Desarrollo de una capa de abstracción de hardware y clases FESA para la renovación del sistema de control de los aceleradores del CERN

Autor:
Rubén Pollán Bella
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Directora:
Anastasiya Radeva
Depto. Beams
Grupo Controls
CERN

Ponente:
José Luis Briz
Depto. de Informática e
Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

16 de agosto de 2011

Índice general

Capítulo 1

Introducción

En este capítulo introduciré el entorno en el que se desarrolla este Proyecto de Fin de Carrera. Para ello explicaré como se organiza el CERN y sus aceleradores, exponiendo el proyecto en el que se engloba este PFC y las tecnologías utilizadas para su desarrollo. Para terminar describiré las peculiaridades de este PFC y los objetivos sobre los que se ha trabajado.

1.1. El CERN

Este proyecto se realiza en el CERN[1], la *Organización Europea para la Investigación Nuclear*. En él he desarrollado un trabajo de renovación del sistema de control de los aceleradores, dentro de un proyecto global de renovación de software y hardware en partes que llevan décadas funcionando. Para ello he dado soporte a módulos de hardware dentro una interfaz común y he desarrollado aplicaciones para los mismos usando las plataformas desarrolladas en el CERN.

El CERN dispone de gran variedad de instalaciones dedicadas a realizar experimentos en física de partículas. La de mayor importancia actualmente es el LHC (*Large Hadron Collider*), un acelerador de partículas de 27 kilómetros de diámetro. En el, tras un año de funcionamiento, en la actualidad se están realizando

[1]Siglas en francés de *Conseil Européen pour la Recherche Nuclé* (Consejo Europeo para la Investigación Nuclear), nombre del consejo provisional creado en 1952 para establecer el laboratorio. Este laboratorio de física de partículas esta situado en la frontera entre Suiza y Francia, a las afueras de Ginebra. El CERN es uno de los centros de investigación científica mas importantes del mundo, gracias a la colaboración de sus 20 países miembros, 8 observadores y decenas de colaboradores.

experimentos sobre dos haces de protones a 3.5 TeV[2], que colisionan a energías de unos 7 TeV, lo que convierte al LHC en el acelerador de mayor energía del mundo. En los diversos experimentos que se llevan a cabo dentro de este acelerador se espera encontrar el Boson de Higgs[3] y extender el conocimiento sobre el Big Bang y la cosmología.

A parte del LHC en el CERN hay todo un complejo de aceleradores (Figura 1.1) de diferentes tamaños y diferentes energías. En ellos se hacen experimentos sobre antimateria, neutrínos, haces de iones etc. Muchos de los aceleradores están interconectados entre si, pues los haces acelerados en unos se usan como entrada para otros aceleradores que requieren las partículas de entrada con cierta energía inicial para poder funcionar. Por lo que los aceleradores conforman una cadena al principio de la cual se encuentran los menos potentes y estos alimentan a aceleradores mas potentes. Por ejemplo para llegar al LHC un haz de partículas ha atravesado antes cuatro aceleradores distintos.

1.2. Contexto tecnológico del PFC

Una vez puesto en marcha el LHC, que ha focalizado los mayores esfuerzos del CERN en los últimos años, se ha decidido renovar el *hardware* y *software* del resto de aceleradores, algunos instalados y en funcionamiento hace décadas. Para ello se ha creado el proyecto ACCOR (ACcelerator COntrol system Renovation).

El proyecto ACCOR se compone de muchos grupos trabajando en diferentes aspectos. Como el grupo de hardware que se encarga de desarrollar drivers y módulos hardware nuevos, o el grupo de aplicaciones que construye aplicaciones para ser usadas por los operadores de los aceleradores. Entre estos dos grupos está el grupo de Front-Ends, que se encarga de crear el software que accede a estos módulos y se comunica con las aplicaciones.

En el sistema de control de los aceleradores del CERN hay una gran variedad de módulos, como módulos con entradas y/o salidas analógicas o digitales, con generadores de señales etc. Cada módulo tiene características y drivers completamente diferentes. Dentro del proyecto ACCOR uno de los desarrollos que se esta realizando consiste en la creación de una capa de abstracción de hardware llama-

[2]Tera electronvoltio, unidad de enrgía equivalente a un electrón acelerado por una diferencia de potencial de 1 voltio.

[3]Partícula elemental hipotética masiva cuya existencia es predicha por el modelo estándar de la física de partículas. Desempeña un papel importante en la explicación del origen de la masa en otras partículas elementales.

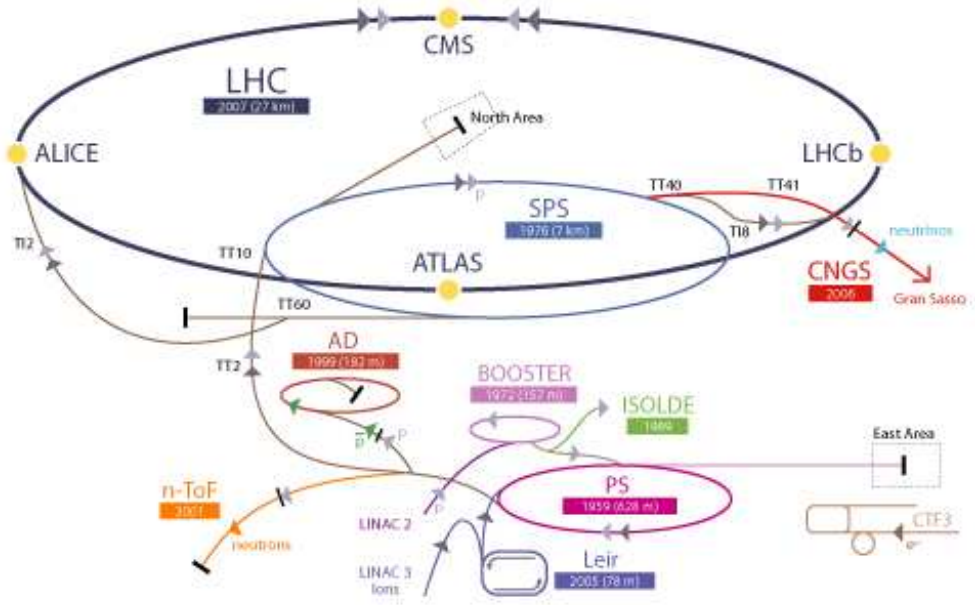


Figura 1.1: Complejo de Aceleradores del CERN
Extraído del archivo gráfico del CERN (`http://cdsweb.cern.ch/record/1260465`)

da COHAL (COntrols Hardware Abstraction Layer). COHAL básicamente es una jerarquía de clases C++ que organiza los módulos por familias con similares características (como Analog Input o Digital Input Output) mostrando una interfaz uniforme para módulos con características parecidas, ocultando las diferencias de drivers, sistemas operativos o arquitecturas.

Para la comunicación con las aplicaciones la sección de Front-Ends desarrolla una plataforma, llamada FESA (Sección 2.3). Con él se crean clases FESA, programas que se ejecutan sobre PCs industriales y que exportan información a las aplicaciones a través de la red interna usando CORBA. FESA autogenera código C++ encargado de la comunicación por CORBA, de la gestión de tareas en tiempo real, coordinación de procesos paralelos etc.

Dentro del proyecto ACCOR se están desarrollando diversas clases FESA, usando COHAL para acceder al hardware. Por cada familia se crea una clase FESA genérica, que permite acceder a la mayoría de funcionalidades del hardware con la que la mayoría de los usuarios pueden crear aplicaciones que usen el hardware sin necesidad de desarrollar ninguna clase FESA. A demás de las clases

genéricas se esta desarrollando clases FESA específicas para necesidades concretas de algunos aceleradores.

Durante mi estancia en el CERN hemos participado tres personas de la sección de Front-Ends en el proyecto ACCOR. Anastasiya Radeva, se ha encargado de supervisar y de coordinar la parte de Front-Ends de ACCOR además de desarrollar algunas partes comunes usadas por el resto. Genady Sivatskiy, es el encargado principalmente de los generadores de funciones (Sección 3.7). Mi Proyecto de Fin de Carrera ha consistido en encargarme del diseño, instalación y puesta en marcha de las tres familias restantes de hardware.

1.3. Contexto organizativo del PFC

El departamento de *Beams* es el encargado de la instalación y mantenimiento de los aceleradores. Este departamento dota de herramientas para los experimentos. No realiza experimentos por si mismo aunque se relaciona muy estrechamente con ellos.

El grupo de *Controls* se encarga de los controles de los aceleradores. Desde la electrónica de los mismos, diseñando módulos hardware propios o comprándolos a terceros o desarrollando drivers, a las aplicaciones como las que usan los operadores que controlan los aceleradores.

La sección *Front-Ends* administra los PCs industriales conectados a través de sensores y actuadores a lo largo de los aceleradores. Desarrolla software para ellos, instalando y manteniendo los sistemas operativos que los hacen funcionar o creando plataformas para que los experimentos puedan crear sus propios programas.

1.4. Naturaleza de este PFC

Mi estancia en el CERN se enmarca dentro de proyecto de *technical student*, unas becas concedidas a estudiantes de todo el mundo para trabajar durante un año en el CERN y realizar su proyecto fin de carrera. Empecé con esta beca a principios de Noviembre del 2009, entrando a trabajar la sección de *Front-Ends*, dentro del grupo de *Controls*, en el departamento de *Beams*. En ella he permanecido trabajando durante 14 meses hasta Diciembre del 2010 realizando las tareas descritas en este proyecto.

Dadas las características del proyecto ACCOR y de mi papel en el mismo, conviene hacer notar que este Proyecto de Fin de Carrera es un tanto atípico en cuanto a que no gira en torno a la resolución de uno o varios problemas estrechamente relacionados o al diseño e implementación de un software determinado, sino que agrupa un conjunto complejo de tareas, repartidas entre una amplia variedad de dispositivos *hardware*, tales como:

- Estudiar y comprender una instalación de gran complejidad, desde su amplia variedad de dispositivos *hardware* hasta las abstracciones ofrecidas a los usuarios mediante diferentes interfaces.
- Diseñar *software* (clases y su interacción) o bien colaborar o mejorar su diseño
- Implementarlas sobre dispositivos específicos, en general sujetos a restricciones de tiempo real.
- Diseñar y realizar ensayos para localizar errores o estudiar características no documentadas tanto en software propio o heredado como en el hardware gestionado por el mismo, proponiendo o realizando mejoras.

En algunos casos esto ha supuesto partir de elementos ya diseñados, en otros diseñar o programar desde cero, y en general utilizar gran parte de los conocimientos y habilidades adquiridos en la carrera para manejar diferentes entornos y sistemas o afrontar problemas nuevos.

El PFC recoge en definitiva el trabajo de 14 meses de estancia en el CERN a tiempo completo. Para facilitar la percepción y evaluación del esfuerzo realizado, el capítulo de Conclusiones resume en la Tabla 5.1 el conjunto de tareas concretas realizadas, indicando sus características y nivel de complejidad.

Los objetivos de este Proyecto de Fin de Carrera en el marco anteriormente descrito son:

- Desarrollar y mantener dentro de COHAL tres familias de hardware: Analog Input, Analog Output y Digital Input Output. Estas clases deben de soportar diferentes módulos de hardware, sobre dos sistemas operativos diferentes (Linux y LynxOS). Parte de las interfaces de estas familias vienen definidas genéricamente por COHAL, pero es preciso hacer un diseño específico, adaptando los conceptos genéricos de COHAL a las características de cada familia y las especificaciones concretas de cada módulo.

Para ello hay que trabajar muy estrechamente con el grupo de hardware, verificar drivers en desarrollo, evaluar si cumplen requisitos, solicitar modificaciones, etc.

- Desarrollar en con COHAL diversas clases FESA para el proyecto ACCOR. Algunas clases genéricas para las familias y algunas especificas para las necesidades de algunos aceleradores.

  El desarrollo de las clases FESA implica relacionarse directamente con el grupo de aplicaciones, discutir los requisitos que necesitan y solucionar los problemas que vayan apareciendo.

- También se incluye la realización de otras tareas relacionadas, fuera de las anteriormente especificadas o consecuencia de las mismas, como el desarrollo de clases FESA para dispositivos auxiliares, etc.

Capítulo 2

Contexto

En este capítulo describiré el contexto en el que se desarrolla el PFC. Para lo que explico el *hardware* usado, los diferentes tipos de PCs, buses de datos y tarjetas. A continuación hago una introducción a los sistemas operativos sobre los que se ejecuta el *software* desarrollado y sus principales características. Para terminar introduciendo dos plataformas (FESA y OASIS) empleadas a lo largo del PFC.

2.1. Hardware

La sección de Front-Ends, en la que yo he realizado el PFC, se encarga de instalar y dar soporte a lo que en el CERN se llama Front-Ends. Son PCs industriales a los que se conectan los equipos del sistema de control.

Los elementos *hardware* usados en el desarrollo de este PFC son:

PCs industriales	VME crate	Sección 2.1.1
	PCI crate	Sección 2.1.2
buses de datos	VME	Sección 2.1.1
	PCI	Sección 2.1.2
	VMOD	Sección 2.1.4
direccionamiento	Logic Unit Number	Sección 2.1.3
	slot	Sección 2.1.3

Cuadro 2.1: Elementos *hardware*

En el CERN se usan principalmente dos tipos de PCs industriales: *VME Crates* y *CompactPCI crates*. Se instalan sin disco duro, haciéndolos arrancar por red, y usando un sistema de archivos compartido entre todos montado a través de NFS[1].

2.1.1. VME crates

Los *VME crates* son dispositivos de bus VME[2] en las que se conecta la CPU y los diversos módulos. Este es el tipo principal de PC industrial que se usa en los sistemas de control del CERN, aunque a lo largo de los años ha ido cambiando el tipo de CPUs que se conectan en ellos.

Desde 1994 en el CERN se han usado CPUs de arquitectura PowerPC, con LynxOS (Sección 2.2.1) como sistema operativo, reemplazando los anteriores MC68K. Se han usado CPUs de CES[3], montadas sobre *VME crates*.

Desde el 2008 esta en proceso el dejar de instalar nuevos sistemas PowerPC, optando por usar CPUs MEN A20[4], que tienen una arquitectura X86-64 y ejecutan Linux (Sección 2.2.2). Al igual que las CPUs CES éstas se conectan a través de un bus VME a los PCs industriales.

2.1.2. PCI crates

Algunos módulos no se encuentran para el bus VME, sino solo para PCI o CompactPCI. Por ejemplo casi no hay osciloscopios VME, pero si que hay una gama muy amplia de ellos en PCI. Esto ha forzado a los sistemas de control del CERN a tener, ademas de VME, PCs industriales con PCI y CompactPCI buses.

Éstos se usan mucho menos que los *VME crates*. Ejecutan Linux con un arranque por red similar a los *VME crates*, compartiendo la mayor parte del sistema de archivos con ellos a través de NFS.

2.1.3. Gestión

En el sistema de control del CERN actualmente hay 1200 Front-Ends instalados. Para gestionarlos, junto con todas las particularidades de cada módulo instalado en ellos, hay una base de datos donde se almacena toda la información

[1] Network File System
[2] `http://www.vita.com/`
[3] `http://www.ces.ch/`
[4] `http://www.men.de/products/vmebus,3,01A020-.html`

de cada Front-End: los módulos que están conectados al Front-End, clases FESA instaladas etc.

De esta base de datos se obtiene de forma automática la información necesaria para el arranque de cada Front-End, como por ejemplo qué *drivers* necesita, qué parámetros necesita cada *driver*, los programas que hay que arrancar etc.

Todos los *drivers* tienen un sistema unificado de numeración de *hardware*. En él caso de módulos VME se accede a el a través del **Logic Unit Number** (LUN), que numera los módulos de un mismo tipo haciéndolos accesible de forma única. En el caso de módulos PCI se usa el número de **slot**, que depende de la posición física de módulo en el bus.

2.1.4. VMOD

Algunos de los módulos usados en el CERN se conectan a través de un conector VMOD. Para ello hay tarjetas que los interconectan con VME o PCI.

En las tarjetas VME caben 3 módulos VMOD, permitiendo conectar varios tipos de módulos en la misma. En el caso de PCI cada tarjeta puede contener 2 módulos VMOD en cada tarjeta.

2.2. Sistemas operativos

Las necesidades de los controles de los aceleradores en cuanto a los Sistemas Operativos (SO) usados son bastante especificas. Hace falta que el SO tenga características de **soft Real Time**, asegurando tiempos mínimos de respuesta a las peticiones. Ademas hace falta SO muy estables, que puedan estar ejecutándose con mucha carga durante años sin dar problemas.

Durante muchos años se ha venido usando LynxOS (Sección 2.2.1) como único SO para los sistemas de control. Pero desde hace poco más de dos años se ha decidido migrar progresivamente hacia Linux (Sección 2.2.2).

2.2.1. LynxOS

LynxOS[5] es un sistema operativo de tiempo real para sistemas empotrados. Es un sistema POSIX, lo que permite disponer de todas las herramientas de los sistemas Unix y facilita el desarrollo de aplicaciones portables entre LynxOS y otros sistemas operativos.

[5] `http://www.lynuxworks.com/`

A lo largo de los años de uso se han ido encontrando muchos problemas relacionados con este sistema operativo. Es un sistema privativo, por lo que no se pueden hacer modificaciones en él. Ello obliga al CERN a depender de la empresa que lo soporta para que arregle los problemas que surgen con el, arreglos que en muchos casos nunca llegan a realizarse o cuyas soluciones son parciales.

Muchas de las herramientas básicas del sistema, como el compilador de C o las bibliotecas stl de C++ están desfasadas y llenas de errores. Esto obliga a diseñar pensando en cómo evitar los problemas que surgen de ellos, teniendo que escribir código específico para evitarlos. Además muchos programas no se puedan compilar sin un gran esfuerzo para portarlos.

2.2.2. Linux

Dados los problemas encontrados en LynxOS se ha decidido montar los nuevos sistemas de control basados en Linux[6]. Este sistema operativo, junto con sus parches de Real Time[7], se adapta mucho mejor a las necesidades del CERN.

Linux, al ser *software libre*, permite al CERN adaptarlo si alguna parte no funciona como se espera. Además es un sistema en activo desarrollo, sus errores se solucionan en poco tiempo.

El CERN, junto con el Fermilab[8], desarrolla Scientific Linux[9] una distribución de GNU/Linux basada en RedHat[10]. El sistema de control de los aceleradores está basado en esta distribución, con algunas partes cambiadas, como el núcleo, que se compila con algunas modificaciones como los parches de tiempo real.

2.3. FESA

FESA es una plataforma para el sistema de control de aceleradores. El desarrollo lo empezó el CERN para su uso interno, pero en la actualidad es usado y desarrollado por GSI[11] además del CERN.

---

[6] `http://www.kernel.org/`

[7] `http://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html`

[8] Laboratorio de investigación en física de partículas estadounidense (`http://www.fnal.gov/`)

[9] `https://www.scientificlinux.org/`

[10] `http://www.redhat.com/`

[11] Gesellschaft für Schwerionenforschung, laboratorio Alemán de investigación de iones pesados. `http://www.gsi.de/`

FESA esta diseñado para crear clases (programas FESA) que se ejecutan en los Front-Ends, accediendo a los módulos de *hardware* instalados en él, y se comunican mediante CORBA[12] con las aplicaciones de control que se ejecutan en otras computadoras. Está compuesto de una serie de herramientas que autogeneran código C++ y se encargan de la instalación de las clases en los Front-Ends.

El flujo de trabajo con FESA (Figura 2.1) se compone de cinco fases:

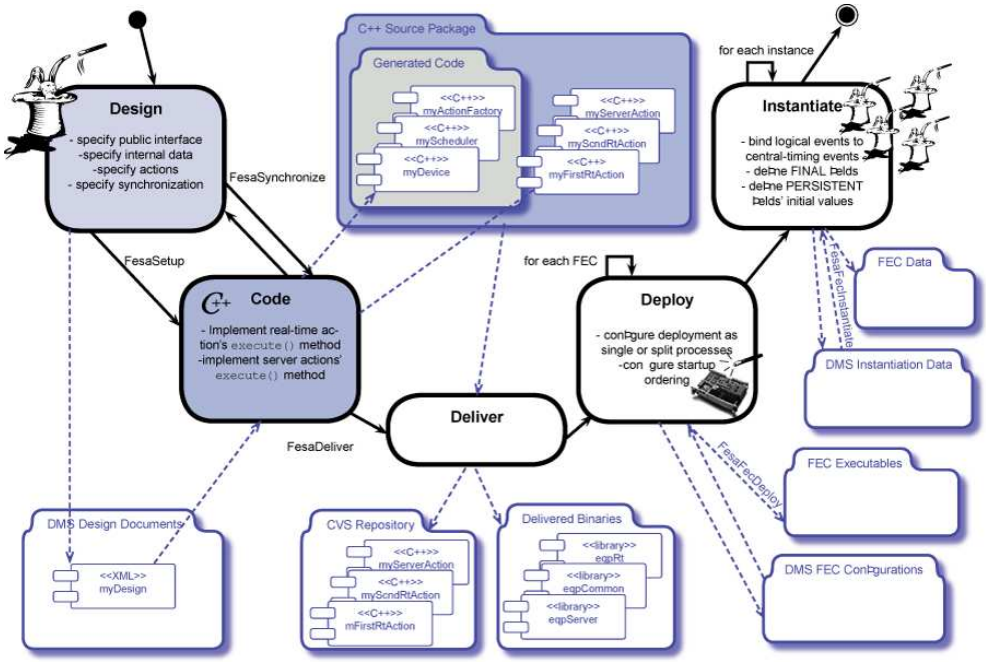


Figura 2.1: Flujo de trabajo de FESA
Extraído de la documentación de FESA

- **Design**. Definición de los diferentes componentes de la clase FESA, como el tipo de eventos que va a tener o la información que exporta a las aplicaciones a través de CORBA.
- **Code**. Una vez autogenerado el código a partir del diseño hay que programar algunas partes de el mismo. FESA crea clases C++ dejando esqueletos de métodos sin código para que el usuario introduzca la lógica en ellos.

[12]Common Object Request Broker Architecture, plataforma estándar de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

- **Deliver**. En esta fase se almacena el código en un CVS[13] centralizado disponible para todo el CERN. Además se compila el código guardando los binarios objeto, qué luego serán utilizados para generar las instalaciones en los Front-Ends a través de la información almacenada en la base de datos (Sección 2.1.3).

- **Deploy**. A FESA hay que indicarle en que Front-Ends se van a instalar la clase desarrollada y qué tipo de instalaciones se va a hacer. Aquí FESA se comunica con la base de datos de *hardware*.

- **Instantiate**. El último paso es indicarle a FESA que tipo de dispositivos va a usar la clase FESA. A través de este paso se le pueden pasar parámetros a la clase FESA definidos en su diseño, permitiendo configurar las características especificas de cada Front-End.

Más información sobre cada fase del flujo de trabajo de FESA en el Apéndice B.

2.4. OASIS

OASIS[14] (Open Analogue Signal Information System) es un sistema para la adquisición y visualización de señales analógicas en el dominio de los aceleradores. Las señales, distribuidas todo alrededor de los aceleradores, son digitalizadas por osciloscopios instalados en Front-Ends. La información recogida es enviada a través de redes Ethernet y mostrada en los puestos de trabajo que ejecutan las aplicaciones dedicadas de OASIS. Cuando el ancho de banda lo permite, las señales analógicas son multiplexadas en los canales de los osciloscopios. Teniendo en cuenta que no todas las señales disponibles son observadas al mismo tiempo, con esta arquitectura se consigue ahorrar digitalizadores, el dispositivo más caro del sistema. Los Front-End son instalados al lado de las fuentes de las señales, con intención de preservar la integridad de las señales en todo lo posible.

OASIS proporciona una abstracción de osciloscopio virtual (Vscope). Un Vscope es un osciloscopio por *software* que recoge la información desde diferentes osciloscopios *hardware* y la muestra como si viniera del mismo módulo. Gracias a esta arquitectura se pueden observar diferentes señales como si estuvieran unas

[13] Concurrent Versions System, un sistema de control de versiones.
[14] `http://project-oasis.web.cern.ch/project-oasis/`

junto a las otras, mientras que en la realidad pueden estar a distancias de cientos de metros. Por supuesto, para que esto funcione, es necesario tener el mismo pulso de trigger y OASIS debe mantener en sincronización los parámetros de configuración usados en las diferentes conexiones dependientes del Vscope.

OASIS tiene una arquitectura en tres capas (Figura 2.2). Una capa *hardware* de osciloscopios y Front-Ends recoge la información usando FESA para ello. Por encima se montan los servidores de aplicación que manejan la información que proviene de los Front-Ends. Y conectándose a estos servidores de aplicación la tercera capa están las aplicaciones de OASIS, que se ejecutan en los escritorios de los operadores.

OASIS maneja mas de 1800 señales en todo el complejo de aceleradores del CERN. Permite acceder a ellas de forma uniforme desde cualquier lado de la red del CERN a través de sus aplicaciones o de las bibliotecas de acceso de OASIS.

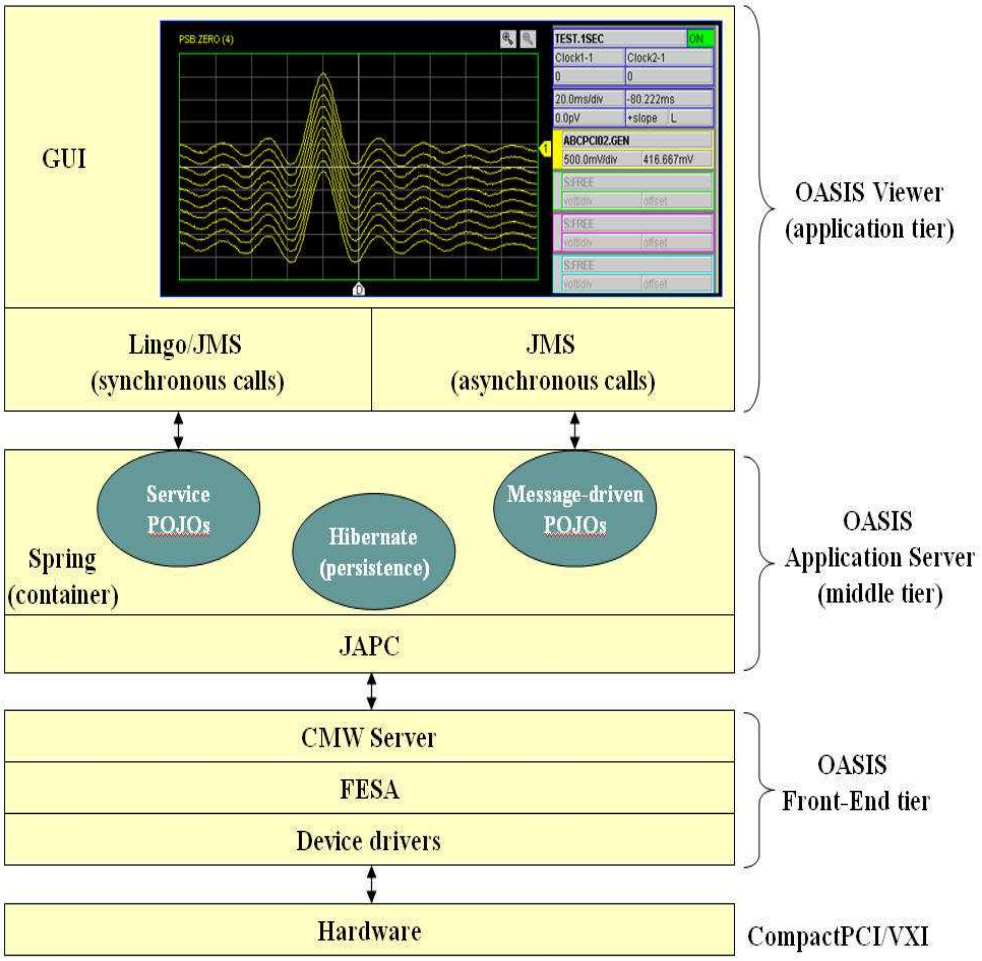


Figura 2.2: Arquitectura de oasis
Extraído de la web de OASIS
(`http://project-oasis.web.cern.ch/project-oasis/`)

Capítulo 3

COntrols Hardware Abstraction Layer

En el sistema de control de los aceleradores del CERN se usa gran variedad de módulos (tarjetas) de *hardware*. A través de ellas recibe el estado de las diferentes partes de cada acelerador mediante sensores o se modifica su estado a través de actuadores.

Para dar soporte de forma coherente a los módulos de *hardware* que se usan en los sistemas de control se ha desarrollado COHAL (COntrols Hardware Abstraction Layer). Es una capa de abstracción de *hardware* en C++, que agrupa los módulos según sus características en familias, dotando de una interfaz común a módulos con características comunes.

Cada módulo se compone de uno o varios canales de entrada y/o salida. COHAL muestra los canales de una forma uniforme entre módulos, usando un direccionamiento común entre módulos, aunque en cada módulo el direccionamiento suele ser diferente. COHAL ademas implementa algunas características no presentes en algunos módulos, como por ejemplo *readBack*[1].

3.1. Arquitectura

COHAL esta diseñado usando las características de orientación a objetos y meta-programación con patrones que ofrece C++. Con clases virtuales y herencia se agrupan los diferentes módulos en familias.

[1] Leer el dato que se ha escrito anteriormente en un canal de salida de un módulo.

COHAL se compone de un serie de clases genéricas, con herramientas comunes para todas las familias, y cuatro familias de *hardware*:

- **Digital Input Output**. Módulos de entrada y salida digital.
- **Analog Input**. Conversores de señales analógicas a digital.
- **Analog Output**. Conversores de digital a analógico.
- **Function Generator**. Generadores de ondas.

La mayor parte del diseño de alto nivel (jeraquía de módulos y clases) estaba ya definido al comienzo de este Proyecto de Fin de Carrera, pero durante el mismo se ha adaptado la arquitectura de COHAL a las necesidades del proyecto ACCOR, en colaboración con otros dos desarrolladores de COHAL. Las tres primeras clases que acabamos de listar las he desarrollado completamente desde cero como parte del PFC, y las he mantenido hasta finalizar mi estancia en el CERN. El Apéndice A proporciona una referencia del contenido de las tres familias desarrolladas.

Cuando se diseñó inicialmente se desconocían en profundidad los usos que iba a tener la capa de abstracción de *hardware*. Por este motivo se recurrió a un diseño genérico susceptible evolucionar al ir descubriendo necesidades concretas.

El Proyecto de Fin de Carrera ha permitido acelerar la evolución de los interfaces, observando las aplicaciones reales que las iban a utilizar, y las características específicas de los módulos *hardware* soportados. En la última fase del PFC se ha trabajado en estabilizar las clases, ya que hay varias instalaciones en el complejo de aceleradores y cada cambio supone realizar actualizaciones.

3.1.1. Jerarquía de módulos

Por cada módulo *hardware* soportado hay una clase a través de la cual se puede acceder a toda la funcionalidad del mismo. Estas clases están integradas en una jerarquía, siendo todas derivadas de la clase genérica de su familia, la cual es derivada de la clase *Module*. La Figura 3.1 muestra la jerarquía módulos de todas las familias que he desarrollado en el PFC.

3.1.2. Inicialización del módulo

Para inicializar los módulos hay dos jerarquías de clases definidas en COHAL: **ModuleConfig** y **ChannelConfig**. Dependiendo de sus características cada módu-

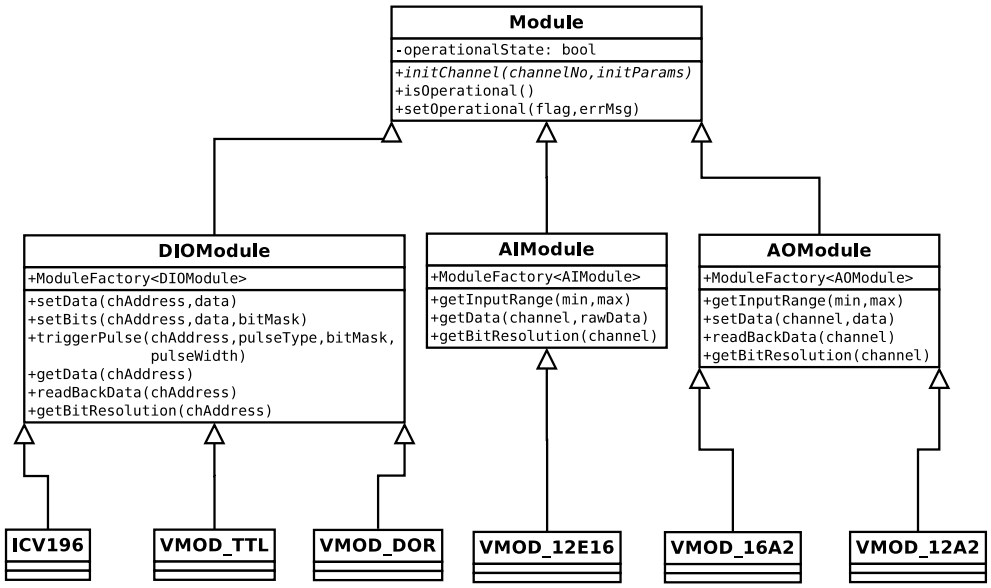


Figura 3.1: Jerarquía de clases para los Módulos COHAL

lo puede no usar ninguna, usar una de las dos o ambas simultáneamente.

En la inicialización del módulo se le da información a COHAL de cómo se va a usar un módulo especifico. Por ejemplo qué canales se van a usar como entrada o como salida, qué voltajes o impedancias va a tener en la entrada. Algunos de estos parámetros se configuran por *software* desde COHAL en el módulo, otros los usuarios tienen que configurarlos a través de pines en el *hardware* y luego informar a COHAL de la configuración que han realizado.

La configuración global del módulo, común para todos sus canales, se hace a través de clases derivadas de *ModuleConfig*, que se pasa al módulo a través del constructor. Puede haber una clase derivada de *ModuleConfig* por módulo y/o uno genérico de la familia, dependiendo de las características comunes o diferentes de cada módulo dentro de la familia.

No es obligatorio el uso de *ModuleConfig*, hay muchos módulos que no necesitan ninguna configuración, por lo que hay familias y módulos que no tienen definido ninguno.

La otra forma de configurar un módulo de las clases derivadas de *Channel-Config*, las cuales se encargan de la configuración de los canales del módulo. Por cada canal a configurar se hace una llamada al método *initChannel* con un *Chan-*

nelConfig.

Al igual que con *ModuleConfig* cada familia o módulo puede tener una clase derivada de *ChannelConfig* o ninguna, dependiendo de si necesita configuración específica por canal y si esta es común para toda la familia o cada módulo necesita una específica.

Como se ha mencionado, la estructura general de estas clases estaba ya definida, y el trabajo en el marco del PFC ha consistido en definir que familias o módulos necesitaban de ellas y que parámetros deberían contener cada una.

3.1.3. Acceso a módulos

Para poder acceder a los módulos de forma global se ha aplicado la metodología de diseño *Abstract Factory*[gamma95], creando una factoría genérica *ModuleFactory* (ver Sección 3.3.3) que está instanciada como un miembro estático dentro de la clase de cada familia.

Para instanciar o acceder a un módulo se hace a través la factoría de su familia, de esta forma que hay un punto único de entrada a los módulos de una familia. Por este motivo un mismo módulo no se puede instanciar dos veces, si tiene el mismo *HWAddress* (ver Sección 3.3.1).

Cada módulo dispone de un mutex, que lo bloquea en caso de estar accediendo al *hardware*, por lo que no se puede dar dos accesos concurrentes al mismo módulo *hardware*.

La infraestructura genérica para el acceso a los módulos estaba definida antes del PFC, cuyo cometido ha sido dotar a las clases de funcionalidad en cada familia.

3.1.4. Gestión de errores

En caso de errores los módulos lanzan excepciones (descritas en la Sección 3.3.4), con información sobre el problema encontrado.

En caso de que el error afecte al funcionamiento del módulo, de tal forma que éste no se pueda usar tras el error, además de la excepción se activa un indicador interno de la clase *Module* llamado *operationalState*. Éste indica si el módulo esta operativo o no, y en caso de no estarlo conserva una cadena de texto con información sobre el error que ha producido.

De esta forma en caso de error irreparable se desactiva el módulo a través del método *setOperational()*. En cada intento de acceso al módulo se comprueba el indicador a traves del método *isOperational()* y en caso de estar activado no se ejecuta la acción, sino que se devuelve un error.

3.2. Test

Cada familia contiene un programa de test. Éste es usado para comprobar el funcionamiento de cada módulo y como ejemplo de cómo usar la familia COHAL.

Los programas de test que se han creado en el PFC para las clases correspondientes son genéricos, permitiendo acceder a toda la funcionalidad de cada uno de los módulos de la familia a través de opciones pasadas a través de la linea de comandos. Esto ha permitido en algunos casos usar módulos de salida para generar datos útiles para comprobar las entradas de otros módulos.

3.3. Generic

COHAL dispone de una serie de clases genéricas usadas por todos los módulos. En ellas residen funciones para el manejo de cadenas, gestión de excepciones, clases virtuales de las que hereda el resto (Module, ModuleConfig, ChannelConfig, ...), clases de abstracción de *hardware* etc.

Todo esto se compila como una librería estática, que se luego se enlazará con el programa que vaya a usar COHAL. Esta librería se llama *libcohal_gen.a*.

Esta parte no ha sido objeto del PFC, pero el desarrollo de las clases específicas de las que nos hemos ocupado ha puesto en evidencia nuevas necesidades para la librería de clases genéricas, y esto ha dado lugar a un estrecho trabajo en equipo para la redefinición de algunas partes.

3.3.1. HWAddress

Tal y como se describía en la Sección 2.1 en el sistema de control de los aceleradores del CERN hay una gran variedad de *hardware*. Diferentes tipos de *hardware* pueden tener diferentes formas de direccionamiento, aunque principalmente se usan el *LUN* o el *Slot* (los sistemas de direccionamiento usados en el CERN, explicados en la Sección 2.1.3).

Con la intención de poder acceder a los módu los de una forma uniforme, independientemente del direccionamiento que usen éstos, COHAL ha creado *HWAddress*. Esta clase identifica un módulo *hardware* tenga el direccionamiento que tenga.

3.3.2. ToolBox

El *ToolBox* recopila funciones comunes usadas en diferentes partes de COHAL. Entre ellas se encuentran funciones para comprobar si un valor está dentro de un rango, funciones que devuelven el mínimo o máximo de una lista o funciones para convertir diferentes tipos de datos a cadenas de texto.

Algunas de las funciones de este paquete están orientadas a solucionar problemas de compatibilidad entre sistemas operativos (ver Sección 2.2). Por ejemplo, algunas funcionalidades de C++ no están bien soportadas en LynxOS, como los *std::strinstream* por lo que en el ToolBox hay una serie de funciones *ToolBox::toString()* para convertir diferentes tipos de datos a *std::string*.

3.3.3. ModuleFactory

En la Sección 3.1.3 se ha explicado que el acceso e instanciación de módulos se hace a través de una factoría. Esta factoría está definida en la clase *ModuleFactory* (Figura 3.2).

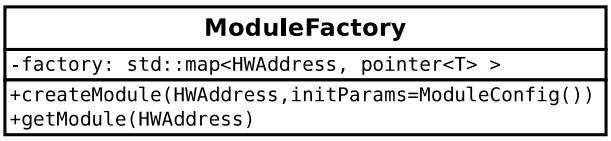


Figura 3.2: Excepciones de Módulos COHAL

La factoría se compone de un *std::map* que relaciona el *HWAddress* (Sección 3.3.1) con su módulo, permitiendo acceder a los módulos a través de su *HWAddress*.

3.3.4. ModuleException

Para indicar problemas de mal funcionamiento o de configuración en módulos en COHAL se usa la infraestructura de excepciones de C++. Se ha definido una clase genérica **ModuleException** y dos clases derivadas de ésta: **ModuleIOError** y **ModuleBadParameter** (ver Figura 3.3).

ModuleException, y a través de la herencia *ModuleIOError* y *ModuleBadParameter*, contiene una cadena de texto con información sobre el error producido. Esta cadena siempre tiene la misma estructura:

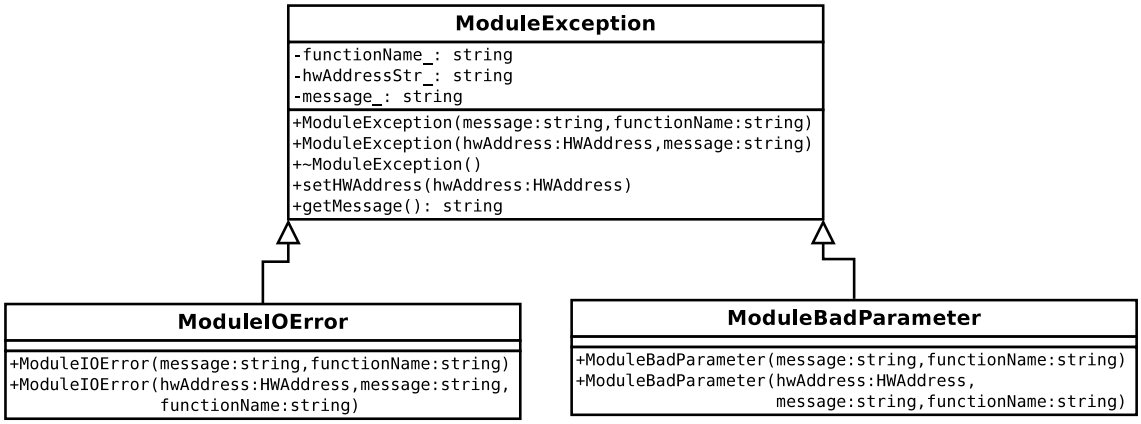


Figura 3.3: Excepciones de Módulos COHAL

```
COHAL::NombreModulo::metodo(...) lun: num: mensaje explicativo
```

En este mensaje se indica el *namespace* (COHAL), la clase que lo ha lanzado la excepción (NombreModulo), el método en el que se ha producido (metodo), su Logic Unit Number[2] (num) y un texto explicativo de que ha producido el error.

ModuleIOError se usa en caso de errores de acceso al *hardware*, por ejemplo si el módulo no está conectado al Front-End o si el *driver* devuelve algún tipo de error en alguna operación.

ModuleBadParameter es usado en caso de configuraciones o parámetros erróneos. Esta excepción es lanzada en la configuración del módulo o de sus canales, si alguno de la configuración no es valido, como por ejemplo en caso de intentar configurar como entrada un canal que solo soporta salida. También es lanzada en caso de pasar algún parámetro inválido a algún método del módulo, como por ejemplo si se intenta acceder al canal tercero en un módulo de dos canales.

3.4. Digital Input Output

Familia de módulos de entrada y salida digital. Los módulos soportados por esta familia tienen canales de entrada y/o salida normalmente de 8 bits y con valores en cada bit de 0 o 5 Voltios.

[2]Descrito en la Sección 2.1.

Esta familia da soporte a tres modelos de *hardware* diferentes: ICV196, VMOD-TTL and VMOD-DOR.

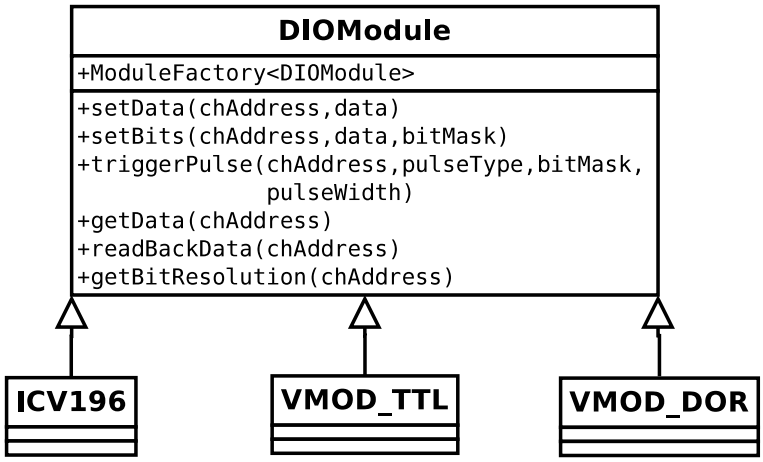


Figura 3.4: Familia DIO COHAL

En esta familia hace varias abstracciones sobre el *hardware* implementando las partes necesarias en *software*, cuando es posible, si el *hardware* no las soporta. Las características implementadas en toda la familia son:

- **Canales con 8, 16 y 32 bits**. Excepto en los módulos que no disponen de canales suficientes para tener 32 bits. Algunos módulos daban soporte por *hardware* para el acceso a canales de diferente ancho de banda, pero no en todos los casos. Por ello en algunos módulos esto se ha tenido que implementar por *software*, ofreciendo una abstracción al usuario para que parezca que el *hardware* lo soporta.

- **Lógica directa o invertida**. En caso de canales TTL lógica directa significa que un 1 lógico se representa con 5 Voltios y un 0 lógico se representa con 0 Voltios; lógica invertida quiere decir que un 1 lógico ésta representado por 0 Voltios y un 0 lógico por 5 Voltios. En otros tipos de canales depende del *hardware*, pero en general 1 lógico en lógica directa será el estado activo.

  Para todos los módulos de esta familia ha hecho falta implementar la inversión de la lógica por *software*, pues aunque algunos la permitían por *hardware* esta era una configuración global a todo el módulo y en COHAL se permite configurar la lógica por cada canal. Además se ha encontrado

que algunos módulos solo usaban lógica invertida y otros solo lógica directa. Por esto he tenido que trasformar cada dato para que haya una salida homogénea se use un módulo u otro.

- **Read back**. Los canales de salida deben de permitir acceder al último dato transmitido. En algunas tarjetas esto se hace por *hardware*, pero no todas lo soportan, por lo que en algunos casos ha habido que implementarlo por *software*.

- **Set bit**. Permite cambiar el valor de unos pocos bits en un canal de salida sin modificar el resto de bits, mediante una máscara de bits.

- **Producir un pulso**. Genera un pulso del tiempo (en microsegundos) que indique el usuario en un canal de salida. Al igual que *setBit* actúa sobre unos bits concretos dejando el resto sin igual.

Ver Apéndice A.3.5 para una descripción mas detallada de la implementación en COHAL de esta familia.

3.4.1. ChannelAddress

Para conseguir un direccionamiento de los canales homogéneo he creado la clase DIOChannelAddress. En ella se indica cuantos canales físicos del módulo componen el canal de COHAL al que se quiere acceder.

Esta clase sirve además internamente para comprobar colisiones entre canales. Detecta cuando se intenta configurar dos canales que se solapan total o parcialmente.

Esta clase no estaba prevista en el marco general inicial del Proyecto. Descubrí que era necesaria al desarrollar esta familia, de modo que la propuse, diseñé e implementé íntegramente en el marco del PFC.

Ver Apéndice A.3.3 para una descripción mas detallada.

3.4.2. ICV196

El ICV196[3] es un módulo con doce canales configurables como entrada o salida, de 8 bits *TTL* cada uno. Los canales son configurables para usarse en parejas, dando como resultado seis canales de 16 bits. Los canales tienen lógica inversa,

[3] `http://www.adas.fr/icv196us.htm`

por lo que en caso de querer escribir un 1 salen 0 voltios y en caso de un 0 sale 5 voltios. Este módulo se conecta a través de un bus *VME* a los Front-Ends.

Este módulo está desarrollado por una empresa externa (Adas Électronique[4]), pero los *drivers* y librerías de acceso al *hardware* se desarrollan en el CERN. El *driver* y la librería de acceso en C al módulo estaban ya desarrollados y funcionado cuando yo empecé a trabajar con el, pero la documentación era algo pobre. Parte de mi trabajo consistió en hablar con el desarrollador de la biblioteca de acceso para entender cómo se usa el módulo y colaborar para mejorar la documentación.

A lo largo de un meses desarrollé las clases necesarias para dar soporte a este módulo dentro de la infraestructura de COHAL y realicé comprobaciones intensivas para asegurar que el *driver* y sus diferentes instalaciones funcionaban correctamente.

Tras el desarrollo inicial ha hecho falta en varias ocasiones modificar el soporte de COHAL para ICV196. Por un lado para adaptarlo a las necesidades de las clases FESA que se desarrollaron usándola (ver por ejemplo la Sección 4.3). Y por otro para darle homogeneidad a la interfaz de la familia *Digital Input Output*, que se iba modificando mientras se añadían otros módulos a ella.

3.4.3. VMOD-TTL

El VMOD-TTL[5] es un módulo con tres canales, dos de 8 bits y uno de 4 bits, configurables como entrada o salida y como lógica directa o negada. En caso de ser de salida los canales se pueden configurar para dar una salida *TTL* o de *colector abierto*. En caso de ser de entrada se pueden programar interrupciones sobre los canales para detectar patrones en ellos.

Este módulo se conecta a través de un conector *VMOD* (Sección 2.1.4), lo que lo permite montar tres tarjetas VMOD sobre una tarjeta *VME* o dos sobre una tarjeta *PCI*.

Al igual que la ICV196 la VMOD-TTL se compra a una empresa externa (Janz[6]) y los *drivers* y bibliotecas de acceso se desarrollan en el CERN. Para LynxOS había un *driver* que se desarrolló en 1995 del que sus desarrolladores dejaron de trabajar en el CERN hace años. Este *driver* contiene muchos *bugs*, que el equipo de desarrollo de *drivers* del CERN no se atreve a intentar arreglar por

[4] `http://www.adas.fr/`
[5] `http://www.janz.de/as/en/modulbus/vmod-ttl.html`
[6] `http://www.janz.de/`

la complejidad del mismo y por ser LynxOS un sistema operativo a eliminar del CERN.

La integración del *driver* heredado supuso largos periodos de pruebas y de soluciones alternativas a los problemas que presentaba, en el curso de las cuales descubrí funcionalidades no documentadas. Por ejemplo, el *driver* disponía de la funcionalidad *read back* descrita en la pág. 27 (lectura del último dato enviado por un canal), oculta tras un error de acceso al *hardware*.

Más adelante el equipo de *drivers* del CERN unificó en un mismo interfaz el acceso al *driver* heredado sobre LynxOS y a un nuevo *driver* sobre Linux. Mi trabajo consistió en eliminar las dependencias del *driver* heredado.

En cuanto estuvo disponible el *driver* para Linux trabajé en integrarlo dentro de COHAL, colaborando muy estrechamente con sus desarrolladores. Participé en el testeo del *driver* durante su desarrollo, ayudándoles a depurarlo, y colaborando en la documentación del mismo. El hecho de tratarse de un *driver* diseñado desde cero me permitió hacer peticiones y debatir con los desarrolladores modificaciones que lo adaptaran mejor a las necesidades de COHAL.

En el desarrollo del soporte para interrupciones el equipo de *drivers* se encontró con limitaciones del *hardware*, no descritas en su *data sheet*. A lo largo de una semana estuve integrando el soporte de interrupciones en COHAL, usando un *driver* preliminar. Pero esa linea de desarrollo se suspendió definitivamente hasta que se consiga aclarar con el fabricante las características reales del *hardware*.

3.4.4. VMOD-DOR

El VMOD-DOR es un módulo con cuatro canales de salida de 4 bits, configurables como dos canales de 8 bits o un canal de 16 bits. Los canales son de *colector abierto*, soportando hasta 70 Voltios y 0.3 Amperios. Se conecta a través de un conector *VMOD*.

Este módulo, junto con su *driver* y bibliotecas de acceso, ha sido desarrollado en el CERN, con soporte para LynxOS y Linux a través de la misma interfaz. Al igual que en el caso de la VMOD-TTL, existe un *driver* viejo no mantenido para LynxOS y uno nuevo para Linux, pero la biblioteca de acceso tiene la misma API.

VMOD-DOR es un módulo sencillo. Al soportar solo escritura en sus canales y necesita pocas opciones de configuración. Por ello pude integrarlo en COHAL en dos semanas, trabajando conjuntamente con el equipo de *drivers* para que realizaran pequeñas modificaciones en el mismo.

Dentro de COHAL no hay soporte para canales de 4 bits, por lo que desde COHAL sólo se pueden utilizar los canales de esta tarjeta como canales de 8 o 16

bits.

3.5. Analog Input

La familia de módulos de entrada analógica, gestiona conversores de señales analógicas a digital. Leen la señal conectada a la entrada de sus canales, indicando su voltaje o intensidad.

Esta familia por el momento solo integra un tipo de módulo, el VMOD-12E16. Está previsto integrar otros módulos en ella por lo que se ha desarrollado de forma genérica para que se pueda adaptar a las necesidades de otros módulos.

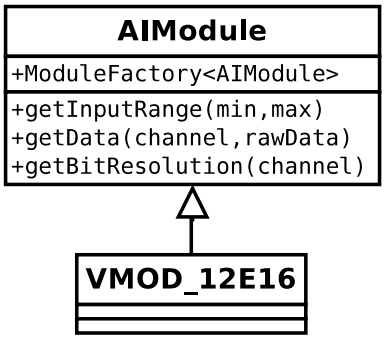


Figura 3.5: Familia DIO COHAL

Las características implementadas en toda la familia son:

- **Leer dato**. Lee el valor en Voltios/Amperios a la entrada del canal dando un valor numérico como resultado. Hace las conversiones pertinentes del valor en bruto proporcionado por la tarjeta al valor real de Voltios o Amperios presente en la entrada.

  Además del dato procesado informa del dato en bruto devuelto por el *hardware*.

Ver Apéndice A.3.1 para una descripción más detallada.

3.5.1. VMOD-12E16

El VMOD-12E16[7] es un módulo con dieciséis canales, configurables como ocho canales diferenciales, con 12 bits de resolución, configurable para leer rangos de ± 5, ± 10, $0-10$ V o $0-20$ mA. Este módulo se conecta a través de un conector *VMOD* (Sección 2.1.4).

En caso de configurarse como canales diferenciales mide la diferencia de tensión o intesidad entre dos canales, sin ninguna intervención del *software*.

Es programable para que amplifique las entradas pudiendo medir valores muy inferiores. Pero el ruido, descubierto durante las pruebas, destruye gran parte de los datos, por lo que hemos decidido no darle soporte a esta funcionalidad dentro de COHAL.

Este módulo es producido por una empresa externa (Janz[8]), pero sus *drivers* y bibliotecas han sido desarrollados en el CERN. Los *drivers* son nuevos y desarrollados solo para Linux.

El desarrollo fue bastante sencillo, pues los *drivers* son estables y sus desarrolladores me han ayudado mucho con todas las dudas y problemas que he ido encontrando. Las pruebas de este módulo las he realizado en conjunción con las del VMOD-12A2 (Sección 3.6.1) y las del VMOD-16A2 (Sección 3.6.2), usando éstos para producir señales senoidales y el VMOD-12E16 para recibirlas, comprobando que cada parte hacía su trabajo y funcionaba correctamente.

3.6. Analog Output

Esta familia de módulos de salida analógica gestiona conversores de digital a analógico. Escriben en sus canales la tensión o intensidad indicada.

He integrado dos módulos a esta familia: VMOD-12a2 y VMOD-16a2.

Las características implementadas en toda la familia son:

- **Escribir un dato**. Escribir un valor en Voltios/Amperios en la salida del canal. A partir del valor numérico de la tensión o intensidad real se calcula el valor en bruto que necesita la tarjeta para generar la salida requerida.
- **Read back**. Da la posibilidad de leer el último dato escrito en un canal de salida.

Ver Apéndice A.3.2 para una descripción mas detallada.

[7] `http://www.janz.de/as/en/modulbus/vmod-12e16.html`
[8] `http://www.janz.de/`

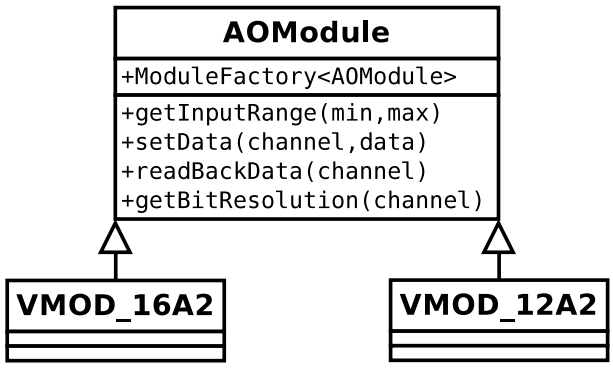


Figura 3.6: Familia DIO COHAL

3.6.1. VMOD-12A2

El VMOD-12A2[9] es un módulo con dos canales de salida. Con 12 bits de resolución, configurable para leer rangos de ± 5, ± 10, $0 - 10$ Voltios o $0 - 20$, $0 - 40$ miliAmperios. Se conecta a través de un conector *VMOD*.

Este módulo es producido por una empresa externa (Janz[10]), pero sus *drivers* y bibliotecas han sido desarrollados en el CERN únicamente para el sistema operativo Linux.

En COHAL no se ha implementado el soporte para salidas en miliAmperios, sino solo para Voltios.

Los *drivers* de este modulo son muy parecidos a los del VMOD-12E16 (Sección 3.5.1), lo que me permitió rápidamente estar usándolos con fluidez.

3.6.2. VMOD-16A2

El VMOD-16A2[11] es un módulo con dos canales de salida, similar al VMOD-12A2. Su principal diferencia es que el VMOD-16A2 dispone de 16 bits de resolución mientras que el VMOD-12A2 dispone de 12 bits de resolución.

La integración del VMOD-16A2 en COHAL la he realizado en paralelo con el VMOD-12A2. Al ser dos módulos muy parecidos y tener los dos una biblioteca de acceso similar, el código en ambos tiene pocas diferencias.

[9] `http://www.janz.de/as/en/modulbus/vmod-12a2-vmod-12a4.html`
[10] `http://www.janz.de/`
[11] `http://www.janz.de/as/en/modulbus/vmod-16a1-vmod-16a2.html`

3.7. Otras familias

Aparte de las anteriormente mencionadas en COHAL está la familia *Function Generator*, encargada de dar soporte a generadores de señales. Yo no he tomado parte en su desarrollo por lo que no menciono aquí su contenido.

En un futuro se espera ampliar COHAL añadiendo más familias, que en estos momentos están soportadas por otros tipos de módulos que actualmente se usan a través de otras herramientas.

Capítulo 4

Clases FESA

FESA, tal y como se describe en la Sección 2.3, es una plataforma para la creación de programas (clases FESA) para ser ejecutados dentro de los Front-Ends del sistema de control de los aceleradores.

Dentro del proyecto ACCOR se están desarrollando diversas clases FESA para sustituir los sistemas de control de los aceleradores más antiguos. Principalmente se desarrollan clases genéricas, que permitan acceder a las funcionalidades de los módulos dadas por COHAL. Para algunas instalaciones con necesidades especiales se desarrollan clases especificas.

Como parte de mi aportación al proyecto ACCOR he desarrollado 3 clases FESA: CGAI, SISL2Watchdog y OasisCursor. También he colaborado en otras 2 clases: CGDIO y OasisRdaClient.

4.1. CGAI

Clase FESA genérica de la familia *Analog Input* (Sección 3.5), que permite acceder a los módulos de dicha familia y a toda su funcionalidad. Con ella cambiando un solo parámetro de instanciación se puede acceder de forma igual a unos u otros módulos de la familia.

El desarrollo de esta clase permitió probar la familia AI, y rediseñar su interfaz para que tuviera más coherencia con las necesidades de FESA. Esto provocó algunos cambios en la familia mientras se diseñaba la clase FESA, como por ejemplo el añadir soporte para leer el dato de la tarjeta en bruto.

Una vez estabilizada la clase CGAI su primera instalación se ha realizado en un laboratorio de mediciones ambientales. En este laboratorio se están probando

sensores y diferentes arquitecturas usando CGAI para sustituir los sensores ambientales de algunos aceleradores.

4.2. CGDIO

Clase genérica de la familia *Digital Input Output* (Sección 3.4).

Esta clase se desarrolló sólo parcialmente, porque aparecieron otras prioridades. El diseño y código realizados han quedado en el CERN para ser completados en el futuro.

4.3. SISL2Watchdog

SISL2Watchdog es parte de la renovación del *Software Interlock System* (SIS) para el LINAC2[1]. SIS requiere un registro persistente para algunos parámetros de configuración, como umbrales, contadores, estados, rangos de tolerancia para los trafos o las fuentes de alimentación. Es necesario tener estos parámetros accesibles para escritura y lectura de forma paralela por las aplicaciones especificas y otras herramientas del sistema de control. Para solucionar esto se ha optado por implementar una clase FESA que implemente este almacenamiento.

Además del registro de parámetros de configuración, hay dos conexiones hardware con el *Interlock crate*[2] del LINAC2. Una dedicada a informar del estado del Interlock a las aplicaciones y otra del SIS al Interlock, la señal *keep alive*, informándole de que SIS sigue activo. Esta es una señal periódica. En caso de no presentarse, el Interlock supone que el SIS ha dejado de funcionar y toma las decisiones por sí mismo.

Para gestionar la señal del SIS al Interlock informando que sigue vivo se usa una clase FESA genérica diseñada anteriormente llamada LTIM, haciendo una instalación de la misma configurada para generar la señal tal como se requiere. Para el resto, el registro persistente y la recepción de señales del *Interlock crate*, se ha decidido implementar una clase FESA especifica.

[1] El primer acelerador en la cadena, en el que se usan átomos de hidrógeno, y separándolos, se empieza a acelerar sus protones. Ver Figura 1.1 para mas información sobre el complejo de aceleradores

[2] Elemento hardware que se encarga de controlar la salida del LINAC2 a otros aceleradores, en caso de que el haz sea defectuoso lo elimina.

He implementado esta clase FESA especifica partiendo un diseño del sistema dado (Figura 4.1) usando la familia DIO de COHAL (Sección 3.4) para las conexiones hardware.

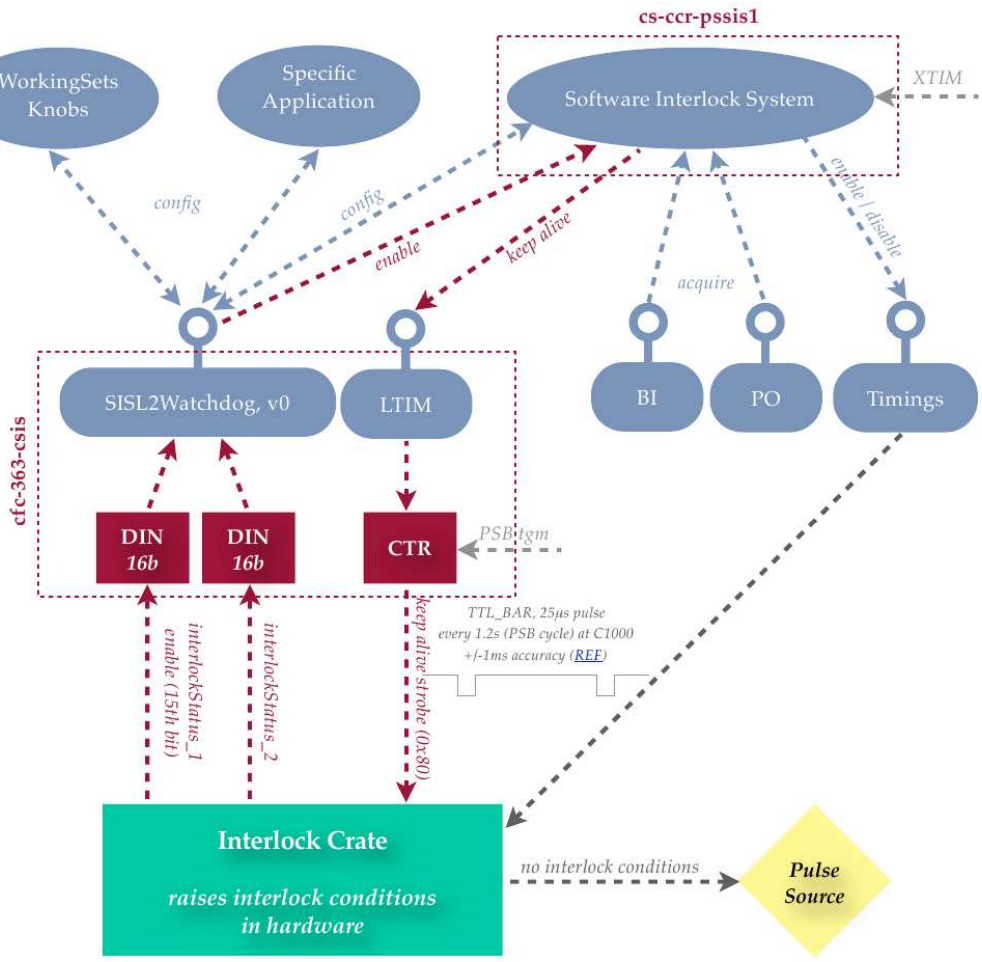


Figura 4.1: Arquitectura del nuevo sistema

Originalmente parecía que iba a ser una clase muy simple y fácil de implementar, pero el sistema que se esta reemplazando con esta clase era muy antiguo y sin documentación. Entender el funcionamiento de las señales del *Interlock crate* ha requerido mucho trabajo, y ha supuesto cambiar varias veces el diseño y las especificaciones sobre las que estaba trabajando y rehacer la clase.

Nuestro primer objetivo no era diseñar la clase para usar el LTIM, sino que pretendíamos usar una única clase para todo usando los módulos de la familia DIO para ello. Tras varias pruebas realizadas mediante un osciloscopio y programas de test que programé específicamente para ello, descubrí que estos módulos no eran capaces de trabajar a las velocidades requeridas por este sistema, por lo que para producir la señal de *keep alive* decidimos optar por un hardware diferente dedicado a producir señales sincronizadas, y utilizar la clase LTIM para controlarlo.

A mi salida del CERN este sistema estaba funcionando en paralelo con el sistema antiguo, en proceso de prueba para verificar que todo funciona bien y decidir si se puede reemplazar el sistema antiguo por el desarrollado en el PFC.

4.4. OasisCursor

Mi primera función en el CERN fue desarrollar la clase OasisCursor, para así familiarizarme con FESA. Esta clase se enmarca dentro del proyecto OASIS (Sección 2.4), el cual no usa COHAL para acceder al hardware, sino una librería propia diseñada con anterioridad.

Esta clase recibe datos de OASIS a través de sus bibliotecas de dispositivos tales como osciloscopios instalados en el mismo Front-End o en otros. Sobre las señales recibidas calcula varios datos: valor máximo, valor mínimo, media e integral. Puede elegirse la ventana sobre las señales en las que se quieren calcular estos datos y configurar si se quiere calcular cada cierto número de señales recibidas o si se quiere que se calcule cada vez que llega una señal de sincronización del *timing* del acelerador.

Además cada vez que se reciben datos se puede recibir una o varias señales en el mismo paquete, dependiendo de cómo este configurada la adquisición de datos del módulo. Para los casos en que se reciban varias señales se han implementado dos algoritmos, uno que primero hace la media de las señales y luego calcula los datos, y otro que calcula dos datos para cada señal y hace el promedio.

Aunque partí de una clase ya diseñada, tuve que realizar varias modificaciones tanto a lo largo del desarrollo como en su despliegue durante los meses posteriores. Al comunicarme con los futuros usuarios de esta clase fui descubriendo las partes del diseño original que había que retocar para que esta clase se adaptara correctamente a sus necesidades.

Al principio se implementó un prototipo rápido, sin pensar en optimizaciones y usando *arrays* de C. Esto suponía un uso excesivo de memoria debido al

tamaño variable de las señales, y unos algoritmos poco eficientes. En una de las instalaciones de la clase se descubrió que ésta no era capaz de recoger datos a la velocidad a la que le llegaban. Por ello desarrollé unos programas de prueba que medían el tiempo consumido para procesar cada señal y optimicé el código. La optimización consistió en sustituir los *arrays* por *std::vector* intentando pasar en todos los lugares posibles el vector como referencia, con lo que conseguí una disminución drástica en el consumo de memoria. También rediseñé los algoritmos para minimizar los cálculos, con lo que conseguí reducir el tiempo a la quinta parte.

4.5. OasisRdaClient

La clase OasisRdaClient es un ejemplo de cómo programar un cliente de RDA, protocolo usado por OASIS (Sección 2.4) para intercomunicar los productores de las señales (osciloscopios) con los receptores de las mismas (clases FESA). Esta clase ya existía cuando se comenzó el PFC, pero tenía serios problemas de rendimiento.

Mi trabajo con ella consistió en optimizarla, aplicando las mismas técnicas que había usado con OasisCursor, como medir rendimientos con los programas de prueba, usar estructuras de datos dinámicas y mejorar los algoritmos.

Capítulo 5

Conclusiones

Tal como se señaló en la introducción de esta Memoria, la Tabla 5.1 resume el conjunto de las principales tareas concretas realizadas, indicando su naturaleza y el nivel de complejidad.

El Proyecto ha supuesto echar mano de recursos y habilidades muy diferentes, moviéndome entre niveles muy altos y abstractos como puede ser el diseño de una jerarquía de clases hasta el nivel físico que requiere la utilización del osciloscopio para estudiar o verificar el comportamiento de las señales, pasando por la complejidad inherente a lo que se suele denominar programación de sistemas.

Proyecto	Apartado	Módulo	Tareas realizadas	Dificultad
COHAL	Generic		Rediseño de partes	baja
	Digital Input Output	ChannelAddress	Programación desde cero Escritura de documentación	media
		ICV196	Programación desde cero Escritura documentación Estudio dispositivo Colaboración documentación dispositivo Diseño de pruebas	alta
		VMOD-TTL	Programación desde cero Escritura documentación Estudio dispositivo Colaboración *driver* Diseño de pruebas	alta
		VMOD-DOR	Programación desde cero Escritura documentación Estudio dispositivo Colaboración *driver* Diseño de pruebas	media
	Analog Input	VMOD-12E16	Programación desde cero Escritura documentación Estudio dispositivo Diseño de pruebas	alta
	Analog Output	VMOD-12A2	Programación desde cero Escritura documentación Estudio dispositivo Diseño de pruebas	media
		VMOD-16A2	Programación desde cero Escritura documentación Estudio dispositivo Diseño de pruebas	media
FESA	CGAI		Programación desde cero Diseño de pruebas Instalación en producción	alta
	CGDIO		Programación desde cero	baja
	SISL2Watchdog		Programación desde cero Diseño de pruebas Instalación en producción	alta
	OasisCursor		Programación desde cero Diseño de pruebas Optimización de código	alta
	OasisRdaClient		Optimización de código	baja

Cuadro 5.1: Tareas realizadas

Bibliografía

[gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[stroustrup04] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley 2004.

[stroustrup94] Bjarne Stroustrup. *The Design and Evolution of C++*. dwAddison-Wesley 1994.

[musser01] David R. Musser, Guillmer J. Derge, Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley 2001.

[prata05] Stephen Prata. *C++ Primer Plus*. Sams 2005.

Apéndice A

COHAL Reference Manual

A.1. COHAL Hierarchical Index

A.1.1. COHAL Class Hierarchy

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

A.2. COHAL Class Index

A.2.1. COHAL Class List

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

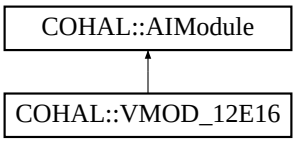
A.3. COHAL Class Documentation

A.3.1. COHAL::AIModule Class Reference

Common interface for Analog Input modules.

`#include <AIModule.h>`

Inheritance diagram for COHAL::AIModule::



Public Member Functions

- AIModule (const HWAddress &hwAddress)

  Constructor of the class.

- virtual ~AIModule ()

  Destructor of the class.

- virtual void init (const ModuleConfig &initParams)=0

  Initialize the module.

- virtual std::string getHwVersion () const =0

  Get hardware version.

- virtual void reset ()=0

  reset the module

- virtual void getInputRange (double &min, double &max)=0

  Get the input range of the module.

- virtual double getData (unsigned long channel, long &rawData)=0

  Read data on the input of the module.

- virtual unsigned long getBitResolution (unsigned long channel)=0

  bit resolution of the analog input

Static Public Attributes

- static ModuleFactory< AIModule > Factory

  Analog Input Factory.

A.3.1.1. Detailed Description

Common interface for Analog Input modules.

A.3.1.2. Constructor & Destructor Documentation

COHAL::AIModule::AIModule (const HWAddress & *hwAddress*) `[inline]`

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

hwAddress hardware address class of the module

A.3.1.3. Member Function Documentation

virtual unsigned long COHAL::AIModule::getBitResolution (unsigned long *channel*) `[pure virtual]`

bit resolution of the analog input

Parameters:

channel number of the channel to use, the lowest one is 1

Returns:

number of bits of resolution

Implemented in COHAL::VMOD_12E16.

virtual double COHAL::AIModule::getData (unsigned long *channel*, long & *rawData*) `[pure virtual]`

Read data on the input of the module.

It is procesed with the input_range to return a meaningful value.

Parameters:

channel number of the channel to use, the lowest one is 1

rawData the data as is readed from the channel

Exceptions:

ModuleException if there is any problem reading the data

Returns:

the meaningful data on the input of the module (after apply the input_range)

Implemented in COHAL::VMOD_12E16.

virtual void COHAL::AIModule::getInputRange (double & *min*, double & *max*) `[pure virtual]`

Get the input range of the module.

Parameters:

min minimum value of voltage/ampere that the module can handle

max maximum value of voltage/ampere that the module can handle

Implemented in COHAL::VMOD_12E16.

virtual void COHAL::AIModule::init (const ModuleConfig & *initParams*) `[pure virtual]`

Initialize the module.

If the constructor was invoked without ModuleConfig this method must be call before use the module. In other case the constructor will initialize the module and this method should not be use.

Parameters:

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

Implemented in COHAL::VMOD_12E16.

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

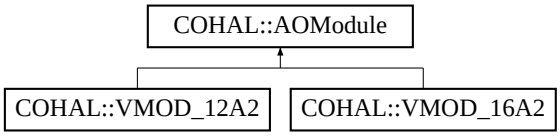
- ai/src/AIModule.h
- ai/src/AIModule.cpp

A.3.2. COHAL::AOModule Class Reference

Common interface for Analog Output modules.

`#include <AOModule.h>`

Inheritance diagram for COHAL::AOModule::



Public Member Functions

- AOModule (const HWAddress &hwAddress)
  Constructor of the class.
- virtual ∼AOModule ()
  Destructor of the class.
- virtual void init (const ModuleConfig &initParams)=0
  Initialize the module.
- virtual std::string getHwVersion () const =0
  Get hardware version.
- virtual void reset ()=0
  reset the module
- virtual void getInputRange (double &min, double &max)=0
  Get the input range of the module.
- virtual void setData (double data, unsigned long channel)=0
  Set data.
- virtual double readBackData (unsigned long channel)=0
  Read back data.
- virtual unsigned long getBitResolution (unsigned long channel)=0
  bit resolution of the analog output

Static Public Attributes

- static ModuleFactory< AOModule > Factory
  Analog Output Factory.

A.3.2.1. Detailed Description

Common interface for Analog Output modules.

A.3.2.2. Constructor & Destructor Documentation

COHAL::AOModule::AOModule (const HWAddress & *hwAddress*) `[inline]`

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

hwAddress hardware address class of the module

A.3.2.3. Member Function Documentation

virtual unsigned long COHAL::AOModule::getBitResolution (unsigned long *channel*) `[pure virtual]`

bit resolution of the analog output

Parameters:

channel number of the channel to use, the lowest one is 1

Returns:

number of bits of resolution

Implemented in COHAL::VMOD_12A2, and COHAL::VMOD_16A2.

virtual void COHAL::AOModule::getInputRange (double & *min*, double & *max*) `[pure virtual]`

Get the input range of the module.

Parameters:

min minimum value of voltage/ampere that the module can handle

max maximum value of voltage/ampere that the module can handle

Implemented in COHAL::VMOD_12A2, and COHAL::VMOD_16A2.

virtual void COHAL::AOModule::init (const ModuleConfig & *initParams*) `[pure virtual]`

Initialize the module.

If the constructor was invoked without ModuleConfig this method must be call before use the module. In other case the constructor will initialize the module and this method should not be use.

Parameters:

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

Implemented in COHAL::VMOD_12A2, and COHAL::VMOD_16A2.

virtual double COHAL::AOModule::readBackData (unsigned long *channel*) `[pure virtual]`

Read back data.

Read back the data set on the module. Some modules don't support this action and will throw an exception.

Parameters:

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if the module can not read data

Returns:

the voltage set on the module

Implemented in COHAL::VMOD_12A2, and COHAL::VMOD_16A2.

virtual void COHAL::AOModule::setData (double *data*, unsigned long *channel*) `[pure virtual]`

Set data.

Set data on the output of the module

Parameters:

data voltage to set

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if there is any problem setting the data

Implemented in COHAL::VMOD_12A2, and COHAL::VMOD_16A2.

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

- ao/src/AOModule.h
- ao/src/AOModule.cpp

A.3.3. COHAL::DIOChannelAddress Class Reference

Channel Address.

`#include <DIOModule.h>`

Public Member Functions

- DIOChannelAddress (unsigned long startChannel, unsigned long endChannel)

  Constructor of the class.

- DIOChannelAddress (unsigned long channel)

  Constructor of the class.

- DIOChannelAddress (const DIOChannelAddress &address)

  Copy constructor.

A.3.3.1. Detailed Description

Channel Address.

Defines a range of channels. If start == end then only one channel is use. The lower valid channel number is 1.

A.3.3.2. Constructor & Destructor Documentation

DIOChannelAddress::DIOChannelAddress (unsigned long *startChannel*, unsigned long *endChannel*)

Constructor of the class.

Parameters:

startChannel the first channel used

endChannel the last channel used

DIOChannelAddress::DIOChannelAddress (unsigned long *channel*)

Constructor of the class.

Create a Channel Address with a single channel

Parameters:

channel the channel used

DIOChannelAddress::DIOChannelAddress (const DIOChannelAddress & *address*)

Copy constructor.

Initialice the class with another channel address

Parameters:

address DIOChannelAddress

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

- dio/src/DIOModule.h
- dio/src/DIOModule.cpp

A.3.4. COHAL::DIOChannelConfig Class Reference

Channel configuration of DIO family.

`#include <DIOModule.h>`

Public Types

- type_none = 0

  channel not configured

- type_in
  input channel
- type_out
  output channel
- logic_ttl = 0
  channel with TTL logic
- logic_ttlbar
  channel with TTL_BAR logic
- enum channel_type { type_none = 0, type_in, type_out }
  Type of the channel.
- enum channel_logic { logic_ttl = 0, logic_ttlbar }
  Channel logic.

Public Member Functions

- DIOChannelConfig (channel_type t=type_none, channel_logic l=logic_ttl)
  Constructor of the class.
- DIOChannelConfig (const DIOChannelConfig &config)
  Copy constructor.
- void setType (channel_type t)
  set channel type
- void setLogic (channel_logic l)
  set channel logic

A.3.4.1. Detailed Description

Channel configuration of DIO family.

A.3.4.2. Member Enumeration Documentation

enum COHAL::DIOChannelConfig::channel_logic

Channel logic.

Logic of the bits.

Enumerator:

logic_ttl channel with TTL logic

logic_ttlbar channel with TTL_BAR logic

enum COHAL::DIOChannelConfig::channel_type

Type of the channel.

The channels can be setted to by input or output.

Enumerator:

type_none channel not configured

type_in input channel

type_out output channel

A.3.4.3. Constructor & Destructor Documentation

DIOChannelConfig::DIOChannelConfig (channel_type *t* = `type_none`, channel_logic *l* = `logic_ttl`)

Constructor of the class.

The default value of type is none (with it the channel will not be configured) the default value of resolution is 8 bits

Parameters:

t is the type of the channel (input or output)

DIOChannelConfig::DIOChannelConfig (const DIOChannelConfig & *config*)

Copy constructor.

Initialice the class with another config

Parameters:

config a DIOChannelConfig

A.3.4.4. Member Function Documentation

void DIOChannelConfig::setLogic (channel_logic *l*)

set channel logic

Parameters:

l the logic (ttl or ttlbar) of the channel

void DIOChannelConfig::setType (channel_type *t*)

set channel type

Parameters:

t the direction (input or output) of the channel

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

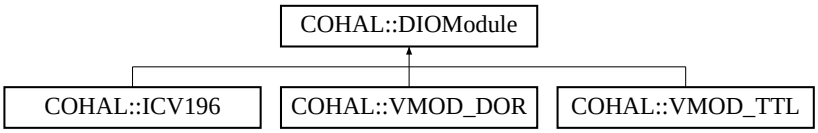
- dio/src/DIOModule.h
- dio/src/DIOModule.cpp

A.3.5. COHAL::DIOModule Class Reference

Common interface for Digital Input/Output modules.

`#include <DIOModule.h>`

Inheritance diagram for COHAL::DIOModule::



Public Member Functions

- DIOModule (const HWAddress &hwAddress)

  Constructor of the class.

- virtual ~DIOModule ()

  Destructor of the class.

- virtual void initChannel (const DIOChannelAddress &chAddress, const ChannelConfig &initParams)=0

  Initialize channel.

- virtual std::string getHwVersion () const =0

  Get hardware version.

- virtual void reset ()=0

  reset the module

- virtual void setData (const DIOChannelAddress &chAddress, unsigned long data)=0

  Set data.

- virtual void setBits (const DIOChannelAddress &chAddress, bool data, unsigned long bitMask)=0

  Set bits.

- virtual void triggerPulse (const DIOChannelAddress &chAddress, unsigned long bitMask, unsigned long long pulseWidth)=0

  trigger pulse

- virtual unsigned long getData (const DIOChannelAddress &chAddress)=0

  Receive data.

- virtual unsigned long readBackData (const DIOChannelAddress &chAddress)=0

  Read back data.

- virtual unsigned long getBitResolution (const DIOChannelAddress &chAddress)=0

  bit resolution of the analog input

Static Public Attributes

- static ModuleFactory< DIOModule > Factory
  Digital Input/Output Factory.

A.3.5.1. Detailed Description

Common interface for Digital Input/Output modules.

A.3.5.2. Constructor & Destructor Documentation

COHAL::DIOModule::DIOModule (const HWAddress & *hwAddress*) `[inline]`

Constructor of the class.

Parameters:

hwAddress hardware address class of the module

Exceptions:

ModuleException if there is any problem initializing the module

A.3.5.3. Member Function Documentation

virtual unsigned long COHAL::DIOModule::getBitResolution (const DIOChannelAddress & *chAddress*) `[pure virtual]`

bit resolution of the analog input

Parameters:

chAddress channel to use

Returns:

number of bits of resolution

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual unsigned long COHAL::DIOModule::getData (const DIOChannelAddress & *chAddress*) `[pure virtual]`

Receive data.

Read data from the input of the module

Parameters:

chAddress channels to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_in

Returns:

the digital value on the input channel

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual void COHAL::DIOModule::initChannel (const DIOChannelAddress & *chAddress*, const ChannelConfig & *initParams*) `[pure virtual]`

Initialize channel.

Before use it every channel needs to be initialized. If the resolution configured needs more than one physical channel it will start from the channel and use the following channels.

Parameters:

chAddress to configure

initParams configuration of the channel

Exceptions:

ModuleException if there is any problem initializing the channel

ModuleException if the channel was already configured with diferent ChannelConfig

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual unsigned long COHAL::DIOModule::readBackData (const DIOChannelAddress & *chAddress*) `[pure virtual]`

Read back data.

Read back the last data put on the output of the module

Parameters:

chAddress channel to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

Returns:

the digital value on the output channel

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual void COHAL::DIOModule::setBits (const DIOChannelAddress & *chAddress*, bool *data*, unsigned long *bitMask*) `[pure virtual]`

Set bits.

Set certain bits on the output of the module

Parameters:

chAddress channels to use

data value to set on each bit of the mask

bitMask mask of bits to be set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual void COHAL::DIOModule::setData (const DIOChannelAddress & *chAddress*, unsigned long *data*) `[pure virtual]`

Set data.

Set data on the output of the module

Parameters:

chAddress channels to use

data digital value to set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if data is bigger than the chAddress capacity

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

virtual void COHAL::DIOModule::triggerPulse (const DIOChannelAddress & *chAddress*, unsigned long *bitMask*, unsigned long long *pulseWidth*) `[pure virtual]`

trigger pulse

Create a pulse of about pulseWidth length in microseconds. The pulse will never be shorter than pulseWidth, but could be longer than that.

From experimental observation: On L865 the maximum error expected is less than 6 microseconds On ppc4 the maximum error expected is less than 15000 microseconds

Parameters:

chAddress channels to use

bitMask mask of bits to be set

pulseWidth the time length of the pulse in microseconds

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implemented in COHAL::ICV196, COHAL::VMOD_DOR, and COHAL::VMOD_TTL.

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

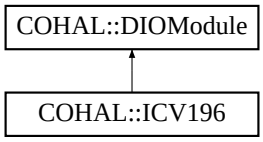
- dio/src/DIOModule.h
- dio/src/DIOModule.cpp

A.3.6. COHAL::ICV196 Class Reference

ICV196 class.

`#include <ICV196.h>`

Inheritance diagram for COHAL::ICV196::



Public Member Functions

- ICV196 (unsigned long lun, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- ICV196 (const HWAddress &hwAddress, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- ∼ICV196 ()

  Destructor of the class.

- void initChannel (const DIOChannelAddress &chAddress, const ChannelConfig &initParams)

  Initialize channel.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void setData (const DIOChannelAddress &chAddress, unsigned long data)

  Set data.

- void setBits (const DIOChannelAddress &chAddress, bool data, unsigned long bitMask)

  Set bits.

- void triggerPulse (const DIOChannelAddress &chAddress, unsigned long bitMask, unsigned long long pulseWidth)

  trigger pulse

- unsigned long getData (const DIOChannelAddress &chAddress)

  Receive data.

- unsigned long readBackData (const DIOChannelAddress &chAddress)

  Read back data.

- unsigned long getBitResolution (const DIOChannelAddress &chAddress)

  bit resolution of the analog input

A.3.6.1. Detailed Description

ICV196 class.

The ICV196 is a 12 channels with 8 bit digital input/output board.

The channels can be configured as 8, 16 or 32 bits. They are addressed by the first physical channel used.

The channels are inverted logic (1=0V, 0=5V) on the hardware, cohal hides that so all the devices of the DIO family will use high voltage for 1 and low voltage for 0.

A.3.6.2. Constructor & Destructor Documentation

ICV196::ICV196 (unsigned long *lun*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

ICV196::ICV196 (const HWAddress & *hwAddress*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

A.3.6.3. Member Function Documentation

unsigned long ICV196::getBitResolution (const DIOChannelAddress & *chAddress*) `[virtual]`

bit resolution of the analog input

Parameters:

chAddress channel to use

Returns:

number of bits of resolution

Implements COHAL::DIOModule.

unsigned long ICV196::getData (const DIOChannelAddress & *chAddress*) `[virtual]`

Receive data.

Read data from the input of the module

Parameters:

chAddress channels to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_in

Returns:

the digital value on the input channel

Implements COHAL::DIOModule.

void ICV196::initChannel (const DIOChannelAddress & *chAddress*, const ChannelConfig & *initParams*) `[virtual]`

Initialize channel.

Before use it every channel needs to be initialized. If the resolution configured needs more than one physical channel it will start from the channel and use the following channels.

Parameters:

chAddress to configure

initParams configuration of the channel

Exceptions:

ModuleException if there is any problem initializing the channel

ModuleException if the channel was already configured with diferent ChannelConfig

Implements COHAL::DIOModule.

unsigned long ICV196::readBackData (const DIOChannelAddress & *chAddress*) `[virtual]`

Read back data.

Read back the last data put on the output of the module

Parameters:

chAddress channel to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

Returns:

the digital value on the output channel

Implements COHAL::DIOModule.

void ICV196::setBits (const DIOChannelAddress & *chAddress*, bool *data*, unsigned long *bitMask*) `[virtual]`

Set bits.

Set certain bits on the output of the module

Parameters:

chAddress channels to use

data value to set on each bit of the mask

bitMask mask of bits to be set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void ICV196::setData (const DIOChannelAddress & *chAddress*, unsigned long *data*) `[virtual]`

Set data.

Set data on the output of the module

Parameters:

chAddress channels to use

data digital value to set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if data is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void ICV196::triggerPulse (const DIOChannelAddress & *chAddress*, unsigned long *bitMask*, unsigned long long *pulseWidth*) `[virtual]`

trigger pulse

Create a pulse of about pulseWidth length in microseconds. The pulse will never be shorter than pulseWidth, but could be longer than that.

From experimental observation: On L865 the maximum error expected is less than 6 microseconds On ppc4 the maximum error expected is less than 15000 microseconds

Parameters:

chAddress channels to use

bitMask mask of bits to be set

pulseWidth the time length of the pulse in microseconds

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implements COHAL::DIOModule.

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

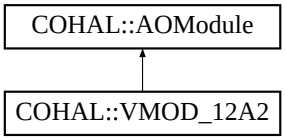
- dio/src/ICV196.h
- dio/src/ICV196.cpp

A.3.7. COHAL::VMOD_12A2 Class Reference

VMOD 12A2 class.

`#include <VMOD_12A2.h>`

Inheritance diagram for COHAL::VMOD_12A2::



Public Member Functions

- VMOD_12A2 (unsigned long lun, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_12A2 (const HWAddress &hwAddress, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_12A2 (unsigned long lun)

  Constructor of the class.

- VMOD_12A2 (const HWAddress &hwAddress)

  Constructor of the class.

- ∼VMOD_12A2 ()

  Destructor of the class.

- void init (const ModuleConfig &initParams)

  Initialize the module.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void getInputRange (double &min, double &max)

Get the input range of the module.

- void setData (double data, unsigned long channel)
  Set data.
- double readBackData (unsigned long channel)
  Read back data.
- unsigned long getBitResolution (unsigned long channel)
  bit resolution of the analog output

A.3.7.1. Detailed Description

VMOD 12A2 class.

The VMOD-12A2 is a 2 channel, 12 bit digital to analog converters, output board.

A.3.7.2. Constructor & Destructor Documentation

VMOD_12A2::VMOD_12A2 (unsigned long *lun*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_12A2::VMOD_12A2 (const HWAddress & *hwAddress*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

VMOD_12A2::VMOD_12A2 (unsigned long *lun*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_12A2::VMOD_12A2 (const HWAddress & *hwAddress*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

hwAddress hardware address class of the module

Exceptions:

ModuleException if there is any problem initializing the module

A.3.7.3. Member Function Documentation

unsigned long VMOD_12A2::getBitResolution (unsigned long *channel*) `[virtual]`

bit resolution of the analog output

Parameters:

channel number of the channel to use, the lowest one is 1

Returns:

number of bits of resolution

Implements COHAL::AOModule.

void VMOD_12A2::getInputRange (double & *min*, double & *max*) `[virtual]`

Get the input range of the module.

Parameters:

min minimum value of voltage/ampere that the module can handle

max maximum value of voltage/ampere that the module can handle

Implements COHAL::AOModule.

void VMOD_12A2::init (const ModuleConfig & *initParams*) `[virtual]`

Initialize the module.

If the constructor was invoked without ModuleConfig this method must be call before use the module. In other case the constructor will initialize the module and this method should not be use.

Parameters:

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

Implements COHAL::AOModule.

double VMOD_12A2::readBackData (unsigned long *channel*) `[virtual]`

Read back data.

Read back the data set on the module. Some modules don't support this action and will throw an exception.

Parameters:

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if the module can not read data

Returns:

the voltage set on the module

Implements COHAL::AOModule.

void VMOD_12A2::setData (double *data*, unsigned long *channel*) `[virtual]`

Set data.

Set data on the output of the module

Parameters:

data voltage to set

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if there is any problem setting the data

Implements COHAL::AOModule.

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

- ao/src/VMOD_12A2.h
- ao/src/VMOD_12A2.cpp

A.3.8. COHAL::VMOD_12A2ModuleConfig Class Reference

Configuration of VMOD-12A2 modules.

`#include <VMOD_12A2.h>`

Public Types

- input_range_neg10_10 = 0

  voltage range from -10 to 10

- input_range_0_10

  voltage range from 0 to 10

- input_range_neg5_5

  voltage range from -5 to 5

- enum input_range { input_range_neg10_10 = 0, input_range_0_10, input_range_neg5_5 }

  Input range on the channels of the module.

Public Member Functions

- VMOD_12A2ModuleConfig (input_range ran=input_range_neg10_10)

  Constructor of the class.

- VMOD_12A2ModuleConfig (const VMOD_12A2ModuleConfig &config)

  Constructor of the class.

Public Attributes

- input_range range

  input range of the module

A.3.8.1. Detailed Description

Configuration of VMOD-12A2 modules.

A.3.8.2. Member Enumeration Documentation

enum COHAL::VMOD_12A2ModuleConfig::input_range

Input range on the channels of the module.

Different hardware have different voltages as input configured by the jumpers.

Enumerator:

input_range_neg10_10 voltage range from -10 to 10

input_range_0_10 voltage range from 0 to 10

input_range_neg5_5 voltage range from -5 to 5

A.3.8.3. Constructor & Destructor Documentation

COHAL::VMOD_12A2ModuleConfig::VMOD_12A2ModuleConfig (input_range *ran* = `input_range_neg10_10`) `[inline]`

Constructor of the class.

The default value of voltage range if is not set is input_range_neg10_10,

Parameters:

ran is the input range that this module uses

COHAL::VMOD_12A2ModuleConfig::VMOD_12A2ModuleConfig (const VMOD_12A2ModuleConfig & *config*) `[inline]`

Constructor of the class.

Initialice the class with another config

Parameters:

config a VMOD_12A2ModuleConfig

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

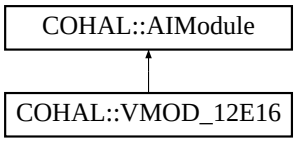
- ao/src/VMOD_12A2.h
- ao/src/VMOD_12A2.cpp

A.3.9. COHAL::VMOD_12E16 Class Reference

VMOD 12E16 class.

`#include <VMOD_12E16.h>`

Inheritance diagram for COHAL::VMOD_12E16::



Public Member Functions

- VMOD_12E16 (unsigned long lun, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_12E16 (const HWAddress &hwAddress, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_12E16 (unsigned long lun)

  Constructor of the class.

- VMOD_12E16 (const HWAddress &hwAddress)

  Constructor of the class.

- ∼VMOD_12E16 ()

  Destructor of the class.

- void init (const ModuleConfig &initParams)

  Initialize the module.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void getInputRange (double &min, double &max)

  Get the input range of the module.

- double getData (unsigned long channel, long &rawData)

  Read data on the input of the module.

- unsigned long getBitResolution (unsigned long channel)

  bit resolution of the analog input

A.3.9.1. Detailed Description

VMOD 12E16 class.

The VMOD-12E16 is a 16 channel (8 channel if configured as differential), 12 bit analog to digital converters, input board.

A.3.9.2. Constructor & Destructor Documentation

VMOD_12E16::VMOD_12E16 (unsigned long *lun*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_12E16::VMOD_12E16 (const HWAddress & *hwAddress*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

VMOD_12E16::VMOD_12E16 (unsigned long *lun*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

lun logic unit number of the module

VMOD_12E16::VMOD_12E16 (const HWAddress & *hwAddress*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

A.3.9.3. Member Function Documentation

unsigned long VMOD_12E16::getBitResolution (unsigned long *channel*) `[virtual]`

bit resolution of the analog input

Parameters:

channel number of the channel to use, the lowest one is 1

Returns:

number of bits of resolution

Implements COHAL::AIModule.

double VMOD_12E16::getData (unsigned long *channel*, long & *rawData*) `[virtual]`

Read data on the input of the module.

It is procesed with the input_range to return a meaningful value.

Parameters:

channel number of the channel to use, the lowest one is 1

rawData the data as is readed from the channel

Exceptions:

ModuleException if there is any problem reading the data

Returns:

the meaningful data on the input of the module (after apply the input_range)

Implements COHAL::AIModule.

void VMOD_12E16::getInputRange (double & *min*, double & *max*) `[virtual]`

Get the input range of the module.

Parameters:

min minimum value of voltage/ampere that the module can handle

max maximum value of voltage/ampere that the module can handle

Implements COHAL::AIModule.

void VMOD_12E16::init (const ModuleConfig & *initParams*) `[virtual]`

Initialize the module.

If the constructor was invoked without ModuleConfig this method must be call before use the module. In other case the constructor will initialize the module and this method should not be use.

Parameters:

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

Implements COHAL::AIModule.

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

- ai/src/VMOD_12E16.h
- ai/src/VMOD_12E16.cpp

A.3.10. COHAL::VMOD_12E16ModuleConfig Class Reference

Configuration of VMOD-12E16.

`#include <VMOD_12E16.h>`

Public Types

- input_range_neg10_10 = 0

  voltage range from -10 to 10

- input_range_0_10

  voltage range from 0 to 10

- input_range_neg5_5

  voltage range from -5 to 5

- enum input_range { input_range_neg10_10 = 0, input_range_0_10, input_range_neg5_5 }

  Input range on the channels of the module.

Public Member Functions

- VMOD_12E16ModuleConfig (input_range ran=input_range_neg10_10, bool isDiff=false)

  Constructor of the class.

- VMOD_12E16ModuleConfig (const VMOD_12E16ModuleConfig &config)

  Constructor of the class.

Public Attributes

- input_range range

  input range of the module

- bool differential

  are the channels differential

A.3.10.1. Detailed Description

Configuration of VMOD-12E16.

It handles information about the channel setup, if they are differential or not.

A.3.10.2. Member Enumeration Documentation

enum COHAL::VMOD_12E16ModuleConfig::input_range

Input range on the channels of the module.

Different hardware have different voltages as input configured by the jumpers.

Enumerator:

input_range_neg10_10 voltage range from -10 to 10

input_range_0_10 voltage range from 0 to 10

input_range_neg5_5 voltage range from -5 to 5

A.3.10.3. Constructor & Destructor Documentation

COHAL::VMOD_12E16ModuleConfig::VMOD_12E16ModuleConfig (input_range *ran* = input_range_neg10_10, bool *isDiff* = false) [inline]

Constructor of the class.

The default value of voltage range if is not set is input_range_neg10_10, the default value of differential if is not set is false

Parameters:

ran is the input range that this module uses

isDiff are the channels configured as differential

COHAL::VMOD_12E16ModuleConfig::VMOD_12E16ModuleConfig (const VMOD_12E16ModuleConfig & *config*) [inline]

Constructor of the class.

Initialice the class with another config

Parameters:

config a VMOD_12E16ModuleConfig

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

- ai/src/VMOD_12E16.h

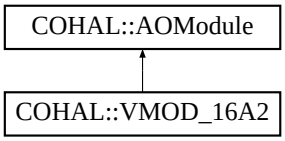
- ai/src/VMOD_12E16.cpp

A.3.11. COHAL::VMOD_16A2 Class Reference

VMOD 16A2 class.

`#include <VMOD_16A2.h>`

Inheritance diagram for COHAL::VMOD_16A2::



Public Member Functions

- VMOD_16A2 (unsigned long lun, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_16A2 (const HWAddress &hwAddress, const ModuleConfig &initParams)

  Constructor of the class.

- VMOD_16A2 (unsigned long lun)

  Constructor of the class.

- VMOD_16A2 (const HWAddress &hwAddress)

  Constructor of the class.

- ∼VMOD_16A2 ()

  Destructor of the class.

- void init (const ModuleConfig &initParams)

  Initialize the module.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void getInputRange (double &min, double &max)

  Get the input range of the module.

- void setData (double data, unsigned long channel)

  Set data.

- double readBackData (unsigned long channel)

  Read back data.

- unsigned long getBitResolution (unsigned long channel)

  bit resolution of the analog output

A.3.11.1. Detailed Description

VMOD 16A2 class.
The VMOD-16A2 is a 2 channel, 16 bit digital to analog converters, output board.

A.3.11.2. Constructor & Destructor Documentation

VMOD_16A2::VMOD_16A2 (unsigned long *lun*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_16A2::VMOD_16A2 (const HWAddress & *hwAddress*, const ModuleConfig & *initParams*)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

VMOD_16A2::VMOD_16A2 (unsigned long *lun*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

lun logic unit number of the module

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_16A2::VMOD_16A2 (const HWAddress & *hwAddress*)

Constructor of the class.

It won't initialize the module, the method init must be call before use the module.

Parameters:

hwAddress hardware address class of the module

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

A.3.11.3. Member Function Documentation

unsigned long VMOD_16A2::getBitResolution (unsigned long *channel*) `[virtual]`

bit resolution of the analog output

Parameters:

channel number of the channel to use, the lowest one is 1

Returns:

number of bits of resolution

Implements COHAL::AOModule.

void VMOD_16A2::getInputRange (double & *min*, double & *max*) `[virtual]`

Get the input range of the module.

Parameters:

min minimum value of voltage/ampere that the module can handle

max maximum value of voltage/ampere that the module can handle

Implements COHAL::AOModule.

void VMOD_16A2::init (const ModuleConfig & *initParams*) `[virtual]`

Initialize the module.

If the constructor was invoked without ModuleConfig this method must be call before use the module. In other case the constructor will initialize the module and this method should not be use.

Parameters:

initParams configuration of the module

Exceptions:

ModuleException if there is any problem initializing the module

Implements COHAL::AOModule.

double VMOD_16A2::readBackData (unsigned long *channel*) `[virtual]`

Read back data.

Read back the data set on the module. Some modules don't support this action and will throw an exception.

Parameters:

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if the module can not read data

Returns:

the voltage set on the module

Implements COHAL::AOModule.

void VMOD_16A2::setData (double *data*, unsigned long *channel*) `[virtual]`

Set data.

Set data on the output of the module

Parameters:

data voltage to set

channel number of the channel to use, the lowest one is 1

Exceptions:

ModuleException if there is any problem setting the data

Implements COHAL::AOModule.

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

- ao/src/VMOD_16A2.h
- ao/src/VMOD_16A2.cpp

A.3.12. COHAL::VMOD_16A2ModuleConfig Class Reference

Configuration of VMOD-16A2 modules.

`#include <VMOD_16A2.h>`

Public Types

- input_range_neg10_10 = 0

  voltage range from -10 to 10

- input_range_0_10

  voltage range from 0 to 10

- input_range_neg5_5

  voltage range from -5 to 5

- enum input_range { input_range_neg10_10 = 0, input_range_0_10, input_range_neg5_5 }

  Input range on the channels of the module.

Public Member Functions

- VMOD_16A2ModuleConfig (input_range ran=input_range_neg10_10)

  Constructor of the class.

- VMOD_16A2ModuleConfig (const VMOD_16A2ModuleConfig &config)

  Constructor of the class.

Public Attributes

- input_range range

input range of the module

A.3.12.1. Detailed Description

Configuration of VMOD-16A2 modules.

A.3.12.2. Member Enumeration Documentation

enum COHAL::VMOD_16A2ModuleConfig::input_range

Input range on the channels of the module.

Different hardware have different voltages as input configured by the jumpers.

Enumerator:

input_range_neg10_10 voltage range from -10 to 10

input_range_0_10 voltage range from 0 to 10

input_range_neg5_5 voltage range from -5 to 5

A.3.12.3. Constructor & Destructor Documentation

COHAL::VMOD_16A2ModuleConfig::VMOD_16A2ModuleConfig (input_range *ran* = `input_range_neg10_10`**)** `[inline]`

Constructor of the class.

The default value of voltage range if is not set is input_range_neg10_10,

Parameters:

ran is the input range that this module uses

COHAL::VMOD_16A2ModuleConfig::VMOD_16A2ModuleConfig (const VMOD_16A2ModuleConfig & *config*) `[inline]`

Constructor of the class.

Initialice the class with another config

Parameters:

config a VMOD_16A2ModuleConfig

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

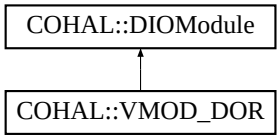
- ao/src/VMOD_16A2.h
- ao/src/VMOD_16A2.cpp

A.3.13. COHAL::VMOD_DOR Class Reference

VMOD DOR class.

`#include <VMOD_DOR.h>`

Inheritance diagram for COHAL::VMOD_DOR::



Public Member Functions

- VMOD_DOR (unsigned long lun, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- VMOD_DOR (const HWAddress &hwAddress, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- ~VMOD_DOR ()

  Destructor of the class.

- void initChannel (const DIOChannelAddress &chAddress, const ChannelConfig &initParams)

  Initialize channel.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void setData (const DIOChannelAddress &chAddress, unsigned long data)

  Set data.

- void setBits (const DIOChannelAddress &chAddress, bool data, unsigned long bitMask)

  Set bits.

- void triggerPulse (const DIOChannelAddress &chAddress, unsigned long bitMask, unsigned long long pulseWidth)

  trigger pulse

- unsigned long getData (const DIOChannelAddress &chAddress)

  Receive data.

- unsigned long readBackData (const DIOChannelAddress &chAddress)

  Read back data.

- unsigned long getBitResolution (const DIOChannelAddress &chAddress)

  bit resolution of the analog input

A.3.13.1. Detailed Description

VMOD DOR class.

The VMOD-DOR is a 2 channel with 8 bit digital output board.

Note: VMOD DOR has support for 4 channels of 4 bits, but here is not implemented. The readBack function is implemented by a buffer on memory, VMOD DOR don't have support for read back.

A.3.13.2. Constructor & Destructor Documentation

VMOD_DOR::VMOD_DOR (unsigned long *lun*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_DOR::VMOD_DOR (const HWAddress & *hwAddress*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

A.3.13.3. Member Function Documentation

unsigned long VMOD_DOR::getBitResolution (const DIOChannelAddress & *chAddress*) `[virtual]`

bit resolution of the analog input

Parameters:

chAddress channel to use

Returns:

number of bits of resolution

Implements COHAL::DIOModule.

unsigned long VMOD_DOR::getData (const DIOChannelAddress & *chAddress*) `[virtual]`

Receive data.

Read data from the input of the module

Parameters:

chAddress channels to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_in

Returns:

the digital value on the input channel

Implements COHAL::DIOModule.

void VMOD_DOR::initChannel (const DIOChannelAddress & *chAddress*, const ChannelConfig & *initParams*) `[virtual]`

Initialize channel.

Before use it every channel needs to be initialized. If the resolution configured needs more than one physical channel it will start from the channel and use the following channels.

Parameters:

chAddress to configure

initParams configuration of the channel

Exceptions:

ModuleException if there is any problem initializing the channel

ModuleException if the channel was already configured with diferent ChannelConfig

Implements COHAL::DIOModule.

unsigned long VMOD_DOR::readBackData (const DIOChannelAddress & *chAddress*) `[virtual]`

Read back data.

Read back the last data put on the output of the module

Parameters:

chAddress channel to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

Returns:

the digital value on the output channel

Implements COHAL::DIOModule.

void VMOD_DOR::setBits (const DIOChannelAddress & *chAddress*, bool *data*, unsigned long *bitMask*) `[virtual]`

Set bits.

Set certain bits on the output of the module

Parameters:

chAddress channels to use

data value to set on each bit of the mask

bitMask mask of bits to be set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void VMOD_DOR::setData (const DIOChannelAddress & *chAddress*, unsigned long *data*) `[virtual]`

Set data.

Set data on the output of the module

Parameters:

chAddress channels to use

data digital value to set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if data is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void VMOD_DOR::triggerPulse (const DIOChannelAddress & *chAddress*, unsigned long *bitMask*, unsigned long long *pulseWidth*) `[virtual]`

trigger pulse

Create a pulse of about pulseWidth length in microseconds. The pulse will never be shorter than pulseWidth, but could be longer than that.

From experimental observation: On L865 the maximum error expected is less than 6 microseconds On ppc4 the maximum error expected is less than 15000 microseconds

Parameters:

chAddress channels to use

bitMask mask of bits to be set

pulseWidth the time length of the pulse in microseconds

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implements COHAL::DIOModule.

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

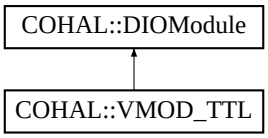
- dio/src/VMOD_DOR.h
- dio/src/VMOD_DOR.cpp

A.3.14. COHAL::VMOD_TTL Class Reference

VMOD TTL class.

`#include <VMOD_TTL.h>`

Inheritance diagram for COHAL::VMOD_TTL::



Public Member Functions

- VMOD_TTL (unsigned long lun, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- VMOD_TTL (const HWAddress &hwAddress, const ModuleConfig &initParams=ModuleConfig())

  Constructor of the class.

- ~VMOD_TTL ()

  Destructor of the class.

- void initChannel (const DIOChannelAddress &chAddress, const Channel-Config &initParams)

  Initialize channel.

- std::string getHwVersion () const

  Get hardware version.

- void reset ()

  reset the module

- void setData (const DIOChannelAddress &chAddress, unsigned long data)

  Set data.

- void setBits (const DIOChannelAddress &chAddress, bool data, unsigned long bitMask)

  Set bits.

- void triggerPulse (const DIOChannelAddress &chAddress, unsigned long bitMask, unsigned long long pulseWidth)

  trigger pulse

- unsigned long getData (const DIOChannelAddress &chAddress)

  Receive data.

- unsigned long readBackData (const DIOChannelAddress &chAddress)

  Read back data.

- unsigned long getBitResolution (const DIOChannelAddress &chAddress)

  bit resolution of the analog input

A.3.14.1. Detailed Description

VMOD TTL class.

The VMOD-TTL is a 2 channel with 8 bit digital input/output board.

A.3.14.2. Constructor & Destructor Documentation

VMOD_TTL::VMOD_TTL (unsigned long *lun*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

lun logic unit number of the module

Exceptions:

ModuleException if there is any problem initializing the module

VMOD_TTL::VMOD_TTL (const HWAddress & *hwAddress*, const ModuleConfig & *initParams* = `ModuleConfig()`)

Constructor of the class.

It will initialize the module.

Parameters:

initParams configuration of the module

hwAddress hardware address class of the module

Exceptions:

ModuleException if the type don't match

ModuleException if there is any problem initializing the module

A.3.14.3. Member Function Documentation

unsigned long VMOD_TTL::getBitResolution (const DIOChannelAddress & *chAddress*) `[virtual]`

bit resolution of the analog input

Parameters:

chAddress channel to use

Returns:

number of bits of resolution

Implements COHAL::DIOModule.

unsigned long VMOD_TTL::getData (const DIOChannelAddress & *chAddress*) `[virtual]`

Receive data.

Read data from the input of the module

Parameters:

chAddress channels to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_in

Returns:

the digital value on the input channel

Implements COHAL::DIOModule.

void VMOD_TTL::initChannel (const DIOChannelAddress & *chAddress*, const ChannelConfig & *initParams*) `[virtual]`

Initialize channel.

Before use it every channel needs to be initialized. If the resolution configured needs more than one physical channel it will start from the channel and use the following channels.

Parameters:

chAddress to configure

initParams configuration of the channel

Exceptions:

ModuleException if there is any problem initializing the channel

ModuleException if the channel was already configured with diferent ChannelConfig

Implements COHAL::DIOModule.

unsigned long VMOD_TTL::readBackData (const DIOChannelAddress & *chAddress*) `[virtual]`

Read back data.

Read back the last data put on the output of the module

Parameters:

chAddress channel to use

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

Returns:

the digital value on the output channel

Implements COHAL::DIOModule.

void VMOD_TTL::setBits (const DIOChannelAddress & *chAddress*, bool *data*, unsigned long *bitMask*) `[virtual]`

Set bits.

Set certain bits on the output of the module

Parameters:

chAddress channels to use

data value to set on each bit of the mask

bitMask mask of bits to be set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void VMOD_TTL::setData (const DIOChannelAddress & *chAddress*, unsigned long *data*) `[virtual]`

Set data.

Set data on the output of the module

Parameters:

chAddress channels to use

data digital value to set

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if data is bigger than the chAddress capacity

Implements COHAL::DIOModule.

void VMOD_TTL::triggerPulse (const DIOChannelAddress & *chAddress*, unsigned long *bitMask*, unsigned long long *pulseWidth*) `[virtual]`

trigger pulse

Create a pulse of about pulseWidth length in microseconds. The pulse will never be shorter than pulseWidth, but could be longer than that.

From experimental observation: On L865 the maximum error expected is less than 6 microseconds On ppc4 the maximum error expected is less than 15000 microseconds

Parameters:

chAddress channels to use

bitMask mask of bits to be set

pulseWidth the time length of the pulse in microseconds

Exceptions:

ModuleException if the channel is not COHAL::DIOChannelConfig::type_out

ModuleException if bitMask is bigger than the chAddress capacity

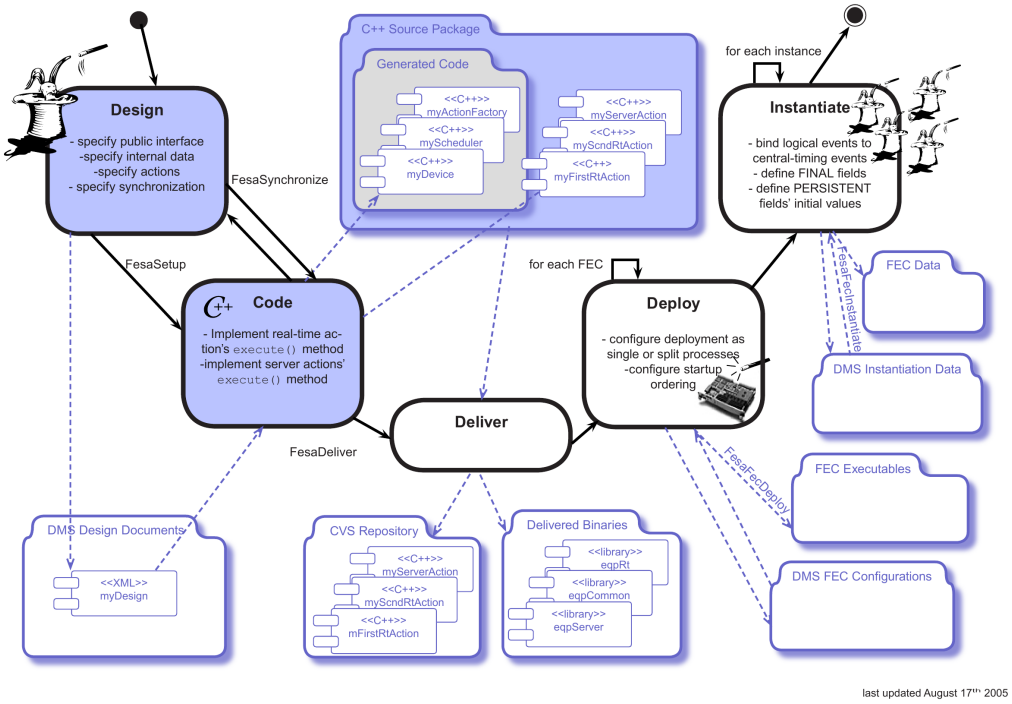
Implements COHAL::DIOModule.

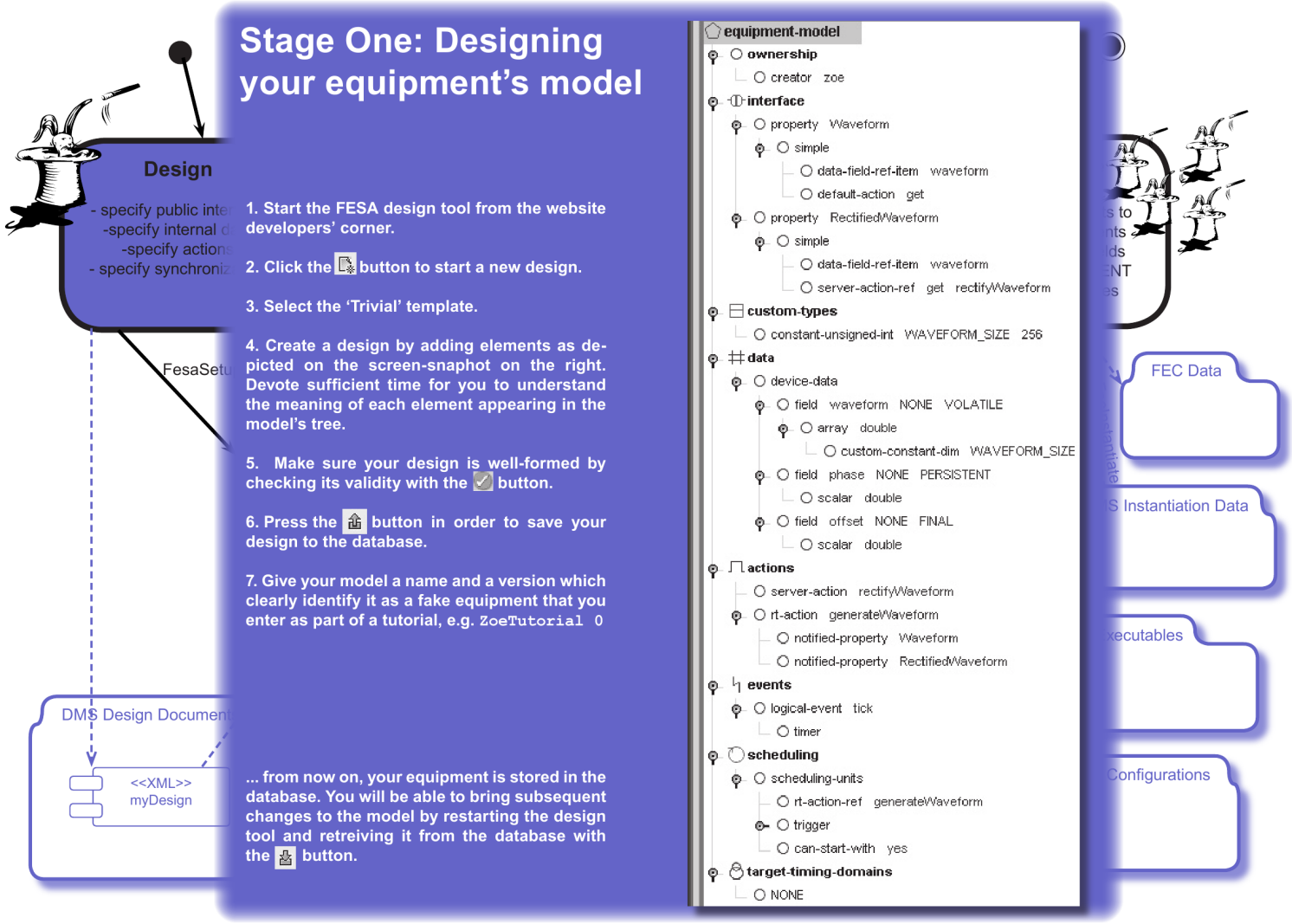
The documentation for this class was generated from the following files:

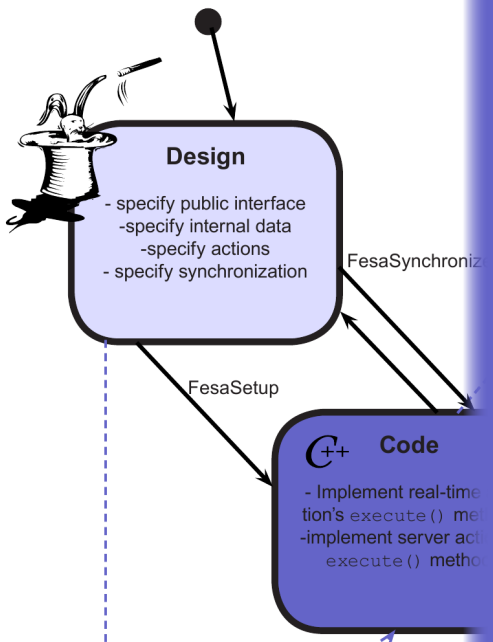
- dio/src/VMOD_TTL.h
- dio/src/VMOD_TTL.cpp

Apéndice B

Fesa Overview




Stage One: Designing your equipment's model
Design
- specify public inte
-specify internal d
-specify action
- specify synchroniz
FesaSetu
1. Start the FESA design tool from the website developers' corner.
2. Click the button to start a new design.
3. Select the 'Trivial' template.
4. Create a design by adding elements as depicted on the screen-snaphot on the right. Devote sufficient time for you to understand the meaning of each element appearing in the model's tree.
5. Make sure your design is well-formed by checking its validity with the button.
6. Press the button in order to save your design to the database.
7. Give your model a name and a version which clearly identify it as a fake equipment that you enter as part of a tutorial, e.g. ZoeTutorial 0
DMS Design Document
<<XML>>
myDesign
... from now on, your equipment is stored in the database. You will be able to bring subsequent changes to the model by restarting the design tool and retreiving it from the database with the button.
equipment-model
ownership
creator zoe
interface
property Waveform
simple
data-field-ref-item waveform
default-action get
property RectifiedWaveform
simple
data-field-ref-item waveform
server-action-ref get rectifyWaveform
custom-types
constant-unsigned-int WAVEFORM_SIZE 256
data
device-data
field waveform NONE VOLATILE
array double
custom-constant-dim WAVEFORM_SIZE
field phase NONE PERSISTENT
scalar double
field offset NONE FINAL
scalar double
actions
server-action rectifyWaveform
rt-action generateWaveform
notified-property Waveform
notified-property RectifiedWaveform
events
logical-event tick
timer
scheduling
scheduling-units
rt-action-ref generateWaveform
trigger
can-start-with yes
target-timing-domains
NONE
FEC Data
Instantiation Data
Executables
Configurations



Preliminary: Fesa Setup

Before being able to enter your specific C++ code, you must first prepare a local work environment on Linux: invoke `Fesa Setup` for initial code-generation (or `Fesa Synchronize` for future design iterations).

```
Zoe> mkdir fesa
Zoe> cd fesa
Zoe/fesa> Fesa Setup ZoeTutorial 0
                          scratch
Zoe/fesa> cd ZoeTutorial/v0
Zoe/fesa/ZoeTutorial/v0> make
```

Stage Two: Coding Server and RT Actions in C++

1. At the heart of your equipment's real-time activity, the `generateWaveform` real-time action class is meant (from your design) to be invoked each time a "tick" event occurs. You now need to enter the code of this action by filling-in the `generateWaveform::execute` method in the `generateWaveform.cpp` C++ implementation file.

```
void generateWaveform::execute(RTEvent * pEv){
    for (unsigned int i=0; i < deviceCollection.size(); i++){
        ZoeTutorialDevice* pDevice = deviceCollection[i];
        double phase = pDevice->phase.get();
        double offset = pDevice->offset.get();
        for (unsigned int i=0;i <WAVEFORM_SIZE; i++){
          pDevice->waveform.setCell(
            i, sin(2*3.14*i/100.0+phase)+offset );
        }
        pDevice->phase.set(phase+1.0);
        log << "device" << pDevice->name.get()
            << " has been woken-up by tick event"
            << " to generate its own waveform"
            << endDebug;
    }
}
```

2. You don't need to provide any piece of code for accessing remotely the device to be able to retreive its waveform since the design specified a default `get` action to serve the `Waveform` property. On the other hand, must supply code that performs data shaping in order to implement the RectifiedWaveform property. To this end you implement the `rectifyWaveform.cpp` file in a fashion similar to the above. The `pWorkingDevice` automatically points to the device actually which is the target of the remote-access request.

```
void rectifyWaveform::execute(RequestEvent * pEv){
  double pRectifiedWaveform[WAVEFORM_SIZE];
  for (unsigned int i=0; i<WAVEFORM_SIZE; i++) {
    double sample = pWorkingDevice->waveform.getCell(i);
    pRectifiedWaveform[i]=fabs(sample);
  }
  data.waveform.set( pRectifiedWaveform, WAVEFORM_SIZE );
  log  << pWorkingDevice->name.get() << " remotely accessed"
       <<endDebug;
}
```

Stage Three: Delivery to the central repository
1 In case you wish to deliver your equipment class to the central system in order to allow subsequent deployment on one of several front-end computers, invoke Fesa Deliver:
Zoe> cd ./fesa/ZoeTutorial/v0
Zoe/fesa/ZoeTutorial/v0> Fesa Deliver ZoeTutorial 0 ppc4
At this stage, the system also takes a snapshot of your C++ source code by commiting your files' changes to the central CVS repository.
2 Check that the CVS commit actually took place by browsing the equipment source-code repository which is accessible from the FESA web site.
C++ Source Package
Generated Code
<<C++>>
<<C++>>
for each instance
Instantiate
- bind logical events to central-timing events
- define FINAL fields
- define PERSISTENT fields' initial values
FesaDeliver
Deliver
DMS Design Documents
<<XML>>
myDesign
CVS Repository
<<C++>>
myServerAction
<<C++>>
mySendRtAction
<<C++>>
mFirstRtAction
Delivered Binaries
<<library>>
eqpRt
<<library>>
eqpCommon
<<library>>
eqpServer
FesaFecDeploy
FesaFecInstantiate
DMS FEC Configurations
FEC Executables
DMS Instantiation Data
FEC Data

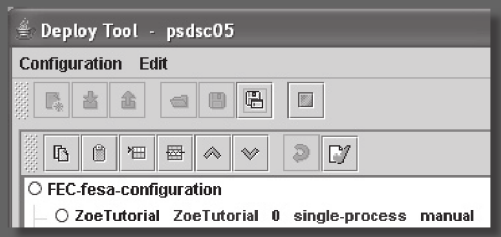
Stage Four: Deploying your class on a FEC

1. Before being able to create new instances of your newly developed equipment-class, you need to prepare one or several front-end computers that can host the class. To this end, you start the FESA Deployment Tool from the FESA website.

2. Retreive configuration of the target FEC, by pressing the button. Add an entry in the FEC's configuration as depicted below (in addition to your class, there might already be other classes deployed on the FEC, in which case you may decide to keep or remove them depending on whether they are actually needed).

3. Select the `single-process` deployment option which means both the server and real-time activities will run different threads within the same process.

4. Select the `manual` startup mode, unless you want the class to start automatically at boot.



4. On Linux, go to the local TEST directory within your C++ development tree.

5. Run `make`:

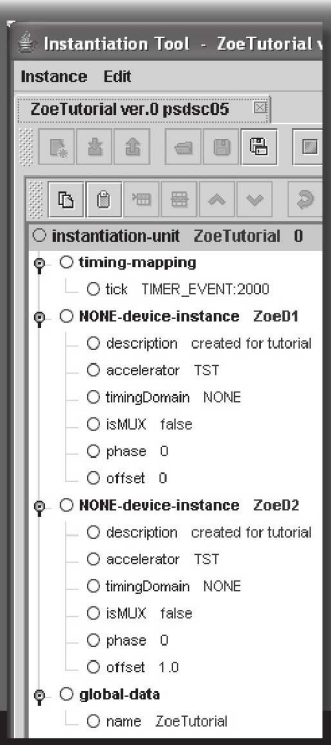
```
Zoe> cd ./fesa/ZoeTutorial/v0
Zoe/fesa/ZoeTutorial/v0> cd TEST
Zoe/fesa/ZoeTutorial/v0/TEST> make
```

Stage Five: Instantiation

1. You are now ready to create new instances of your class. In general, each time you would need to configure a new instance each time you physically connect a corresponding new hardware device to the front-end computer (through a dedicated electronic board or through a field-bus). To this end, launch the Instantiation Tool from the FESA web site.

2. For test purposes, create two new instances named (*) `ZoeD1` and `ZoeD2` and set different offset fields for each of them, as depicted on the right.

(*) Note that for operational instances, you would have to comply with the strict CERN standard for naming devices.



3. For test purposes, you can extract instantiation data from the data-base and into your local `TEST` directory. To this end, you invoke the `Fesa Instantiate` script:

```
Zoe> cd ./fesa/ZoeTutorial/v0
Zoe/fesa/ZoeTutorial/v0> cd TEST
Zoe/fesa/ZoeTutorial/v0/TEST> Fesa Instantiate ZoeTutorial 0 psdsc05 .
```

4. Everything is now ready for local testing (i.e. your local executable and the instantiation data for the front-end computer). Remote log to the targeted `ppccodv12` front-end computer and launch your equipment software:

```
Zoe> ssh psdsc05
psdsc05: cd /user/Zoe/fesa/ZoeTutorial/v0/TEST
psdsc05: ZoeTutorialSingleProcessServer &
```

5. Congratulations! You can now remotely access devices ZoeTutorialD1 and ZoeTutorialD2 accross the middleware. To this end, you may launch the Navigator Tool from the FESA website.