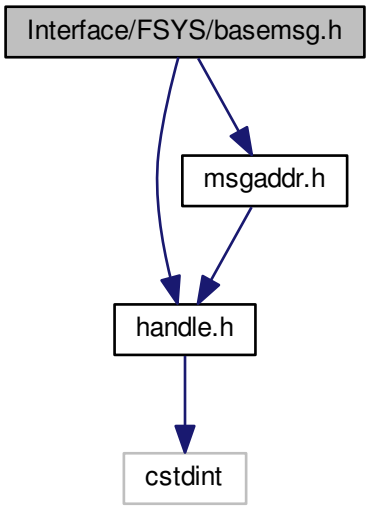
Copyright

13.3 Interface/FSYS/basemsg.h File Reference

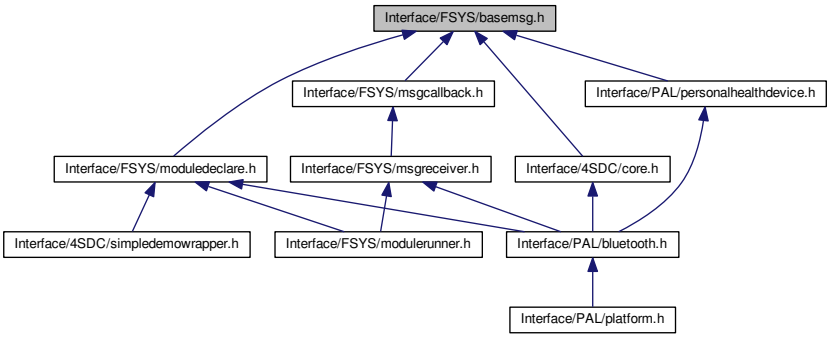
Contains interface declaration for the FSYS::BaseMsg class.

```
#include "handle.h"
#include "msgaddr.h"
```

Include dependency graph for basemsg.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::BaseMsg

  The BaseMsg class is the base class for all messages.

13.3.1 Detailed Description

/see FSYS::BaseMsg

Author

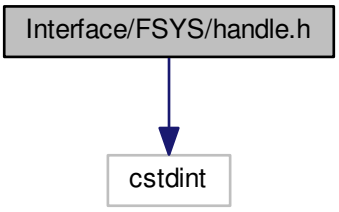
Mike Kristoffersen, The Alexandra Institute.

Copyright

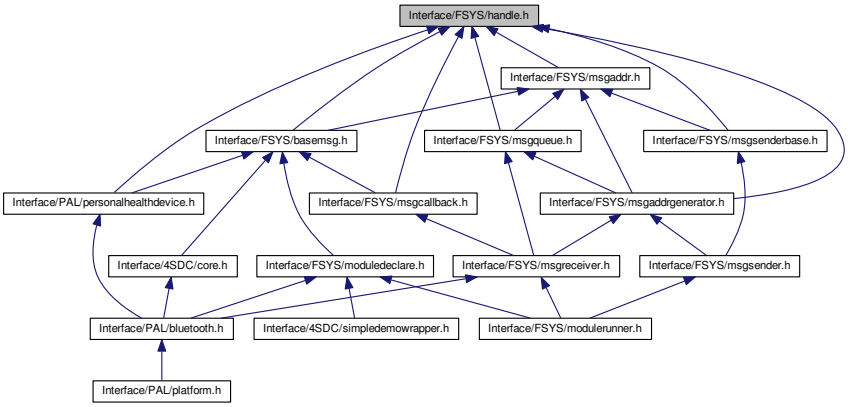
13.4 Interface/FSYS/handle.h File Reference

Contains interface declaration for the FSYS::Handle class.

`#include <cstdint>`
Include dependency graph for handle.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::Handle

  Class that holds and assigns handles.

13.4.1 Detailed Description

/see FSYS::Handle

Author

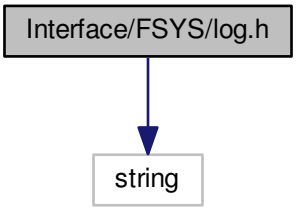
Mike Kristoffersen, The Alexandra Institute.

Copyright

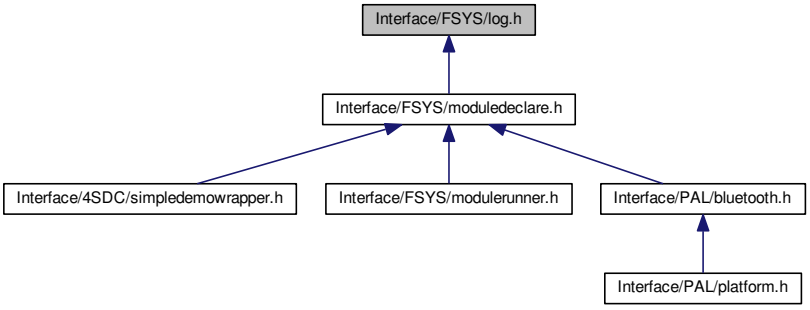
13.5 Interface/FSYS/log.h File Reference

Contains interface declaration for the FSYS::Log class.

`#include <string>`
Include dependency graph for log.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::Log

  Log class used for logging information about program execution.

Macros

- #define **DECLARE_LOG_MODULE**(module) namespace {FSYS::Log eRRORlOG(module);}
- #define **FATAL**(msg) eRRORlOG.fatal(msg, __FILE__, __LINE__);
- #define **ERR**(msg) eRRORlOG.err (msg, __FILE__, __LINE__);
- #define **WARN**(msg) eRRORlOG.warn (msg, __FILE__, __LINE__);
- #define **DEBUG**(msg) eRRORlOG.debug(msg, __FILE__, __LINE__);
- #define **INFO**(msg) eRRORlOG.info (msg, __FILE__, __LINE__);

13.5.1 Detailed Description

/see FSYS::Log

Author

`Mike Kristoffersen`, The Alexandra Institute.

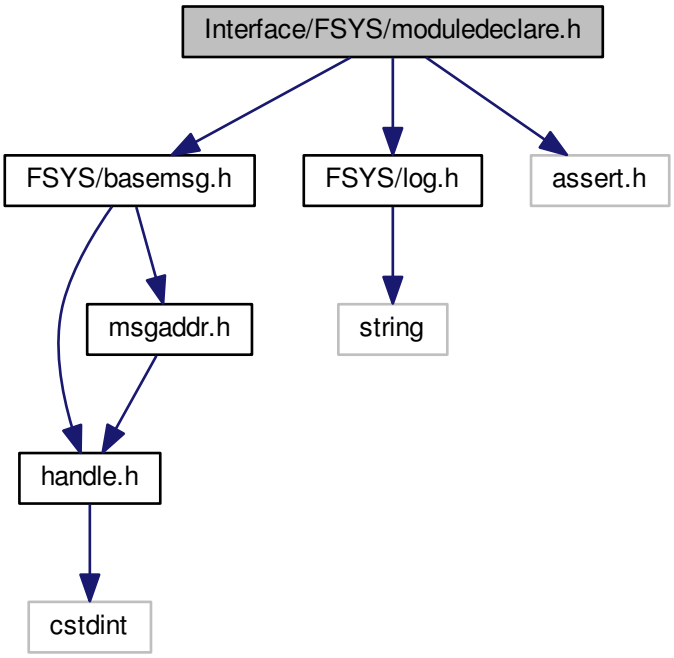
Copyright

13.6 Interface/FSYS/moduledeclare.h File Reference

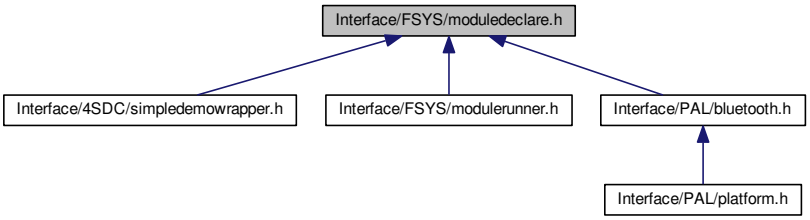
Contains the declaration of the module defining macros and classes.

```
#include "FSYS/basemsg.h"
#include "FSYS/log.h"
#include <assert.h>
```

Include dependency graph for moduledeclare.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::Module

  Base class for all modules.
- class FSYS::MsgSystemReady

  Message broadcasted when the system is initialised.
- class FSYS::MsgGoingDown

  Message broadcasted when the system is being terminated.
- class FSYS::MsgGroupReady

Message broadcasted when all static modules in a group are ready.

- class FSYS::MsgDestroy

  Last message broadcasted to threads.
- class FSYS::MsgAllDestroyed

  Last message broadcasted in system.
- class FSYS::ModulePrivate::LauncherBase

  The LauncherBase class is the non-template class for other Launchers.
- class FSYS::ModulePrivate::Launcher< TypeToLaunch, mt >

  The Launcher class is used for dynamically launching the user clases.
- class FSYS::ModulePrivate::LauncherEnd

  Dummy launcher class to mark the end of the launcher list.

13.6.1 Detailed Description

The macros and classes are used for defining the modules in the system.

A module is in this context defined as an object that needs to be instantiated (potentially in its own thread) and isn't instantiated dynamically or as a static member of or in a different object.

A module must inherit from the FSYS::Module class.

A number of different module types exists:

A "Static" module is one that will be created when the program is launched. A static module can't and won't be deleted/destroyed until the program is terminating.

A "Delayed" module will be instantiated when a message of the type given in its declaration macro is broadcasted in the system.

A "Delayed" module can destroy it self by calling its deleteMe() function.

If a "Delayed" module has destroyed it self it will be instantiated again the next time a message of the type given in the template parameter is broadcasted. (Beware of raceconditions here)

A "Dynamic" and "DynamicThread" module will be instantiated each time a message of MsgType is broadcasted in the system. A DynamicThread module will be spawned in its own thread where a dynamic module will be created in the thread defined by its module group.

"Dynamic" and "DynamicThread" modules must destroy them self when they are done, by calling their deleteMe() function, to avoid resource leaking and CPU cycle leaking.

A Module is defined in the module definition file by using the MODULE∗ macros.

MODULE_GROUP(GroupName) MODULE_STATIC(ModuleClassName) MODULE_DELAYED(ModuleClass↩ Name) MODULE_GROUP_END

MODULE_GROUP(GroupName) MODULE_DYNAMIC(ModuleClassName) MODULE_DYNAMIC_THREAD(↩ ModuleClassName) MODULE_GROUP_END

All modules in a group are guaranteed to run in the same thread, modules in different groups, might or might not run in the same thread, depending on the configuration of the platform that the code is run on (With MODULE_D↩ YNAMIC_THREAD as an obvious exception to this rule)

Currently only MODULE_STATIC is implemented

The ModuleX baseclass will implement and call the following virtual functions that can be used by the module to act, depending on where in its life cycle it is, by overriding the virual functions

void msgReady(void)

This function is called when the module can start to use the message system. It is guarantied that this function is called before any messages are received by the module.

void msgGroupReady(void)

This function is called when all the static modules in the group are ready to use the message system (this means that the module can interact through the message system with the other static modules in the group).

void msgSystemReady(void)

This function is called when all static modules in all groups are ready to use the message system (this means that the module can interact through the message system with all other static modules in the system (in all groups))

void msgGoingDown(void)

This function is called to inform the module that the shutdown sequence of the system has been initiated and that it should release all its system resources.

void msgDestroy(void)

After this function is called the module can no longer rely on the message system and should prepare for imminent destruction.

The msgSystemReady() and msgGoingDown() function calls will also be broadcasted as standard messages

FSYS::MsgSystemReady

Broadcasted when all static modules are started. If a module is created after this message is broadcasted, then it won't be able to receive this message, but it will still have its void msgSystemReady(void) function called.

FSYS::MsgGoingDown

Broadcasted when the system is shutting down. If a module is created after this message is broadcasted, then it won't be able to receive this message, but it will still have its void msgGointDown(void) function called.

The following messages are being used internally.

FSYS::MsgGroupReady

Broadcasted by a group when all of its static modules have launched

FSYS::MsgDestroy

Broadcasted when all the threads should terminate

FSYS::MsgAllDestroyed

Broadcasted when all threads are destroyed

Author

Mike Kristoffersen, The Alexandra Institute.

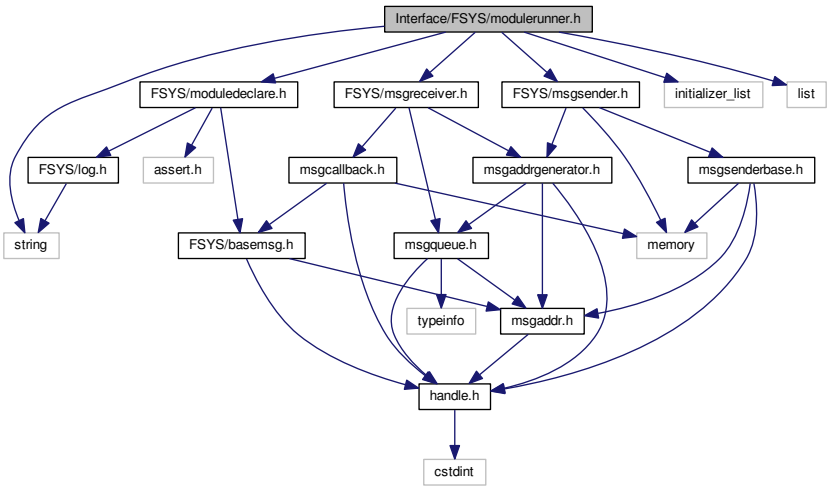
Copyright

13.7 Interface/FSYS/modulerunner.h File Reference

Interface file for the module runner class.

```
#include "FSYS/moduledeclare.h"
#include "FSYS/msgreceiver.h"
#include "FSYS/msgsender.h"
#include <initializer_list>
#include <list>
#include <string>
```

Include dependency graph for modulerunner.h:



Classes

- class FSYS::ModulePrivate::Runner

13.8 Interface/FSYS/modulethreader.h File Reference

Interface file for the module threader class.

Classes

- class FSYS::ModulePrivate::ModuleThreader

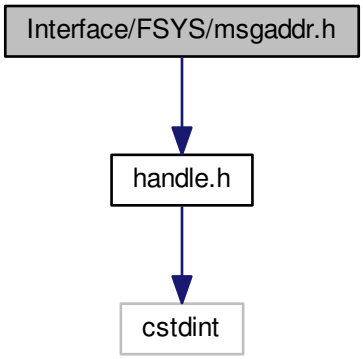
  The ModuleThreader class.

13.9 Interface/FSYS/msgaddr.h File Reference

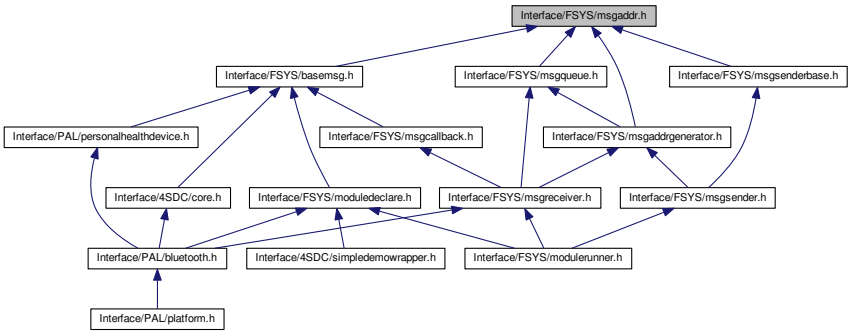
Contains interface for a message address.

```
#include "handle.h"
```

Include dependency graph for msgaddr.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgAddrGenerator< T >

  Constructor to MsgAddrGenerator object.
- class FSYS::MsgAddr

  The MsgAddr class contains a unique address in the message system.

13.9.1 Detailed Description

/see FSYS::MsgAddr

Author

Mike Kristoffersen, The Alexandra Institute.

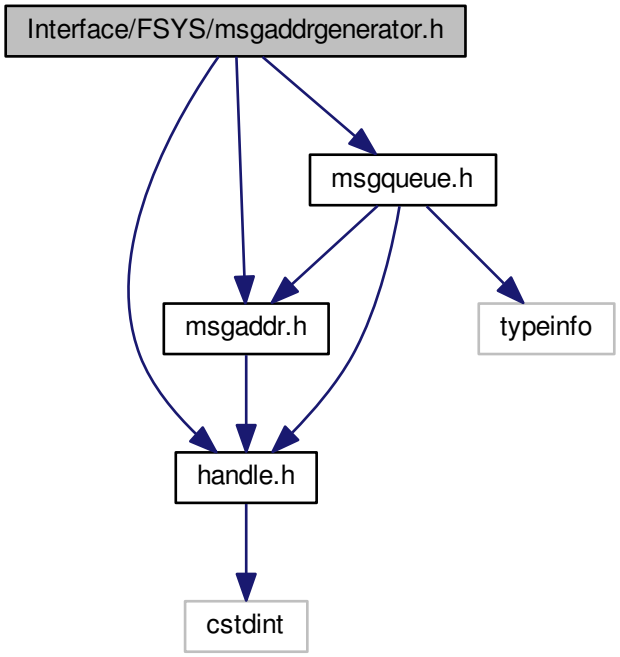
Copyright

13.10 Interface/FSYS/msgaddrgenerator.h File Reference

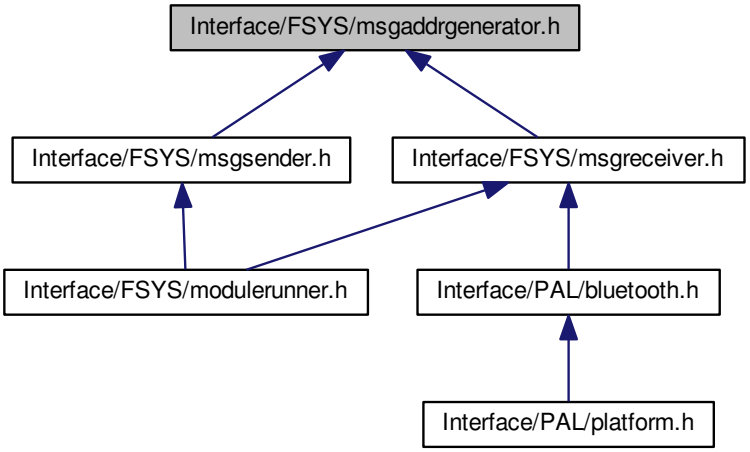
Contains interface for a message address generator class.

```
#include "handle.h"
#include "msgaddr.h"
#include "msgqueue.h"
```

Include dependency graph for msgaddrgenerator.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgAddrGenerator< T >

  Constructor to MsgAddrGenerator object.

13.10.1 Detailed Description

/see FSYS::MsgAddrGenerator

Author

`Mike Kristoffersen`, The Alexandra Institute.

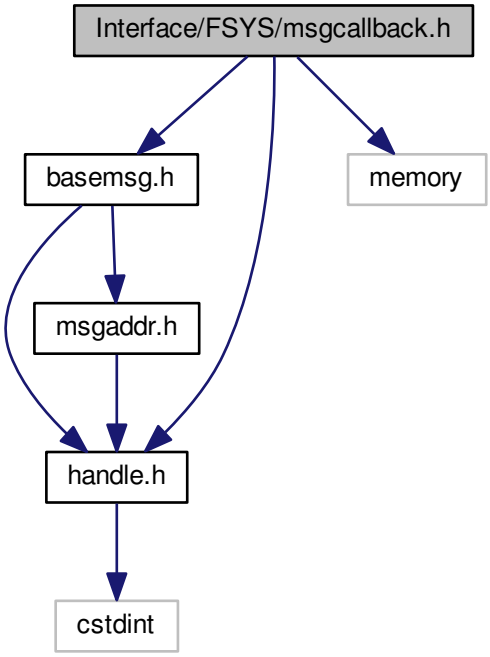
Copyright

13.11 Interface/FSYS/msgcallback.h File Reference

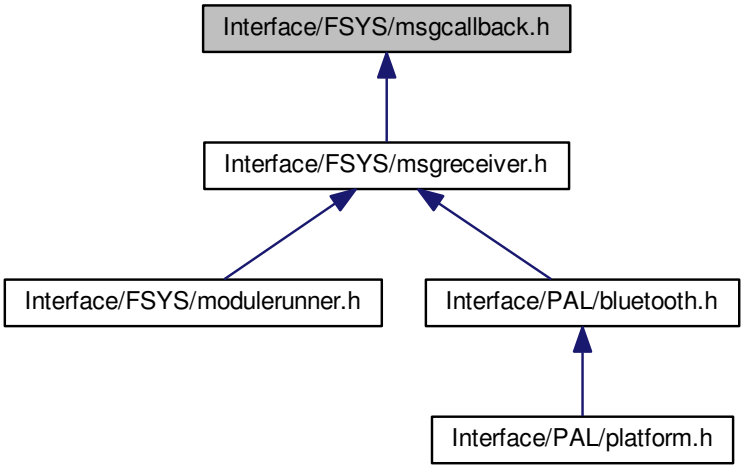
Contains interface declaration for the FSYS::MsgCallBack class.

```
#include "basemsg.h"
#include "handle.h"
#include <memory>
```

Include dependency graph for msgcallback.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgCallBack

  The MsgCallBack class is an internal class to the message system.
- class FSYS::MsgCallBackT< T >

  Template class that inherits from MsgCallBack.

13.11.1 Detailed Description

/see FSYS::MsgCallBack

Author

Mike Kristoffersen, The Alexandra Institute.

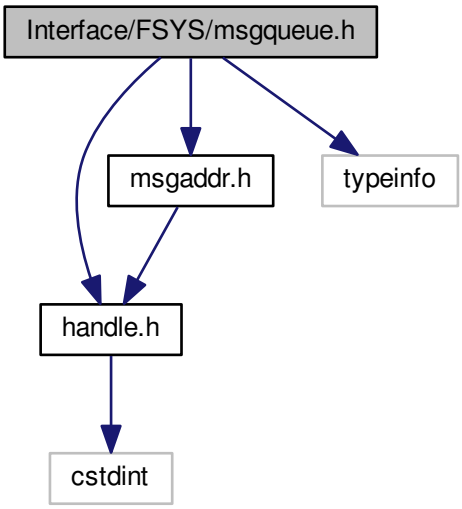
Copyright

13.12 Interface/FSYS/msgqueue.h File Reference

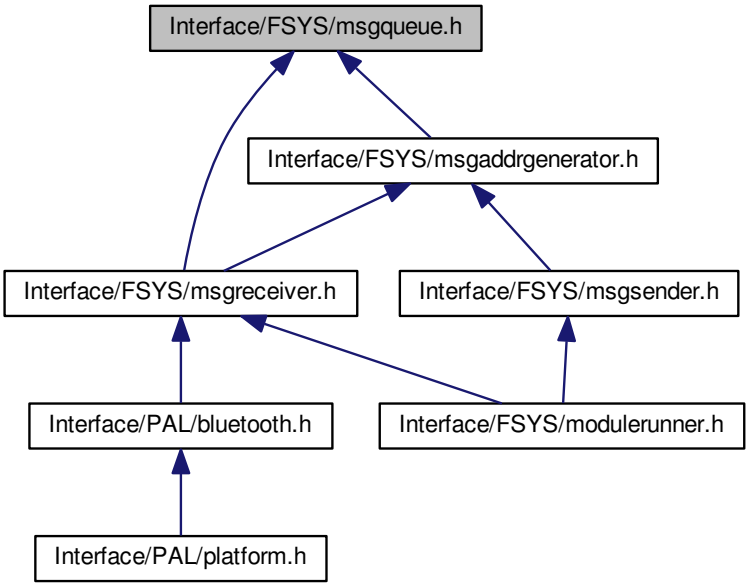
Contains interface declaration for the FSYS::MsgQueue class.

```
#include "handle.h"
#include "msgaddr.h"
#include <typeinfo>
```

Include dependency graph for msgqueue.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgQueue

  This is a wrapper class for the class that does all the hard work.

13.12.1 Detailed Description

/see FSYS::MsgQueue

Author

Mike Kristoffersen, The Alexandra Institute.

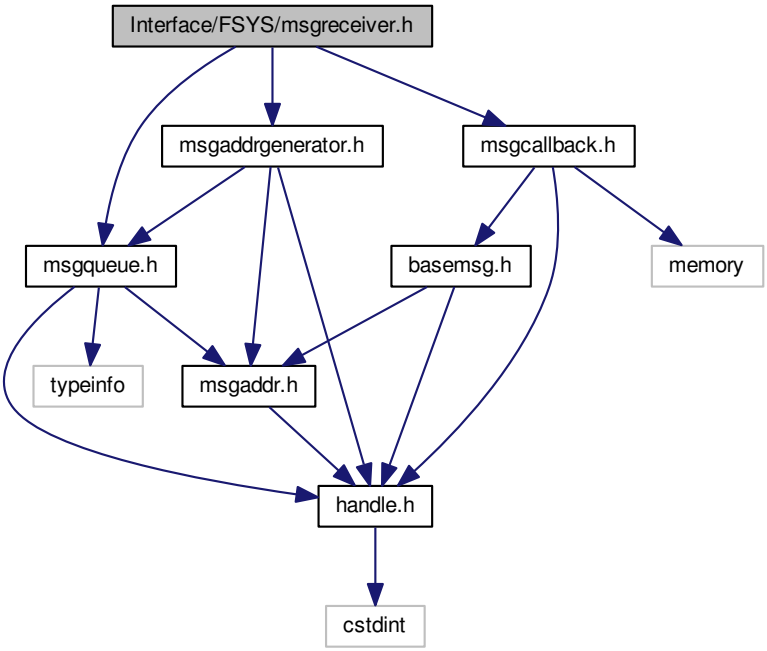
Copyright

13.13 Interface/FSYS/msgreceiver.h File Reference

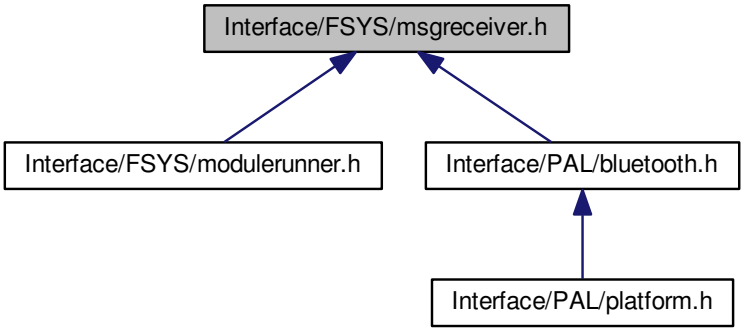
Contains interface declaration for the FSYS::MsgReceiver class.

```
#include "msgaddrgenerator.h"
#include "msgcallback.h"
#include "msgqueue.h"
```

Include dependency graph for msgreceiver.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgReceiver< TReceiverClass, TMsg >

  Template class allowing other classes to receive messages.

13.13.1 Detailed Description

/see FSYS::MsgReceiver

Author

Mike Kristoffersen, The Alexandra Institute.

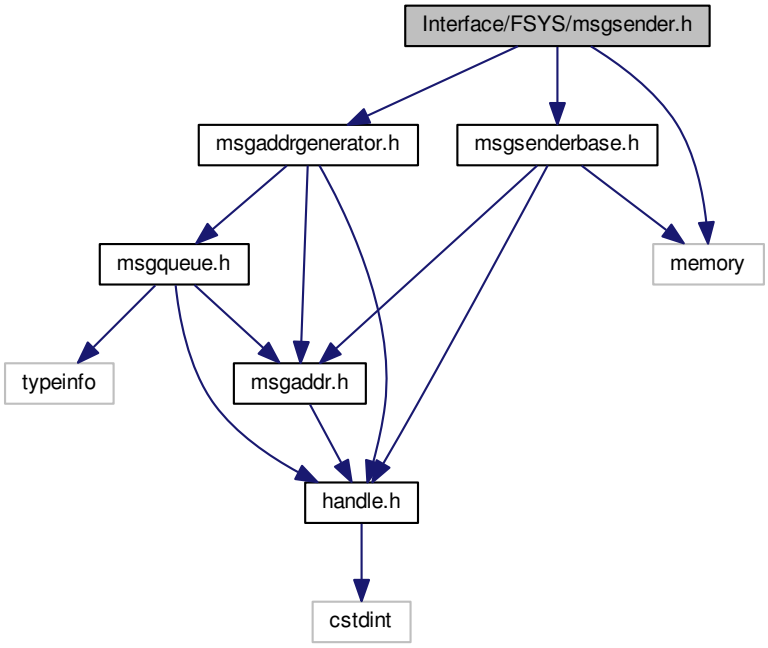
Copyright

13.14 Interface/FSYS/msgsender.h File Reference

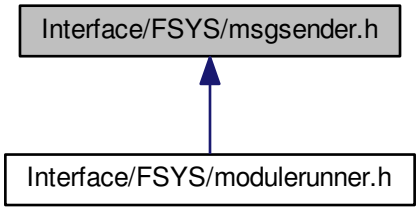
Contains interface declaration for the FSYS::MsgSender class.

```
#include "msgsenderbase.h"
#include "msgaddrgenerator.h"
#include <memory>
```

Include dependency graph for msgsender.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgSender< parentClass >

  Class enabling other classes to send messages.

13.14.1 Detailed Description

/see FSYS::MsgSender

Author

Mike Kristoffersen, The Alexandra Institute.

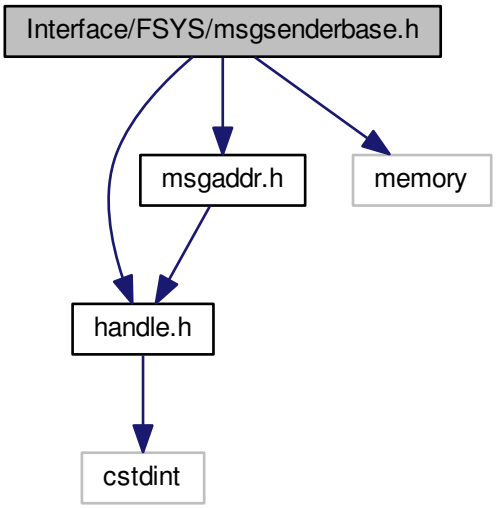
Copyright

13.15 Interface/FSYS/msgsenderbase.h File Reference

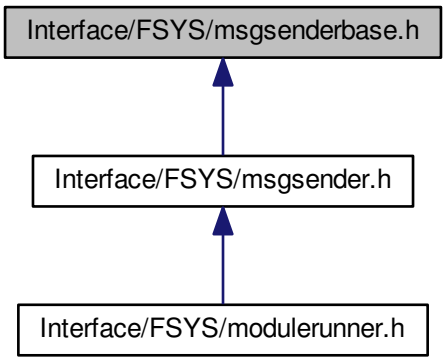
Contains interface declaration for the FSYS::MsgSenderBase class.

```
#include "handle.h"
#include "msgaddr.h"
#include <memory>
```

Include dependency graph for msgsenderbase.h:



This graph shows which files directly or indirectly include this file:



Classes

- class FSYS::MsgSenderBase

  Type neutral message sender class.

13.15.1 Detailed Description

/see FSYS::MsgSenderBase

Author

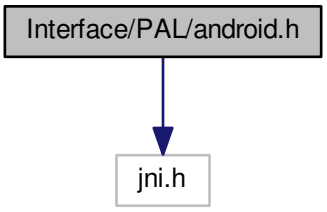
Mike Kristoffersen, The Alexandra Institute.

Copyright

13.16 Interface/PAL/android.h File Reference

Handles to the Java VM running our Android service.

`#include <jni.h>`
Include dependency graph for android.h:



Classes

- class PAL::Android

  Handles to the Java VM running our Android service.

Macros

- #define JNIVER JNI_VERSION_1_6

  The minimal required JNI/JRE version.

13.16.1 Detailed Description

This component bridges the C++ and Java worlds of the Android service by providing the handles into the Android/↩ Java world needed by the C++ components of the PAL layer.

Author

Jacob Andersen, The Alexandra Institute, © 2015

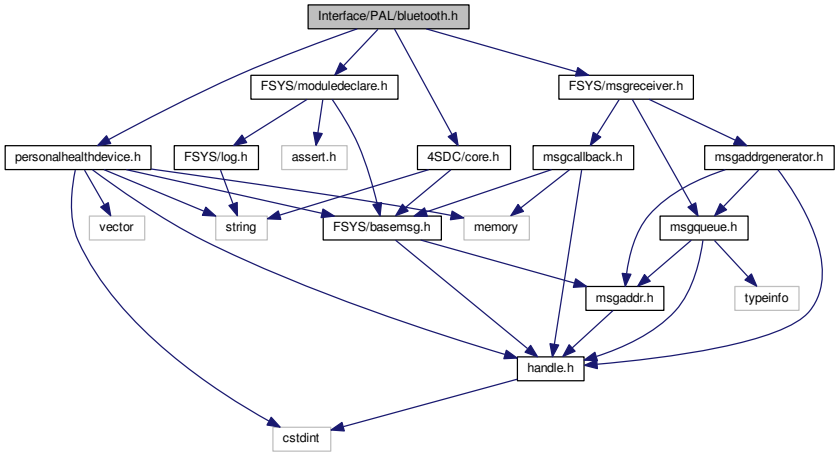
Copyright

13.17 Interface/PAL/bluetooth.h File Reference

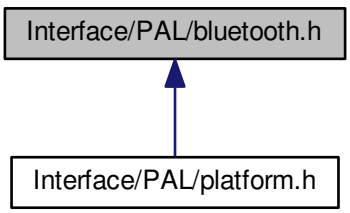
Bluetooth PAL-layer module.

```
#include "personalhealthdevice.h"
#include "FSYS/msgreceiver.h"
#include "4SDC/core.h"
#include "FSYS/moduledeclare.h"
```

Include dependency graph for bluetooth.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PAL::BluetoothModule

  Bluetooth PAL-layer module wrapper class.

13.17.1 Detailed Description

This module provides connections to Bluetooth devices.

Currently, this module only provides access to HDP devices, but in the future, support for BLE ("Smart") devices and device management operations (scanning, pairing etc.) should be added to this module.

Author

Jacob Andersen, The Alexandra Institute, © 2014

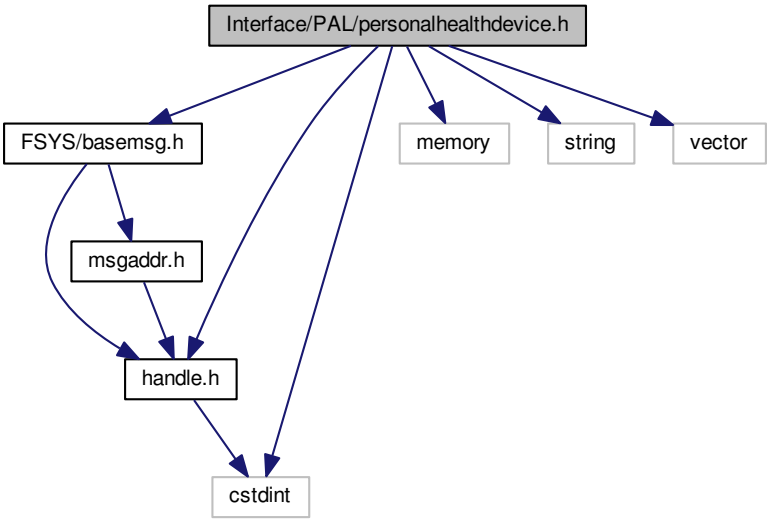
Copyright

13.18 Interface/PAL/personalhealthdevice.h File Reference

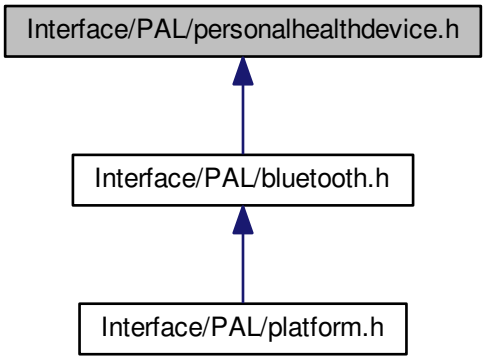
PAL layer interface for personal health device communication.

```
#include "FSYS/basemsg.h"
#include "FSYS/handle.h"
#include <cstdint>
#include <memory>
#include <string>
#include <vector>
```

Include dependency graph for personalhealthdevice.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PAL::PHDRegisterDatatypeMsg
- class PAL::PHDUnregisterDatatypeMsg
- class PAL::PHDConnectIndicationMsg
- class PAL::PHDDisconnectIndicationMsg
- class PAL::PHDDataTransferMsg
- class PAL::PHDDisconnectMsg
- class PAL::PHDConnectIndicationBluetoothMsg
- class PAL::PHDConnectIndicationUSBMsg
- class PAL::PHDConnectIndicationZigBeeMsg
- class PAL::PhysicalPHD
- class PAL::VirtualPHD

  The base interface of a personal health device seen from the perspective of a transport module.
- class PAL::PersonalHealthDeviceConnector

  The PAL interface for communication with personal health devices.
- class PAL::PHDBluetoothDevice

  Bluetooth specialization of the PersonalHealthDevice.
- class PAL::PHDBluetoothConnector

  The PersonalHealthDeviceConnector class specialization for communication with bluetooth devices (PHDBluetooth↩Device).
- class PAL::PHDUSBDevice

  A USB specialization of the PersonalHealthDevice.
- class PAL::PHDUSBConnector

  The PersonalHealthDeviceConnector class specialization for communication with USB devices (PHDUSBDevice).
- class PAL::PHDZigBeeDevice

  A ZigBee specialization of the PersonalHealthDevice.
- class PAL::PHDZigBeeConnector

  The PersonalHealthDeviceConnector class specialization for communication with ZigBee devices (PHDZigBee↩Device).
- class PAL::PersonalHealthDeviceHandler

The abstract class, session-layer components must implement to handle communication with personal health devices from any transport class.

- class PAL::PersonalHealthDeviceProviderBase

  The abstract class, PAL-layer components must implement to provide communication with personal health devices.

- class PAL::PHDBluetoothProvider

  Bluetooth specialisation of the PersonalHealthDeviceProviderBase.

Typedefs

- typedef uint32_t **errortype**

13.18.1 Detailed Description

The communication between transport components in the PAL layer and protocol handlers in the session layer is defined by the classes in this file.

The central interface (pure virtual class) is the PersonalHealthDeviceConnector, which, conceptually, may be thought of as an multiconductor electrical connector with signals flowing in both directions; with each method of the interface representing one of those uni-directional signals.

Two abstract classes, PersonalHealthDeviceHandler and PersonalHealthDeviceProviderBase implements the two "ends" of this connector – conceptually like a male and female electrical connector. These abstract classes will manage all the communication across the FSYS message system, leaving subclasses to focus only on handling (and generating) the messages. A component in the session layer must derive from the PersonalHealthDevice↩ Handler and implement the handling of upward signals, while a component in the PAL layer must derive from one of the subclasses of PersonalHealthDeviceProviderBase and implement the downward signals.

Author

Jacob Andersen, The Alexandra Institute.

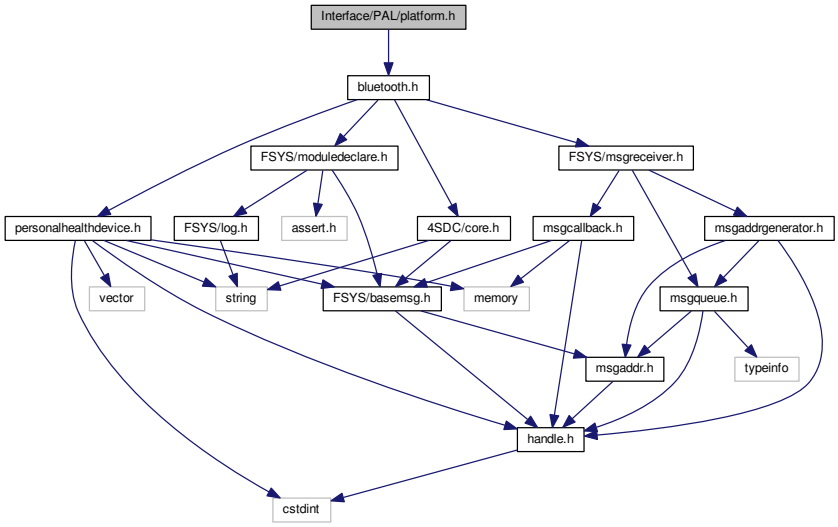
Copyright

13.19 Interface/PAL/platform.h File Reference

Platform-dependent declarations.

```
#include "bluetooth.h"
```

Include dependency graph for platform.h:



Classes

- class PAL::PALModuleLauncher

13.19.1 Detailed Description

This header declares the interface of platform-dependent PAL modules, which must be implemented by the platform libraries.

Author

Jacob Andersen, The Alexandra Institute, © 2014

Copyright

Bibliography

[1] The International Electrotechnical Commission, *IEC 62304:2006, Medical Device Software – Software Life Cycle Processes*, 1st ed., May 2006. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=38421 5

[2] National eHealth Authority, *Reference Architecture for Collecting Health Data From Citizens*, version 1.0, Jun 2013. [Online]. Available: http://www.ssi.dk/~/media/Indhold/DK%20-%20dansk/Sundhedsdata%20og%20it/NationalSundhedsIt/Standardisering/Referencearchitecture%20for%20collecting%20health%20data%20from%20citizens%20v%201%200.ashx 5

[3] National Sundheds-it, *Referencearkitektur for Opsamling af Helbredsdata hos Borgeren*, version 1.0, Jun 2013. [Online]. Available: http://www.ssi.dk/~/media/Indhold/DK%20-%20dansk/Sundhedsdata%20og%20it/NationalSundhedsIt/Standardisering/Referencearkitektur%20for%20opsamling%20af%20helbredsdata%20hos%20borgeren%20v%20%201.ashx 5

[4] Personal Connected Health Alliance, *Continua Design Guidelines*, Genome ed., 2015. [Online]. Available: http://www.continuaalliance.org/products/design-guidelines 5, 7

[5] ——, *Fundamentals of Data Exchange White Paper*, Sep 2015. [Online]. Available: https://cw.continuaalliance.org/document/dl/13473 5

[6] Medical Devices WG, *Health Device Profile*, Bluetooth SIG, version 1.1, Jul 2012. [Online]. Available: https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=260864 5, 8

[7] ISO/IEEE, *Health informatics – Personal health device communication – Part 20601: Application Profile – Optimized exchange protocol*, 1st ed., 2010. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=54331 5, 7, 8

[8] Medical Devices WG, *Personal Health Devices Transcoding White Paper*, Bluetooth SIG, version 1.5, Oct 2014. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=272346 8

[9] USB Implementers Forum, Inc., *Universal Serial Bus Device Class Definition for Personal Healthcare Devices*, version 1.0 (+errata), Nov 2007. [Online]. Available: http://www.usb.org/developers/docs/devclass_docs/Personal_Healthcare_1.zip 8

[10] The ZigBee Alliance, *ZigBee Health Care Profile Specification*, version 1.0, Mar 2010. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/10/docs-10-5619-00-0zhc-zigbee-health-care-profile-1-0-public.pdf 8

[11] NFC Forum, *Personal Health Device Communication*, version 1.0, 2013. 8

[12] Integrating the Healthcare Enterprise, *IHE Patient Care Device (PCD) Technical Framework, Volume 2, IHE PCD TF-2, Transactions*, version 4.0, Nov 2014. [Online]. Available: http://www.ihe.net/uploadedFiles/Documents/PCD/IHE_PCD_TF_Vol2.pdf 8

Index