UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

FACULTAD DE FÍSICA
Departamento de Física de Partículas

Diseño y Programación orientados a Objetos de la Reconstrucción de Sucesos en el Experimento HADES de Colisiones Núcleo-Núcleo

Memoria presentada para
optar al Grado de Licenciado
en Ciencias Físicas por:

Manuel Sánchez García
Marzo, 1999

A mis padres

Agradecimientos

En Julio de 1997 el profesor J.A. Garzón me ofreció la posibilidad de integrarme en la colaboración HADES como miembro del grupo de Santiago. Gracias al profesor Garzón por ofrecerme esta oportunidad.

Quizá sea uno de los pocos españoles que está contento con sus jefes; pero quiero agradecer a Hans, Carmen y Nacho su paciencia y la libertad con que me han permitido trabajar. Especialmente la profesora Carmen Fernández, sin cuya dirección, guía y experimentado consejo este trabajo jamás se hubiera parecido a lo que hoy es.

También quiero dar las gracias a Antonio, sin el que no hubiera sobrevivido a mi primer mes en Alemania; Beatriz, nuestra ya legendaria administradora de las Alphas y principal betatester de HYDRA; Héctor, y su interminable lista de correcciones; Enrique, a quien por fin parece que le compilan las librerías del CERN; Pepe y Lola, oscilando cuánticamente entre España y Alemania, y mi primo Toño, que durante todos estos años de carrera me ha mantenido medianamente cuerdo.

Quiero dar las gracias también a todos los demás compañeros del departamento, mis amigos y familia, y en general a todos los que me han acompañado durante el último año y medio en los buenos y no tan buenos momentos por los que he pasado.

Para terminar quiero que en esta página de agradecimientos aparezcan las personas más importantes de mi vida, mis padres y mi hermana Paula. Todo lo que soy se lo debo a ellos.

Índice general

Índice de figuras

Capítulo 1

Introducción

El proyecto experimental HADES tiene por finalidad principal el estudio de la materia nuclear densa y caliente y las propiedades de los hadrones en la materia nuclear mediante experimentos de colisiones nucleares con haces de iones pesados, piones y protones. Concretamente, se investigarán las propiedades de los mesones vectoriales en la materia nuclear, midiendo el canal de desintegración de estos mesones en dileptones (pares electrón-positrón), en todo el intervalo de masa invariante hasta 1 GeV/c^2, con suficiente resolución para separar los mesones ρ, Φ y ω, temas sobre los que, hasta la fecha, hay muy poca información experimental.

El proyecto HADES fue propuesto por cerca de un centenar de científicos de 18 institutos diferentes de Alemania, Francia, Italia, Rusia, Polonia, Chipre, República Checa, República Eslovaca y España y fue aprobado en 1994. El Grupo Experimental de Núcleos y Partículas de la Universidad de Santiago de Compostela se unió a la Colaboración HADES a finales de 1995, contribuyendo desde entonces a distintos espectos de la preparación del espectrómetro.

El espectrómetro HADES está siendo instalado en el acelerador SIS del laboratorio GSI (Gesellschaft für Schwerionenforschung) de Darmstadt (Alemania). Se trata de un complejo conjunto de detectores con simetría cilíndrica alrededor del eje determinado por el haz de partículas incidente, dividido en seis sectores iguales, cada uno de los cuales consta básicamente de los siguientes detectores:

- Un detector Cherenkov RICH de gas para identificar los electrones.
- Cuatro módulos de cámaras de deriva MDC para la reconstrucción de las trayectorias de las partículas cargadas.

- Un imán toroidal superconductor para la determinación del momento de las partículas cargadas.
- Un detector de tiempo de vuelo TOF y un detector PreShower para la determinación de la multiplicidad de la colisión y la separación de leptones y hadrones.

Actualmente el proyecto se encuentra en una fase muy avanzada. El imán fue instalado en abril de 1998, todos los subdetectores han sido completamente diseñados y 2 de los 6 sectores del detector se encuentran prácticamente finalizados y han sido verificados con éxito bajo haz. Los prototipos de los módulos electrónicos necesarios han sido igualmente diseñados y comprobados. Dos de los seis sectores estarán finalizados en el verano de 1999, cuando comenzarán los primeros experimentos. Se prevé una duración mínima de 4 años de funcionamiento para el detector completo.

El programa de reconstrucción de un experimento es el conjunto de código y herramientas informáticas que permiten, a partir de los datos proporcionados por la electrónica del espectrómetro, elaborar toda la información con sentido e interés físico para su posterior análisis. Las señales recogidas en los detectores de HADES en las colisiones núcleo-núcleo y grabadas en soporte magnético deben sufrir un complejo procesado hasta ser transformadas en las magnitudes físicas de interés para el análisis de la colisión. En nuestro caso el fin principal es reconstruir la masa efectiva de los pares electrón-positrón producidos en la colisión núcleo-núcleo, pero no es el único. Se pueden obtener adicionalmente otros resultados físicos interesantes si se reconstruye la cantidad de movimiento y se identifican las partículas cargadas producidas en el vértice de la interacción.

Un aspecto fundamental del "software" del experimento HADES para la reconstrucción "offline" es la apuesta por las nuevas metodologías de Programación Orientada a Objetos. Después de muchos años en los que era el FORTRAN el lenguaje universal en la Física de Partículas y Física Nuclear, las innegables ventajas que ofrece la estructura en objetos para el tratamiento de problemas de muy alta complejidad hace de los lenguajes basados en dicha filosofía una sugestiva alternativa. Entre todos aquellos lenguajes informáticos orientados a objetos, el elegido para el desarrollo del "software" del experimento ha sido C++.

Una dificultad añadida en el desarrollo de los programas de reconstrucción es la escasez de librerías específicas para este tipo de Física en C++. Efectivamente, a lo largo de una treintena de años, el trabajo de los miles de investigadores en este campo en todo el mundo había producido una coleción de rutinas y librerías de programas, fundamentalmente en lenguaje FORTRAN. Dicha colección cubría

casi todos los aspectos presentes en un experimento típico: tratamiento dinámico de datos, reconstrucción, análisis, representación gráfica,... Tal carencia debe de suplirse, bien buscando las herramientas apropiadas entre las existentes en el mercado, bien desarrollando aquellas para las que no exista una alternativa razonable. En este sentido la colaboración HADES decidió desarrollar los programas de reconstrucción en el lenguaje C++ y la utilización del paquete ROOT de programas del CERN.

El diseño e implementación de la estructura de clases del programa de reconstrucción de HADES es el trabajo objeto de esta memoria. Ello conlleva la definición de clases, la definición de sus relaciones y operaciones y determinar sus atributos, así como sus relaciones de herencia. Esta tarea es de una importancia trascendental ya que de ella dependerá la versatilidad y potencia del programa. Un diseño incorrecto impediría sacarle a la filosofía de la Programación Orientada a Objetos las indudables ventajas que esta metodología aporta a la calidad del "software".

En el Capítulo 2 de esta memoria se describen en líneas generales los principales objetivos físicos para los que ha sido diseñado el experimento HADES, las principales características de los detectores que lo integran y la función de los distintos componentes del software necesario para obtener, monitorizar, procesar y analizar los datos.

El Capítulo 3 aborda el objetivo de este trabajo: el programa de reconstrucción de los sucesos de HADES. Previamente al análisis mismo del problema, el equipo de software tomó una serie de decisiones relativas a sistemas operativos, lenguajes, herramientas y métodos de programación a utilizar, que son explicadas al comienzo del capítulo. Ya que una de las decisiones fue utilizar programación orientada a objeto, los rasgos generales de este tipo de programación se exponen brevemente. Para el análisis y diseño de la solución adoptada para la reconstrucción de sucesos nos guiamos por el método de Booch, que se introduce y resume también en este capítulo. Los requerimientos del programa y sus posibles modos de utilización conducen a un diseño detallado que se traduce en un conjunto de clases que son implementadas en el lenguaje C++.

El capítulo 4 trata en detalle la implementación y utilización de las clases que constituyen el programa. Se describen las clases por grandes familias, que provienen en último término de los conceptos más básicos involucrados en el problema: datos tomados por los detectores, procedimientos de entrada/salida, parámetros de reconstrucción y algoritmos y procedimientos de reconstrucción. Hay una clase fundamental, llamada HADES, que contiene, coordina y proporciona métodos para manejar todos los conceptos anteriormente mencionados. Se han implementado además, clases para contener, organizar y gestionar los datos, clases para realizar la lectura y escritura de los datos, clases para contener, organizar y gestionar los parámetros

de la reconstrucción, y clases básicas para gestionar los algoritmos y procedimientos de reconstrucción. Se explican además, detalladamente, los pasos para inicializar el programa y algunos ejemplos de aplicación.

El resultado del trabajo que se describe en esta memoria es la primera versión de la estructura general del programa de reconstrucción. En el seno de esta estructura, los programadores usuarios especializados en cada uno de los detectores de HADES pueden implementar y utilizar modularmente clases específicas para contener los datos y parámetros de un detector y algoritmos para procesarlo. Pueden existir diferentes contenedores de datos y algoritmos de transformación para cada detector sin que ello afecte a la estructura general del programa. Como ejemplo de utilización del programa con datos reales se expone en el capítulo 5 la reconstrucción de sucesos en el detector de tiempo de vuelo TOF. Para realizarla se han implementado clases específicas para este detector, que son descritas. En Diciembre de 1998 se tomaron datos con este detector, que son reconstruídos a título de prueba del programa de reconstrucción.

Finalmente se exponen las principales conclusiones de este trabajo en el Capítulo 6. En una serie de Apéndices se añade información relativa a herramientas de programación utilizadas y se muestra parte de la documentación sobre las clases del programa producida para uso del experimento HADES.

Capítulo 2

El experimento Hades

La investigación de las propiedades de la materia nuclear en condiciones extremas (es decir, altas densidades y temperaturas) nos proporciona claves para el entendimiento de procesos como, por ejemplo, los que dieron lugar a la formación del Universo en el Big Bang, ya que en ese momento la interacción fuerte entre partículas tuvo lugar a muy altas temperaturas y densidades. Hoy podemos encontrarnos con condiciones cercanas dentro de estrellas de neutrones densas. Esta línea de investigación contribuye asimismo al estudio de uno de los principales objetivos de la física nuclear actual, como es obtener la ecuación de estado de la materia nuclear. Este conocimiento no sólo es importante para la Física Nuclear en sí, sino también para poder entender procesos astrofísicos que tienen lugar en la fase final de la evolución de las estrellas.

Cálculos basados en QCD, así como diversos modelos hadrónicos, predicen cambios detectables en la masa y anchura de resonancia de, por ejemplo, los mesones vectoriales, cuando éstos se producen y desintegran en el seno de materia hadrónica. Desde el punto de vista de QCD tales modificaciones pueden tener su raíz en la restauración de la simetría quiral del Lagrangiano a altas temperaturas y densidades.

Como consecuencia de estas predicciones, en la actualidad se desarrollan varios experimentos destinados al estudio de la materia nuclear densa y caliente. Uno de esos experimentos es HADES[24, 31, 35, 32, 1] (High Acceptance Dielectron Spectrometer), en el "Instituto para la Investigación en Iones Pesados" GSI[1], localizado en Darmstadt, Alemania.

[1]Las siglas GSI corresponden a Gesellshaft für Schwerionenforschung

2.1. La física de Hades

A la hora de observar los efectos anteriormente mencionados sobre las características de los mesones vectoriales producidos en materia nuclear densa, se puede pensar en diversas técnicas, todas ellas consistentes en producir reacciones con iones pesados y analizar distintas variables en el estado final. Así se han estudiado piones, kaones, mesones, antiprotones y dileptones. La técnica más en auge hoy en día y la usada en HADES consiste en estudiar el espectro de dileptones (e^+e^- en realidad) producidos por desintegraciones de mesones vectoriales en el medio nuclear. Sin embargo, el espectrómetro no está limitado a la detección y medida de leptones, sino que también puede ser usado para la detección de hadrones.

HADES trata de ver las modificaciones de las propiedades en el medio hadrónico de los mesones vectoriales a través de su canal de desintegración en electrón-positrón. Las ventajas de esta técnica son la existencia de sondas (mesones vectoriales) con vidas medias suficientemente cortas como para desintegrarse en la zona hadrónica que se forma durante la colisión y que tiene una vida media del orden de 10 fm/c. No ocurre así con, por ejemplo, los mesones pseudoescalares como el π^0 o el η^0; aunque éstos han sido estudiados en el canal de desintegración a fotones por la colaboración TAPS [16].

La ventaja de fijarse en pares dileptónicos es que éstos, una vez producidos, no sufren interacción hadrónica en el medio nuclear y "guardan memoria" de las condiciones con las que fueron producidos. El problema, en cambio, es la pequeña proporción de desintegraciones en el canal dileptónico con respecto a otros canales, lo que hace necesaria una elevada estadística y por tanto gran aceptancia en el espectrómetro así como una elevada intensidad en el haz usado para la realización del experimento. HADES proporcionará una aceptancia del 40 % y trabajará con intensidades máximas de 10^8 partículas por segundo.

Las carácteristicas de los mesones vectoriales que analizará HADES se pueden observar en la siguiente tabla:

Mesón	Masa $(\frac{MeV}{c^2})$	Anchura $(\frac{Mev}{c^2})$	cτ (fm)	Canal dominante	e^+e^- branching ratio
ρ	768	152	1.3	$\pi\pi$	$4{,}4 \times 10^{-5}$
ω	782	8.43	23.4	$\pi^+\pi^-\pi^0$	$7{,}2 \times 10^{-5}$
ϕ	1019	4.43	44.4	K^+K^-	$3{,}1 \times 10^{-4}$

Si nos fijamos en las vidas medias; el mejor candidato a ser utilizado como sonda es el mesón ρ; que tiene la desventaja de que su anchura de resonancia es

extremadamente grande ($\sim 150MeV$). Sin embargo, las predicciones también nos dicen que podemos esperar cambios en las anchuras de los mesones ω y ϕ en factores 5 y 15 respectivamente, a una densidad nuclear tres veces superior a la normal. Se pueden conseguir mesones ω y ϕ desintegrados en la materia nuclear entre aquellos que son producidos hacia atrás.

A la hora de hacer predicciones o de explicar los datos, existen diversos modelos; de los cuales podemos encontrar un buen par de revisiones en [34, 13]. A potencial químico 0, la disminución del condensado de quarks en el vacío al aumentar la temperatura, debido a la restauración de la simetría quiral en el lagrangiano de QCD, se ha observado en simulaciones de retículo en QCD así como en cálculos basados en la teoría de perturbaciones quiral y el modelo de gas de piones. La disminución del condensado de quarks puede llevar a una reducción de la masa de los hadrones en el medio nuclear, tal como se ve en estudios basados en reglas de suma en QCD [33]. A la hora de estudiar las colisiones de iones pesados desde un punto de vista teórico, se utilizan modelos de transporte como el modelo relativista de transporte, el modelo Walecka (QHD) o el modelo $\sigma - \omega$ no lineal.

En diversos laboratorios se han realizado experimentos anteriores a HADES que han trabajado sobre éstos temas a más altas energías; como CERES[21, 14, 22], HELIOS1-3[25]. En todos los casos se encuentra un incremento en la producción de electrones frente a lo que se esperaría de la superposición de reacciones elementales; pero los datos obtenidos no cuentan con la suficiente resolución como para sacar conclusiones definitivas. Tampoco la colaboración EPoS II[15] aporta resultados concluyentes.

En el régimen de energías de HADES, los únicos datos existentes son los de la colaboración DLS[12, 29, 23, 19, 30] en el BEVALAC de Berkeley. La principal aportación de esta colaboración ha sido demostrar la posibilidad de medir el espectro de dileptones; sin embargo la baja aceptancia y estadística del espectrómetro impide hacer un estudio eficaz. Además la resolución en masa invariante es insuficiente para separar los picos ρ y ω. Las conclusiones no son claras ya que los datos publicados últimamente por la colaboración DLS entran en contradicción con otros publicados anteriormente así como con cálculos teóricos publicados en los últimos años.

2.2. El acelerador

El experimento HADES se lleva a cabo en el instituto alemán GSI, localizado en la ciudad de Darmstadt. El instituto cuenta con un sistema de aceleradores de iones pesados formado fundamentalmente por tres componentes: el UNILAC, que es un

acelerador lineal; el sincrotrón de iones pesados SIS y el anillo de almacenamiento ESR. En la figura 2.1 se puede ver un esquema de los aceleradores.

El UNILAC se usa tanto para suministrar iones a experimentos a bajas energías como en el papel de preacelerador para el SIS. A su vez, a la salida del SIS, el haz puede llevarse a algún experimento, al separador de fragmentos FRS o al ESR. El ESR permite enfriar el haz y utilizarlo para experimentos internos en el anillo. Aunque también se puede llevar el haz enfriado a algún experimento o reinyectarlo en el SIS.

El UNILAC constituye el primer acelerador del GSI y fue construído en 1975. Este acelerador está diseñado para acelerar iones con una carga positiva hasta +10, dado que cuando fue construído éste era el número máximo de electrones que se esperaba poder quitar a un átomo de Uranio en la fuente de iones. Al comienzo de la línea del haz hay dos inyectores, norte y sur, que contienen las fuentes de iones de donde son extraídos los átomos ionizados con la ayuda de un campo eléctrico.

La primera parte del UNILAC la compone un acelerador tipo Wideröe, que mide 30m de longitud y contiene en torno a 130 electrodos aceleradores. La frecuencia de operación del campo eléctrico es de 27MHz asegurando una polaridad correcta cada vez que los iones se encuentran entre los electrodos. Todos los tipos de iones salen del Wideröe a una energía de 1.4 MeV por nucleón. A continuación los iones interaccionan con un gas, perdiendo más electrones; hasta alcanzar estados de ionización tales como el 28+ en Uranio. Recientemente la primera parte del UNILAC ha sufrido una profunda actualización, sustituyéndose el acelerador Wideröe por otro.

La penúltima parte del UNILAC la constituye un acelerador del tipo Álvarez de 55m de longitud. La frecuencia de operación de las cavidades de radiofrecuencia es de 108 Mhz. En cuanto a la energía de salida, se llega a los 11.6 MeV por nucleón para todos los iones. Por último el UNILAC cuenta con una estructura formada por 15 resonadores; cada uno de los cuales cuenta sólo con un hueco alimentado por un generador de radiofrecuencia ajustable; permitiendo así acelerar o decelerar los iones para que salgan con energías que van de los 2 a los 18 MeV por nucleón. Aparte de toda esta estructura, en 1992 se colocó un nuevo inyector entre el Wideröe y el Álvarez que permite cambiar el tipo de iones acelerados en un tiempo inferior a $15ms$; ésto junto con otras mejoras que permiten cambiar la energía de salida en menos de 15ms hacen posible elegir de forma independiente la energía e ión usados para experimentos en el UNILAC e inyección en el SIS.

El SIS, como ya se ha dicho, es un sincrotrón con una circunferencia de 216 metros; 24 imanes de curvatura y 36 lentes magnéticas. Antes de entrar en el SIS,

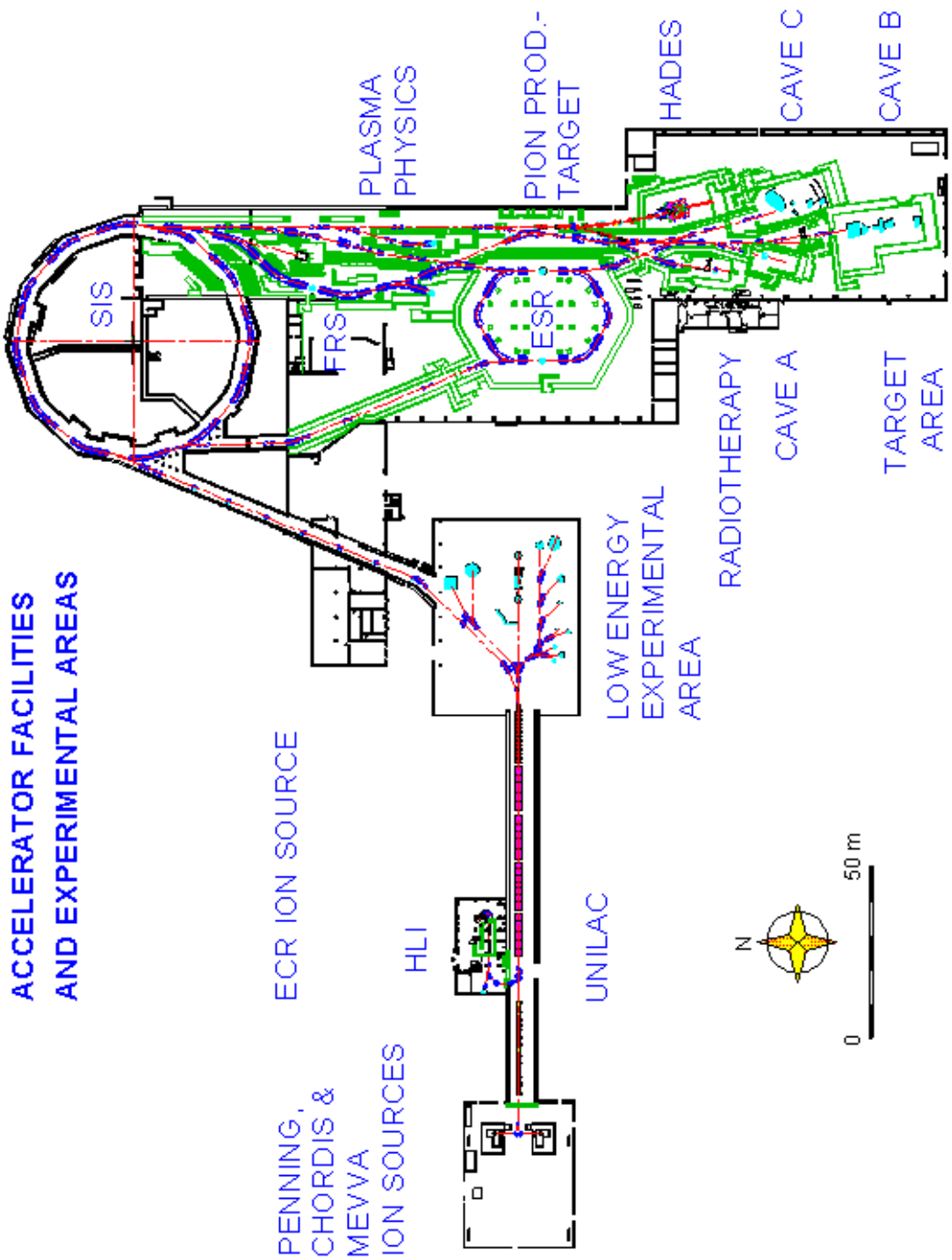


Figura 2.1: Vista general del GSI

los iones interaccionan con una hoja de carbón alcanzando así estados de ionización hasta 72+ en el caso del Uranio. En estos casos, el SIS permite llegar a energías de 1GeV por nucleón; en el caso de elementos ligeros (Neón por ejemplo) se puede alcanzar ionización total y en consecuencia incrementar la energía hasta los 2 GeV por nucleón. La aceleración se realiza en dos cavidades de radiofrecuencia diametralmente opuestas; en las que los iones ven una caída de potencial de 15000 voltios. La frecuencia de operación va de los 800KHz a 5.6MHz; en cuanto al vacío en la línea del haz, la presión debe ser inferior a 10^{-11} torr.

La otra gran estructura es el ESR, con un perímetro de 108 metros tiene forma hexagonal con dos lados de mayor longitud que el resto. En uno de ellos está el enfriador por electrones mientras que en el otro tenemos un blanco gaseoso. En total hay 6 imanes deflectores y 20 lentes magnéticas formando parte del ESR, que permite almacenar iones de Uranio de hasta 560 MeV por nucleón, o iones de Neón de hasta 830MeV por nucleón durante tiempos que van desde las 10h para iones ligeros hasta de 30 a 60 minutos para iones pesados.

2.3. El espectrómetro

HADES es un experimento de segunda generación en espectroscopía de alta resolución de pares de electrones. Debido a la pequeña constante de acoplo electromagnética, el canal de desintegración en dielectrones corresponde típicamente a una proporción $\leq 10^{-4}$ del total de las desintegraciones. De ahí que las características fundamentales de HADES sean:

- Una elevada aceptancia geométrica en conjunción con capacidad para un elevado ritmo adquisición de sucesos
- Resolución en la reconstrucción de la masa invariante comparable a la anchura natural del meson ω ($\frac{\Delta M}{M} \simeq 1\,\%$)
- Una relación señal-ruido significativamente mayor que 1 para masas invariantes hasta $M \simeq 1 Gev/c^2$
- Elevada granularidad, para poder trabajar con los sistemas más pesados; como U+U a 1A $\cdot$ GeV)

En la figura 2.2 se puede observar una vista esquemática del espectrómetro. Éste está dividido en azimutalmente en 6 sectores, con simetría hexagonal y cubriendo

ángulos polares $18^0 \leq \theta \leq 85^0$, lo que da una aceptancia del 40 % [27]. Un detector RICH rápido, insensible a hadrones, está situado en torno a un blanco segmentado y es usado para la identificación de electrones. Para medir el momento de las partículas se cuenta con 4 cámaras de deriva en cada sector y 6 imanes superconductores. Como complemento al RICH para la identificación de electrones se cuenta también con varios detectores Pre-Shower (a ángulos polares inferiores a 45^0) y un detector de tiempo de vuelo (TOF). El trigger de primer nivel se obtiene a partir de una señal de multiplidad en el TOF. Un trigger de segundo nivel se realiza combinando las señales del RICH, TOF y Pre-Shower.

2.3.1. El detector RICH

El detector RICH[31] consiste en un radiador gaseoso en torno al blanco, un espejo VUV esférico y un detector gaseoso de fotones con un fotocátodo de CsI; véase figura 2.3. El detector de fotones está localizado delante del blanco para minimizar el impacto de partículas cargadas que puedan venir de la zona de interacción. Debido a la limitada sensibilidad del fotocátodo, la detección de fotones Cherenkov está limitada a la región VUV[2] del espectro. En el rango de longitudes de onda accesibles y con la geometría usada se producen del orden de 100 fotones por electrón incidente.

El radiador escogido es un perfluorobutano (C_4F_{10}) que, debido a un umbral de producción de luz Cherenkov de $\gamma_{th} \sim 18{,}3$, es ciego al paso de hadrones en el rango de energías del experimento. Dicho gas radiador es transparente para longitudes de onda ≥ 145 nm y no muestra una emisión significante de luz de centelleo.

La luz Cherenkov emitida por los electrones se refleja en 18 segmentos de espejo colocados sobre una capa soporte situada detrás del blanco. Los substratos de los segmentos de espejo están realizados en un nuevo tipo de material orgánico parecido a la cerámica que proporciona una rigidez extrema y una pequeña longitud de radiación al mismo tiempo. El cono de luz atraviesa una capa de CaF_2 de 6mm de grosor separándose así el gas de contaje del gas radiador.

El detector de fotones, parte clave del RICH, está diseñado para detectar de forma eficiente fotones individuales en la región VUV. Consiste en una cámara proporcional de hilos (MWPC) con una estructura en pads debajo de la capa foto-sensible de CsI; ésta cámara se utiliza con CH_4 puro y con una ganancia de 10^5. Las señales se leen de una vez de un conjunto de 30000 pads, cada uno de ellos con una superficie de $6{,}6 \times 6{,}6\ mm^2$ y con una eficiencia superior al 95 % para fotoelectrones individuales. El rendimiento global del RICH se puede resumir en una constante

[2] ultravioleta lejano

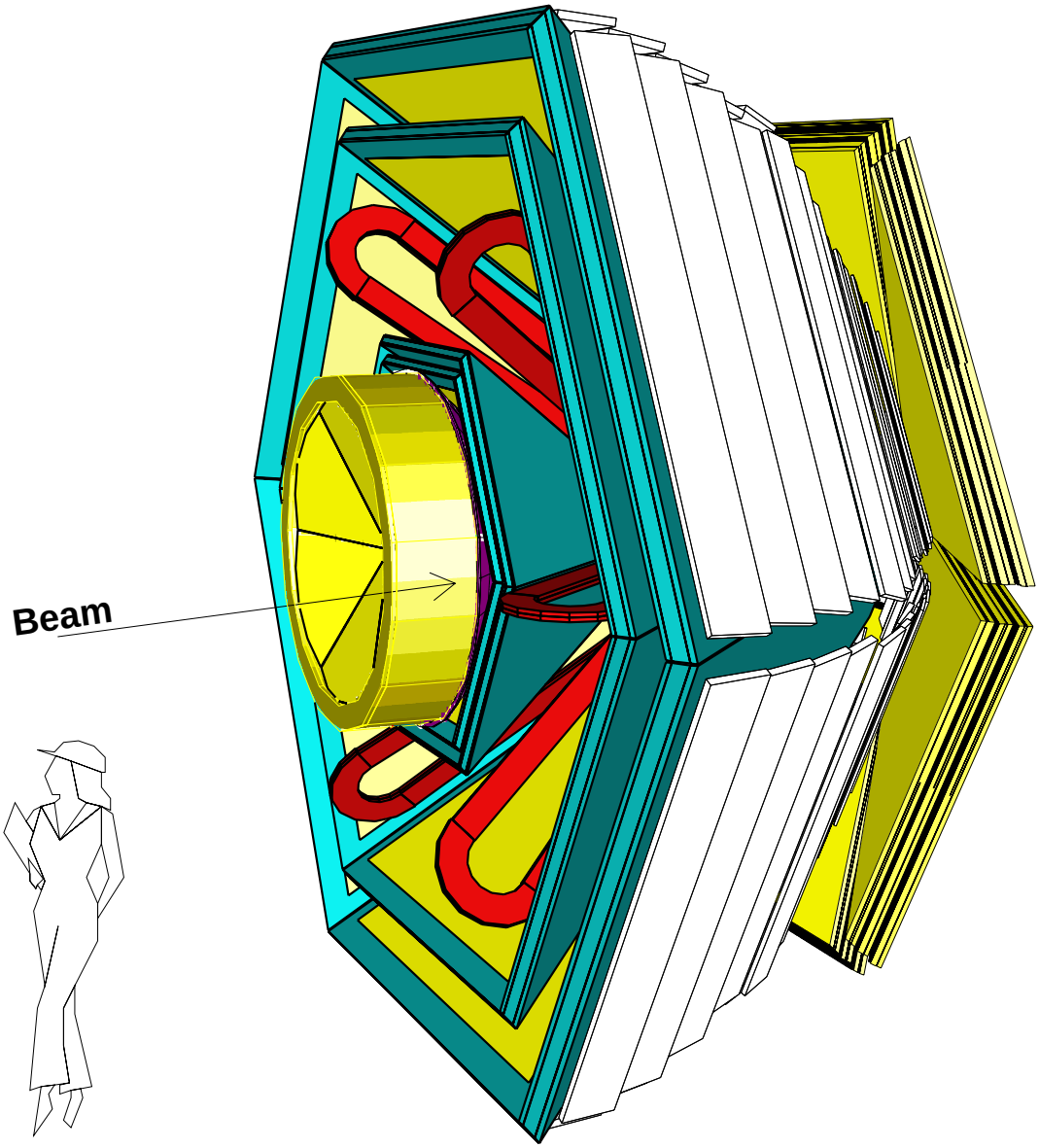


Figura 2.2: Vista esquemática del espectrómetro HADES

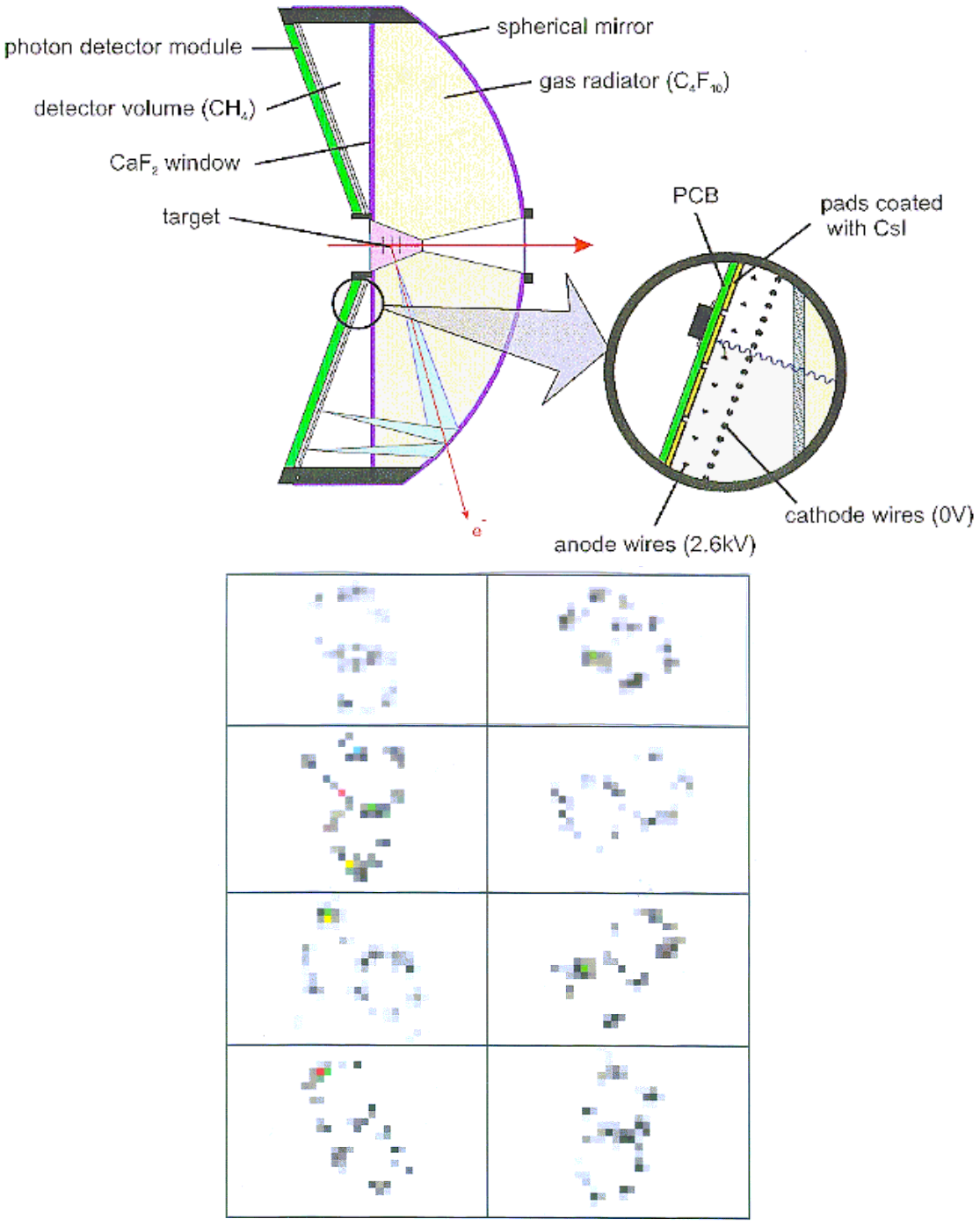


Figura 2.3: Vista esquemática del detector RICH con algunos anillos dobles

de mérito $M_0 = 108$ dando más de 10 fotoelectrones detectados por anillo. Las principales causas de fotones de fondo son la luz de centelleo en el gas radiador ($\sim$ 100 fotones por colisión central) y la producción externa de pares a partir de rayos γ($\sim$ 0,5 anillos por colisión central). Ejemplos de anillos se pueden observar en la figura 2.3.

2.3.1.1. Rendimiento del detector

Se ha trabajado en varias ocasiones con un prototipo del RICH para la determinación de velocidades de iones pesados y con la geometría de HADES para detectar electrones provinientes de conversión de pares de rayos γ en colisiones U+Pb a 1 $AGeV$[26]. Los anillos obtenidos no están oscurecidos por fondo de partículas cargadas y la calidad es suficiente para reconocimiento rápido de anillos "online".

2.3.2. La reconstrucción de trazas

Para conseguir una resolución en masa invariante del 1 % en la región del meson ω se debe ser capaz de reconstruir trazas de electrones con una resolución en momento del orden del 1 % para electrones con momentos mayores 0.1 GeV/c. Para este propósito se dispone de un espectrómetro magnético consistente en un imán toroidal y 4 cámaras de deriva (MDCs) por cada uno de los seis sectores (véase figura 2.4). El imán está constituido por 6 bobinas superconductoras montadas por separado en cajas de 80 mm de grosor, produciendo un campo máximo de 0.7 Teslas con una densidad de corriente de 120 A/mm^2 por bobina (las bobinas toroidales se aprecian claramente en la figura 2.2). El campo magnético creado por las bobinas es inhomogéneo, lo que complica el posterior algoritmo de búsqueda de trazas; que estará basado, en principio, en el "análisis de componentes principales"[17].

Para una región de 0.5 m de longitud del campo magnético la variación en el momento es menor de 100 MeV/c; manteniéndose así la aceptancia para electrones con momento bajo. La resolución requerida en la medida del momento se logra optimizando la resolución en posición intrínseca del sistema de la reconstrucción de trazas y limitando el scattering múltiple.

Las 24 cámaras de deriva proporcionan una superficie activa total de unos 24 m^2, que permite detectar tanto los electrones como otras partículas cargadas que provengan de la zona de colisión. Los marcos laterales de las cámaras caen en la región de sombra de las bobinas magnéticas con lo que la aceptancia geométrica es máxima. En cada sector, dos cámaras están situadas antes del imán y las otras dos

Figura 2.4: Corte transversal de HADES

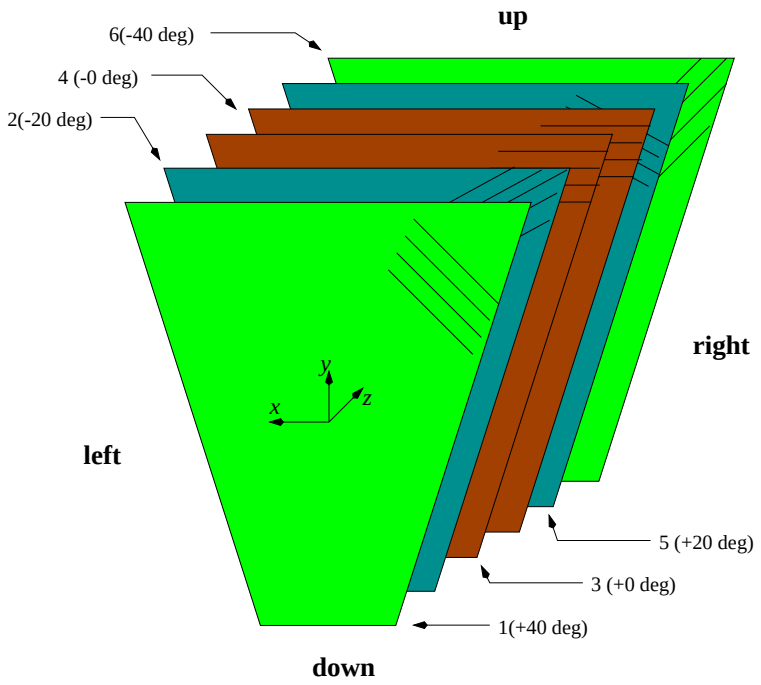


Figura 2.5: Estructura de una cámara de deriva

después del mismo. El tamaño de la celda va desde los 2.5 a los 7 mm, manteniendo una granularidad constante suficientemente alta para trabajar con multiplicidades de 0.6 cm^{-1} (en la región de menor ángulo polar). Cada cámara tiene seis planos de hilos sensores con orientaciones $+40^0$,-20^0,0^0,0^0,$+20^0$,-40^0 para proporcionar suficiente redundancia, de forma que se reconstruyan sin ambigüedad las coordenadas polar y azimutal de las trayectorias. Se puede ver un esquema de esta estructura en la figura 2.5

Para conseguir la resolución en momento requerida, se debe prestar especial atención a la minimización del múltiple scattering a lo largo de la trayectoria de la partícula. La contribución del scattering múltiple, la cual domina sobre la resolución en la posición para momentos inferiores a 0.4 GeV/c, se minimiza usando materiales ligeros en las cámaras y llenando con He la región del campo magnético. Los hilos de tungsteno plateados en oro (sensores) tienen un diámetro de $20\mu m$, los de aluminio (cátodos y campo) tienen $80\mu m$ de grosor. Las ventanas están hechas de capas de "mylar" aluminizado de $12\mu m$ de grosor. La mezcla de gases usada es $He - iC_4H_{10}$, la cual ha sido optimizada en términos de plateau de eficiencia (típicamente se consiguen 300V) y de resolución en la medida de la posición.

La velocidad de deriva para la mezcla satura en torno a los 4 $cm/\mu s$ para los voltajes usados; consiguiéndose una relación casi lineal entre tiempo de deriva y posición de la traza. La resolución en la medida para un sólo hit es $< 80\mu m$ sobre más del 80 % de la celda, de modo que la resolución total en la medida del momento se mantiene por debajo del 1 % para momentos superiores a 0.2GeV/c, incluyendo la contribución del múltiple scattering.

Para poder tratar con el elevado ritmo de adquisición de datos de HADES se ha diseñado un TDC (convertidor tiempo-digital) a medida. Se trata de un TDC con 8 canales basado en el TDC2001 construído en Mainz. Puede trabajar a 25 MHz y está construído con tecnología de 0.6 micras.

2.3.3. El detector de tiempo de vuelo TOF

El TOF[5] está diseñado para discernir electrones de piones hasta 0.5 GeV/c y de protones hasta 2 GeV/c, así como para medir la multiplicidad de partículas cargadas en cada colisión. Un total de 1056 plásticos centelladores aseguran una eficiencia en la detección de impactos superior al 90 %. La longitud y ancho de los plásticos van desde los 40 a los 237 cm y desde 1 a 3 m respectivamente, a medida que nos movemos hacia ángulos polares mayores. La luz es recogida por fotomultiplicadores en sendos extremos de cada plástico centellador. La estructura del TOF se puede ver en la figura 2.6

La capacidad para discriminar electrones de piones y protones que tiene este detector se usa en la identificación "online" de electrones. Además también se prevé hacer con las señales de los centelladores un análisis de la altura de sus pulsos, lo que contribuye a la identificación de la partícula así como a mejorar la medida de la posición.

La señal de "start" para el TOF será suministrada por un detector de diamante segmentado, situado tras el blanco para dar el tiempo inicial de una interacción. Las principales ventajas de este nuevo tipo de detector son una mayor velocidad de respuesta y menor daño a la radiación que otros detectores rápidos más convencionales.

2.3.4. El detector Pre-shower

Debido al elevado momento de las partículas a ángulos polares $\theta \leq 45^0$se introduce un detector Pre-shower detrás del los centelladores centrales del TOF, que permita mejorar la eficiencia en la separación entre piones y electrones.

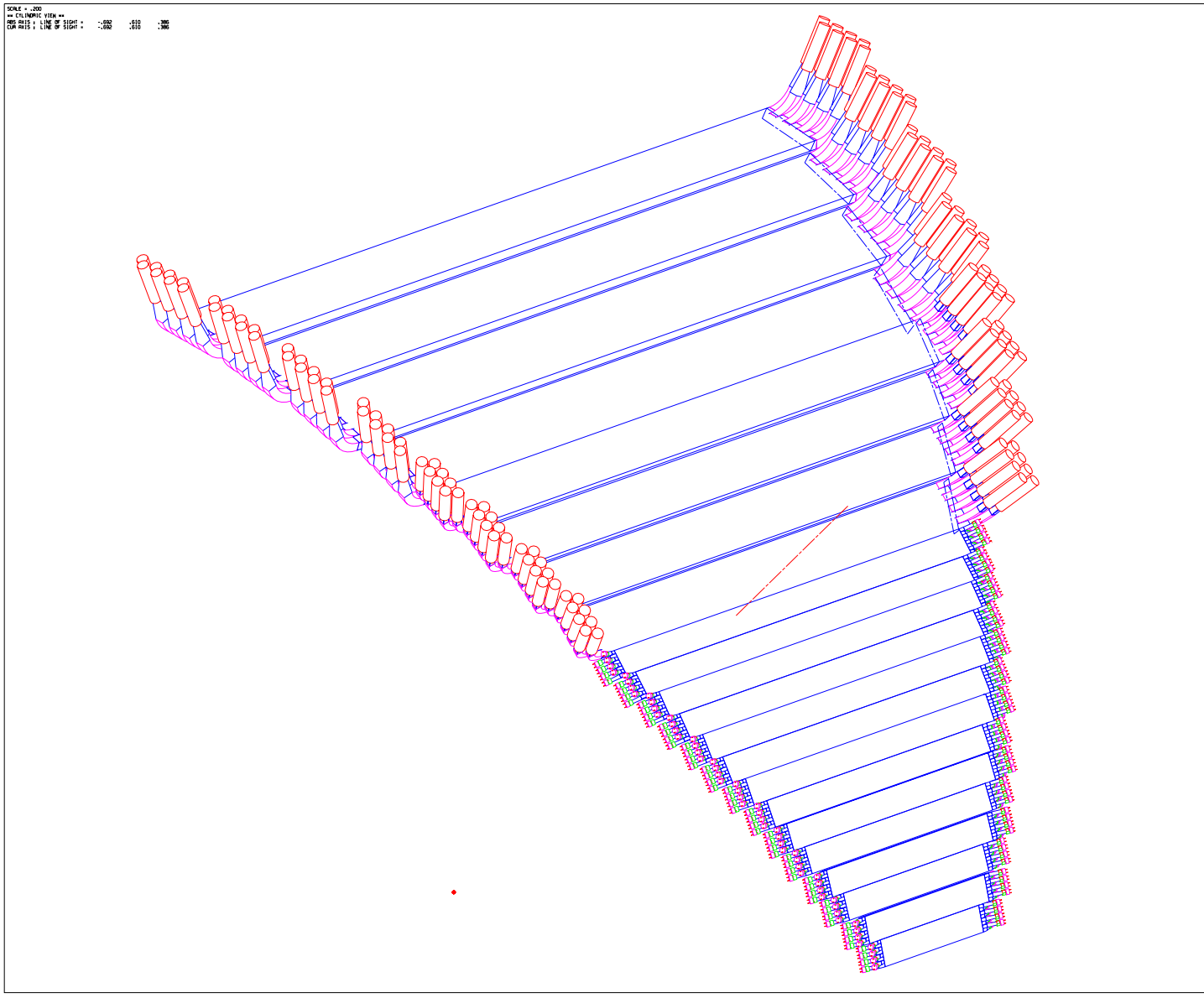


Figura 2.6: Vista esquemática del detector de tiempo de vuelo

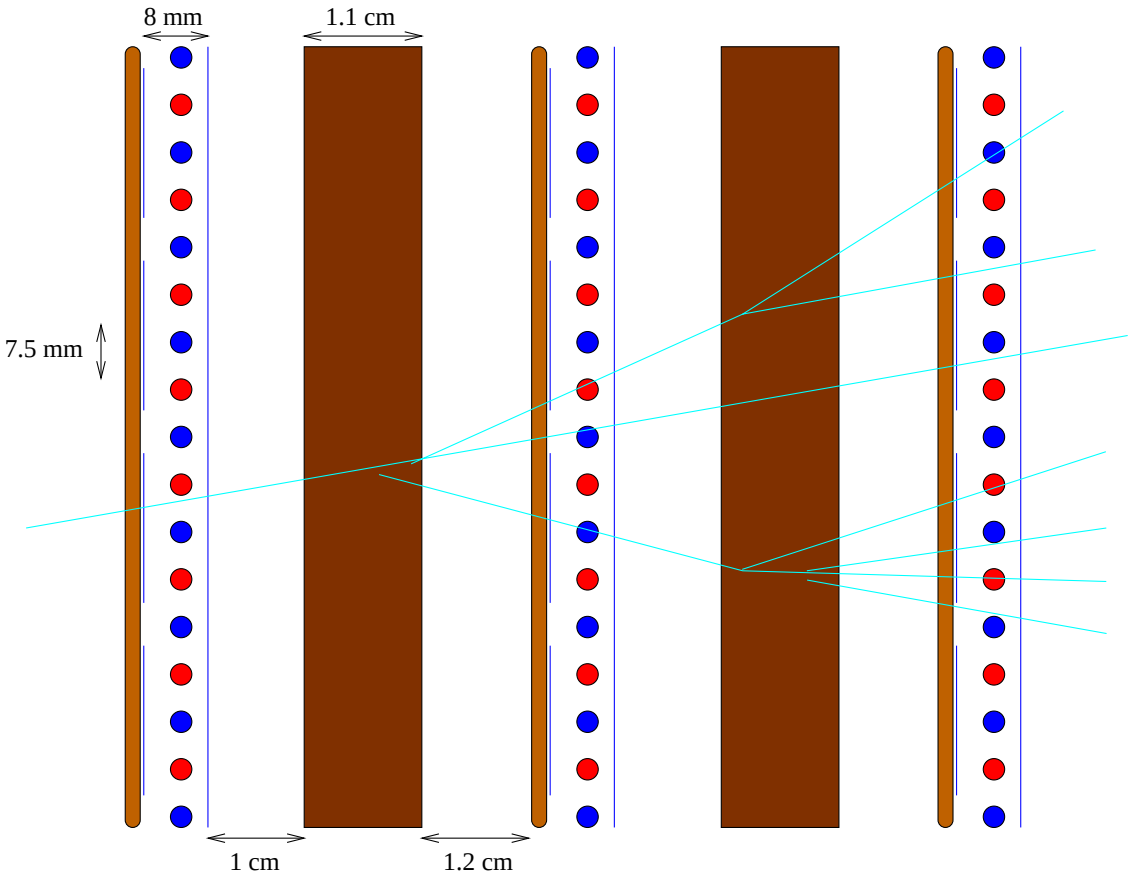


Figura 2.7: Esquema del detector Pre-Shower

El detector Pre-Shower[31, 18] está formado por 6 sectores idénticos distribuidos simétricamente en torno al eje definido por el haz. Cada sector está formado por 3 cámaras de hilos leídas por pads y separadas por capas de plomo (2) de 1.1 cm de grosor (2.5 longitudes de radiación). El grosor de las capas de plomo está optimizado para maximizar la probabilidad de producción de una cascada electromagnética en el rango de energías usado; al mismo tiempo que se mantiene la probabilidad de cascadas hadrónicas en límites tolerables. Cada cámara de hilos está formada por un plano con hilos de campo y sensores alternados separados una distancia de 7.5 mm, que se sitúa entre dos planos catódicos separados 8 mm; uno de ellos está fabricado con acero inoxidable y el otro con fibra de vidrio con 1024 celdas de cobre organizadas en 32 filas por 32 columnas. Esta granularidad limita la probabilidad de dobles hits a menos del 2 % para colisiones centrales $Au + Au$ a 1 $A \cdot GeV$. La figura 2.7 muestra un esquema con la estructura del detector Pre-Shower.

La carga inducida es leída en el plano de fibra de vidrio, dando lugar a un pulso con tiempo de subida de 10 ns y de caída de 20-30 ns. La corriente llega a los 0.5 mA para una carga de 10 pC. Varios pulsos deben ser integrados en una celda en caso de una cascada electromagnética.

Las cámaras funcionan en el régimen llamado "self-quenching streamer mode" (SQS) con lectura de la carga en las celdas del detector. Una comparación de señales antes y después de la capa de plomo es lo que permite la separación entre electrones con cascada y hadrones que no producen cascada. Se espera que las identificaciones erróneas de hadrones y electrones en el Pre-Shower se sitúen en torno al 10 % para colisiones Au+Au; siendo las fuentes de error dominantes la resolución del detector, cascadas hadrónicas y señales en múltiples celdas para un solo hit.

2.3.4.1. Rendimiento del detector

Un prototipo a tamaño real y con 2 cámaras ha sido probado con haces de electrones, protones e iones pesados. De los datos obtenidos se infiere una eficiencia del 82 % para cascadas de un sólo electrón en ausencia de hadrones.

También se ha investigado la respuesta del detector a hadrones mínimamente ionizantes (protones a 2.1 GeV/c). Por comparación con los electrones a 850 MeV la probabilidad de error en la identificación se sitúa por debajo del 10 %. Para electrones a momentos más bajos (200 MeV/c) la probabilidad de error llega a $\simeq 25\,\%$, mientras que la eficiencia de detección pura para electrones se queda en el 40 %

La prueba con el haz de iones pesados se ha realizado en conjunción con un detector TOF para identificar electrones el presencia de iones pesados. Para las partículas más rápidas $\beta \simeq 1$ las diferencias en las cargas medidas muestran un ensanchamiento significante e indican la sensibilidad esperada a electrones. A bajas velocidades, la capacidad de discriminación entre electrones y hadrones se reduce considerablemente. Los protones lentos son decelerados en el plomo de modo que se producen considerables pérdidas de energía en la capa siguiente (post-conversor). Sin embargo la saturación de la ionización, esperada del modo SQS, da como resultado diferencias de carga aproximadamente constantes entre las capas antes y después del plomo y limita la ineficiencia en la identificación de los leptones a valores aceptables.

2.3.5. El Trigger

El sistema de trigger para el experimento HADES[36] está organizado en 3 niveles y está diseñado para seleccionar de forma eficiente sucesos con candidatos a dielectrones en colisiones de iones pesados. Con una intensidad de 10^8 partículas por segundo y una interacción en el blanco de un 1 % se esperan del orden de 10^6 sucesos de este tipo por segundo. De éstos sólo nos interesan los sucesos centrales; seleccionados por un trigger de primer nivel y que representan un 10 % de las interacciones. De modo que el sistema de adquisición debe aceptar datos cada 10 μs para 80k canales. Así pues inicialmente tenemos un flujo de datos de 2Gbytes/s que es reducido a través de un procesado online en dos etapas hasta los 4 Mbytes/s para colisiones Au+Au.

El trigger de primer nivel se basa en la multiplicidad registrada en el TOF y representa una reducción de los datos en un factor 10 como se ha dicho. A continuación, sólo los datos del RICH, TOF y Pre-Shower son transmitidos a una unidad de procesado de imágenes manteniendo el resto de los datos en una memoria intermedia hasta que se tome la decisión del trigger de segundo nivel. Gracias a una arquitectura masivamente paralela basada en chips FPGA (Field Programmable Gate Arrays) se puede tomar una decisión en 40-50 μs reduciendo el flujo de datos en un factor 100 hasta llegar a los 20-40 Mbytes/s. Por último se transmiten los resultados del trigger de segundo nivel junto con los datos recogidos en las MDC a un trigger de tercer nivel, opcional, usando para ello una red ATM. En este punto se utiliza un procesador suficientemente potente para un mejor emparejamiento de trazas antes y después del campo magnético, reduciéndose el flujo de datos en otro factor 10 para así llegar a unos 1-4 Mbytes/s.

2.4. La adquisición de datos

El sistema de adquisición de HADES está basado en una red ATM que conecta crates VME para la lectura de los detectores y trigger de segundo nivel con la estación de trabajo destinada a construir el suceso y con un trigger de tercer nivel opcional. Cada crate VME destinado a la lectura de un detector cuenta con varios módulos específicos para ese detector, un módulo para trigger que sirve como interfaz para el sistema de distribución del trigger, y un Power PC con el sistema operativo LynxOs destinado a ensamblar los datos de cada detector en un subsuceso.

Para optimizar el rendimiento los ordenadores encargados de formar cada subsuceso recogen una gran cantidad de datos del trigger de segundo nivel antes de

mandarlos asíncronamente al constructor de sucesos via ATM. Un número de identificación, único para cada suceso, es usado para emsamblar los sucesos; éstos sucesos ensamblados son escritos en cinta para su posterior análisis.

2.5. El software de simulación y análisis

El software de simulación desarrollado para el estudio del rendimiento del espectrómetro está basado en GEANT 3.21[28]. Un nuevo interfaz de ORACLE a GEANT permite que datos geométricos de los detectores extraídos del programa de diseño CATIA, con el que se han dicho diseñados todos los detectores y soportes del espectrómetro, sean importados directamente en el paquete de simulación. Este mecanismo permite una actualización continua del paquete de simulación.

En cuanto al software de reconstrucción, es éste precisamente el tema central del presente trabajo, por lo que será tratado en detalle en los capítulos sucesivos.

Capítulo 3

La reconstrucción de sucesos en Hades

Como ya hemos dicho en el capítulo anterior, el principal objetivo del experimento HADES es estudiar dileptones producidos en colisiones núcleo-núcleo. Como en la mayoría de los experimentos de "Física nuclear y de partículas" el software de HADES abarca cuatro grandes áreas de trabajo: Adquisición y monitorización de datos, Simulación, Reconstrucción y Análisis. En esta memoria trataremos concretamente sobre el software de reconstrucción de los sucesos.

La reconstrucción supone distintas tareas: leer la información de los sucesos de alguna base de datos con distintos niveles de reconstrucción, hacer reconocimiento de formas en todos o algunos de los detectores y subdetectores de HADES, identificar partículas en caso de ser posible, ajustar las trazas reconstruídas para obtener el momento de las partículas, y proporcionar información de los sucesos reconstruídos así como toda la información acerca del proceso de reconstrucción. No hay una forma única de reconstruir sucesos; así por ejemplo, el usuario puede decidir reconstruir sólo leptones o reconstruir cualquier tipo de partícula: leptón o hadrón.

3.1. Decisiones previas

Los primeros requerimientos con los que nos encontramos son las decisiones tomadas por la colaboración HADES en cuanto a las herramientas que se van a utilizar para el desarrollo del programa de reconstrucción. Así pues, hemos decidido:

- Usar **programación orientada a objetos**. La razón es que este tipo de pro-

gramación permite abordar con mayores garantías proyectos de complejidad media o elevada. Facilitando un estilo de programación modular, que dota al programa final de una gran flexibilidad y facilita el desarrollo cuando éste es realizado por diversos grupos de programadores separados geográficamente.

- **Lenguaje de programación: C++**. El lenguaje de programación C++ es uno de los muchos lenguajes orientados a objetos; las principales razones para escoger éste y no otros son su amplio abanico de funciones, el rendimiento de los programas y su creciente aceptación en el mundo de la "Física Nuclear y de Partículas", lo cual facilita el acceso a librerías y utilidades (como puede ser ROOT).

- **Compilador: g++**. Se ha elegido este compilador por ser gratuíto y por estar disponible en un gran número de plataformas. La elección de un compilador no significa que se puedan usar en el software extensiones propietarias de ese compilador. De hecho el desarrollo se restringirá en lo posible al estándar ANSI C++[2] y en lo no contemplado por dicho estándar, se usará POSIX[37]. El hecho de que se haya escogido el compilador g++ tan sólo quiere decir, por lo tanto, que las nuevas versiones del software son probadas con dicho compilador antes de hacerse oficiales.

- **Notación gráfica: UML**. El desarrollo de programas en un equipo internacional hace necesario el uso de herramientas de comunicación bien definidas; una de ellas es una notación estándar para los diagramas de clases que acompañan a la documentación del software. La notación gráfica UML[7] ("Unified modelling language", ver apéndice **??**) se escoge sobre otras posibilidades, como OMT[3] ("Object modelling language") o BOOCH[10], debido a su creciente aceptación en el mundo de la programación orientada a objetos (de hecho UML significa "Unified modeling language" haciendo alusión a que se trata de una unificación de las notaciones BOOCH y OMT, promovida por los propios creadores de éstas). El uso de una notación particular sería muy engorroso si no dispusiéramos de una herramienta especializada para producir los diagramas; en nuestro caso dicha herramienta será el paquete Rose/C++.

- **Sistema de control de versiones: CVS[6]**. La utilización de un sistema central de control de versiones es otra de las consecuencias de un esquema de desarrollo distribuído (diversos grupos escribiendo diferentes partes del código en distintos lugares del mundo). De entre todos los posibles sistemas se ha escogido CVS por su gratuidad y su disponibilidad en un amplio rango de plataformas.

- **El software está basado en ROOT**. ROOT[11] es un "framework" especialmente diseñado para el desarrollo de aplicaciones en física de altas energías (véase apéndice A). ROOT se usa fundamentalmente para:

  - Proporcionar una interfaz de usuario
  - Proporcionar sistemas para almacenar objetos en ficheros de salida
  - Hacer gráficos e histogramas
  - Generar documentación a partir de los comentarios en el código

  El uso de ROOT impone una serie de ligaduras al software que se puede desarrollar. En concreto, no se puede hacer uso de templates o excepciones (por el momento); del mismo modo, el intérprete de ROOT (i.e. la interfaz de usuario) no es totalmente compatible con C++

- Las **plataformas soportadas** son:

  - Sistemas Unix (DEC Unix,AIX,HPUX ...)
  - Linux (PC)
  - otras (Windows NT)

- En cuanto a la **organización** del programa se pretende que el código aparezca dividido en librerías independientes; es decir, distintas partes del código (módulos) van a distintas librerías de enlace dinámico que pueden ser cargadas o descargadas en tiempo de ejecución. Debe haber una librería con código común que es cargada por defecto; el resto de las librerías se cargan bajo demanda del usuario. De este modo el código que no es necesario en un momento dado no ocupa la memoria de la máquina. Ejemplos de partes del programa que pueden ir a librerías de enlace dinámico distintas de la central son el "event display", los algoritmos específicos de un reconstructor, así como sus objetos de datos etc. Para conseguir este objetivo es necesario que el programa de reconstrucción esté diseñado de una forma totalmente modular.

3.2. Programación orientada a objetos

Al igual que la programación procedural, la programación orientada a objetos[10, 20] es un paradigma para la modelación de productos software. No se debe asociar por tanto a un lenguaje en concreto sino a una forma de escribir código y de diseñar.

En efecto, podemos hacer programación orientada a objetos en lenguajes procedurales como C, Cobol o Fortran; del mismo modo que podemos hacer programación procedural en ensamblador. Sin embargo existen lenguajes que soportan de forma directa el estilo de programación orientado a objetos, como Smalltalk, Python, Object Pascal o C++. Siendo C++ el más extendido, tanto por su rendimiento, como por el amplio soporte que da para el paradigma orientado a objetos; hasta tal punto, que muchos consideran a C++ un lenguaje experimental donde se prueban los nuevos avances en este tipo de programación.

Uno de los mayores problemas con que se encuentra la programación a gran escala es la complejidad inherente al software. Complejidad que viene fundamentalmente de cuatro fuentes:

1. La complejidad del problema a resolver.

2. La dificultad de gestionar el proceso de desarrollo. Dada la magnitud del trabajo a realizar se necesita un equipo de desarrollo, idealmente lo más pequeño posible. En cualquier caso ésto dificulta la comunicación y coordinación.

   Cuando el desarrollo es realizado por un equipo, sobre todo si está disperso geográficamente, el principal problema radica en mantener una unidad en el diseño del sistema.

3. La flexibilidad posible a través del software. Esta flexibilidad tiene innegables ventajas pero obliga a definir explícitamente cada pieza constituyente de un programa, haciendo de la programación un trabajo intensivo.

4. Los problemas para caracterizar el comportamiento de los sistemas discretos. El software es un sistema discreto porque el estado de una aplicación viene determinado por un conjunto discreto de variables y controles de flujo. De modo que en sistemas de gran tamaño se produce una explosión combinatoria de los estados posibles que hace imposible comprobarlos todos.

La programación orientada a objetos nos proporciona nuevas herramientas para gestionar toda esta complejidad; como, por ejemplo, la encapsulación, la herencia o el polimorfismo.

La **encapsulación** la obtenemos del concepto de objeto. Un objeto es un conjunto de datos y funciones que aparecen integrados en una sola unidad software. Las funciones se suelen llamar *métodos* y los datos o variables se llaman *miembros* del objeto. No todos los métodos y miembros de un objeto tienen que ser públicos, es decir, accesibles desde el exterior del objeto; más bien, un objeto sólo hace

públicos parte de sus miembros y funciones, definiendo así un contrato con el resto del mundo. Siempre que este contrato sea respetado podemos cambiar los detalles internos del objeto. Ésto tiene dos ventajas fundamentales, una, la reusabilidad del código y otra, la acotación del terreno para buscar errores. Una vez que sabemos que un objeto cumple bien con su contrato ya no es necesario buscar errores en él de nuevo, puesto que el desarrollo en el resto del código no le afecta.

Los objetos son instancias de clases. Una clase es como una plantilla que nos dice cómo crear (instanciar) objetos de esa clase. Por lo tanto describe sus miembros y sus funciones; indicando cuáles de ellos son públicos o privados. Cada objeto creado a partir de una clase tendrá una copia de sus miembros; mientras que los métodos son compartidos por todos los objetos de la clase.

La **herencia** es una herramienta que nos permite especializar una clase. Cuando decimos que una clase hereda de otra, significa que sus objetos contendrán todos los miembros de la clase padre así como todas sus funciones. La herencia nos define una relación entre clases del tipo "es un"; así pues decir que la clase "Cuadrado" hereda de la clase "Figura" equivale a decir que "un Cuadrado es una Figura".

Típicamente la complejidad toma forma de jerarquía. Siendo muy importante reconocer explícitamente esta jerarquía. Esencialmente hay dos tipos de jerarquía:

Jerarquía "es parte de". Este tipo de jerarquía se modela directamente en programación orientada a objetos haciendo que un objeto sea un miembro de datos de otro. Un ejemplo de este tipo de jerarquía es un "avión que puede ser estudiado descomponiéndolo en un sistema de propulsión, un sistema de control, etc."

Jerarquía "es un". Este tipo de jerarquía se modela directamente a través de relaciones de herencia entre clases. Es el tipo de jerarquía que aparece en una pieza de conocimiento como, por ejemplo, "un motor turbofan es un tipo de motor a propulsión,un Pratt y un Whitnew TF30 son tipos de motores turbofan"

La herencia abre además las puertas al **polimorfismo**. Para explicarlo pensemos en un ejemplo concreto. Supongamos que queremos programar un sistema en el que es necesario dibujar distintas formas en pantalla, como cuadrados, círculos y elipses. En el paradigma procedural, la función de dibujado de la pantalla sería algo como:

```
while (...) {
 int figura=escogerFigura();
 switch(figura) {
  case 1 : dibuja_rectangulo();
  case 2 : dibuja_cuadrado();
  case 3 : dibuja_circulo();
 }
}
```

En este pedazo de código tenemos un bucle "while" en el que primero le pedimos al usuario que escoja una figura. Luego, en función del código que identifica a cada figura, vamos a una estructura "switch" y llamamos a la función que la dibuja. Si ahora quisiésemos introducir un cuarto tipo de figura, por ejemplo el triángulo, tendríamos que modificar la estructura interna del programa con la consecuente posibilidad de introducir errores en ella.

En un paradigma orientado a objetos, primero definiríamos una clase "Figura" que representa a cualquier tipo de figura, dotándola de un métdo llamado "dibujar", que la dibuja. Resulta evidente que este método no tiene por ahora ningún significado concreto ¿Cómo dibujar una figura sin saber que figura concreta es?. Se dice entonces que "dibujar" es un *método virtual* cuyo significado será definido más tarde y que "Figura" es una *clase abstracta*. No se pueden instanciar objetos de una clase abstracta ya que no sabemos como operar sobre ella, ¡Sus métodos no están definidos!

Luego podemos definir una clase heredada de "Figura" por cada tipo de figura: cuadrado, elipse. A este proceso se le llama *derivar una clase* de la clase "Figura"; la clase así definida se llama *clase derivada* o *subclase*.

Como nuestras clases "Cuadrado", "Elipse" y "Círculo" derivan de "Figura" tendrán todas ellas un método "dibujar". Para cada una de ellas ya sabemos como dibujarla, por tanto ya podemos escribir el código del método dibujar que será diferente en cada clase. A este proceso se le llama *sobrecargar* el método "dibujar". Hecho ésto, el anterior fragmento de código se convierte en:

```
while (...) {
 Figura *figura=escogerFigura();
 figura->dibujar();
}
```

La diferencia fundamental con el caso anterior es que ya no hay un "switch". Ahora, la función "escogerFigura()" nos devuelve un puntero a un objeto figura del

tipo elegido por el usuario y nosotros simplemente llamamos al método "dibujar" de ese objeto. De modo que la implementación del método es automáticamente escogida en función de la clase del objeto figura; es decir, en función del tipo de figura. Decimos entonces que el método "dibujar" es *polimorfo* porque toma distintas implementaciones en distintos momentos del programa.

Nótese como ahora es trivial añadir los triángulos. Basta definir una nueva clase "Triangulo" que deriva de "Figura" y que da una nueva implementación al método "dibujar". No es necesario tocar el código central del programa, reduciendo así notoriamente la posibilidad de introducir nuevos errores y facilitando el mantenimiento.

3.3. El método de Booch aplicado a la reconstrucción de sucesos en HADES

Existen muchos métodos de análisis y diseño orientados a objeto en el mercado, la mayoría de ellos comerciales. Una lista de los más comunes se puede encontrar, por ejemplo, en *http://www.well.com/user/ritchie/oopm-uic.html*. Pero la mayoría de los experimentos de Física Nuclear y de Partículas que están desarrollando programación orientada a objetos utilizan el método de Booch. Es por ésto que hemos seguido, en líneas generales, este método.

El método de Booch es, pues, un método que sirve como ayuda para diseñar sistemas usando el paradigma de objetos; cubriendo las fases de análisis y diseño. Las características fundamentales son que se trata de un método dirigido por casos de uso y que es iterativo e incremental. Sin embargo también hace hincapié en una secuencia de actividades recomendada, así como en productos de las distintas etapas. Normalmente estos productos toman la forma de diagramas de alguno de los tipos introducidos en la notación de Booch. Como todo método de programación, no se trata de una receta que haya que seguir al pie de la letra; sino de una guía para ayudar a orientar diversas fases del ciclo de diseño.

El método es iterativo y se organiza en dos ciclos anidados: los macro procesos y los micro procesos:

Un **macro proceso** es una actividad a largo plazo, que se realiza fundamentalmente en varios pasos

- Establecimiento de los **requerimientos** fundamentales del sistema
- **Análisis** del dominio y del comportamiento del sistema

- **Diseño** de la arquitectura del sistema
- **Evolución** de la implementación a través de sucesivas versiones
- **Mantenimiento** de las sucesivas versiones producidas

Un **micro proceso** es la actividad diaria que lleva a la producción de nuevas clases objetos y escenarios de uso. Las varias actividades que se pueden identificar en este proceso son:

- Identificar **clases** y **objetos**
- Identificar la **semántica** de clases y objetos
- Identificar **relaciones** entre clases y objetos
- **Implementar** esas clases y objetos, así como las estructuras de datos y los algoritmos

3.3.1. Requerimientos del sistema

En esta etapa se determina qué es lo que queremos que haga el sistema. Es una etapa de alto nivel en que se definen las características clave del sistema; en qué dominio nos centramos y a qué tipo de situaciones es sometido el software.

Productos de esta etapa son el *Establecimiento del Problema* ("Problem Statement"), donde se describe de una forma muy general en qué consiste el sistema, la *Descripción del Sistema* ("System charter"), que contiene más específicamente las responsabilidades del sistema y, eventualmente, los requerimientos concretos desde el punto de vista "software" así como "hardware" y el *Establecimiento de las Funciones del Sistema* ("System Function Statement"), que describe el comportamiento previsto del sistema a través de los escenarios (casos) de uso más importantes.

3.3.1.1. Establecimiento del problema

Como ya hemos mencionado con anterioridad, lo que pretendemos, en último término, es reconstruir, para su análisis físico, los sucesos tomados por el espectrómetro HADES. Una descripción general del espectrómetro y de sus objetivos físicos puede verse en el capítulo 2 de ésta memoria.

3.3.1.2. Descripción del sistema

Responsabilidades del sistema

El **objetivo principal** del programa de reconstrucción es, pues, la reconstrucción, para su utilización en la física, de los sucesos obtenidos por el espectrómetro HADES.

HADES, como ya ha sido mencionado, es un espectrómetro con simetría axial, dividido en seis sectores iguales. En cada uno de los seis sectores hay varios detectores diferentes (RICH, MDCs, imán, TOF, Pre-Shower). A su vez, en cada sector, los detectores pueden estar constituídos por varios módulos: por ejemplo, en cada sector hay cuatro cámaras de deriva MDCs a las que denominamos módulos MDC1, MDC2, MDC3 y MDC4, respectivamente.

Esta estructura geométrica modular de HADES debe estar incorporada en el programa de reconstrucción de forma que, a elección del usuario, se pueda realizar la reconstrucción hasta un determinado nivel para sectores, detectores o módulos determinados.

Esta modularidad geométrica se refleja, a su vez, en las estructuras de datos y en los algoritmos de reconstrucción.

La reconstrucción de sucesos progresa en pasos sucesivos. Los datos originalmente obtenidos por el sistema de Adquisición de Datos (DAQ) son elaborados gradualmente. Cada etapa de la reconstrucción es realizada por un algoritmo o procedimiento modular y produce un nivel de datos elaborados.

La modularidad de los algoritmos o procedimientos de transformación permite su sustitución por otros sin que el resto del sistema se vea afectado, en particular las estructuras de datos existentes. Para conseguir un óptimo aprovechamiento de esta característica, la selección y secuenciación de procedimientos de transformación a ejecutar es realizada por el usuario en tiempo de ejecución.

El número de niveles de datos parcialmente elaborados es dependiente del detector, e incluso puede ser dependiente del algoritmo.

Los datos obtenidos de la adquisición de datos son "desempaquetados", obteniéndose así el primer nivel de datos de cada detector: son los llamados datos "Raw". El "desempaquetamiento" clasifica la información recibida de la electrónica y que viene ordenada por canales en datos ordenados por detectores, módulos etc. Posteriormente los datos de cada detector son calibrados, lo que puede realizarse en uno o varios pasos, dependiendo de la complejidad del detector. Como resultado

de la calibración obtenemos información física de las partículas que atravesaron los detectores: posición, cantidad de movimiento, energía, carga, masa, etc.

Niveles superiores de la reconstrucción pueden combinar datos de diferentes detectores con el fin de llegar al conocimiento del cuadrimomento total de las partículas de interés involucradas en la colisión.

El sistema debe ser capaz, pues, de ser utilizado con fines diferentes, aunque su objetivo final sea la reconstrucción de las colisiones núcleo-núcleo. Utilizaciones requeridas son, por ejemplo:

- Análisis técnicos de detectores
- Calibración de un detector
- Alineamiento
- Optimización de los algoritmos de reconstrucción
- Fusionado de distintos ficheros de input de sucesos
- etc.

El sistema requiere una serie de entradas (**"inputs"**) para ser ejecutado:

1. Los **datos a reconstruir**, que pueden venir de diversas fuentes:
   - El Sistema de Adquisición de Datos, que graba los datos tomados por el espectrómetro en cintas, las cuales deben poder ser leídas y analizadas.
   - Sucesos simulados, que son generados por GEANT y almacenados en disco o cinta.
   - Sucesos parcialmente reconstruídos, que son generados por el propio programa de reconstrucción; de modo que si se quieren probar distintos métodos de reconstrucción, por ejemplo, no se tiene que reconstruir los sucesos empezando de cero en cada prueba.
   - Otros[1]
2. El número y tipo de los distintos niveles de datos, así como la organización en memoria de éstos (es decir, la **estructura del suceso**).

[1] Con "otros" se pretende decir que el programa debe ser lo suficientemente flexible para permitir la incorporación de nuevas fuentes de entrada de datos (por ejemplo: fusionado de distintos inputs)

3. La fuente de donde serán leídos los **parámetros de reconstrucción** (es decir, parámetros geométricos,parámetros de calibración, parámetros de desempaquetamiento...). En principio estos parámetros podrán ser leídos de distintas fuentes:

   - una base de datos central basada en ORACLE
   - un fichero ROOT
   - un fichero ASCII
   - También se debe dar la libertad al usuario para introducir parámetros a mano si lo deseara.

4. Los **algoritmos y procedimientos** con los que se llevará a cabo la reconstrucción. El especificar ésto como una entrada más al sistema permite ensayar distintos algoritmos de reconstrucción sin necesidad de recompilar o "relinkar" el programa.

   Por otra parte se debe dar la posibilidad de incorporar nuevos algoritmos al programa de reconstrucción manteniendo el tiempo de compilado y linkado al mínimo posible. Esto implica que el diseño sea modular, de modo que se pueda reutilizar la mayor parte del código a la hora de probar distintos algoritmos y se mantenga el tiempo de compilación bajo (ya que sólo es necesario recompilar los nuevos módulos sin tocar el resto).

   El mecanismo o mecanismos para seleccionar una configuración de algoritmos o un formato para los datos de salida debe combinar una gran versatilidad (sobre todo en el caso de la selección de algoritmos) para los usuarios expertos junto con la sencillez necesaria para los usuarios noveles. Así pues, un usuario experto debe poder hacer una definición exhaustiva del conjunto de algoritmos a utilizar para la reconstrucción, mientras que un usuario novel se limita a escoger entre unos conjuntos estándar ya preparados previamente.

El sistema produce una serie de salidas ("outputs"), a elección del usuario, entre los que se encuentran los niveles de datos reconstruídos, gráficos e histogramas, mensajes de error, estadísticas, resúmenes etc.

La salida del programa de reconstrucción debe ser tal que se guarde información de con qué conjunto de algoritmos y parámetros se ha hecho la reconstrucción, junto con los datos reconstruídos.

Interfaz del sistema con el usuario

El programa de reconstrucción puede funcionar tanto en modo "batch" como en modo interactivo. Se ha decidido la utilización del paquete ROOT para implementar la interactividad del programa.

En cualquiera de los dos modos de funcionamiento el usuario puede decidir los "inputs" y "outputs" anteriormente enumerados en el caso de que no quiera utilizar las opciones por defecto.

Plataformas de desarrollo y de destino

Las decisiones acerca de las plataformas "Hardware" y herramientas "software" con las que se implementará el sistema ya han sido especificadas en la sección 3.1 de este capítulo.

3.3.1.3. Establecimiento de las funciones del sistema

El sistema, como hemos ya comentado debe posibilitar diversos escenarios o "casos de uso". Entre ellos, debe permitir:

- Seleccionar el tipo de ejecución: "batch" o interactiva, "depurado" o "producción", ...
- Seleccionar la parte o partes del espectrómetro a procesar (qué sectores, detectores, módulos ...)
- Seleccionar las tareas de reconstrucción: desempaquetar, calibrar, alinear, identificar, reconstruir el momento, etc.
- Seleccionar entre varios algoritmos existentes para una misma tarea.
- Seleccionar parámetros de reconstrucción
- Decidir el output.
- etc.

3.3.2. Análisis del dominio y del comportamiento del sistema

Se define un modelo orientado a objetos preciso de aquella parte del problema en el mundo real que es de interés para el programa. De esta manera es como se obtiene un conocimiento más preciso del problema. Los pasos fundamentales pueden ser:

- Definir las clases
- Definir relaciones entre clases
- Definir los métodos de las clases
- Encontrar los miembros, es decir las propiedades que describen a las clases
- Definir la estructura de herencia
- Validar el modelo e iterar de nuevo

3.3.3. Diseño de la arquitectura del sistema

Las etapas de análisis y diseño están interrelacionadas, de modo que no es fácil separarlas. La etapa de diseño, sin embargo, proporciona ya suficiente detalle como para conseguir una implementación efectiva, definitiva y eficiente para llevar a cabo el modelo anteriormente creado. Los pasos son:

- Determinar la arquitectura inicial
- Planificar las versiones que se irán realizando
- Desarrollar las versiones
- Refinar el diseño

Productos típicos de las etapas de análisis y diseño son los diagramas de clases, los diagramas de interaciones entre objetos y los escenarios de uso.

La metodología que acabamos de exponer en líneas generales nos ha llevado, a través de las etapas de Análisis y Diseño, a una implementación de la estructura general del programa de reconstrucción que se describe detalladamente en el siguiente capítulo de esta memoria.

Capítulo 4

Descripción del programa de reconstrucción

Éste capítulo conforma la parte central de la documentación del software de reconstrucción para el experimento HADES. Por esa razón se reproduce aquí directamente en inglés, tal y como fue originalmente escrito.

4.1. Conventions

The written code follows some conventions in order to make it more readable. One of these conventions is also important to be kept on mind when reading the following description. The convention is that all classes developed for the reconstruction program have a name beginning with "H"; meanwhile the classes which are taken from ROOT begin with "T". The detailed convention rules are the following:

1. File names and directory names are written in lower case, using only a-z and 0-9.

2. Suffixes are .h for header files, .cc for code files and .C for ROOT macros. Furthermore, plain Fortran files have .f, Fortran files with preprocessor statements have .F and Fortran include files have .inc.

3. To avoid multiple inclusions, .h files should contain a precompiler definition, e.g. for myfile.h:

```
#ifndef MYFILE_H
#define MYFILE_H
[...]
#endif  /*!MYFILE_H*/
```

4. For class definition files, the file name should equal the class name.

5. Names of data types, classes, typedefs, structures/unions start with a capital, e.g. MyClass.

6. All instances (variables, constants, classes, functions) start lower case, e.g. setValue, nextChoice().

7. Use capitals to separate words in a name, e.g. myFloatVariable, HerSmartClass.

8. Globals must start with a g, e.g. gHades, and we recommend to start constants with a k, e.g. kUnit.

9. Preprocessor constants and macros are all upper case with no leading or trailing underscores, e.g. TWO_PI, SUM(A,B).

10. The following convention should be used for identifiers of larger scope:

    - Hades base classes start with an H
    - RICH-relevant classes start with HRich
    - MDC-relevant classes start with HMdc
    - SHOWER-relevant classes start with HShow
    - TOF-relevant classes start with HTof
    - START-detector classes begin with HStart
    - ORACLE-relevant classes start with HOra

11. Use ROOT types whenever possible, e.g. Int_t instead of int. This rule is mandatory for streamed data members.

12. Decrease the number of #includes in .h files, put them into the .cc files and use class forward declarations in the .h (if only pointers to a classes are declared), e.g.:

```
class MyClass;        rather than     #include "myclass.h"
 class YourClass{                      class YourClass{
    MyClass* f();                         MyClass* f();
 };                                    };
```

13. Comment your code. To be compatible with the automatic ROOT html documentation scheme, comments should be placed in 3 places: 1) a class description on top of the .cc and the .h files, data member comments in the .h file and function member comments following the function names in the .cc file. Critical or not so obvious spots of the code should also be commented.

4.2. Overview

The general analysis of the problem of producing a reconstruction program for HADES involves some basic concepts, already mentioned when speaking about the requirements:

- Data (real, simulated, reconstructed, ...)
- Input/Output procedures.
- Parameters (geometry of the detectors, calibration of the detectors, unpacking information ...)
- Algorithms, tasks.

These concepts have been incorporated in the following sets of classes:

- Fundamental class: Hades
  Hades is the class which encapsulates the whole reconstruction program, providing methods to control the different tasks which can be realized. This includes methods to set the different inputs as well as methods to launch the reconstruction process itself.

  The Hades class holds objects of different classes; coordinating its behavior and getting from them the necessary services to make its job. The composition of the Hades class can be seen in figure 4.1. See also section 4.3

- Classes to contain data
  Since the reconstruction process is done event per event, the basic data structure is the **event**. From the point of view of Physics, an event holds all the information collected by the different detectors in the spectrometer regarding one interaction between one beam particle and the target; moreover, it also stores the totally or partially reconstructed data which are derived from the original signals. But we can have other types of events which do not contain information about an interaction in the target; for example, signals produced in the detectors by a particle with the purpose of calibration. There can be, therefore, different kind of events: real events, simulated events, calibration events, ...

  An event is represented by a HEvent object; being HEvent an abstract class. This allows (and enforces) to inherit from HEvent other classes which correspond to the different kind of events we can find. Within an HEvent data are organized in "categories", being a category a set of data of the same kind (i.e. raw data, calibrated data for MDC ...). Categories, on their side, hold the data objects (inheriting from HDataObject) in an arbitrary way but allowing the access to those data as if they were in a matrix; i.e., through an array of indexes which is encapsulated in a HLocation object. See also section 4.4

- Classes to manage input/output of data
  Once we have a place where to store an event's information, we need a tool to read that information in. Since the data can come from different sources (as has been said in the requirements chapter) we must set a generic interface so the Hades class only needs to know that Application Program Interface (API); and then we can have a different implementation for that API for each data source. To accomplish this, we only need to set an Abstract Base Class (ABC) defining the desired API; this class is HDataSource. In this way, the inherited classes can provide different implementations corresponding to the different kind of data sources. See also section 4.5

- Classes to contain and to manage all the numerical information needed to process the data (parameters)
  In order to do the event reconstruction, several numerical parameters are needed, as for example, calibration parameters or geometry positions of detectors. One common characteristic to most of these parameters is that they will go through several different versions; corresponding, for example, to changes in the spectrometer or any other condition which can make them change. This makes necessary to have a parameter repository with some versioning system. The runtime database (realized through the HRuntimeDb class) is such

a repository. See also section 4.6

- Classes to perform tasks
  For each event we need to accomplish a certain task; which is represented by an HTask object. Again, HTask is an abstract class defining a generic API allowing to execute one task and to navigate through a task set.

  HReconstructor and HTaskSet are two particular subclasses inheriting from HTask; the first of them is meant for reconstruction algorithms, while the second one allows us to group several tasks in one. For more information see section 4.7.

4.3. The HADES class

The Hades class is the fundamental class which controls and coordinates all the different parts of the reconstruction software. Essentially it is formed by (see figure 4.1):

- a data source where to read event's data from
- a HTaskSet storing the tasks to perform for each event
- a HEvent where to store the information in process
- a HSpectrometer created during initialization and storing information about the spectrometer's structure
- a database where to read reconstruction parameters from
- a ROOT output tree
- an output file

There must be one and only one object instantiating the Hades class for a execution of the program, that is, the Hades class is a "soliton". This object is accessible from every part of the program through a global pointer which is called "gHades". For more information on this class and the services provided by it, check the reference documentation.

Figura 4.1: Hades class structure.

4.4. Classes to contain data

4.4.1. The event

A physical event is an interaction between a beam particle and the target, and it can be real or simulated. A calibration event contains the response of one or several detectors to one or several particles or to a calibration signal (a laser signal, for example). The event is the unit for the data processing. From the reconstruction program's point of view, an event is an object instantiating some HEvent subclass and holding all relevant information coming from a beam-target interaction or resulting from a calibration signal. The event can contain both the original data coming from the spectrometer (raw data) and the more elaborated data which result from the reconstruction process.

One event is reconstructed in steps, so each step in the reconstruction process produces one level of reconstructed data. The number and kind of reconstruction levels (or data levels) which are stored in an event is not fixed beforehand, since it can change as a function of the kind of event (simulated, real ...) as well as the specific task we are accomplishing at a given moment. If, for example, we are studying the calibration parameters for the MDC it makes no sense to carry neither every data level for the other detectors nor those MDC levels which are not used.

There is only one HEvent object within the Hades soliton. This HEvent object acts as a central repository, globally accesible, with all the information for one event; storing also structural information about that event. In this way, the different components of the reconstruction program (event display, data input, reconstruction algorithms ...) can access the event information in an independent way.

Data contained in an event are HDataObject objects. Within each event, these objects are organized in categories; that is, the event holds "categories" [1] and these categories hold the data objects. During the initialization of the program the user decides which categories (how many and what kind of) he wants to have in the event; as well as the kind of data objects stored in each category.

To access a particular category within an event, we can use the **getCategory** (Cat_t aCat) function from HEvent. This function returns the event's category identified by "aCat", being "aCat" the value of a constant which univocaly identifies one particular category (for example, catMdcRaw for the category holding raw data in the MDC).

As for the event storage in an output file, ROOT provides us with an automatic

[1] HCategory objects

mechanism to store any ROOT object into a file, this can seem enough at a first glance. However, it turns out to be convenient to store the event's information in a more adequate and ordered form for its further analysis. In particular, we want to store event's information using a ROOT tree (see appendix A). This is the reason for a function **makeBranch**() appearing at HEvent's declaration[2].

In addition to storing the data objects we must be able to clear the information held in a HEvent so to leave free place for the next event. This can be better understood watching at the basic reconstruction cycle; the steps are the following:

1. Clear information in the current event
2. Read information from the active data source
3. Launch the reconstruction for the current event
4. Store the event's data in an output file

To accomplish the first step in this list we can use the functions **Clear**() and **clearAll**(Int_t level). The first of them deletes all data objects in the event but preserving its structure. The second one, on the other side, deletes both the data objects and the part of the event's structure which is selected in the parameter "level" [3]

These are the fundamental characteristics of a general event. Now we will see different kind of events and how the previous functions are implemented for each case. If you prefer to know more about the "categories" before; just jump to section 4.4.2.

4.4.1.1. The event under reconstruction

An event under reconstruction is represented by an HRecEvent object and corresponds to a physical, totally or partially reconstructed event. This event is divided in several partial events corresponding to the different detection systems (MDC, RICH ...); and these hold the categories corresponding to the reconstruction levels for each partial event (you can see a diagram with HRecEvent's structure in figure 4.2. When you ask a HRecEvent for a category, it translates the request to the partial event, which is the real container of the categories.

[2]In principle, ROOT has an automatic mechanism to build a tree from any object; however this mechanism doesn't qualify the flexibility demands we have for an object as complex as an HEvent

[3]For example, if level=0 every data object as well as the whole event's structure will be deleted

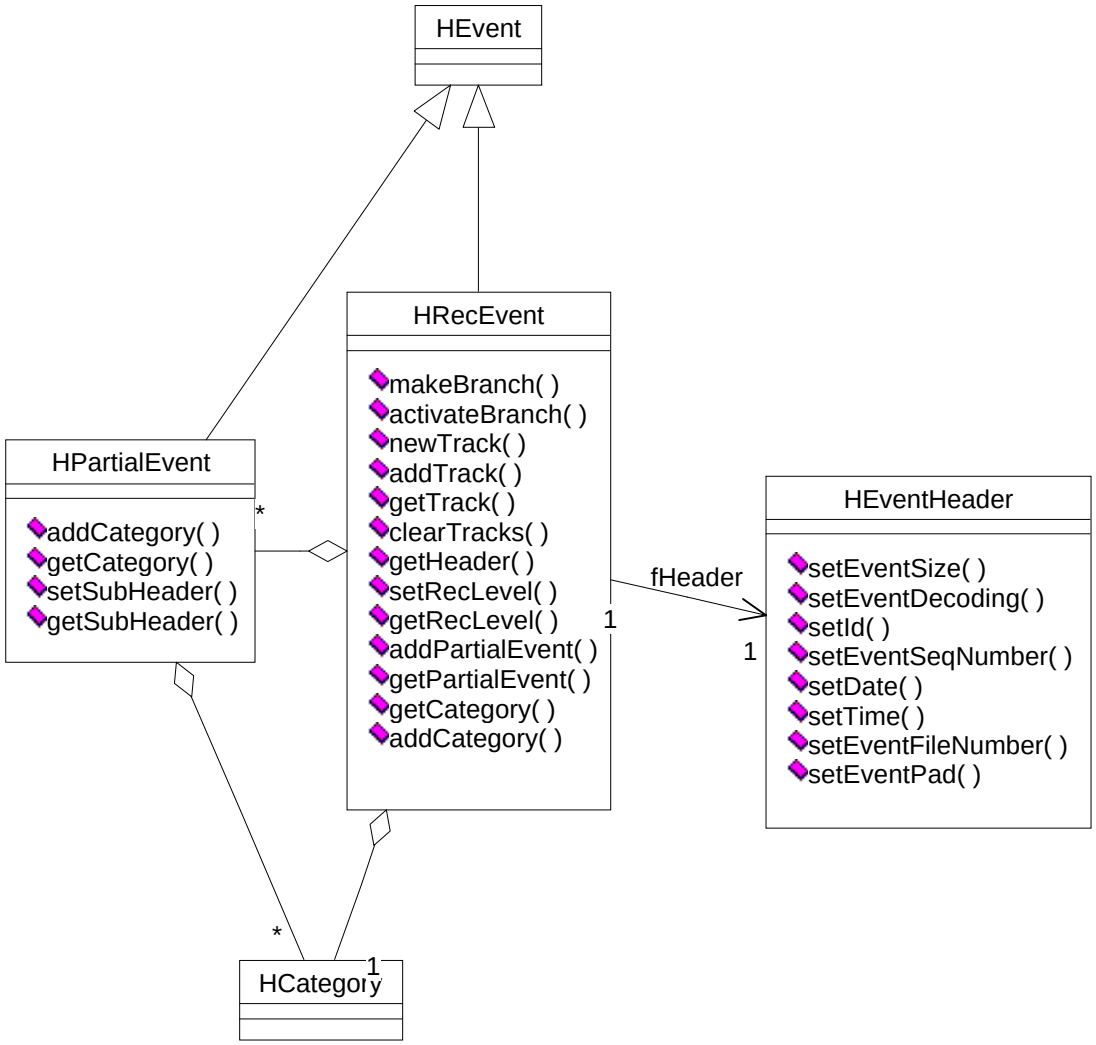


Figura 4.2: HRecEvent structure.

The HRecEvent contains one or several HPartialEvent objects. In addition it also contains a category with tracks.

Aside partial events and their categories we have another category in a HRecEvent which doesn't belong to any partial event. This category holds the final tracks once they are reconstructed and it is identified by the constant "catTrack".

One needs to store more information in a physical event than the data objects themselves; therefore each HRecEvent has also a header (object of class HEventHeader). This header stores global information about the event, like event number, run number, date, trigger ... To access an HRecEvent's header one can use the **getHeader**() method from HRecEvent.

Another information stored in this kind of event is a number meaning its reconstruction level. Essentially that number can take just two values: "rlRaw" and "rlHit" which correspond to an non reconstructed event and a completely reconstructed one. To get or change the reconstruction level in a event, one can use **getRecLevel**() and **setRecLevel**().

Each partial event is identified within the HRecEvent to which it belongs by a numerical constant whose value is that of the first category one can store within that partial event. These numerical constants, which identify each category in a HEvent, are defined in the "haddef.h" header file as well as the different "XXXdef.h" files for each detector (like "tofdef.h" and "mdcdef.h") and they have names like "catMdcRaw" for the category containig "raw" data for the MDCs. By including declarations for these constants in the "XXXLinkDef.h" header files, they are also available within the ROOT interpreter.

4.4.1.2. The partial event: HPartialEvent

As its name suggests, a partial event is part of an event under reconstruction. In fact it is each part of an HRecEvent which has to do with a particular detection system. So, for Hades, we have one partial event for the RICH, another one for the MDC, etc.

Each partial event holds an array with all the categories belonging to it; so we can get any of them through the **getCategory**() method.

Aside the categories, each partial event maintains a "reconstruction level" in the same way as the HRecEvent. This allows one to know what is the state of the reconstruction for some event.

Obviously, this kind of event has also all the functions required to any "HEvent" like those intended to build an output ROOT tree starting from the categories array held by the event.

4.4.1.3. The simulated event

The simulated events are the events produced by the Hades simulation program, which is based on GEANT. Simulated events can be used as input for the reconstruction program instead of real ones in order to test the software, therefore the simulated events must conform to the same structure as the real ones, so the software can seamlessly process both real and simulated events. This is automatically granted by inheriting our simulated event class (HSimEvent) from the generic HEvent, provided that all the reconstruction algorithms are only based on the interface defined by that class.

On the other hand, a simulated event holds more information than a real one; in particular in a simulated event we know the collision kinematics, or the dilepton tracks, for example. Therefore the simulated event class needs to hold more information which is not available in a real event.

The most likely implementation to deal with these requirements will be, then, to inherit a new HSimEvent class from HEvent so that this class contains a HRecEvent and some more categories for kinematic information and global collision information. This grants the software will run without modification on both kind of events and, at the same time, allows the simulated event to hold some extra information which can be used for some specific tasks; mainly while debugging the reconstruction algorithms.

4.4.2. The categories

A category is essentially a container of objects within an event, with the extra point that every object in a category belongs to (instantiate) the same class. For example, the raw data for MDC make up a category; but raw data in the RICH correspond to a different category since they are instances of a different class. Other categories can be: the one storing calibrated MDC data, etc.

The category concept is represented by the HCategory class; in fact this is an abstract class with declares a basic API which must be implemented by any kind of category. These implementations correspond to different strategies for storing data both in memory and in file; being ones better than the others depending on the situation. In figure 4.3 you can see the HCategory's definition as well as some inherited classes.

A category's API must have functions to access the data objects held by it. This access can be of two kind; we can ask for one single data object or a set of them

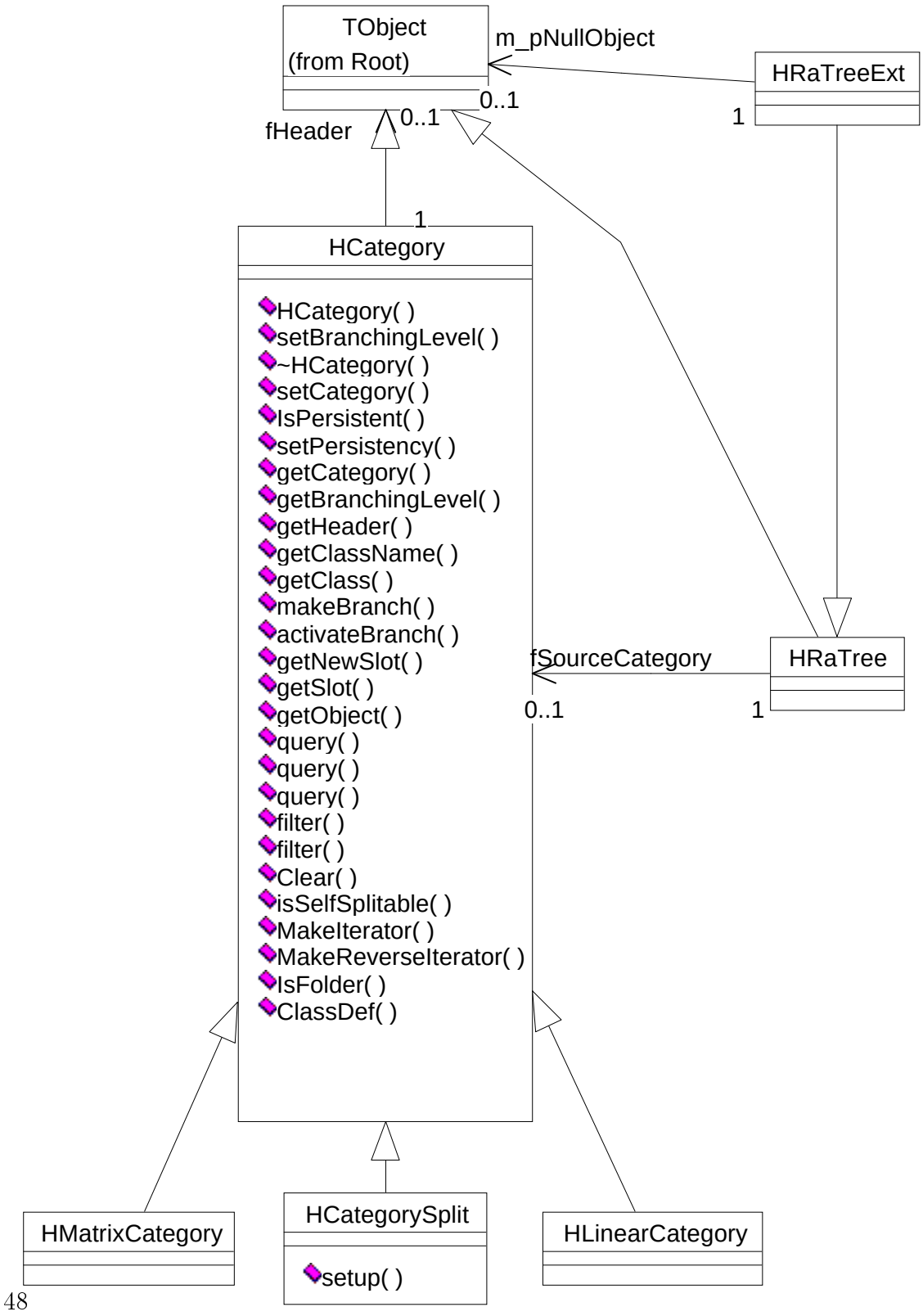


Figura 4.3: HCategory structure

verifying some condition. Another kind of access is iteration on all or part of the objects held by a category.

To access a particular object in a category we need "something" which identifies it in a univocal way. This "something" is an object instantiating the HLocation class and it's no more than an array of indexes. So, as we can see, it's like if data objects in a category were stored in a matrix. Summarizing, each object is stored in a category at the direction defined by a set of indexes encapsulated in a HLocation object; then, to access the data object we can use **getLocation**(HLocation &loc). The next example will help to make it clearer:

```
{
//Let's say cat is a category with raw MDC data.
HCategory *cat;
//A raw hit in a MDC
HMdcRaw *raw;
//A new location object
HLocation loc;

//Let's set loc pointing to the fourth hit at the first layer
//of MDC 2 in second sector. For this, we need to call
//HLocation::set(n,...) with the number of indexes in the
//location as the first argument, and then the actual indexes
//themselves, in order.
loc.set(4,2,2,1,4);

//Let's set raw pointing to the desired data object. This is
//accomplished by calling the getObject(loc) method from
//class HCategory. This method returns a pointer to the
//object in the category at the location given by the
//method's argument (loc).
raw=cat->getObject(loc);
}
```

If we want a set of data verifying some condition, then our friend is **query** (TCollection *aCol, HLocation &loc, HFilter &filter)) This function places within the collection aCol every object in the category which verifies the condition given by the filter "filter"[4] and corresponding to the location "loc". If "loc" is omitted, then

[4]See the HFilter class in the reference documentation

any location is valid. If "filter" is not specified every object corresponding to the location "loc" is added to the collection. Let's see an example:

```
{
//Let cat be a category with MDC raw data. Each raw data is
//identified by 4 indexes: sector,module,layer and cell.
HCategory *cat;
//Let array be the target array of selected data objects
TObjArray *array;

//Let's set loc pointing to the first module in the first
//sector
HLocation loc;
loc.set(2,0,0);

//Let filter be a filter implementing condition ``cond1''
HCond1Filter filter;

//Do the job. Now we have array filled with those data
//objects in the category which correspond to the first
//module in the first sector of the MDC and verify the
//condition ``cond1''
cat->query(array,loc,filter);
}
```

At the end we have an iteration on all or part of a category; this is accomplished using iterators, in the Standard Template Library (STL) way. To get an iterator for a category we can use the function **MakeIterator**(); this function will return an HIterator object iterating on the whole category. If we want to restrict the iteration to a location we can use the **gotoLocation**(HLocation &loc) method from HIterator. Let's see now one example with an iterator running on all raw data for chambers 1 and 2:

```
{
//The usual stuff
HCategory *cat;
HMdcRaw *raw;
HLocation loc;
```

```
//Set loc pointing to sector 1, module 2
loc.set(2,1,2)

//Build the iterator up
HIterator *iterator=cat->MakeIterator();

//Now we do a loop on the data objects using the iterator we
//got before. This is accomplished with a "while" loop whose
//condition equals te pointer "raw" to the next data object
//in the category and checks if "raw" is different from
//NULL; once raw==NULL the iteration stops.
while ( (raw=(HMdcRaw *)iterator->Next())!=NULL) {
  raw->Dump(); //print the data object pointed by "raw"
}
}
```

Besides having objects stored in a category we must be able to add new objects to that category. The adopted solution consists in the user having to ask the category for a place in memory (a slot) where to place the new object; then the user instantiates the object using the "new with placement"[5] operator. In case the object is not instantiated using the "new" operator, what you have is just a piece of memory, not a real object; that means, for example that the virtual table is not built and therefore no virtual function can be called. Since each object in a category is associated to a location, to get a slot we use the method **getSlot**(HLocation &loc) if we now all the indexes in the location, or we can use **getNewSlot**(HLocation &loc) if we know all the indexes in the desired location but the last one. Any of these two functions will return a pointer to the requested slot, or NULL if no slot was available at the requested location.

The main reason to let the category do the memory management instead of simply using the C++ "new" operator comes from the high number of data objects instantiated per event, and the high number of events to process. The "new" operator calls a cost-full routine in the operating system to get the requested memory. However a category can have a preallocated block of memory for the data objects which are going to be instantiated; this can speed memory management up because

[5]The "new with placement" operator is used to instantiate an object at a given memory address. The syntaz to instantiate an object of class, let's say HMdcRaw, at the address pointed by a pointer named "pMemAddress" is: "raw=new(pMemAddress) HMdcRaw". Where, "raw" is a pointer to HMdcRaw. Note that the "new" operator doesn't need to actually allocate memory but uses the memory pointed by "pMemAddress" assuming it is already allocated

the category knows beforehand the size of the data objects which are going to be instantiated, as well as the kind of memory requests it will be asked for. Let's see now an example:

```
{
 //The usual stuff
 HLocation loc;
 HMdcRaw *raw;
 HCategory *cat;
 ...
 //Set loc pointing to sector 2, module 2, layer 1,cell 1
 loc.set(3,2,2,1,1)
 //Ask for a slot
 raw=cat->getSlot(loc);

 //If the slot is valid (raw!=NULL) instantiate the object
 if (raw!=NULL) raw=new(raw) HMdcRaw;
 else Error("No slot available");
}
```

Following is the description of some kind of categories which have been implemented; this description deals with specific issues for each category, in particular its implementation.

4.4.2.1. The HMatrixCategory

This kind of category store data objects in a matrix-like structure. In this way, when we ask for an object in the category, the location indexes which identify the objects are the same as the indexes of the underlying matrix. To initialize a matrix category one needs to provide the following data in the constructor:

- Number of indexes in the matrix
- Maximum value for each of the indexes (that is, matrix dimensions)
- fillRate, this is a number between 0 and 1 which corresponds to the maximum fraction of occupied locations we expect.

Looking in more detail into this category's implementation we don't see the mentioned matrix anywhere. That's because in practice, the data objects are stored in an array (a TClonesArray from ROOT).

The internal structure of the category is: on one side we have a TClonesArray A with every data object, and we have a HIndexTable object T which behaves as a matrix of integers. When we are looking for an object associated to a location, we get from T the matrix element corresponding to the indexes of that location; this matrix element is an integer giving the position in A of the desired data object.

In this way it is not needed to get for A all the memory which would be used if every location was full; and we can keep the TClonesArray without holes (becoming this fact critical when we want to store the array in an output file)

We have already said that HIndexTable behaves as an integer matrix; however, again, we can see that internally we have an array of integers. This is done like that so to be able to work with an arbitrary number of indexes in our matrixes.

4.4.2.2. The HCategorySplit

To understand what this category does, we have to define beforehand the idea of "terminal" which will be used in the remaining of this section. Given a category where each data object is identified by n indexes, we call "terminal" to each location with n-1 indexes. An example will make this clearer: let's consider raw data in MDC; each data object is identified by 4 indexes (sector,module,layer,cell), therefore a "terminal" corresponds to a layer (location with $4 - 1 = 3$ indexes).

What makes a HCategorySplit special is its ability to store the data objects for each "terminal" in an independent TClonesArray. So that when generating the ROOT output tree we have one branch for each "terminal"

The category is internally made up of a matrix of pointers to TClonesArray objects; these, on their side, hold the data objects for each "terminal". As usual, the mentioned pointer matrix is realized, in practice, as an array. You can see this structure in figure 4.4

Using TClonesArrays directly brings about an important consequence: one should not leave holes on the nth index when filling in a HCategorySplit. If this rule is not respected one will get a "segmentation violation" when storing the category in split mode. This means we will not be able to write to a file in split mode if we have one object at (1,2,1,0) and another at (1,2,1,2) and nothing in (1,2,1,1). But there is no problem having one in (1,2,1,0) and another at (1,2,3,0); or if we store data in non split mode.

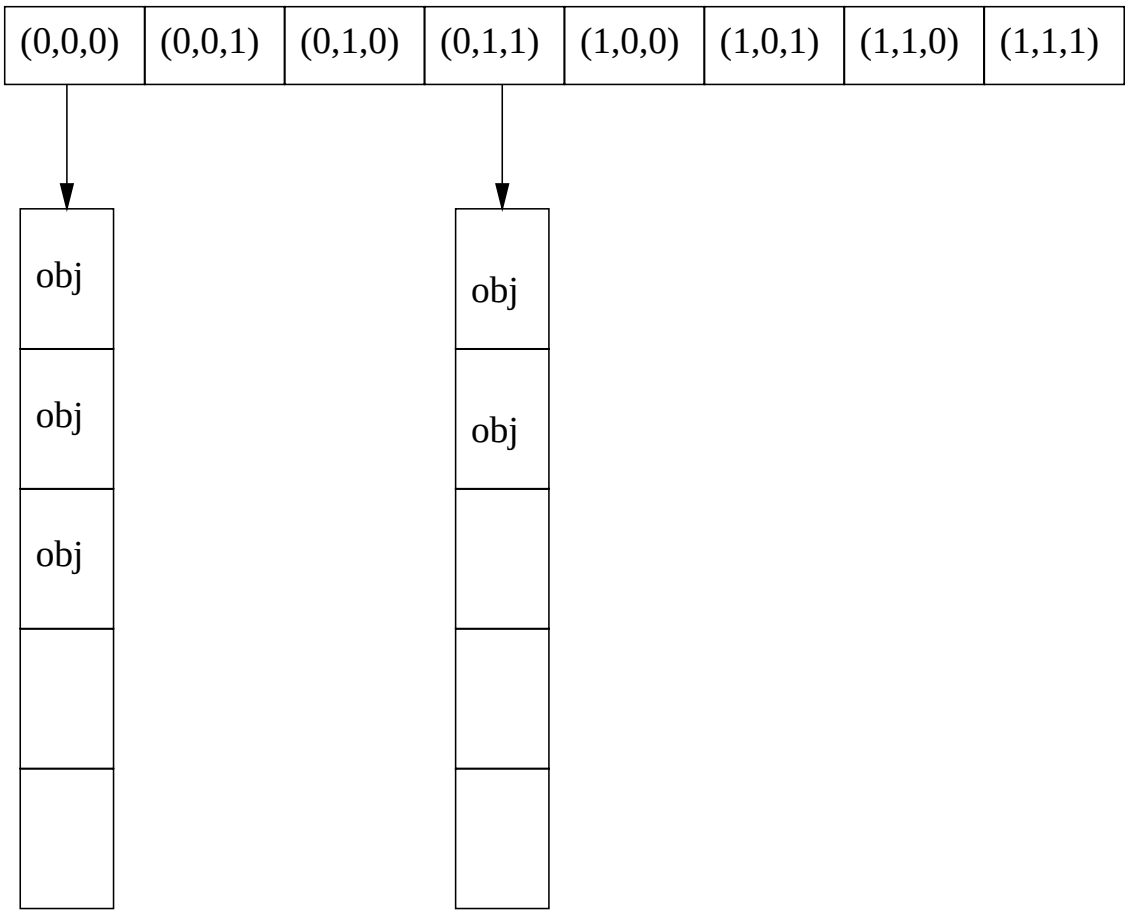


Figura 4.4: HCategorySplit structure

As for initialization, this is done in two steps. In the first step, when the category is instantiated, one must set:

- Class name for the data objects to store in the category.
- Number of indexes needed to identify one "terminal"
- Dimensions of the "terminals" matrix.
- Pattern to name each of the branches for the different "terminals". In order to produce those names, a loop is done on all the active "terminals" in the category and for each "terminal", its location is matched against the beforementioned pattern in order to produce one unique name. The matching is done by copying each character in the pattern to the branch's name until we find a sequence like " %i %", which is substituted by the value plus 1 of i-th index in the location for the current "terminal", then the following characters in the pattern are copied to the branch's name until another sequence like " %i %" is found and so on until one reaches the end of the pattern. For example, if our category has 3 indexes and the "terminal" matrix has dimensions $2 * 2$; a pattern like "S %0 %.M %1 %" will cause the branches to be created with names:
  - S1.M1
  - S1.M2
  - S2.M1
  - S2.M2

The second step consists in calling one of the **setup**() functions to set the active "terminals". That is, which modules we want memory and an output branch for. In order to set this, two ways are allowed; whether one provides the number of active "terminals" and their number, or a table of integers (one per module) where -1 means an inactive "terminal" and a number greater than 0 corresponds to the numbers of data objects expected for that "terminal"

4.4.2.3. The HCategoryMatrixSplit

Essentially it is the same as the HCategorySplit. In fact, it inherits from HCategorySplit. The main difference between the two is that HCategoryMatrixSplit uses HClonesTable objects instead of TClonesArray. A HClonesTable is a descendant of

TClonesArray, but modified in order to allow having holes even in split mode. On the other hand they are more complex and slower when accessing one particular data object.

4.4.2.4. The HLinearCategory

This is the simplest kind of category we'll speak about. In fact, a HLinearCategory is nothing more than a wrapper to a TClonesArray so it can be used within the framework.

Therefore, the data stored in a category like this are identified by one single index (the location has just one index) which corresponds to the position of the data object in the underlying TClonesArray.

This category can be useful in a variety of processes like calibration. Let's say, for example, that we want to go from raw data in MDC to calibrated data; each raw data is identified by four indexes (sector, module, layer, cell). The first step is to read data from the acquisition system and place them in the "catMdcRaw" category. After that the data are calibrated.

In this example, one possibility is to place the data in the category without an order (putting data in a HLinearCategory as we read them) and store the four indexes as a data member of the data object. Later, during calibration, we iterate over all data objects, and for each of them we do the calibration with the parameters specified by the indexes stored in the data object.

4.5. Classes to manage input/output of data

This sections describes essentially which mechanisms are foreseen in the framework both for data reading and writing. In the first case, the adopted solution must be able to deal with several input sources; on the other hand, data output is always realized through ROOT files and using, essentially, but not only, ROOT trees.

4.5.1. Data input

In this section we will describe how the data are read from the different available sources.

The only thing the Hades class needs to know is the definition of "data source"

in terms of C++; that is, which methods are provided by a "data source" and their meaning. In this way we can call those methods without knowing which concrete source is used.

The abstract class defining a data source is HDataSource, and mainly defines one function **getNextEvent**() which must be implemented by all the inherited classes. When this function is called one new event is read from the data source into the event structure. The returned value of the operation can be one of the following:

kDsOk: the event was successfully read

kDsEndFile: we have reached the end of one file (set of data with the same reconstruction parameters), but more data are available

kDsEndData: we have reached the end of the data source

kDsError: error

Up to now there is place for only one data source within the Hades soliton; so if we want to combine several data sources we would have to implement one new data source as the union of them.

Another very important function of the HDataSource class is the **init**() method, used during initialization. Within this method each particular data source must check whether an event object exists or not and if it doesn't exist then it is the data source's responsibility to instantiate an event object. Usually, the data source will also have to add to the instantiated event object those categories where data will be read. Note that if an event object or the needed categories exist, then the data source must not destroy them, but use them directly.

4.5.1.1. Data input from the Data Acquisition System: HldSource

HldSource is the base class for those data sources reading data from the HADES acquisition system (DAQ). The class structure can be seen in figure 4.5.

The HldSource reads raw data in the order and format provided by the DAQ and places them at their place within the event structure; this usually implies some reordering. This process is what is known as unpacking and is realized by unpackers (objects instantiating the HldUnpack class) within a HldSource.

HldUnpack is an abstract class from which several different unpackers are inherited, as HRichUnpacker or HTofUnpacker. In fact, we have one different unpacker for each detection system in HADES (MDC, TOF, RICH, SHOWER, START), so each unpacker only knows how to deal with a particular kind of data. The most important method of this class is the **execute**() function, where the unpacking process is realized. Another important function is **init**() which is used during the initialization procedure. Within this function, the unpacker has to do the following:

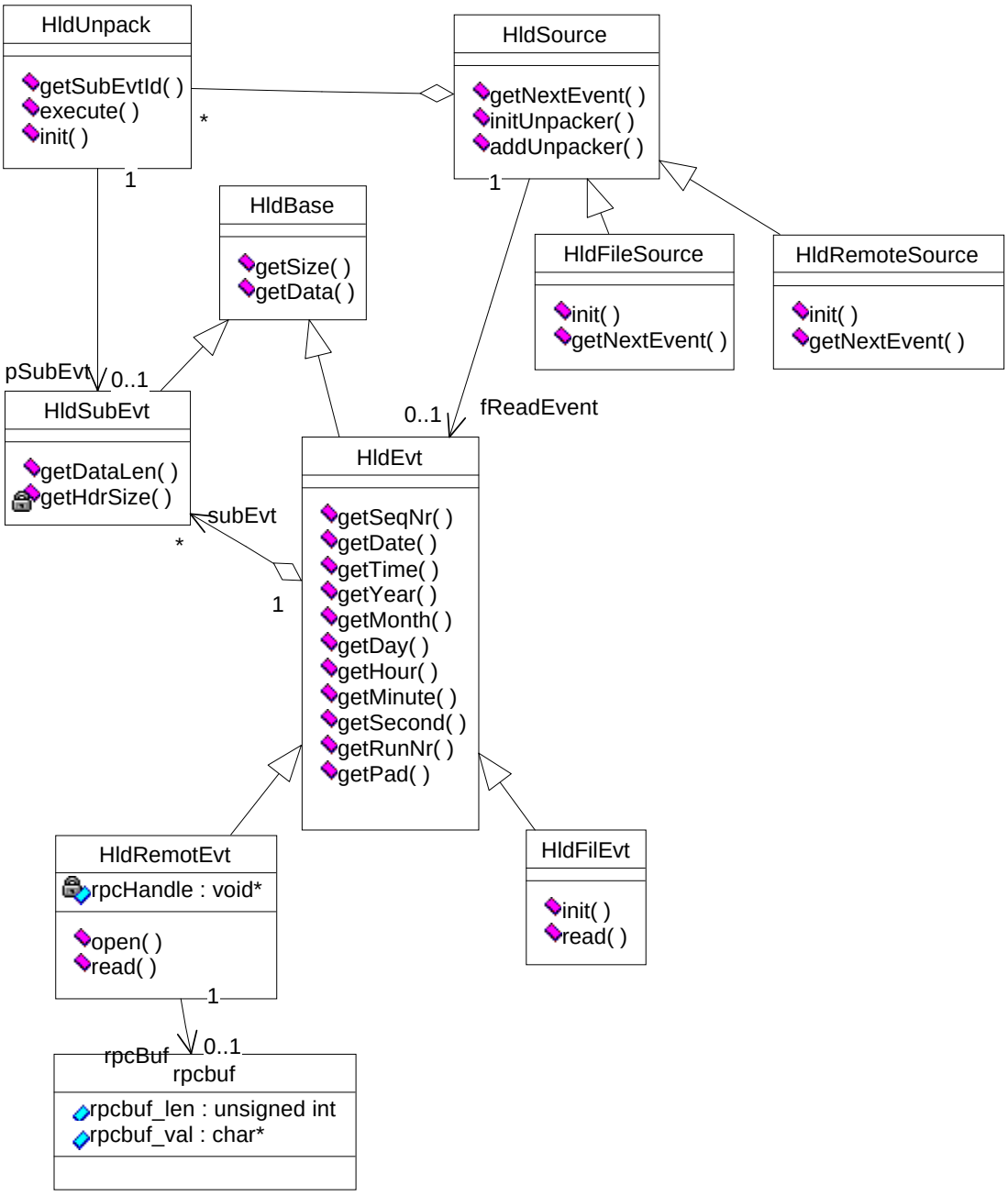


Figura 4.5: HLdSource

- Get from the event structure pointers to the category where read events will we written. If the needed category is not in the event structure, then it is responsibility of the unpacker to instantiate it and add it to the event structure. The recommended way to do such a instantiation is through the HDetector classes which will be seen later.
- Get pointers to the parameter containers from the runtime database. If a needed container is not in the data base, then it is responsibility of the unpacker to instantiate it and add it to the data base; but not to initialize the container.
- Do other specific initialization.

The HldSource maintains a list of active unpackers at a given moment [6] so that only the information corresponding to those unpackers is actually processed. This kind of modular organization allows to select which kind of information we want to read as well as supporting the situation of ".hld" files only containing data for part of the spectrometer (which is an usual situation); furthermore it makes easier to incorporate not previously foreseen changes of the spectrometer into the analysis software (as adding a new detector, modifying the data format for a detector ...)

This has been the general information about a HldSource. However, in practice we will use always one of its subclasses like HldFileSource or HldRemoteSource. Both of them work in a very similar way, being the main difference the fact that the first one reads data from a file and the second one reads data from a RPC connection through the net. Therefore, the first one is more suitable for offline analysis, while the first one is more suitable for online analysis.

An example of initialization for a HldFileSource is:

```
{
 HldFileSource *source=new HldFileSource;
 source->addUnpacker(new HRichUnpacker);
}
```

Note that the unpackers used both in HldFileSource and HldRemoteSource are just the same; this is possible thanks to the common infrastructure in HldSource.

Describing with more detail, what happens when the **getNextEvent**() function is called is the following:

[6] This list is built by the user himself in the initialization of the HldSource using **addUnpacker** (HldUnpacker *unpacker)

1. A buffer is filled with the information to be unpacked. This buffer is a HldEvt object inheriting from HldBase; it stores generic information about the read event (event number, bytes used ...). Each HldEvt is made of sub-events, HldSubEvt objects, which are read at the same time as the HldEvt.

2. The **execute**() function is called for each of the active unpackers. Each unpacker has an associated HldSubEvt where it reads data from, transforming those data into objects and placing them into the event structure.

4.5.1.2. Simulated data input: HGeantSource

HGeantSource is another kind of data source which allows us to read into the event's structure data stored in ntuples in one or several files. This data source is intended to read output data from GEANT, which are stored in ntuples.

As with HldSource, the ntuples format depends on the detection system, so, like before, the adopted solution consists in defining a class for every system. Therefore, we have a HGeantReader class playing here the same role as HldUnpack in HldSource, and different subclasses for the different detection systems, like HTofGReader or HMdcGReader.

In addition to this, HGeantSource also manages a list with all the files where the ntuples are located, in such a way that the reader classes don't need to worry about their ntuples being in one single file or spread through several of them.

The list of readers as well as the input files used by the HGeantSource are specified by the user during the program's initialization. One other important question is that, unlike HldSource, those data in the input file for which no HGeantReader exist are not read into any intermediate buffer.

4.5.1.3. Partially reconstructed data: HRootSource

In this case, the data source is a ROOT file holding an event tree. Usually this tree would have been generated by the reconstruction program itself and holds completely or partially reconstructed events.

As for the internals, the only remarkable thing is the use of **activateBrach**() and **activateTree**() from the HEvent, HCategory (the earlier) and Hades classes. These methods are used to associate the memory position where data are read to the corresponding branch.

4.5.2. Data output

The ROOT facilities are used for data output; both object serialization and ROOT trees.

The Hades soliton itself manages an output file if the user wants to have one. In this file both the reconstruction events and the relevant information about how those events were reconstructed are stored. That is, besides the reconstructed events it is also stored:

- The event structure. That is, how many categories and of which kind the event contains
- Which algorithms were used for the reconstruction
- The parameters used by the reconstruction algorithms. That is, geometry calibration parameters, etc.

To set the output file we have to call the **setOutputFile**(Tex_t *name, Option_t *opt, Text_t *title, Int_t comp) during initialization (see section 4.8). Where:

"name": is the file name

"opt": indicates if the file is opened for writing (opt="UPDATE"), reading, etc.

"title": is an optional title for the file

"comp": indicates the compression level for the output file from 0 to 9

As for how data are stored in the output file, in first place a new entry for an Hades object is created in the output file, so the global object "gHades" is stored there. Even though the event structure and the event tree are parts of gHades, entries are also created for them in the output file's top level for convenience. In this way, we can access them in two different ways: through gHades or directly.

The events are stored using a ROOT tree whose structure[7] is determined by the event structure and by the so called "split level". The split level is a number, stored in the Hades class, which controls the branching level in the output ROOT tree. In principle the allowed values for this "split level" are:

[7] The branch layout

0: Only one branch is created for the whole event, which is stored as a whole

1: There is one branch for each partial event, which is stored as a whole. However, the header, final track and some other data are stored creating one branch per data member

2: One branch is created for each category, and hanging of it, one branch per data member of the class contained in the category. However, each category still can decide how the branching is going to be done.

So we see how the split level tells us till which level the event structure is expanded in the output tree. In any case the value of the "split level" is just a hint and how the splitting is actually done is decided by the event classes (HRecEvent, HPartialEvent, ...). The split level can be set with **setSplitLevel**(Int_t sl) from the Hades class.

Any category can decide how it is going to split its data. For example, the HMatrixCategory creates one single branch for all its data, and hanging from that branch one sub branch per data member in the class held by the category. However, HCategorySplit builds one independent branch per "terminal" [8] with sub branches for each data member in the class stored in the category.

One common characteristic for all categories and which affects the output file is the persistence. We can decide on a per category basis if a category is or is not persistent; that is, if it will be stored or not in the output file. A category's persistence is controlled through **setPersistency**(Bool_t per).

4.6. Classes for the reconstruction parameters

The reconstruction parameters include all those numbers, which are needed for the reconstruction process, as for example, positions or dimensions of the detectors, calibration parameters, pattern recognition parameters, etc. These numbers are organized in sets of functionally related parameters. Each of this sets is represented by a subclass of HParSet, so our generic "parameter set" is a HParSet. Each set of parameters can also have different versions, corresponding to changes in the spectrometer ...So, for example, the MDC calibration parameters can have a different version for each event file since the numbers can change for each event file[9]. Furthermore, there can be different versioning sequences. This gets clear with

[8] See the definition of "terminal" in section4.4.2.2

[9] An event file is a sequential set of data with the same reconstruction parameters

an example: let's consider the calibration parameters for MDC; there must be an official set of this parameters for each event file; but when the responsible of the MDC calibration is computing the calibration parameters he is likely to have lots of temporal preliminary versions of the parameters, which cannot go into the official repository, even though they are versions and require a proper version management.

The parameters can come from different sources, to manage this fact the HParIo class is defined. This class manages input and output of the parameters from or to the different sources. In principle three sources are foreseen:

ORACLE: it is a commercial database where the official parameters will be stored. This data base is maintained at GSI

ASCII file: this possibility is intended to allow users an easy access to the parameters so they can easily play with them

ROOT file: this mechanism is automatically provided by ROOT and it is a convenient way of having local copies of the reconstruction parameters in the laboratories. In this way downloading the parameters from ORACLE is not needed

Now that we have a place where to put data and a mechanism to read and write data we need "somebody" to manage all this stuff. This job is done by a runtime database, which is a HRuntimeDb object within the Hades soliton. This object is responsible of the version management and it's the owner of all the parameter containers; providing functions to get/add parameter containers to the database as well as functions to update the database.

We will see now in more detail how the HParIo and HRuntimeDb work.

4.6.1. Input and output: HParIo

A HParIo is, essentially, an array of HDetParIo objects. The HDetParIo class defines the generic interface used to actually read and write the parameter containers; this API consists mainly of two functions:

init(HParSet *par,Int_t *set): Fills the "par" container in. Being "set" an array of active modules.

write(HParSet *par): Writes out the "par" container

The way down to the concrete implementation of these functions has two levels. One first level sets the "source" by inheriting a class from HParIo and another from HDetParIo for the particular data source; let's call them HParXXXIo and HDetParXXXIo. The first of them handles generic questions about the particular source, while the second one handles those details which can differ from detector to detector. The second level of concretion consists in defining a HDetParXXXIo subclass for each detection system; this subclass will have a **init**() and **write**() function for each supported parameter container. Let's consider, for example, input and output for ROOT file and for MDC parameters. The first level sets the "source", which in our case is ROOT, defining two classes: HParFileIo and HDetParFileIo, which have nothing to do with MDC and are also used for TOF or RICH. The second level sets the detector, MDC for us, by derivating the HMdcFileIo from HDetParFileIo. This class, HMdcFileIo, has several **init**() and **write**() methods, one for each kind of parameter container managed by the class.

4.6.2. The runtime database

You can see the structure of the runtime database in figure 4.6. Essentially it consists of 3 HParIo objects and a list of parameter containers (HParSet objects).

Each container in the list is identified by a name, so you can retrieve a container given its name with the function **HRuntimeDb::getContainer**(). You can also add new containers to the list with **HRuntimeDb::addContainer**().

As for the 3 HParIo objects; two of them correspond to inputs, one primary input and one secondary input; while the third corresponds to the output, if any. Having two inputs has the advantage that if some data are not available on the first one, the database will look for data on the second input before returning an error. This is specially useful to combine part of the data you have locally (in a ROOT file) with data from ORACLE, for example.

The version management is done with what is called "event files" (HEventFile objects). An event file identifies a set of events for which the reconstruction parameters remain unchanged; each event file holds a list of HParVersion objects, one per parameter container. On its side, a HParVersion object holds version numbers for a particular container in different sources. When the active data source reports the end of an event file, the runtime database is notified to update the containers in it. Then the runtime database uses the HEventFile object to see, for each container, if the container's version id has changed, in this case the container is updated and the "changed" flag is set with **HParSet::setChanged**(kTRUE).

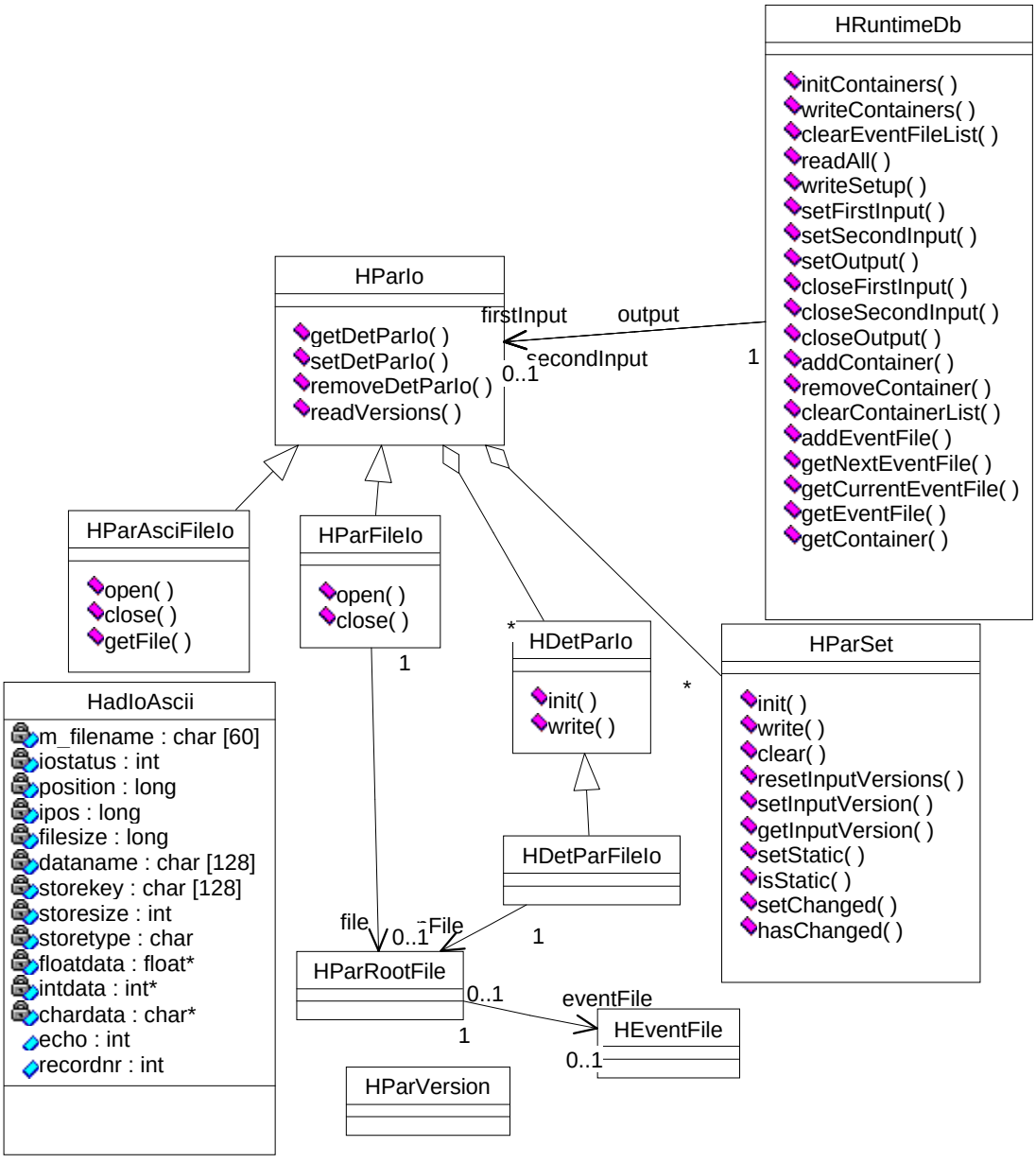


Figura 4.6: Runtime Database structure

Another interesting possibility of the HParSet objects is that they can be made static (**HRuntimeDb::setStatic**()); this means the container is not updated when the runtime database receives an update signal. So the user can initialize the container with some "hand-made" parameters at the very beginning of the initialization and these parameters won't we overwritten by the versioning mechanism.

4.7. Classes to perform tasks

One of the requirements we've seen in the previous chapter was to have a flexible system allowing us to select which algorithms to use for event reconstruction, as well as in which way those algorithms are going to be combined.

The adopted solution to get this objective is the following. We define an abstract class HTask representing a generic task. Each task has a name as well as several possible exits; so we can connect another task to any of those exits using the function **HTask::connectTask**(HTask *task,Int_t n), where "task" is the task to connect and "n" is the exit to which it is connected.

To perform a task one calls the function **HTask *next**(Int_t &errCode). This function performs the task and returns back the next task to be executed; that is the task connected to the resulting exit. If any problem was found, an error code must be written to errCode. Note that it is the task itself who decides which task is going to be executed the next, making possible to control the execution flow of the program. In particular one can define a task to have two possible exits, so when the **next**() method is called it just checks some condition and selects one of the two exits depending on the outcome. One concrete example where such a functionality is useful would be to run a specific analysis code for some special events, a task could look at the event header and depending on a flag in that header select an analysis chain or other.

Other important functions of the HTask class are **Bool_t init**() and **Bool_t finalize**() which should be called before the first execution of the task and after the last one respectively.

The **init**() function, as its name suggests, is used during initialization. Essentially what this function has to do can be summarized in the following points:

- Try to get pointers to parameter containers in the runtime database using **HParSet *HRuntimeDb::getContainer**(Text_t name[]). There are two possibilities:

1. The returned value isn't NULL. Then that pointer is used.
2. The returned value is NULL. In this case it is responsibility of the task to instantiate the "container" and add it to the runtime database.

- Try to get pointers to the needed HCategorys. Typically this is done using **HCategory *HEvent::getCategory**(Cat_t cat). If it returns NULL, it is the task's responsibility to instantiate the category and add it to the event structure. To instantiate the category, it is recommended to use the **HDetector::buildCategory**() function[10] instead of instantiating it directly with the "new" operator.

- Do the specific initialization for the concrete task. For example, calculate local parameters starting from those in the database [11]

There are two kind of standard tasks: reconstructors, which represent particular algorithms or procedures to transform the data, and task sets (see figure 4.7). Task sets are important because they allow to group several tasks into one. In fact, what the Hades class executes for each event is a task set. This task set is built during initialization, at execution time (see section 4.8).

4.7.1. The reconstructors

Reconstructors are a particular kind of tasks; for which a HReconstructor class is defined. The reconstructors are represented by objects instantiating the HReconstructor class; this is an abstract class which defines the common interface for every algorithm. Examples of reconstructors are calibration of raw data in MDC; a particular algorithm for segment finding in MDC or a calibration function for the TOF. Every reconstructor has a function **Int_t execute**(), available for the user to call it, where the real reconstruction process takes place. So that the HReconstructor class overloads the function **next**() so it calls **execute**(). If the value returned by **execute**() is less than 0; then it is interpreted as an error code. If the value is greater than 0 (or equal) it is associated to one of the possible exits in the reconstructor so the task connected to that exit is returned by the **next**() function.

[10] See section 4.8.3 for more information

[11] Note that this is not possible without an initialized parameter container in the database; and we cannot initialize containers within this init function. This only happens at the very beginning of the analysis, in this case what happens is that the initialization procedure is such that the init function is called once, so the containers are added to the runtime database; then the runtime database initializes those containers and the init function is called again

Figura 4.7: Tasks structure

4.7.2. Task sets

These are another fundamental kind of task, and they are defined by the class HTaskSet. It represents a set of tasks arbitrarily connected among them. To add tasks to a task set one of the following functions must be used:

Bool_t connect(HTask t)
It is used to connect the first task (the head task) to the task set

Bool_t connect(HTask t,HTask w,Int_t n=0)
Connects task "t" to the exit number "n" of task "w"

Bool_t connect(HTask t,Text_t where,Int_t n=0))
Connects task "t" to the n-th exit of the task named "where"

Bool_t connect(Text_t task[],Text_t where[],Int_t n=0))
Connects task named "task" in the task set to the n-th exit in task named "where"

The **connect**() methods which take a task's name as an argument, are provided for convenience. So the user doesn't need to keep pointers to those tasks in order to connect them to other tasks.

The tasks connected using these methods belong to the task set where they live, so they are destroyed at the same time as the task set. You should not connect tasks in a HTaskSet directly using **HTask::connectTask**() unless you really know what you are doing.

When calling function **next**() in a HTaskSet, its tasks are executed starting from the first one and following the order dictated by the **next**() function in each executed task until we get a NULL. At this moment we stop the execution of the internal tasks in the task set and the task set's **next**() function returns a pointer to the next task connected to the task set (or NULL if none exists).

Note also that a task set is a task, so we can always put a HTaskSet within another; building a recursive structure.

4.8. Initialization

In the initialization of the program the user stablishes his requests about the different customizable issues in the system. These include:

- What detectors are going to be used. That is, the spectrometer configuration
- What inputs (up to a maximum of two) and output are going to be used for the runtime database. That is, where reconstruction parameters will be read from and where will they be stored (if the user wants to store them)
- What versions of the reconstruction parameters are going to be used for data analysis. For example, in the calibration, we have to select which calibration parameters are we going to use to calibrate a file's data
- What tasks will be performed for each event
- What data source will be used to read the event's data from
- What structure is going to be used to store event's data in memory. However; if the user doesn't explicitly set an event structure then a default one will be used. This default is determined by the selected tasks to be performed

The user not only can select among a set of precoded options, but can add his own options. This is possible thanks to the modular design organized in dynamically linked libraries; which can be loaded at any moment using functions provided by ROOT.

Initialization is done in a file with C++ code which is interpreted at execution time (From now on we'll call this file the configuration macro). This allows a direct interaction with every part of the analysis, since it is also written in C++. In fact, one of the possible ways of working is to use a C++ macro as the main program and call the different services provided by the analysis when needed.

The initialization procedure is strongly automated, so the user can chose to only customize a minimum set of issues (or choose a pre-made configuration macro). In this situation, default values are set for those aspects not explicitly customized by the user. These default values are determined depending on the tasks the user has chosen to perform, and they are considered optimal for that set of tasks.

However, if the user makes a selection it will be respected, overriding the default values. Let's see this with an example: in principle, if we tell the program that we want to calibrate the MDCs, we are not interested in the data structure used by the developers of such calibration, so we leave it uninitialized. However, at a given moment we may be interested in using another data structure[12], then we only have

[12]For example, at mass production we may want a linear structure because of its performance; but when doing detector studies we may want a very ramified structure for the data so to make every kind of correlations easier

to initialize the data structure we want to use and our selection will be respected. As a consequence of this freedom we must store the data structure along with the output data; or it would be difficult to know which structure was used to analyze a given set of data.

One should note here that using default values is a safe bet, but setting them manually is not. So, one user is expected to know what he is doing before overriding default values; taking the risk that the reconstruction software refuses to work.

In a typical initialization macro the main steps are:

1. Ask for the **shared libraries** to be used
2. Instantiate **inputs and output for the runtime database** and select them
3. Select **detectors** to use instantiating objects of classes representing those detectors and adding those objects to the HSpectrometer object in the Hades soliton
4. Select which versions of the **reconstruction parameters** are going to be used by the runtime database
5. Build the **list of tasks** to be performed for each event; to build this list we can use the detector classes
6. Select the **data source** instantiating an HDataSource object and setting it as the current data source with **Hades::setDataSource**()
7. Call function **Hades::init**() and check if the return value is kTRUE
8. Set if an **output file** is or not required and if we want an event tree

The numbering in this list is important and it indicates the ordering for the different initialization steps. Now we'll go into more detail for the different aspects of initialization. At the end we'll see some examples.

4.8.1. Spectrometer configuration

The HADES spectrometer is represented within the analysis by a HSpectrometer class, which holds a list of detectors (HDetector objects, like HMdc, HTof, ...). On its side, the Hades soliton has a HSpectrometer available through the function **Hades::getSpectrometer**(). To access a particular detector in the spectrometer,

HSpectrometer::getDetector(Text_t *name) is used; where "name" is the detector's name.

In this way, to select which detectors we want to use for analysis we only have to choose the corresponding detectors and add them to the HSpectrometer object in Hades, using the function **void HSpectrometer::addDetector**(HDetector *det).

On their side, the HDetector objects store configuration information for each particular detector: number of sectors, active modules in each sector, etc. This configuration parameters can be set by calling the appropriate functions for each detector and will be used thoroughly by other parts of the software.

One of the places where this configuration information is used is in functions **buildCategory**() and **buildTask**() of the HDetector class. These two functions set the default values for the data structure and task structure for each particular detector. Therefore they are a very important part of the initialization mechanism and deserve further attention.

buildCategory(): The complete signature of this method is: **HCategory *buildCategory**(Cat_t cat). It is a virtual function, whose behavior depends on the particular detector we are working with. Given a category identifier "cat", this function instantiates a category of the appropriate type and with its configuration adapted to that of the detector. That is, a HCategory subclass is selected and an object of this class is instantiated according to the configuration parameters in the detector. If the "cat" identified is not recognized, then NULL is returned.

buildTask(): The complete signature of this method is: **HTask *buildTask**(Text_t task[], Text_t opt[]). This function builds a task identified by "task" with the options in "opt". Again, it is a virtual function which only gets a concrete meaning for each detector; where the valid values for both "task" and "opt" are defined.

This procedure frees the user of knowing a task's internal structure (that is, its subtasks and how are them connected). Simplifying initialization.

4.8.2. Data base initialization

During the database initialization what the user sets is:

inputs: At least one input must be set, but the user can set up to a maximum of two. To set one input one only needs to create the Io object (see 4.6.1) and

use whether **setFirstInput**() or **setSecondInput**(), from the HruntimeDb class, depending on what the user wants.

output: The procedure is the same as before: create a HParIo object and call **HRuntimeDb::setOutput**() giving a pointer to the object as a parameter.

event files: The next step is to set which event files are going to be used. This is typically accomplished by calling **HRuntimeDb::addEventFile**() and giving the file's name as an argument.

set current event file: This is done calling the method **setCurrentEventFile** () from HRuntimeDb and giving the event file number as a parameter. -1 to start from the beginning.

Of course this functions are called for the HRuntimeDb object in the Hades soliton. This you can get with **Hades::getRuntimeDb**()

4.8.3. Tasks selection

Selecting tasks means instantiating objects for the task we want to be performed for each event, and adding those objects to the HTaskSet within the Hades soliton. For that we need a pointer to the HTaskSet which can be got with **Hades::getTask** (). Once we've that pointer, we only need to use the **connect**() functions seen in section 4.7 to connect the different tasks we want to execute.

To instantiate the task objects the instantiated detector's **buildTask**() functions can be used. Or we can create those objects directly by calling the "new" operator. Choosing one ore another option will depend on the situation. The first of these methods is an easy-to-use way of selectinf a premade task set which is built by the corresponding HDetetor class; on the other hand, when there is not a premade task set fulfilling our needs, we should exhaustively define our own task set by using with the "new" operator.

4.8.4. Selecting the data source

Selecting a data source is as trivial as instantiating an object of the chosen data source and activate it as the current data source using the **setDataSource** (HDataSource *dataS) method from the Hades class.

Obviously each data source can get some different initialization parameters. As, for example, the server's IP direction if reading data from Internet, a file name,

or nothing. Since our configuration file is a C++ macro, it is enough to call the functions specified in each data source's documentation to set these parameters.

4.8.5. Event structure

As has already been said; it is not needed to explicitly define an event structure in the configuration macro, it can be created automatically. In any case it can be sometimes useful to do so.

In this situation it is enough to create an event object (an HRecEvent typically) where we can manually add all or part of the needed categories. Then we can set this object as the current event by calling **void Hades::setCurrentEvent**(HEvent *ev)

4.8.6. Examples

Here we'll see a couple of configuration macros which are thoroughly commented. The first of them sets the full analysis chain for the TOF, and it's quite simple:

```
{
//Configuration macro for Tof unpacking

//In this simple example we don't need connections to
//a parameter database so it's left uninitialized

//Create detectors and their setup
  HTofDetector *tof=new HTofDetector;
  Int_t mods[22]={1,2,3,4,5,6,7,8}; //8 modules are active
  tof->setModules(0,mods);  //but only in the first sector
  gHades->getSetup()->addDetector(tof); //add the detector
                                        //to the spectrometer

//Set the event file in the runtime database.
  gHades->getRuntimeDb()->addEventFile("t002.hld");


//Set the split level
  gHades->setSplitLevel(2);
```

```
//Instantiate the data source (here from Lmd file).
//Note you shouldn't declare variables static here.
//Use pointers instead.
  HldFileSource *source=new HldFileSource;

//Add the Tof unpacker to the data source
  source.addUnpacker(new HTofUnpacker);

//Set source as the active data source
  gHades->setDataSource(source);

  gHades->getRuntimeDb()->setCurrentEventFile(-1);

//Call the initialization and chech for an error
  if (!gHades->init()) printf("Error during initialization\n");

//Set output file ``test.root''. The parameter RECREATE means
//if a file test.root exists it is replaced.
//The last argument is the compresion level we want for the
//file.
  gHades->setOutputFile("test.root","RECREATE","Test",2);

//Build output tree
  gHades->makeTree();
}
```

The second example is a little more complicated and shows a configuration macro with both the Tof and Shower detectors, together with some tasks which are exhaustively defined, rather than use the **buildTask**() methods.

```
{
//Configuration macro for Tof and Shower

//Set the input for reconstruction parameters. In this case
//we supose we have our parameters in a local file named
//params.root
 HFileIo *io=new HFileIo("params.root");
 gHades->getRuntimeDb(&io);
```

```
//Create detectors and their setup
  HTofDetector *tof=new HTofDetector;
  HShowerDetector *shower=new HShowerDetector;
  Int_t mods[22]={1,2,3,4,5,6,7,8}; //8 modules are active
  Int_t smods[3]={1,2,0};           //only 2 shower modules active
  tof->setModules(0,mods);          //but only in the first sector
  shower->setModules(0,smods);      //but only in the first sector
  gHades->getSetup()->addDetector(tof);
  gHades->getSetup()->addDetector(shower);

//Set the event file in the runtime database.
  gHades->getRuntimeDb()->addEventFile("t002.hld");


//Set the split level
  gHades->setSplitLevel(2);

//Instantiate the data source (here from Lmd file).
//Note you shouldn't declare variables static here.
//Use pointers instead.
  HldFileSource *source=new HldFileSource;

//Add the Tof unpacker to the data source
  source->addUnpacker(new HTofUnpacker);
  source->addUnpacker(new HShowerUnpacker);

//Set source as the active data source
  gHades->setDataSource(source);

//Build some tasks
 HTaskSet *showertasks=new HTaskSet("shower","shower");
 HTaskSet *toftasks=new HTaskSet("tof","tof");
 showertasks->connect(new HShowerCalibrater("shower.cal",
"shower.cal");
 showertasks->connect(new HShowerHitFinder("shower.hitf",
"shower.hitf"), "shower.hitf");
 showertasks->connect(NULL,"shower.cal")
 toftasks->connect(new HTofCalibrater("tof.cal","tof.cal"));
 toftasks->connect(new HTofHitF("tof.hitf","tof.hitf"),"tof.cal");
```

```
 toftasks->connect(NULL,"tof.hitf");
 gHades->getTask()->connect(toftasks); //First process the TOF
 gHades->getTask()->connect(showertasks); //Then the shower stuff
 gHades->getTask()->connect(new HMatcher("match","match"),
    "shower"); //Match tof and shower hits
 gHades->getTask()->connect(NULL,"match"); //Set terminator

  gHades->getRuntimeDb()->setCurrentEventFile(-1);
//Call the initialization and chech for an error
  if (!gHades->init()) printf("Error during initialization\n");

//Set output file ``test.root''. The parameter RECREATE means
//if a file test.root exists it is replaced.
//The last argument is the compresion level we want for the
//file.
  gHades->setOutputFile("test.root","RECREATE","Test",2);

//Build output tree
  gHades->makeTree();
}
```

4.8.7. Initialization internals

Here we'll explain in detail how the initialization procedure works. The procedure starts when **Hades::init**() is called; what happens then is:

- the event's address is set in the active data source
- the **init**() function is called for the data source. This function does some specific initialization; for example in the case of HldSource it calls the init function for each unpacker. The data source is the responsible of creating an event object if none exists.
- the init function is called for each task in the task set to be performed

Note that the tasks' **init**() function will also be called in the event loop each time it starts the processing of a new event file; this includes the first event file. So during **Hades::init**() new parameter containers are added to the runtime database, and categories are added to the event structure. In the event loop the runtime

database is initialized and therefore the parameter containers are read; then the **init**() function is again called for each event file allowing each task to calculate some local parameters if needed.

4.9. Event processing

In this section we will see in more detail how the loop on events is realized. This is, in fact, an explanation in pseudocode of the **Hades::eventLoop**(Int_t nEvents) implementation; reading the source code for that function is recommended. This function does the following:

1. Ensure there is a current event, that is, an event structure and a data source.
2. Clear the event structure.
3. While the number of processed events is less than "nEvents" and the data source doesn't return an error code or and end of data code.
   a) Initialize the task list.
   b) While the number of processed events is less than "nEvents" and there are data in the current data source's file.
      1) Read a new event from the data source
      2) Execute the task set for the current event
      3) Fill the output ROOT tree if one exists
      4) Clear the event structure
4. Check if the data source has returned an error code and notify it

4.10. Running the program

There are several ways to run this software. One of them is with the executable file "hydra". This runs the software in batch mode. The syntax is very simple: "hydra filename [numEvents]" where "filename" is the configuration macro used to initialize the analysis and "numEvents" is an optional parameter which specifies the maximum number of events to be processed. By default all available events will be processed.

Another way is to use the software as a extension to ROOT and so work in a ROOT interactive session or write some macros. In this case the user is responsible

of some more things, as creating the Hades soliton at the very beginning of the session and destroying it at the end.

There is, to the user disposal, a set of standard macros to make working with the analysis software easier. This macros automatically do some tedious work which otherwise would fall into the user's back; the following subsections document those macros:

4.10.1. Analysis macros

This macro defines a generic interface for data analysis. This includes reconstruction as well as post-analysis.

The macro is called "analfunc.C" and it is automatically loaded when starting ROOT if it is properly configured. Once the macro is loaded we have a new function in the ROOT interpreter, this function is called "analyze" and it is overloaded to have a variety of behaviors.

Bool_t analyze(Text_t file[],Int_t numEvents=0) If we call this function the reconstruction program is launched using the configuration macro in the file "file". The second parameter corresponds to the number of events to be processed, if this parameter is omitted all events will be processed.

Bool_t analyze(char *in,char *out,char *user,Int_t ne,Int_t fe) This function is useful for those analysis tasks over reconstructed data which are relatively complex. The parameters have the following meanings:

"in" is a ROOT file with totally or partially reconstructed data as generated by the analysis

"out" is the name for an output file, with the results of this analysis (histograms ...)

"user" is a file name where a User class is defined. This class must have an **void execute**(HEvent *event) function where the code which analyzes each event is placed.

"ne" is the number of events to process. If it is ommited all events are processed.

"fe" number of the first event to process. If it is omitted the first event in the input file will be used

What this function does is to open the input file and for each event call the function **execute**() of the User class as defined in the "user" file; giving the current event as a parameter. Within this function the user can place the code to do the desired task.

The advantage of this method is that the user only needs to worry about what he wants to do (the execute function) and not about things like the split level of the event tree or its detailed structure.

4.11. Documentation

The program's documentation as well as the program itself, installation instructions and examples can be found on the Internet at *http://fpddux.usc.es/~hades/reconstruction/structure/index.html* [13]. From this page you will also be able to access the reference documentation; part of which is reproduced in appendix D

[13] You can see also the entry page at appendix C

Capítulo 5

Primeras pruebas del programa de reconstrucción

Aunque anteriormente se habían realizado varias pruebas de rendimiento bruto del programa de reconstrucción y el código se había probado con datos simulados, la primera prueba en condiciones reales del programa se produjo a mediados de Diciembre de 1998 en el GSI, cuando se tomaron datos con haz para probar la parte del espectrómetro ya construído.

Durante la toma de datos de Diciembre de 1998 se utilizaron módulos de los detectores TOF y SHOWER así como un detector START. Se utilizó un haz de ^{209}Bi cuya energía era de $1AGeV$, sobre un blanco de plomo.

Como ilustración de la utilización y funcionamiento del programa de reconstrucción presentaremos en este capítulo el análisis preliminar de los datos tomados por el detector TOF.

5.1. La reconstrucción de sucesos en el detector de tiempo de vuelo TOF

Como ya se ha explicado en el capítulo 2 de esta memoria el detector de tiempo de vuelo TOF está dividido en 6 sectores; cada uno de ellos formado por 22 módulos con 8 plásticos centelleadores. En ambos extremos de cada plástico hay un fotomultiplicador. Conceptualmente el funcionamiento del TOF se basa en usar un TDC para medir en cada fotomultiplicador el tiempo transcurrido desde una señal de START hasta que hay señal en ese fotomultiplicador. La señal de START nos

indica el momento en que se produce la interacción; mientras que la del fotomultiplicador nos da el momento en que la luz, emitida en el plástico centelleador al ser atravesado por una partícula, es recogida en el fotomultiplicador. Por tanto, ese tiempo es la suma del tiempo de vuelo más el tiempo empleado por la luz en propagarse a través del plástico centelleador.

En la práctica los TDC no nos dan directamente un tiempo de vuelo, sino un número de canal, como corresponde a un dispositivo digital. El paso de número de canal a tiempo de vuelo es lo que se conoce como "calibración". En nuestro caso, los TDC con que trabajamos tienen 4096 canales y la calibración que se utiliza es de tipo lineal. Es decir, la relación entre tiempo transcurrido y número de canal es:

$$tiempo = a \times numcanal + b \tag{5.1}$$

Además, también tenemos una medida de la amplitud de la señal proporcionada por el fotomultiplicador.

Una vez que tenemos los datos calibrados podemos usar la diferencia de tiempos entre ambos fotomultiplicadores en cada plástico para así conocer el punto por el que ha pasado la partícula. En efecto, cuando una partícula atraviesa el plástico centelleador, se produce luz que se propaga hacia los dos fotomultiplicadores a una cierta velocidad v_g. Si t_1 y t_2 son los tiempos registrados por los dos fotomultiplicadores y x es la coordenada de impacto, tomando como x=0 el centro del plástico centelleador, entonces:

$$x = \frac{v_g \cdot (t_1 - t_2)}{2} \tag{5.2}$$

Por su parte el tiempo de vuelo real t será:

$$t = \frac{t_1 + t_2}{2} + cte \tag{5.3}$$

El tiempo de vuelo, una vez reconstruído el momento de la partícula, puede ser utilizado para separar los leptones de los hadrones, es decir, para identificar la partícula; por lo que podemos ignorar la constante en la ecuación anterior.

5.2. Las clases C++ específicas del detector TOF

El escenario correspondiente a la reconstrucción de los datos del detector TOF hasta el nivel descrito comprendería los siguiente pasos:

1. Lectura de los datos provinientes del sistema de adquisición.
2. Desempaquetamiento de los datos y volcado sobre una estructura de datos ordenada (datos "raw")
3. Calibración de los datos y volcado sobre una estructura de datos calibrados
4. Reconstrucción de la posición de impacto de la partícula en el centelleador y de su tiempo de vuelo, así como almacenamiento de estas magnitudes reconstruídas en una estructura de datos

Este escenario requiere la implementación de las siguientes clases (véase figura 5.1):

- Una clase de "desempaquetado" de datos que transforme el formato de datos del sistema de adquisición de datos DAQ a la estructura de suceso que se esté utilizando. Ésto se hace escribiendo una clase HTofUnpacker y derivándola de HldUnpack. Tan sólo es necesario implementar los métodos **init**() y **execute** () de ésta clase. **init**() es llamado por el programa durante la inicialización y **execute**() realiza el desempaquetamiento propiamente dicho.
- Clases para almacenar tanto los datos en formato número de canal, como los datos calibrados, es decir, los tiempos. En nuestro caso estas son las clases HTofRaw y HTofCal. Ambas son clases de datos, derivadas por tanto de HDataObject. La única tarea de estas clases es almacenar un número de canal o un tiempo de vuelo respectivamente. Así pues cada objeto HTofRaw almacena 4 números de canal que se corresponden a los dos tiempos medidos por los dos fotomultiplicadores de un plástico centelleador y las dos amplitudes de las correspondientes señales en HTofCal.
- Un contenedor de parámetros en la base de datos que nos dé los valores de las constantes de calibración "a" y "b" para cada TDC. Basta pues con crear una clase HTofCalPar que derive de HTaskSet. La estructura interna de esta clase es tal que tenemos un par (a,b) por cada plástico centelleador (celda). Es decir, se trata de una matriz tridimensional de pares de números.

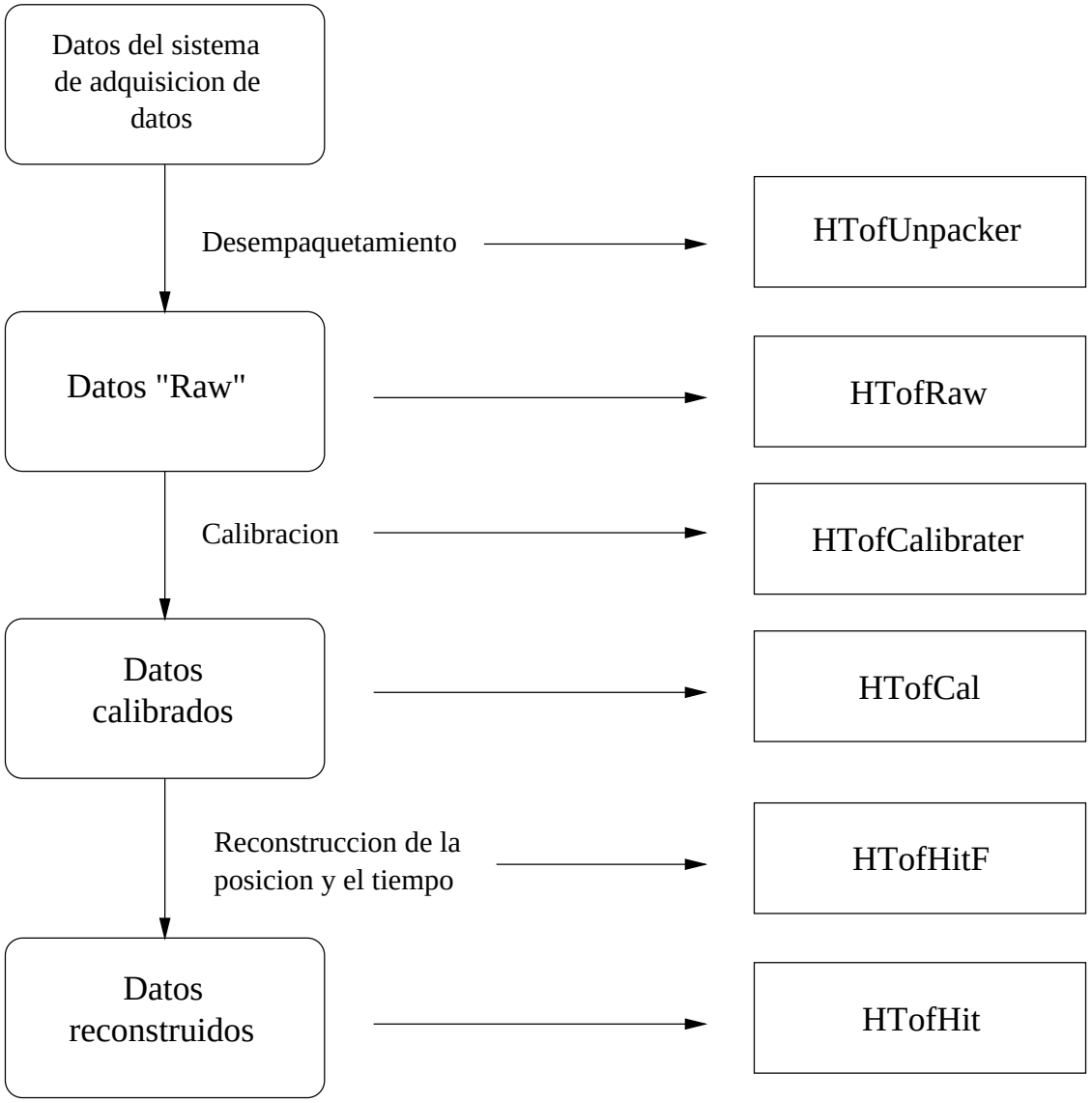


Figura 5.1: Esquema conceptual del proceso de reconstrucciónde los datos del detector TOF.

A la derecha aparecen las clases de datos correspondientes a cada nivel de datos así como las clases de transformación correspondientes a cada tarea realizada.

- Una clase de transformación que "calibre", es decir, que transforme los números de canal del TDC en tiempos. Esta clase es HTofCalibrater y debe derivar de HReconstructor. Escribir esta clase implica escribir dos funciones:

  - Una función **init**() que es usada durante la inicialización y sirve para que el reconstructor le diga al resto del programa qué elementos necesita para desempeñar su función.
  - Una función **execute**() que hace un loop en la categoría que almacena los HTofRaw. Para cada uno de ellos lee los parámetros de calibración del HTofCalPar tal y como son rellenados por la base de datos (run-time data base); y con esos datos hace la multiplicación, creando así un nuevo dato calibrado (HTofCal).

- Una clase de datos HTofHit que almacene los datos reconstruídos: el tiempo de vuelo y la posición.

- Una clase de transformación HTofHitF derivada de HReconstructor que pase de HTofCal a HTofHit y que será muy similar a HTofCalibrater.

- Una clase HTofDetector derivada HDetector; de modo que almacene la lógica necesaria para crear una lista de tareas con los reconstructores que hemos introducido, así como el tipo de categoría por defecto para cada nivel de reconstrucción.

Por último tan sólo nos queda derivar una clase HTofDetector de HDetector; de modo que almacene la lógica necesaria para crear una lista de tareas con los reconstructores que hemos introducido. Así como el tipo de categoría por defecto para cada nivel de reconstrucción.

5.3. Pruebas del programa con datos del TOF

El primer paso de la reconstrucción es el desempaquetamiento de los sucesos. En la tabla siguiente se muestran tiempos y tamaños de fichero de salida referentes al desempaquetado de 1.222.646 sucesos; donde sólo se lee la información proviniente del Sistema de Adquisición de Datos para 8 módulos, con ocho módulos centelleadores cada uno, de un sector del TOF. Se muestran los resultados utilizando una splitlevel 0,2 y sin salida:

	tiempo	*tamaño*
HMatrixCategory (sl=2)	$832 \pm 10s$	63.5 Mb
HMatrixCategory (sl=0)	$978 \pm 12s$	72 Mb
sin salida	$118 \pm 1s$	

Vemos entonces que el caso más eficiente es el primero. Otra dato importante surge de comparar el caso número 1 con el tercero; la única diferencia entre ambos es que en el tercero no se escribe la salida a un fichero. De donde podemos inferir que la escritura a fichero representa en torno al 80 % del tiempo consumido por el desempaquetamiento. En principio no debe ser necesario escribir un fichero durante la monitorización online, de hecho, la anterior tabla de resultados nos demuestra que dicha monitorización debe ser diseñada para funcionar sin escritura a fichero.

Para generar estos resultados se ha escrito una macro de configuración base sobre la que se hacen pequeños cambios resultando en las distintas medidas de tiempos. La macro es la siguiente:

```
{
//Config Macro for Tof and Start

//Create detectors and their setup
  HTofDetector *tof=new HTofDetector;
  Int_t mods[22]={1,2,3,4,5,6,7,8}; //4 modules active
  tof->setModules(0,mods);   //but only in the first sector
  gHades->getSetup()->addDetector(tof);

//Set the runtime database;
  gHades->getRuntimeDb()->addEventFile("/mnt/cdrom/t002.hld");


//Set the split level
  Int_t splitLevel=2;      //split level of the output tree
  gHades->setSplitLevel(splitLevel);

//Set the data source (here from Lmd file). Don't declare
//variables static. Use pointers instead.
//  HldRemoteSource source("acheron"); is an error
  HldFileSource *source=new HldFileSource;

//Define the unpackers
```

```
  source.addUnpacker(new HTofUnpacker);
  source.addUnpacker(new HStartUnpacker);

//Set source as the active data source
  gHades->setDataSource(source);

  gHades->getRuntimeDb()->setCurrentEventFile(-1);
  if (!gHades->init()) printf("Error during initialization\n");

//Set output file
  Int_t compLevel=0;       //compression level of the output file
  gHades->setOutputFile("test.root","RECREATE","Test",compLevel);

//Build output tree
  gHades->makeTree();
}
```

Como resultado del desempaquetamiento obtenemos las señales directamente del detector. Las figuras 5.2 y 5.3 muestran como ejemplo las distribuciones de señales para los fotomultiplicadores izquierdo y derecho de cada uno de los 8 plásticos centelleadores del módulo 5. La vigilancia de este tipo de histogramas nos asegura, durante y después de la toma de datos, que el detector y la electrónica asociada funcionan correctamente.

Consideremos ahora la reconstrucción completa para el Tof. Para ello usamos la siguiente macro; que esencialmente coincide con la anterior:

```
1: {
2: gHades->getRuntimeDb()->addEventFile("/mnt/cdrom/t011.hld");
3: HTofDetector *tof=new HTofDetector;
4: Int_t mods[22]={1,2,3,4,5,6,7,8};
5: tof->setModules(0,mods);
6: gHades->getSetup()->addDetector(tof);
7:
8: gHades->getTask()->connect(tof.buildTask("hitF","raw"),NULL);
9: gHades->getTask()->connect(NULL,"tof.hitF");
10:
11: gHades->setSplitLevel(2);
12:
13: HldFileSource *datos=new HldFileSource;
```

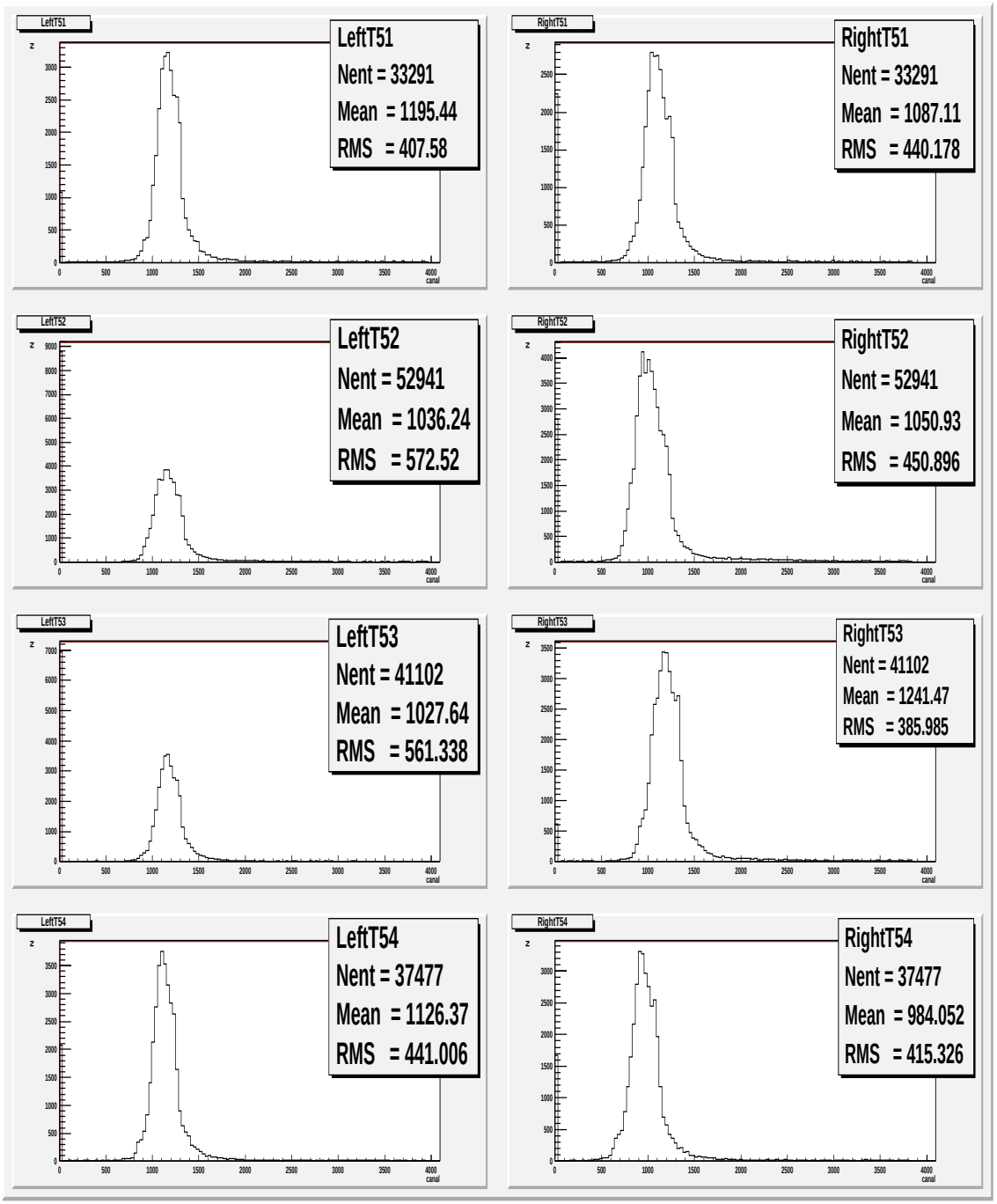


Figura 5.2: Datos raw para los plásticos 1 a 4 del primer módulo del TOF

En la figura se representan los datos raw (números de canal) para los fotomultiplicadores izquierdo y derecho correspondientes a los plásticos de 1 a 4 del módulo 5. Los histogramas aparecen ordenados de arriba a abajo, correspondiendo los de la izquierda al fotomultiplicador izquierdo y los de la derecha al derecho.

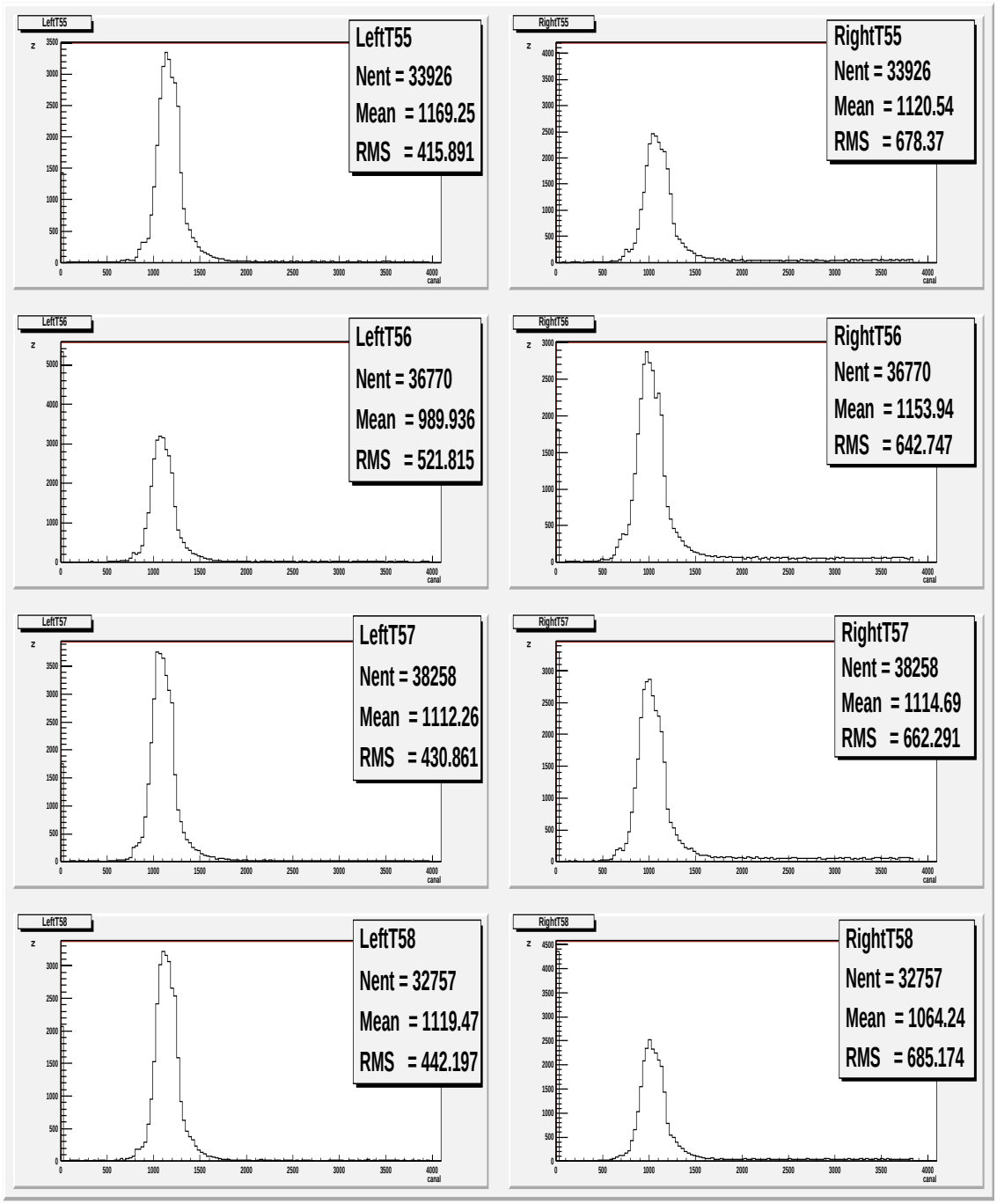


Figura 5.3: Datos raw para los plásticos 5 a 8 del primer módulo del TOF

En la figura se representan los datos raw (números de canal) para los fotomultiplicadores izquierdo y derecho correspondientes a los plásticos de 5 a 8 del módulo 5. Los histogramas aparecen ordenados de arriba a abajo, correspondiendo los de la izquierda al fotomultiplicador izquierdo y los de la derecha al derecho.

```
14: datos->addUnpacker(new HTofUnpacker);
15: gHades->setDataSource(datos);
16:
17: gHades->getRuntimeDb()->setCurrentEventFile(-1);
18: if (!gHades->init()) printf("Error\n");
20:
21: gHades->getCurrentEvent()->getCategory(catTofRaw)
        ->setPersistency(kFALSE);
22: gHades->getCurrentEvent()->getCategory(catTofCal)
        ->setPersistency(kFALSE);
23:
24: gHades->setOutputFile("test.root","RECREATE","Test",1);
25:
26: gHades->makeTree();
27: }
```

Tal como se presenta esta macro, se lanza el análisis para el TOF usando como entrada el fichero "t011.hld", que tiene 202.688 sucesos. Además, se usa la HMatrixCategory para los datos de las distintas fases del análisis (dado que esa es la categoría por defecto) aunque sólo se escribe a fichero la categoría con los "hits" finales. El "splitlevel" se selecciona en la línea 11 y es 2.

A continuación se presentan medidas de tiempos y tamaños de fichero realizadas con variaciones de esta macro; en particular tenemos los siguientes casos:

1. almacenar todas las categorías con splitlevel=2. Para ello basta con comentar las líneas 21 y 22 de la anterior macro.

2. el caso 1 con splitlevel=0. Para ir a este caso se cambia el splitlevel en la línea 11.

3. sin fichero de salida. Basta comentar las líneas 24 y 26 de la macro obtenida de aplicar los cambios para los casos anteriores.

4. sólo hits con splitlevel 2. Es el caso ilustrado en la macro anterior tal y como se presenta.

Los resultados obtenidos para estas pruebas son los que se muestran en la siguiente tabla:

caso	*tiempo*	*tamaño*
1	$272 \pm 6s$	61 Mb
2	$308 \pm 4s$	37 Mb
3	$45 \pm 2s$	
4	$131 \pm 4s$	14 Mb

Por comparación de los casos 1 y 2 con el 3 vemos de nuevo como la mayor parte del tiempo se emplea en la escritura a fichero de salida de los datos procesados. Precisamente ésto explica que si en lo único en que estamos interesados es en los datos finales, el escenario más favorable es el 4. Especial atención merece la comparación de los casos 1 y 2; en el primero de ellos la ramificación del árbol de salida consigue que en cada rama aparezcan datos homogéneos, de modo que el algoritmo de compresión de ROOT alcanza un rendimiento muy superior al del caso 2 donde la única rama del árbol almacena datos heterogéneos. Sin embargo, no toda la diferencia viene de ROOT sino que el el caso 1 parte de la información de la estructura del suceso se almacena de forma externa al propio objeto suceso, de modo que conseguimos menos redundancia en la información almacenada.

En la figura 5.4 se pueden ver los datos "raw" y calibrados para un plástico centelleador concreto. Este histograma debe tomarse simplemente como un test de funcionamiento del programa. De hecho el TOF aún no está calibrado; es decir, no se han calculado los parámetros de calibración del mismo. En el histograma mostrado se ha usado un conjunto de parámetros preliminar. Si avanzamos un paso más, haciendo la reconstrucción de tiempos de vuelo y posiciones obtenemos histogramas del tipo del mostrado en la figura 5.5 para un centelleador.

Para hacer estos histogramas, una vez tenemos el fichero de salida producido por el análisis, se ha hecho uso de las herramientas que a nuestra disposición pone ROOT. Una de las más potentes de estas herramientas es el "Explorador" (figura 5.6), a través del cual podemos acceder a toda una serie de elementos de ROOT como son los ficheros abiertos. Un fichero aparece en el explorador como una carpeta, de modo que si abrimos esta carpeta nos encontramos todos los objetos contenidos en el fichero; entre ellos el árbol de sucesos "T" producido por el análisis así como una copia del solitón gHades. Algunos de estos objetos aparecen como carpetas que a su vez contienen otros objetos, así por ejemplo podemos abrir la carpeta Hades y ver con qué fuente de datos, algoritmos etc. se ha realizado la reconstrucción de los sucesos en el fichero. Si abrimos la carpeta del árbol "T" tenemos una subcarpeta por cada rama del árbol ROOT que contiene alguna subrama, mientras que en la parte derecha del explorador aparecen aquellas ramas que no tienen subramas (ver figura 5.6), cada una de estas ramas corresponde a una de las variables almacenadas (tiempo de vuelo, posción, etc.) de modo que si hacemos doble click con el botón

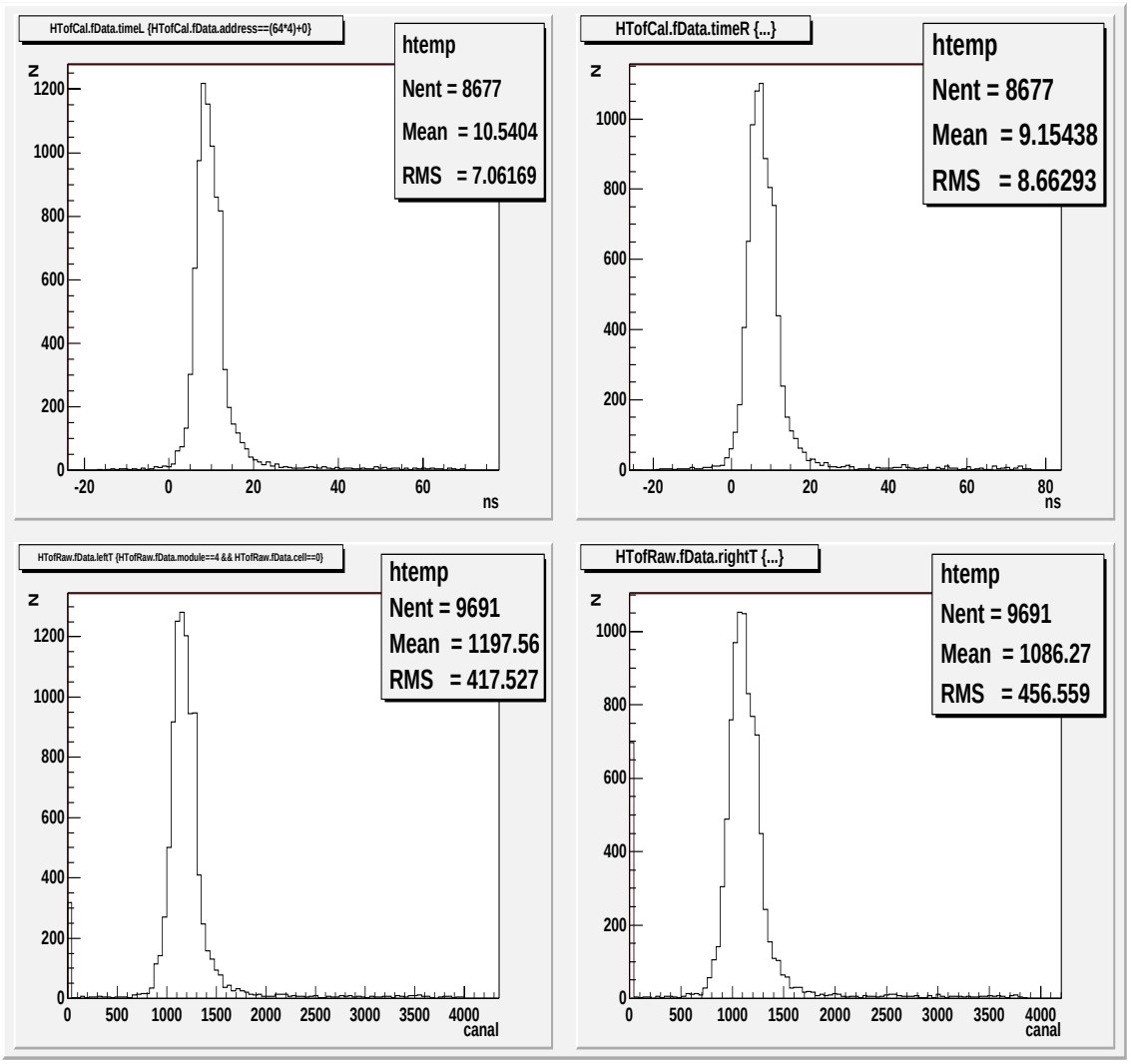


Figura 5.4: Datos raw y calibrados para el plástico 1 del módulo 5.

Los dos histogramas de la parte inferior muestran los datos raw recogidos por los fotomultiplicadores izquierdo y derecho respectivamente; mientras que los dos histogramas de la parte superior corresponden a datos calibrados.

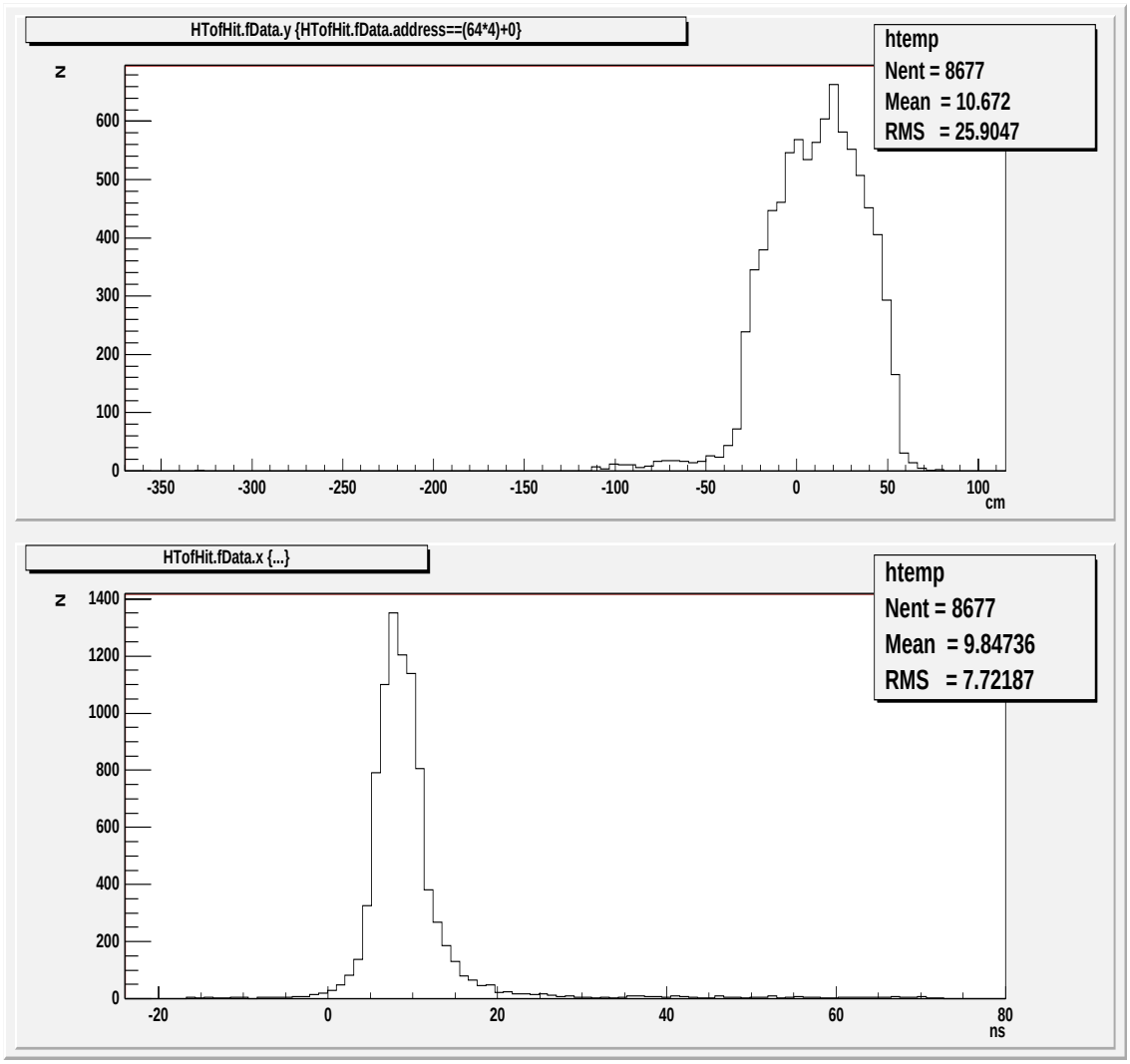


Figura 5.5: Posición y tiempo de vuelo en el plástico 1 del módulo 5.

El histograma de la parte superior muestra posiciones, mientras que el de la parte inferior corresponde a tiempos de vuelo.

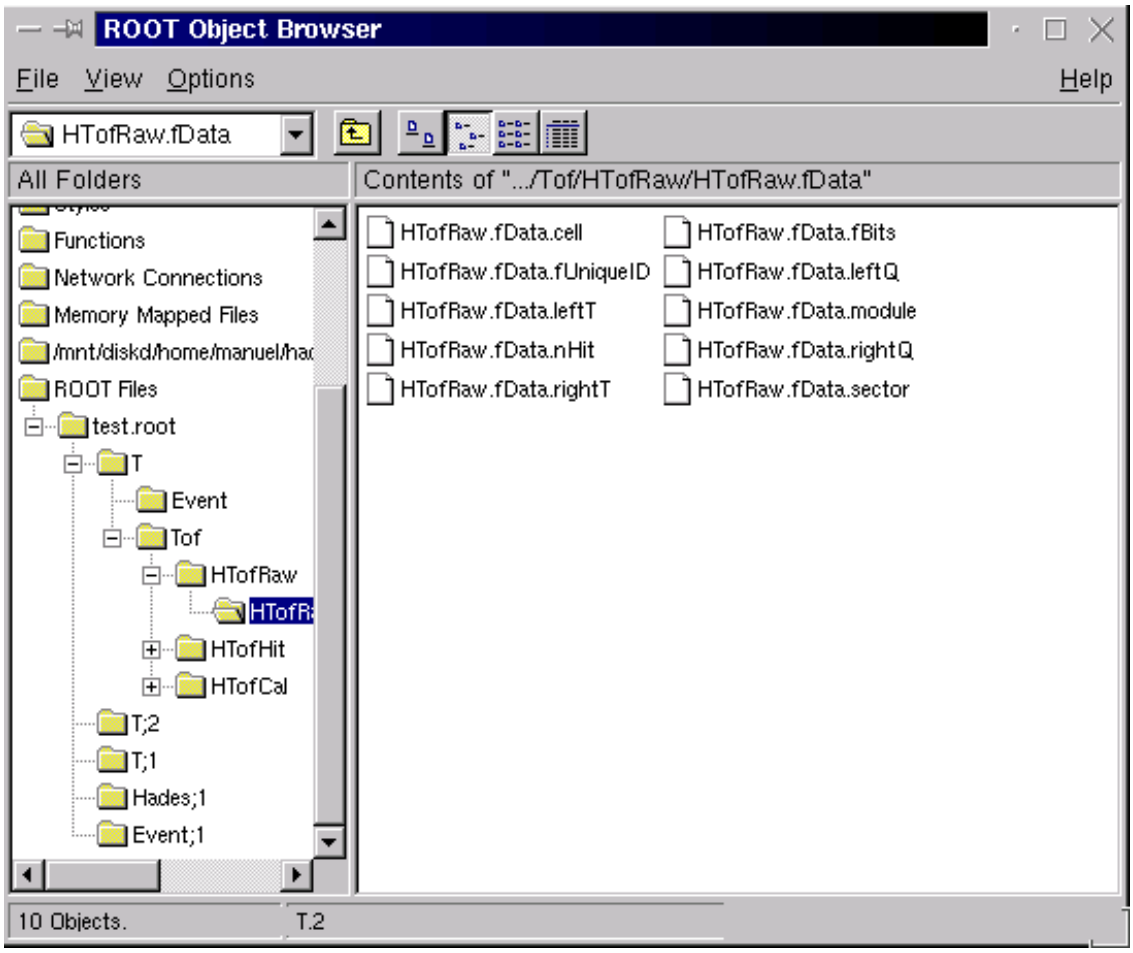


Figura 5.6: El explorador de ROOT

izquierdo del ratón sobre cualquiera de ellas obtenemos un histograma de la variable correspondiente.

La otra gran herramienta es el intérprete de C++ interactivo. A través de el podemos conseguir un mejor control de lo que queremos histogramar, en particular resulta muy sencillo hacer selecciones, cosa que no podemos hacer desde el explorador. Por ejemplo si, utilizando el browser, hacemos doble click en la rama "HTofCal.fData.leftT" obtendremos un histograma suma de los tiempos recogidos por todos los fotomultiplicadores izquierdos. Si lo que queremos es un histograma para el fotomultiplicador 1 del módulo 5, la forma de conseguirlo es usar en el intérprete un comando como:

```
T->Draw("HTofCal.fData.leftT","HTofCal.fData.address==(64*4)+0");
```

Capítulo 6

Conclusiones

El proyecto experimental HADES tiene por finalidad principal el estudio de la materia nuclear densa y caliente y las propiedades de los hadrones en la materia nuclear mediante experimentos de colisiones nucleares con haces de iones pesados, piones y protones. Concretamente, se investigarán las propiedades de los mesones vectoriales en la materia nuclear, midiendo el canal de desintegración de estos mesones en dileptones (pares electrón-positrón).

En el experimento HADES colaboran en la actualidad unos 140 físicos de 18 Institutos de Investigación y Universidades europeos. El espectrómetro HADES está siendo instalado en el acelerador SIS del laboratorio GSI (Gesellschaft für Schwerionenforschung) de Darmstadt (Alemania).

Las señales recogidas en los detectores de HADES en las colisiones núcleo-núcleo y grabadas en soporte magnético deben sufrir un complejo procesado hasta ser transformadas en las magnitudes físicas de interés para el análisis de las colisiones. Este complejo proceso lo realiza el llamado programa de reconstrucción de los sucesos. El experimento HADES es uno de los primeros experimentos de Física Nuclear y de Partículas cuya reconstrucción de sucesos se programa orientada a objeto y en lenguaje C++.

El objeto del trabajo que se presenta en esta memoria fue el análisis, diseño e implementación en lenguaje C++ de la estructura general del programa de reconstrucción de sucesos del experimento HADES.

- Siguiendo, en líneas generales, los pasos del Método de Boch, en primer lugar se discutieron y establecieron los requerimientos básicos del sistema a desarrollar. En esta etapa se decidió "qué es lo que debe hacer el programa".

- En la etapa de análisis del sistema se definieron las clases por grandes familias, provinientes de los conceptos más básicos involucrados en el problema: los datos tomados por los detectores, los procedimientos de entrada/salida, los parámetros de reconstrucción y los algoritmos y procedimientos de reconstrucción nos condujeron a la creación de clases para contener, organizar y gestionar los datos, clases para realizar la lectura y escritura de los datos, clases para contener, organizar y gestionar los parámetros de la reconstrucción, y clases básicas para gestionar los algoritmos y procedimientos de reconstrucción.

- Un diseño detallado de las clases la definición de sus relaciones y operaciones y sus atributos así como sus relaciones de herencia condujo a su implementación en el lenguaje C++.

- Un programa de estas características, pensado para ser utilizado y desarrollado para utilizaciones específicas, por un equipo de programadores separados en el espacio, necesita una buena documentación. Hemos hecho un gran esfuerzo en comentar detalladamente el código y en producir una documentación, tanto con ayuda del programa ROOT, que a partir de los comentarios incluídos en el código produce documentación en formato accesible desde Internet, como redactando una descripción aparte con ejemplos de uso y accesible igualmente desde Internet.

- Se ha probado el programa con datos reales tomados en Diciembre de 1998 por uno de los detectores de HADES, el detector de tiempo de vuelo TOF. La reconstrucción de los datos de un detector supone implementar clases específicas para ese detector en el marco del programa general. En el caso del TOF se han implementado las clases necesarias para contener cada unos de los niveles de reconstrucción del detector y para realizar las transformaciones que pasan los datos de un nivel al siguiente. El programa funciona correctamente, proporcionando la información requerida sobre la reconstrucción a través de histogramas, que el usuario puede producir a partir de los datos reconstruídos de una forma muy simple.

- Concluímos de las pruebas efectuadas que hemos implementado una estructura lo suficientemente modular y flexible para que pueda ser desarrollada por una equipo de programadores que trabajan en distintos laboratorios, y para que se puedan utilizar alternativamente y comparar entre sí algoritmos o procedimientos de reconstrucción distintos para los mismos datos, o estructuras de datos diferentes para diferentes necesidades. La flexibilidad de la estructura permitirá, a partir de ahora, construir y utilizar en su marco las soluciones

más eficientes en cuanto a tiempo de cálculo y espacio en memoria para la reconstrucción de los sucesos globalmente y en cada detector.

- Por lo que respecta a la utilización del programa herramienta ROOT del CERN, concluímos que es un valioso instrumento que resuelve a satisfacción una serie de necesidades de nuestro sistema, como son, principalmente, la lectura y escritura de estructuras complejas de objetos a fichero, la representación gráfica, la interactividad del usuario con el programa, e incluso su documentación.

Bibliografía

[1] The HADES proposal.

[2] *The ISO/ANSI C++ Draft.* Web page: http://www.cygnus.com/misc/wp.

[3] *OMT Object Model.* See http://wwwis.cs.utwente.nl:8080/dmrg/MEE/misop007/index.html.

[4] *PAW.* See http://wwwinfo.cern.ch/asd/paw/index.html.

[5] *The TOFW for the HADES spectrometer.*

[6] *CVS Manual*, 1993. See http://www.cyclic.com.

[7] *UML Notation guide*, 1997. See http://www.rational.com/uml/resources/documentation.

[8] *UML Semantics*, 1997. See http://www.rational.com/uml/resources/documentation.

[9] *UML Summary*, 1997. See http://www.rational.com/uml/resources/documentation.

[10] Grady Booch. *Object Oriented Modeling and Desing with Applications.* Addison-Wesley, 1993.

[11] Rene Brun and Fons Rademakers. ROOT- an object oriented data analysis framework. *Nuclear Instruments and Methods*, A(389), 1997. See also http://root.cern.ch.

[12] Jim Carroll. Measurements of e^+e^- pair production at the BEVALAC. *Nuclear Physics*, A(495):09c–422c, 1989.

[13] W. Cassing, V. Metag, U. Mosel, and K.Ñiita. Production of energetic particles in heavy-ion collisions. *Physics Reports*, 188(6):363–449, 1990.

[14] CERES Collaboration. First results from CERES/NA45 on low-mass electron pair production in Pb-Au collisions. *Nuclear Physics*, A(610):317c–330c, 1996.

[15] EPoS II Collaboration. Search for e^+e^- pairs with narrow sum-energy distributions in heavy-ion collisions. *Physics Letters*, B(389):4–12, 1994.

[16] J. Díaz, G. Martínez, and Y. Schutz. Gamma ray and particle production in heavy ion reactions. *World Scientific*, 1994.

[17] D. Dutta, A.K. Mohanty, R.K. Choudhury, and Phool Chand. Pattern recognition of particle tracks using principal component analysis and artificial netwrk. *Nuclear Instruments and Methods in Physics Research*, A(404):445–454, 1998.

[18] A. Balanda et al. Development of a fast pad readout system for the HADES shower detector. *Nuclear Instruments and Methods in Physics Research*, A(417):360–370, 1998.

[19] C.Ñaudet et al. Threshold behavior of electron pair production in p-Be collisions. *Physical Review Letters*, 62(23):2652–2655, 1989.

[20] Erich Gamma et al. *Design Patterns.* Addison-Wesley.

[21] G. Agakichiev et al. Enhanched production of low-mass electron pairs in 200 GeV/nucleon S-Au collisions at the CERN super proton synchrotron. *Physical Review Letters*, 75(7):1272–1275, 1995.

[22] G. Agakichiev et al. CERES results on low-mass electron pair production in Pb-Au collisions. *Nuclear Physics*, A(638):159c, 1997.

[23] G.Roche et al. First observation of dielectron production in proton-nucleus collisions below 10 GeV. *Physical Review Letters*, 61(9):1069–1072, 1988.

[24] Joachim Stroth et al. Dilepton spectroscopy with HADES at SIS. In *Hirschegg '95 Worshop on Dynamical properties of Hadrons in Nuclear matter*, 1995.

[25] M. Masera et al. Dimuon production below mass 3.1 gev/c^2 in p-w and s-w interactions at 200 gev/c/a. *Nuclear Physics*, A(590):93c, 1995. Colaboración DLS.

[26] R. Gernhäuser et al. Photon detector performance and radiator scintillation in the HADES RICH. *Nuclear Instruments and Methods in Physics Research*, A(371):300–304, 1996.

[27] Reiner Schicker et al. Acceptance and resolution simulation studies for the dielectron spectrometer HADES at GSI. *Nuclear Instruments and Methods in Physics Research*, A(380):586–596, 1996.

[28] René Brun et al. *GEANT 3.21*. CERN, CERN DD/EE/84-1 edition, 1987.

[29] R.J. Porter et al. Dielectron cross section measurements in nucleus-nucleus reaction at 1.0 AGeV. *Physical Review Letters*, 79(7):1229–1232, 1997.

[30] S. Beedoe et al. Measurements of dielectron production in niobium-niobium collisions at 1.05 gev/nucleon. *Physics Review*, C(47):2840, 1993.

[31] Jurgen Friese. HADES, the new electron pair spectrometer at GSI. Report for the HADES collaboration.

[32] Chilo Garabatos. The HADES dilepton spectrometer. Elsevier Preprint.

[33] Tetsuo Hatsuda and Su Houng Lee. QCD sum rules for vector mesons in the nuclear medium. *Physical Review C*, 46(1):34–46, 1992.

[34] C.M. Ko and G.Q. Li. Medium effects in high energy heavy-ion collisions. *Nuclear Particle Physics*, (22):1673–1725, 1996.

[35] Heike Nuemann. HADES - a high acceptance dielectron spectrometer proposed for relativistic heavy-ion collisions. *Acta physica slovaca*, 44(3):195–205, 1994.

[36] Reiner Shicker, Mario Axiotis, and H. Tsertos. A versatile dielectron trigger for nucleon-nucleon and nucleus-nucleus collisions. Preprint submitted to Elsevier Preprint.

[37] Fred Zlotnick. *The POSIX.1 standard*. Benjamin/Cummings, 1990.

Apéndice A

ROOT

ROOT es un conjunto de "frameworks" más un intérprete de C++ destinados origirariamente al análisis de datos en física de altas energías; así como al desarrollo en C++ de aplicaciones tanto de análisis como de reconstrucción o adquisición de datos.

En las secciones siguientes se introduce ROOT, de una forma genérica al principio, para luego estudiar ligeramente más en detalle el intérprete de C++ y el sistema de entrada-salida. Aspectos ambos, que son clave para entender el funcionamiento y diseño del software desarrollado en esta memoria.

En todo caso aquí sólo estamos haciendo una introducción, para conocer en profundidad este sistema, un buen sitio para empezar es la página web de ROOT que se encuentra en *http://root.cern.ch* donde se puede encontrar documentación, así como varias publicaciones y manuales al respecto.

A.1. ROOT a vista de pájaro

El esquema general de ROOT[11] está formado por una librería de clases de las llamadas "universales"; es decir, todas las clases en la librería derivan de una clase base TObject. Aún así, existen algunas clases, como la destinada a operar con cadenas de caracteres, que no derivan de TObject.

En total tenemos unas 250 clases organizadas en 20 frameworks divididos en 9 categorías; que pasamos a describir un poco más en detalle a continuación:

Las clases base constituyen los bloques fundamentales de ROOT; a esta categoría pertenecen clases como TObject, que define el comportamiento básico de todos los

objetos ROOT, o TClass que proporciona información de la estructura de una clase en tiempo de ejecución. Otras clases importantes son el gestor de memoria TStorage y TSystem, que es una capa de abstracción del sistema operativo.

Otro grupo importante de clases son los contenedores, que proporcionan estructuras de datos como árboles binarios, listas, arrays etc. para ser usados con objetos ROOT.

Se dispone además de toda una serie de clases destinada al análisis de datos y a la realización de histogramas. Como pueden ser las subclases de TH1 para histogramas o TMinuit para minimización de variables.

El sistema de árboles ROOT constituye el mecanismo fundamental de entrada salida de datos para el software de análisis. Este sistema se explica con mayor detalle en la sección A.3.1 de este apéndice.

Otros dos grupos de clases de gran importancia son el que nos define las primitivas gráficas 2D basado en la clase TGXW fundamentalmente; además del que nos define las primitivas 3D que se construyen a partir de primitivas 2D y tienen su uso fundamental en la representación de la geometría de detectores.

Se dispone además de toda una librería para la definición de interfaces gráficos basada en XClass'95. Esta libreria, aunque no es del todo completa, está integrada dentro del framework de ROOT. Por ahora tan solo está disponible para sistemas Unix, aunque hay una versión prevista para Windows.

Otro elemento fundamental de ROOT es un intérprete de C++ que nos permite interactuar directamente con las clases del sistema. De modo que el lenguaje de programación, el lenguaje de macros y el empleado en las sesiones interactivas son el mismo, y es C++.

Las clases de documentación (THtml) parmiten la generación de documentación en formato HTML a partir de la información contenida en las cabeceras C++ y comentarios introducidos en lugares predeterminados del código fuente.

A.2. El intérprete de C++

CINT es el intérprete de C++ utilizado por ROOT. Este intérprete convierte a C++ en un lenguaje interpretado; con las ventajas que ello supone para tareas como realización rápida de prototipos. Reduciendo el ciclo de programación-compilado-linkado-prueba a programación-prueba.

Dada la complejidad del C++, lenguaje que no fue diseñado para ser interpre-

tado, tan sólo el 95 % del C y el 85 % del C++ están soportados por CINT. En particular, las deficiencias más destacadas son el no poder derivar una clase interpretada de una compilada; la no existencia de soporte para templates, o el no poder usar excepciones. De todas formas es una herramienta de gran valor por la flexibilidad que aporta al sistema de configuración del software de reconstrucción.

A.3. El sistema de entrada-salida

Uno de los pilares del sistema ROOT es su base de datos jerárquica orientada a objetos. Es decir, podemos escribir cualquier objeto ROOT directamente a un fichero y después recuperarlo; posibilidad no soportada directamente por el lenguaje C++ y que resulta de gran importancia para nuestros propósitos. Los punteros a objetos contenidos en uno dado son almacenados adecuadamente; teniendo cuidado de resolver las referencias cíclicas. Además existen objetos especiales de la clase TDirectory que nos permiten estructurar el fichero de salida en directorios.

Todo objeto ROOT cuenta con una función **Streamer**(TBuffer &b) que se encarga de serializarlo; es decir de convertirlo en una secuencia de bytes. Ésta función puede ser generada automáticamente junto con otras por una herramienta llamada "rootcint", o puede ser escrita directamente por el programador. Ésta última posibilidad permite almacenar físicamente solo parte de los atributos del objeto, mientras que el resto son reconstruídos durante la lectura a partir de los almacenados.

Cuando queremos almacenar un objeto en un fichero lo que ocurre es que se llama a la función Streamer del objeto para que coloque su información en una memoria intermedia (objeto de la clase TBuffer). Luego, dependiendo del nivel de compresión seleccionado para el fichero de salida, este "buffer" es comprimido y la información comprimida se almacena en otro TBuffer que es escrito al fichero físico.

Para identificar la clase de cada objeto escrito en un fichero de salida; de modo que se pueda restaurar adecuadamente, hay un objeto TKey por cada objeto escrito en el fichero. Éste proporciona la información antedicha.

A.3.1. Los árboles ROOT

Los árboles ROOT son una propuesta para sustituir a las n-tuplas del programa PAW[4], tradicionalmente utilizadas en experimentos de Física Nuclear como contenedores de datos para el análisis; permitiendo trabajar con estructuras mucho más complejas que éstas. Por tanto los árboles ROOT son un medio para almacenar

estructuras de datos en un fichero de salida; en particular objetos.

Un "árbol" (TTree) está hecho de "ramas" (TBranch). Cada "rama" está descrita por sus "hojas" (TLeaf). Las "hojas" pueden ser variables simples, estructuras, arrays o objetos. Un array puede ser de longitud variable, siendo esta longitud una variable almacenada en la misma u otra rama del árbol. En general las ramas representaran objetos; teniendo una hoja para almacenar el objeto en cuestión (modo "no split") o bien teniendo una subrama por cada atributo del objeto (modo "split").

Una aplicación típica del modo split se da cuando utilizamos un árbol para almacenar un objeto, el cual a su vez contiene un array de objetos. En este caso se puede almacenar el array como un objeto en sí mismo creando una rama para él. Pero también se puede crear una rama para el array, con una subrama para cada atributo de los objetos en el array; de modo que cada una de esas ramas cuenta con una hoja almacenando un vector. Cada elemento i del vector es el valor del atributo para el objeto i del array de objetos.

Éste es el esquema más idóneo si, por ejemplo, tenemos un objeto suceso que contiene un array de trazas. Ya que entonces cuando queramos mirar a la componente x del momento de las trazas no será necesario cargar todo el objeto traza en memoria, sino tan sólo un array con la variable deseada para cada traza.

La estructura de datos TTree permite un acceso directo a cualquier suceso, cualquier rama y cualquier hoja incluso en el caso de estructuras de longitud variable.

Apéndice B

Notación UML

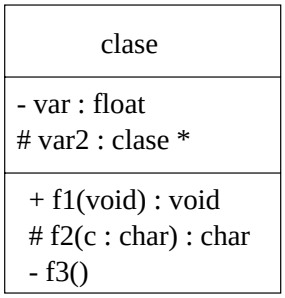
La notación UML [9, 7, 8] consiste en una representación gráfica para el modelado de proyectos software en base a un diseño orientado a objetos. En concreto las siglas "UML" corresponden a "Unified Modelling Language", haciendo referencia a que UML es un lenguaje de modelado y no solo una notación. La idea detrás de las notaciones como UML está en definir una representación gráfica de los componentes y relaciones en un proyecto software, y hacerlo de una forma estándar; es decir, el ideal perseguido es contar para el software con una herramienta como son los diagramas de circuitos en electrónica.

La notación tiene cabida para diversos tipos de diagramas en los que se muestran desde la estructura de clases estática de un programa hasta la distribución física de los módulos de código pasando por casos de uso del software diseñado o diagramas temporales donde se muestran interacciones entre objetos.

Como se puede imaginar la descripción completa de la notación es demasiado extensa para reproducirla aquí. Nos centraremos, pues, únicamente en aquellos aspectos de la documentación que son necesarios para entender los diagramas que acompañan a este trabajo. Tan sólo veremos los diagramas de clases, utilizados para describir la estructura estática, y aún de ellos no veremos todas los elementos que los pueden componer.

Un diagrama es esencialmente una estructura en 2-D formada por iconos, líneas (o flechas) y texto. A continuación veremos los elementos que conforman un diagrama de clases:

Clase Un rectángulo como



representa una clase de nombre "clase". Como se ve, el rectángulo que representa la clase está dividido en 3 áreas; en la primera de ellas aparece el nombre de la clase; en la segunda se colocan los miembros de la clase. Cada miembro viene representado por una cadena del tipo "nombre : tipo" indicando el nombre de la variable y su tipo; esta cadena de caracteres viene precedida de un símbolo que indica el tipo de acceso a la variable:

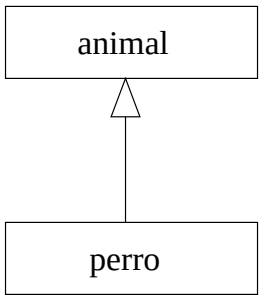
+: el miembro es público

#: el miembro es protegido (sólo las clases derivadas pueden acceder a él)

-: el miembro es privado (sólo es visible en la implementación de esta clase y sus "amigas")

En el tercer área otro conjunto de cadenas de caracteres nos muestra los métodos de la clase, que definen su contrato con el exterior (son como las patillas de salida de un circuito integrado). Las cadenas son del tipo *"nombre(argumentos) : tipo_ devuelto"* indicando el nombre de la función y el valor que devuelve; "argumentos" es un conjunto de cadenas como las utilizadas para indicar los miembros de datos separadas por comas, y nos define los argumentos que toma la función. Al igual que antes, éstas cadenas pueden ir precedidas de '+', '-' o '#' según el tipo de acceso. Cabe hacer notar que algunas utilidades, como Rose/C++, sustituyen éstos símbolos por dibujos. Así pues, el '+' viene representado por un rectángulo rosa inclinado hacia la izquierda.

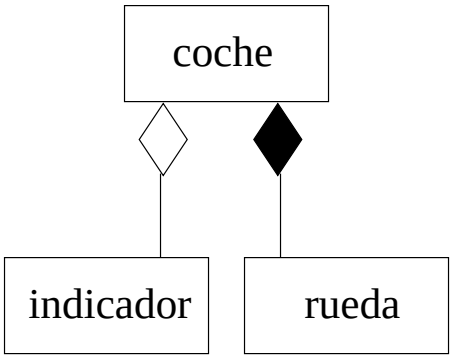
Herencia Se representa por una flecha terminada en triángulo hueco como en



Donde la clase "perro" deriva de "animal" heredando por tanto todos sus atributos y métodos.

Asociación Se representa por una línea sólida entre las dos clases relacionadas y expresa una relación entre ambas. En caso de dibujarse una flecha, ésta indica el sentido de la relación. Típicamente se usa para el caso de una clase conteniendo un puntero a otra; pero sin que la segunda "forme parte" de la primera conceptualmente. En caso de que la línea sea discontinua se expresa una relación de uso; es decir, una clase usa los servicios de la otra. En cualquiera de los extremos de la línea se puede indicar el nombre de la relación así como su multiplicidad (1, 0..1, 1..n, *).

Agregación Es un tipo de relación que consiste en que una clase "forma parte" de la otra. Para indicar la clase contenedora se dibuja un rombo en el extremo de la línea correspondiente a esa clase; este rombo puede estar hueco indicando agregación por referencia, o relleno en el caso de agregación por valor. Así por ejemplo, a continuación vemos un diagrama representando una clase "coche" que contiene por valor 4 "ruedas" y por referencia un número arbitrario de "indicadores"



La diferencia fundamental entre agregación por valor y por referencia es que en el primer caso el objeto contenido es destruído al mismo tiempo que el

continente; mientras que en el segundo caso ésto dependerá de la semántica de la clase.

Apéndice C

Página web del programa

HADES links | GENP PAGE | RECONSTRUCTION | SLOW CONTROL | HOME

Object-Oriented event reconstruction for the HADES experiment

This page describes and provides the code and documentation of the general structure of the object-oriented event reconstruction program written in the University of Santiago de Compostela for the HADES experiment at GSI.
The program is under development so changes in the contents of this page are expected from time to time.

- General description of the project (ps version).
- Index of classes documented using ROOT.
  - Version 0.01
  - Version 0.02
- Class diagrams in UML.
- How to install and run the program (ps version).
- Examples.
- Download the latest version of the code.

Please, send your comments and suggestions to Manuel Sánchez.

Private documents

HADES links | GENP PAGE | RECONSTRUCTION | SLOW CONTROL | HOME

Last modification: 26/6/1998, HAPOL
Comments and suggestions.
Héctor Alvarez Pol, Beatriz Fuentes Arenaz.

Apéndice D

Documentación de referencia

A continuación se incluye parte de la documentación de referencia del software de reconstrucción. La documentación completa se puede encontrar en la página de computing de Hades: http://hades.gsi.de/computing/anal/ClassIndex.html

D.1. Clases base

D.1.1. class HCategory

- Base classes: public TObject
- public methods
  - virtual void ~HCategory()
  - virtual void activateBranch(TTree* tree, Int_t splitLevel)
  - TClass* Class()
  - virtual void Clear()
  - virtual Bool_t filter(HFilter& aFilter)
  - virtual Bool_t filter(HLocation& aLoc, HFilter& aFilter)
  - Int_t getBranchingLevel()
  - Cat_t getCategory()
  - virtual TClass* getClass()
  - virtual const Text_t* getClassName()
  - TObject* getHeader()
  - virtual TObject*& getNewSlot(HLocation& aLoc)

- virtual HDataObject* getObject(HLocation& aLoc)
- virtual TObject*& getSlot(HLocation& aLoc)
- virtual TClass* IsA()
- virtual Bool_t IsFolder()
- Bool_t IsPersistent()
- virtual Bool_t isSelfSplitable()
- virtual void makeBranch(TBranch* parent)
- virtual TIterator* MakeIterator(Bool_t dir = kIterForward)
- TIterator* MakeReverseIterator()
- virtual Bool_t query(TCollection* aCol, HFilter& aFilter)
- virtual Bool_t query(TCollection* aCol, HLocation& aLoc)
- virtual Bool_t query(TCollection* aCol, HLocation& aLoc, HFilter& aFilter)
- void setBranchingLevel(Int_t nLevel)
- void setCategory(Cat_t aCat)
- void setPersistency(Bool_t per)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

■ Data Members

- protected:
  - Bool_t fPersistency: *Indicates whether this category is stored in output.*
  - Cat_t fCat: *Identifier for this category*
  - Int_t fBranchingLevel: *Number of levels for the data in the category*
  - TObject* fHeader: - *Categorys header.*

D.1.1.1. Class Description

```
HCategory (ABC)

  The HCategory class is an abstract base class. So the real work is
made by the derived classes. These derived classes correspond to different
strategies to store the data objects in memory and in file. The classes
derived from HCategory can also indicate the way they want to be stored in
a Root tree (the TBranch layout), for that purpose the makeBranch() function
can be overloaded.

  A HCategory encapsulates one category of data; that is one kind of data
(mdc raw data,rich rings...), and it is responsible of giving the user
access to the objects held by that category. The category is also able to
hold one header with common information for all the objects in the category.
```

```
  The class provides functions to access the objects in the category:

  Each data object in the category is stored in a particular location
(see HLocation), so to access an object you must give its location and use
the method: getObject(HLocation &aLoc). This method returns one only object,
if you want a collection with all the objects corresponding to a particular
location (i.e, all the raw data in the second chamber of the first sector...)
then use query(TCollection *col,HLocation &aLoc) with col being the
collection where to store the result. You can also iterate on all the
objects in the category or the objects corresponding to a particular location
using the HIterator created by MakeIterator() or MakeReverseIterator()

  The category is also responsible of allocating new objects, this can be
done through the functions getSlot(HLocation &aLoc) and
getNewSlot(HLocation &aLoc)  which return a place in memory where to
allocate the new object (a slot) corresponding to the location aLoc. The
strategy of letting the HCategory to manage the memory has the advantage
that it allows the category to have the memory preallocated.

  Example:
   {
    HLocation loc; //Allocates loc pointing to a location
    HMdcRaw *raw; //Declaration of a pointer to a HDataObject
    HCategory *cat; //Pointer to a generic category
    ...
    //loc is set to point to the location (2,2,1); this could be something
    //like: sector 2, mdc 2, plane 1.
    loc.set(3,2,2,1);
    //Ask for the slot at location "loc" and stores its address in raw
    //if there is not such a slot, the getSlot() method will return NULL
    raw=cat->getSlot(loc);
    //If we have a valid slot (raw!=NULL) then allocate a new object of
    //class HMdcRaw at the addres given by raw using the operator
    //"new with placement"
    if (raw!=NULL) raw=new(raw) HMdcRaw;
   }

  Each category can be persistent or not; so if it's persistent it
will be stored in the output file (if any) and if it's not, then it won't
be stored in the output file. To control the persistency of a category
the setPersistency() method is provided

  Here follows a description of the common methods every class inherited
from HCategory provides:

 TObject *&getNewSlot(HLocation &aLoc)
   Returns a memory slot for location aLoc where you can place a new object
```

```
   aLoc gives the indexes for the location of the new object except the last
   one. So the function returns the first empty slot in aLoc.

  TObject *&getSlot(HLocation &aLoc)
     The same as before, but now aLoc gives the complete location of the new
    object
  HDataObject *getObject(HLocation &aLoc)
    Returns the object at the location aLoc

  TClonesArray *getClones(HLocation &aLoc)
    Returns a clones array with the objects corresponding to the HLocation
    aLoc.

  Bool_t query(TCollection *aCol,HLocation &aLoc,HFilter &aFilter)
    Adds to aCol all objects in the category corresponding to the location
    aLoc and passing the filter aFilter. There are also functions without
    aFilter or without aLoc

  Bool_t filter(HLocation &aLoc,HFilter &aFilter)
  Bool_t filter(HFilter &aFilter)
    The same as before but now the objects not verifying the conditions are
   deleted off the category
```

D.1.1.2. ~HCategory(void)

```
Destructor.
```

D.1.1.3. void activateBranch(TTree *tree,Int_t splitLevel)

```
 If a category generates its own branches in a different way than that
provided with HTree::makeBranch (see isSelfSplitable()) then it must
override the makeBranch function as well as the activateBranch function.

 The activate branch is intended to activate those branches in the TTree
tree whose names correspond to the names of the branches that would be
created by makeBranch() in this category. To do such a thing the Root
methods: TTree::SetBranchAddress() and TTree::SetBranchStatus() need to
be called.

 As a default activateBranch() does nothing
```

D.1.1.4. void setBranchingLevel(Int_t nLevel)

```
Sets the branching level.
```

```
 The branching level is directly related with the number of indexes needed
to identify an object in the category (the number of indexes in the object's
location). For example, in the HMatrixCategory those two numbers are equal;
however in HSplitCategory the branching level is equal to the number of
indexes needed to unambiguosly identify an object in the category minus 1.
```

D.1.1.5. void setCategory(Cat_t aCat)

```
Sets the identifier for this particular category
```

D.1.1.6. Cat_t getCategory(void)

```
Returns the identifier of this particular category
```

D.1.1.7. Int_t getBranchingLevel(void)

```
Returns the branching level for this category.

 The branching level is directly related with the number of indexes needed
to identify an object in the category (the number of indexes in the object's
location). For example, in the HMatrixCategory those two numbers are equal;
however in HSplitCategory the branching level is equal to the number of
indexes needed to unambiguosly identify an object in the category minus 1.
```

D.1.1.8. Bool_t IsPersistent(void)

```
Returns kTRUE if the category is persistent, and kFALSE if it's not.
```

D.1.1.9. void setPersistency(Bool_t per)

```
Sets the persistency of the category

Input:
  per=kTRUE --> The category is persistent
  per=kFALSE --> The category is not persistent.
```

D.1.1.10. Bool_t IsFolder(void)

```
Returns kTRUE
```

D.1.1.11. Bool_t query(TCollection *aCol,HFilter &aFilter)

```
Stores in the collection aCol pointers to all the objects in the category
verifying the condition given by the filter aFilter.

Returns kTRUE if everything works Ok, kFALSE in other case
```

D.1.1.12. Bool_t query(TCollection *aCol,HLocation &aLoc)

```
Stores in the collection aCol pointers to all the objects in the category
corresponding to the location aLoc.

Returns kTRUE if everything works Ok, kFALSE in other case
```

D.1.1.13. Bool_t query(TCollection *aCol,HLocation &aLoc,HFilter &aFilter)

```
Stores in the collection aCol pointers to all the objects in the category
verifying the condition given by the filter aFilter and corresponding to
the location aLoc

Returns kTRUE if everything works Ok, kFALSE in other case.
```

- Autor: Manuel Sanchez

D.1.2. class HDataObject

- Base classes: public TObject
- public methods
  - HDataObject HDataObject()
  - HDataObject HDataObject(HDataObject&)
  - virtual void ~HDataObject()
  - TClass* Class()
  - virtual Cat_t getCategory()
  - HDataObjId* getId()
  - HLocation*& getLocation()
  - Bool_t HasIdentifier()
  - virtual TClass* IsA()
  - void setLocation(HLocation& aLoc)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)

- virtual void Streamer(TBuffer& b)

- Data Members
  - protected:
    - HLocation* fLocation: *! Location of this object in the event*

D.1.2.1. Class Description

```
HDataObject

  ABC for all the data objects
```

D.1.2.2. HDataObject(void)

D.1.2.3. ~HDataObject(void)

D.1.2.4. void setLocation(HLocation &aLoc)

```
Sets the location of this object to that given in aLoc

 For more information on the HLocation role in a event, see HEvent
```

D.1.2.5. HDataObjId* getId(void)

```
 Returns an HDataObjId which unambigously identifies this data object
within the event structure
```

D.1.2.6. Cat_t getCategory(void)

```
Returns the category this data object belongs to
```

D.1.2.7. Bool_t HasIdentifier(void)

```
 Returns kTRUE if the object knows about its own location in the event
structure, kFALSE in other case.
```

- Autor: Manuel Sanchez

D.1.3. class HDataSource

- Base classes: public TObject

- public methods
  - void ~HDataSource()
  - TClass* Class()
  - virtual EDsState getNextEvent()
  - virtual Bool_t init()
  - virtual TClass* IsA()
  - void setEventAddress(HEvent** ev)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - HEvent** fEventAddr: *! Address of the event to fill*

D.1.3.1. Class Description

```
HDataSource

 This is an abstract base class. Its derived classes provide the data needed
for the reconstruction, reading them from different sources.

 The main method of this class is getNextEvent() which is responsible of
reading data from the source (file...) and put them into the event pointed
by fEventAddr

 The return value is:
     kDsOk --------> no error.
     kDsEndFile ---> file's end.
     kDsEndData ---> data's end.
     kDsError -----> error.

 Another important method is init(). In this method the data source must
setup all what it needs to run (get pointers to the target categories, and
if the returned pointer is NULL, create those categories ...).


 Within the whole framework, the data sources (classes inherited from
HDataSource) are the responsible for reading the event's data before its
proccessing.
```

D.1.3.2. void setEventAddress(HEvent **ev)

```
Used to give the data source the address of the event it must fill in
Input:
 ev --> Address of a pointer to the event that will be filled in by the
        data source
```

- Autor: Manuel Sanchez

D.1.4. class HDetParIo

- Base classes: public TNamed
- public methods
  - HDetParIo HDetParIo()
  - HDetParIo HDetParIo(HDetParIo&)
  - virtual void ~HDetParIo()
  - TClass* Class()
  - Int_t getInputNumber()
  - virtual Bool_t init(HParSet*, Int_t*)
  - virtual TClass* IsA()
  - void setInputNumber(Int_t n)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
  - virtual Int_t write(HParSet*)
  - virtual Bool_t write(HDetector*)
- Data Members
  - protected:
    - Int_t inputNumber: *input number (first or second input in runtime database)*

D.1.4.1. Class Description

```
HDetParIo

  Abstract class for parameter input/output needed by a
  detector
```

D.1.5. class HDetector

- Base classes: public TNamed
- public methods
  - HDetector HDetector()
  - HDetector HDetector(Text_t* name, Text_t* title)
  - HDetector HDetector(HDetector&)
  - virtual void ~HDetector()
  - virtual void activateParIo(HParIo*)
  - virtual HCategory* buildCategory(Cat_t)
  - virtual HTask* buildTask(Text_t*, Option_t*)
  - TClass* Class()
  - Int_t getMaxModules()
  - Int_t getMaxSectors()
  - virtual Int_t getModule(Int_t sector, Int_t mod)
  - virtual Int_t* getModules()
  - virtual Bool_t init(Text_t*)
  - virtual TClass* IsA()
  - void print()
  - virtual void setModules(Int_t sec, Int_t* modules)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
  - virtual Bool_t write(HParIo*)
- Data Members
  - protected:
    - Int_t maxModules: *maximun number of modules per sector*
    - TArrayI* modules: *Module's array.*

D.1.5.1. Class Description

```
HDetector

  class to store the actual setup of a detector

  The setup is defined in the macro via the memberfuntion
  setModules(Int_t sector,Int_t* modules). The array must
  have the size of the maximum mumber of modules in a
  sector for this detector. If the module if not active the
```

```
  according number is 0.
  e.g. for Mdc:   Int_t mod[4]={1,2,0,0};
                  mdc.setModules(0,mod);
       This activates the modules 1 and 2 in sector 0

 It also holds the intelligence to build tasks related to this particular
 detector as well as to create data structures for it.

  This intelligence is stored in the functions: buildTask() and
buildCategory() and has the form of compiled C++ code.

  The function HTask *HDetector::buildTask(Text_t *task, Option_t *opt)
 builds a task object corresponding to the task identified by the string
 "task". This can be an atomic task (a HReconstructor) or a set of them, in
 this case the tasks will be packed in a HTaskSet so that buildTask() returns
 a pointer to that HTaskSet. "opt" is a string where extra options can be
 given; the available options differ from detector to detector.

   The function HCategory* HDetector::buildCategory(Cat_t cat) uses the
 setup information in the HDetector to build a category object. The category
 class and it's setup will be defined by "cat"; i.e. the HDetector has built
 in the information of which category subclass is recommended for each kind
 of data.
```

D.1.5.2. HDetector(void)

```
Constructor
```

D.1.5.3. HDetector(Text_t* name,Text_t* title) : TNamed(name,title)

```
Constructor with name
```

D.1.5.4. ~HDetector(void)

```
Destructor
```

D.1.5.5. void setModules(Int_t s,Int_t* m)

```
 stores the modules givven in 'm' as active modules in sector 's'
```

D.1.5.6. Int_t* getModules()

```
 returns a linear array of all modules
```

D.1.5.7. Int_t getModule(Int_t s,Int_t m)

```
returns the number of the module in a sector
returns 0 if this module is not active
```

D.1.5.8. void print()

```
prints the detector setup
```

- Autor: I. Koenig, D. Bertini, M. Sanchez

D.1.6. class HEvent

- Base classes: public TObject
- public methods
  - virtual void ~HEvent()
  - virtual void activateBranch(TTree* tree, Int_t splitLevel)
  - virtual Bool_t addCategory(Cat_t aCat, HCategory* cat, Option_t* opt)
  - TClass* Class()
  - virtual void Clear()
  - virtual void clearAll(Int_t level)
  - virtual HCategory* getCategory(Cat_t aCat)
  - virtual HEventHeader* getHeader()
  - virtual const Text_t* GetName()
  - virtual const Text_t* GetTitle()
  - virtual TClass* IsA()
  - virtual void makeBranch(TBranch* parent)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - TString* fName: -*Event's name*
    - TString* fTitle: -*Event's title*

D.1.6.1. Class Description

```
HEvent Begin_Html
<a href="Estructura_suceso.gif">(UML Diagram)</a> End_Html

  ABC for the different kind of events (HPartialEvent,HRecEvent,
HSimulatedEvent...)

  An HEvent holds the data corresponding to an event, those data are arranged
 in categories (mdc raw data, mdc cal data, rich rings,...) which are objects
 instantianting one HCategory, to access one of these categories the user
 can call HEvent::getCategory(). Within the categories, each data object has
 a location (given by an object instantiating HLocation) which identifies
 that object in the category (in some sense, the location is a generalized
 index).

  So to acces one data object, firstly one needs to get the category and then
 the particular data object.

  This class defines the most important functions any event must provide to
access the data contained into it:

  getCategory(Cat_t aCat):
    returns the category identified by aCat

  Clear(void)
    clears the event. (Before loading a new event)

  clearAll(Int_t level)
    Clears the data in the event and the event structure
    (list of subevents...) depending on the value of "level".

  makeBranch(TBranch *parent)
    generates Root TTree branches for the data contained in the event and
    hangs them from "parent"

  addCategory(Cat_t cat,HCategory *category,Option_t *opt)
    This function adds the category "category" identified by "cat" to the
    event's structure.
```

D.1.6.2. ~HEvent(void)

D.1.6.3. void Streamer(TBuffer &R__b)

```
 Stream an object of class HEvent.
```

- Autor: Manuel Sanchez

D.1.7. class HEventFile

- Base classes: public TNamed
- public methods
  - HEventFile HEventFile()
  - HEventFile HEventFile(Text_t* name, Text_t* refName)
  - HEventFile HEventFile(HEventFile& ef)
  - virtual void ~HEventFile()
  - void addParVersion(HParVersion* pv)
  - TClass* Class()
  - Long_t getFileId()
  - HParVersion* getParVersion(Text_t* name)
  - TList* getParVersions()
  - const Text_t* getRefFileName()
  - virtual TClass* IsA()
  - void print()
  - void resetInputVersions()
  - void resetOutputVersions()
  - void setFileId(Long_t n)
  - void setRefFileName(Text_t* s)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - Long_t fileId: *Id number of the event file*
    - TList* parVersions: *List of container names with the versions*
    - TString refFileName: *! name of the reference event file for initialization*

D.1.7.1. Class Description

```
HEventFile

  class for event file

  Each event file has a name and an id number. The name is
```

```
used to find the parameters valid for this file in Oracle
and in the ROOT file. The id number identifies the event
file. Each event has this number in the event header. It
is also stored in Oracle and can be used to find the
name of the corresponding event file and the valid
parameter versions.
Associated with the event file is a list of container
names with the versions of the containers in the two
possible inputs and the output (class HParVersions).
The input versions are used during the initialisation
used during the initialisation of the containers.
```

D.1.7.2. HEventFile(Text_t* name,Text_t* refName=) : TNamed(name,"event file")

```
constructor with the name of the event file
```

D.1.7.3. HEventFile(HEventFile &ef)

```
copy constructor
```

D.1.7.4. ~HEventFile()

```
destructor
```

D.1.7.5. void addParVersion(HParVersion *pv)

```
adds a container version object to the list
```

D.1.7.6. HParVersion* getParVersion(Text_t* name)

```
return a container version object called by the name of
the container
```

D.1.7.7. void resetInputVersions()

D.1.7.8. void resetOutputVersions()

D.1.7.9. void print()

```
prints the list of container versions for this event file
```

D.1.8. class HEventHeader

- Base classes: public TObject
- public methods
  - HEventHeader HEventHeader()
  - HEventHeader HEventHeader(HEventHeader&)
  - virtual void ~HEventHeader()
  - TClass* Class()
  - UInt_t getDate()
  - UInt_t getEventDecoding()
  - UInt_t getEventFileNumber()
  - UInt_t getEventPad()
  - UInt_t getEventSeqNumber()
  - UInt_t getEventSize()
  - UInt_t getId()
  - UInt_t getTime()
  - virtual TClass* IsA()
  - void setDate(UInt_t date)
  - void setEventDecoding(UInt_t eventDecod)
  - void setEventFileNumber(UInt_t evFileNr)
  - void setEventPad(UInt_t evPad)
  - void setEventSeqNumber(UInt_t evSeqNr)
  - void setEventSize(UInt_t eventSize)
  - void setId(UInt_t id)
  - void setTime(UInt_t time)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - private:
    - UInt_t fEventSize:
    - UInt_t fEventDecoding:
    - UInt_t fId: *Event id*
    - UInt_t fEventSeqNumber: *Event number*
    - UInt_t fDate: *date*
    - UInt_t fTime: *time*
    - UInt_t fEventFileNumber:
    - UInt_t fEventPad: *64 bit alignment*

D.1.8.1. Class Description

```
HEventHeader

 This class contains the header information of an event (run number...)
 See Data Members for more detailed information.
```

- Autor: Manuel Sanchez Manuel Sanchez

D.1.9. class HFilter

- Base classes: public TObject
- public methods
  - virtual void ~HFilter()
  - virtual Bool_t check(HDataObject* obj)
  - TClass* Class()
  - virtual TClass* IsA()
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)

D.1.9.1. Class Description

```
HFilter

  ABC defining the basic interface of filters.

  A filter is a objects which is able to take one HDataObject and tell
if it pass or not that particular filter.

  The most important method of this class is
HFilter::check(HDataObject *dataObject), if it returns kTRUE "dataObject"
passes the filter and if the function returns kFALSE, then "dataObject"
doesn't pass the filter.
```

D.1.9.2. ~HFilter(void)

- Autor: Manuel Sanchez

D.1.10. class HIterator

- Base classes: public TIterator
- public methods
  - virtual void ~HIterator()
  - TClass* Class()
  - virtual HLocation& getLocation()
  - virtual Bool_t gotoLocation(HLocation& aLoc)
  - virtual TClass* IsA()
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)

D.1.10.1. Class Description

```
HIterator

  Base class for categories iterators. This class inherits from TIterator
so you can iterate on categories very much like on Root collections.
```

D.1.10.2. ~HIterator(void)

- Autor: Manuel Sanchez

D.1.11. class HLocatedDataObject

- Base classes: public HDataObject
- public methods
  - virtual void ~HLocatedDataObject()
  - TClass* Class()
  - virtual Int_t getLocationIndex(Int_t n)
  - virtual Int_t getNLocationIndex()
  - virtual TClass* IsA()
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)

D.1.11.1. Class Description

```
HLocatedDataObject

 This is a data object which also stores it's location in the event structure
 This location is retrieved in a standard way through the functions
getNLocationIndex() (returns the number of indexes needed to locate the
data object) and getLocationIndex(Int_t n) (which returns the value of the
index number "n").
```

D.1.11.2. ~HLocatedDataObject(void)

- Autor: Manuel Sanchez

D.1.12. class HLocation

- Base classes: public TObject
- public methods
  - HLocation HLocation()
  - HLocation HLocation(HLocation& aLoc)
  - virtual void ~HLocation()
  - TClass* Class()
  - virtual void Dump()
  - Int_t getIndex(Int_t aIdx)
  - Int_t getLinearIndex(TArrayI* sizes)
  - Int_t getNIndex()
  - Int_t getOffset()
  - Int_t getUncheckedIndex(Int_t aIdx)
  - void incIndex(Int_t nIndex)
  - virtual TClass* IsA()
  - int operator++()
  - void operator+=(Int_t shift)
  - void operator–()
  - void operator-=(Int_t shift)
  - HLocation& operator=(HLocation& loc)
  - Int_t& operator[](Int_t aIdx)
  - void readIndexes(HLocation& loc)

- void set(Int_t nIndex, int)
- void setIndex(Int_t aIdx, Int_t aValue)
- void setNIndex(Int_t nIdx)
- void setOffset(Int_t aOfs)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

- Data Members
  - protected:
    - TArrayI fIndexes: *High indexes indicating the location*
    - Int_t fOffset: *Less significant index of the location*

D.1.12.1. Class Description

```
HLocation

  ATENCION! Se ha eliminado el index check en operator[]

  A location is a set of indexes defining the location of a particular
data object within the HEvent structure; so within a loop is recommended to
directly use one HLocation instead of a set of indexes (i,j,k,l...) (and
better an HIterator instead of HLocation).

  An HEvent holds the data corresponding to an event, those data are arranged
 in categories (mdc raw data, mdc cal data, rich rings,...) which are objects
 instantianting one HCategory, to access one of these categories the user
 can call HEvent::getCategory(). Within the categories, each data object has
 a location (given by an object instantiating HLocation) which identifies
 that object in the category (in some sense, the location is a generalized
 index).
```

D.1.12.2. HLocation(void)

```
Default constructor
```

D.1.12.3. HLocation(HLocation &aLoc)

```
Copy constructor
```

D.1.12.4. ~HLocation(void)

```
Destructor
```

D.1.12.5. void set(Int_t nIndex,...)

```
Sets all the indexes in the HLocation at once:

The first argument is the number of indexes to be used and the next
arguments are the actual indexes (as many as nIndex)
```

D.1.12.6. void setNIndex(Int_t nIdx)

```
Sets the number of indexes to be used and resets all indexes to 0
```

D.1.12.7. void incIndex(Int_t nIndex)

```
Increments the index nIndex and sets all the following indexes to 0
```

D.1.12.8. void readIndexes(HLocation &loc)

```
Sets the loc.getNIndex() first indexes to the value given in loc
and the others are set to 0
```

D.1.12.9. void Dump(void)

```
Dumps the object to cout in the format:

index1:index2:.....:offset
```

- Autor: Manuel Sanchez

D.1.13. class HMatrixCategory

- Base classes: public HCategory
- public methods
  - HMatrixCategory HMatrixCategory()
  - HMatrixCategory HMatrixCategory(Text_t* className, Int_t nSizes, Int_t* sizes, Float_t fillRate = 0.1)
  - HMatrixCategory HMatrixCategory(HMatrixCategory&)
  - virtual void ~HMatrixCategory()

- virtual void activateBranch(TTree* tree, Int_t splitLevel)
- virtual void Browse(TBrowser* b)
- TClass* Class()
- virtual void Clear()
- virtual Bool_t filter(HFilter& aFilter)
- virtual Bool_t filter(HLocation& aLoc, HFilter& aFilter)
- virtual TClass* getClass()
- virtual const Text_t* getClassName()
- virtual TObject*& getNewSlot(HLocation& aLoc)
- virtual HDataObject* getObject(HLocation& aLoc)
- Int_t getSize(Int_t aIdx)
- TArrayI* getSizes()
- virtual TObject*& getSlot(HLocation& aLoc)
- virtual TClass* IsA()
- virtual Bool_t isSelfSplitable()
- virtual void makeBranch(TBranch* parent)
- virtual TIterator* MakeIterator(Bool_t dir = kIterForward)
- virtual Bool_t query(TCollection* aCol, HFilter& aFilter)
- virtual Bool_t query(TCollection* aCol, HLocation& aLoc)
- virtual Bool_t query(TCollection* aCol, HLocation& aLoc, HFilter& aFilter)
- void setup(Text_t* className, Int_t nSizes, Int_t* sizes, Float_t fillRate = 0.1)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

■ Data Members

- protected:
  - HIndexTable* fIndexTable: *Table indicating the position in fData for the data objects*
  - Int_t fNDataObjs: *Number of data objects actually stored.*
  - TClonesArray* fData: *Container for the data in the Matrix category;*

D.1.13.1. Class Description

```
HMatrixCategory

  In this particular HCategory all the objects belonging to the category
 are stored in a matrix-like fashion; so before start using the category it
 is needed to indicate the number of indexes in this matrix and the range of
```

```
each index

 When the matrix category is stored in a tree it creates one superbranch
holding one subbranch per data member in the objects held by the category.
All the objects held by the category are stored through that superbranch.
```

D.1.13.2. HMatrixCategory(void)

```
Default constructor
```

D.1.13.3. HMatrixCategory(Text_t *className, Int_t nSizes, Int_t *sizes, Float_t fillRate=0.1)

```
 Creates a HMatrixCategory holding objects of class "className" clasified
by nSizes indexes (giving their location) whose maximun values are the
indicated in the vector "sizes".

 However it only allocates memory for fillRate*(maximun number of objects)
data objects.
```

D.1.13.4. ~HMatrixCategory(void)

```
Destructor
```

D.1.13.5. void setup(Text_t *className,Int_t nSizes,Int_t *sizes,Float_t fillRate=0.1)

```
This method sets up a category.

Input:
  className ---> a string with the name of the object's class stored in
                 the category
  nSizes ---> Number of indexes needed to access an object (fBranchingLevel)
  sizes ---> a vector of ints containing the maximum value for each of the
             indexes before.
  fillRate---> Expected fill factor in the category
               (number of slots used)/(maximun number of slots)
```

D.1.13.6. const Text_t* getClassName(void)

```
Returns the class' name of the objects held by the category
```

D.1.13.7. TClass* getClass(void)

D.1.13.8. TArrayI* getSizes(void)

returns an array of ints with the maximun value for the indexes used to get an object

D.1.13.9. Int_t getSize(Int_t aIdx)

Returns the maximun value of the index aIdx

D.1.13.10. void activateBranch(TTree *tree,Int_t splitLevel)

Activates the branches in tree matching the appropiate names.

D.1.13.11. void makeBranch(TBranch *parent)

This function should not be called, since HMatrixCategory is not self splittable

D.1.13.12. Bool_t isSelfSplitable(void)

Determines if the category can be automatically split or it has
its own splitting algorithm (in this last case the function makeBranch is called for splitting).

returns kFALSE

D.1.13.13. HDataObject* getObject(HLocation &aLoc)

Returns the object at the HLocation aLoc (see HCategory)

D.1.13.14. Bool_t query(TCollection *aCol,HFilter &aFilter)

Puts all objects in the category which pass the HFilter aFilter in the collection pointed by aCol.

D.1.13.15. Bool_t query(TCollection *aCol,HLocation &aLoc)

Puts all objects in the category corresponding to the HLocation aLoc in the collection pointed by aCol

D.1.13.16. Bool_t query(TCollection *aCol, HLocation &aLoc, HFilter &aFilter)

```
Puts all the objects in the category passing the HFilter aFilter and
corresponding to the HLocation aLoc into the collection pointed by aCol
```

D.1.13.17. Bool_t filter(HFilter &aFilter)

```
see HCategory
```

D.1.13.18. Bool_t filter(HLocation &aLoc,HFilter &aFilter)

```
see HCategory
```

D.1.13.19. TIterator* MakeIterator(Bool_t dir)

```
Returns an iterator, which iterates in the direction dir and (as default)
throuhg the whole category.
```

D.1.13.20. void Clear(void)

```
 see HCategory
```

D.1.13.21. void Browse(TBrowser *b)

```
Browse objects in this category
```

D.1.13.22. void Streamer(TBuffer &R__b)

```
 Stream an object of class HMatrixCategory.
```

- Autor: Manuel Sanchez

D.1.14. class HParIo

- public methods
  - HParIo HParIo()
  - HParIo HParIo(HParIo&)
  - virtual void ~HParIo()
  - virtual void cd()

- virtual Bool_t check()
- TClass* Class()
- virtual void close()
- virtual HDetParIo* getDetParIo(Text_t*)
- virtual TClass* IsA()
- virtual void print()
- virtual void readVersions(HEventFile*)
- virtual void removeDetParIo(Text_t*)
- virtual void setDetParIo(HDetParIo*)
- virtual void setDetParIo(Text_t*)
- void setInputNumber(Int_t)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

■ Data Members

- protected:
  - TList* detParIoList: *list of detector I/Os*

D.1.14.1. Class Description

```
HParIo

  Base class for different parameter input/output sources:
  Oracle, Rootfiles, Asciifiles
  It contains a list of detector I/Os.
```

D.1.14.2. HParIo()

```
default constructor creates an empty list of detector I/Os
```

D.1.14.3. ~HParIo()

```
 default destructor
```

D.1.14.4. void setDetParIo(HDetParIo* detParIo)

```
 stores pointer of the input/output class for a detector
 used for I/O from ROOT file or Asci file
```

D.1.14.5. void setInputNumber(Int_t num)

```
sets in all detector I/Os the number of the input
```

D.1.14.6. HDetParIo* getDetParIo(Text_t* detName)

```
returns pointer to input/output class for a detector
```

D.1.14.7. void removeDetParIo(Text_t* detName)

```
removes input/output class for a detector
```

D.1.15. class HParSet

- Base classes: public TNamed
- public methods
  - HParSet HParSet()
  - HParSet HParSet(HParSet&)
  - virtual void ~HParSet()
  - TClass* Class()
  - virtual void clear()
  - Int_t getInputVersion(Int_t i)
  - Bool_t hasChanged()
  - virtual Bool_t init()
  - virtual Bool_t init(HParIo* io)
  - virtual Bool_t init(HParIo*, Int_t*)
  - virtual TClass* IsA()
  - Bool_t isStatic()
  - virtual void print()
  - void resetInputVersions()
  - void setChanged(Bool_t flag = kTRUE)
  - void setInputVersion(Int_t v = -1, Int_t i = 0)
  - void setStatic(Bool_t flag = kTRUE)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
  - virtual Int_t write()
  - virtual Int_t write(HParIo*)

- Data Members
  - protected:
    - Text_t detName[10]: *! name of the detector the container belongs to*
    - Int_t versions[3]: *! versions of container in the 2 possible inputs*
    - Bool_t status: *! static flag*
    - Bool_t changed: *! flag is kTRUE if parameters have changed*

D.1.15.1. Class Description

```
HParSet

  Base class for all parameter containers
```

D.1.15.2. HParSet()

```
constructor sets default values of data elements
```

D.1.15.3. Bool_t init(HParIo* io)

```
intitializes the container from an input in run time
database. If this is not successful it is initialized from
the second input. If this failes too, it returns an error.
(calls internally the init function in the derived class)
```

D.1.15.4. Int_t write()

```
writes the container to the output defined in the runtime
database
returns the output version in the ROOT file
returns -1 if error occured
(calls internally the init function in the derived class)
```

D.1.15.5. void print()

```
prints information about container (versions,status,hasChanged...)
```

D.1.15.6. void resetInputVersions()

```
resets the input versions if the container is not static
```

D.1.16. class HPartialEvent

- Base classes: public HEvent
- public methods
  - HPartialEvent HPartialEvent()
  - HPartialEvent HPartialEvent(Text_t* aName, Text_t* aTitle, Cat_t aBaseCat)
  - HPartialEvent HPartialEvent(HPartialEvent&)
  - virtual void ~HPartialEvent()
  - virtual void activateBranch(TTree* tree, Int_t splitLevel)
  - void addCategory(Cat_t aCat, HCategory* category)
  - virtual Bool_t addCategory(Cat_t aCat, HCategory* cat, Option_t* opt)
  - virtual void Browse(TBrowser* b)
  - TClass* Class()
  - virtual void Clear()
  - virtual void clearAll(Int_t level)
  - TObjArray* getCategories()
  - virtual HCategory* getCategory(Cat_t aCat)
  - Int_t getRecLevel()
  - TObject* getSubHeader()
  - virtual TClass* IsA()
  - virtual Bool_t IsFolder()
  - virtual void makeBranch(TBranch* parent)
  - void setRecLevel(Int_t aRecLevel)
  - void setSubHeader(TObject* header)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - Int_t fRecLevel: *Reconstruction level of the event*
    - Cat_t fBaseCategory: *Identifier for the first category stored in this partial event*
    - TObject* fHeader: *-Sub event's header.*
    - TObjArray* fCategories: *- Categories*

D.1.16.1. Class Description

```
HPartialEvent Begin_Html
<a href="HPartialEvent.gif">(UML Diagram)</a>
End_Html

  Each partial events holds the data (i.e. the HDataObjects) corresponding
to each main detection system; like Mdcs, Rich...

  These data in a partial event are organized in categories, which stand
for the different types of data in the partial event. For example, in the
partial event for the Mdcs there will be a category holding the objects with
raw data (i.e. the objects instatiating a HMdcRaw). So conceptually the
categories are "kind of data" and they hold the objects of a particular
kind (instatiating a particular class).

  The categories are the responsible of giving access to the objects they
hold (see HCategory), but a partial event must also allow the user to
access the data stored into it; that's done using the function getCategory()
which returns a particular category, so the user can ask that category for
the data object he wants. For example, let's say we have a HLocation object
loc pointing to any location, and a HPartialEvent par with the Mdc's data;
if we want to get the raw data object pointed by loc we should use:

   par->getCategory(catMdcRaw)->getObject(loc);

   And this function will return the pursued data object.
```

D.1.16.2. HPartialEvent(void)

```
Default constructor
```

D.1.16.3. HPartialEvent(const Text_t *aName, const Text_t *aTitle, Cat_t aBaseCat) : HEvent(aName,aTitle)

```
Allocates a new HPartial event with name aName and title aTitle

 aBaseCat is the identifier of the lower category that will be stored in
this partial event, for example: if the user wants to allocate a partial
event holding the Mdc data, then aBaseCat should be: catMdc; which is a
constant defined in HadDef.h
```

D.1.16.4. ~HPartialEvent(void)

```
Clears the event
```

D.1.16.5. void activateBranch(TTree *tree,Int_t splitLevel)

```
 Activates those branches in the TTree tree whose names are equal to those
of the branches that would be generated with the current event structure if
the function makeBranch() is called, being the split level=splitLevel

 This way if we have a TTree with a lot of branches and we only need some
of them to fill data in the event structure we are using at a given moment,
then we only activate those branches and the others are not read
```

D.1.16.6. void makeBranch(TBranch *parent)

```
 Makes branches for each category in the partial event and for the split
tree, then those branches are added to the list of subbranches of "parent"

 This method is called by Hades::makeTree()
```

D.1.16.7. Bool_t addCategory(Cat_t aCat,HCategory *cat,Option_t *)

```
 Adds the category "cat", identified by "aCat" to the partial event.
```

D.1.16.8. void addCategory(Cat_t aCat,HCategory *category)

```
 Adds the category "category", which is identified by aCat, to the partial
event

 Before using this function the user should setup the HCategory "category"
```

D.1.16.9. TObjArray* getCategories(void)

```
 Returns an array with all the categories held by this event

 To add categories to the partial event use HPartialEvent::addCategory()
```

D.1.16.10. HCategory* getCategory(Cat_t aCat)

```
 Returns a pointer to the category identified by "aCat"
```

D.1.16.11. void Clear(void)

Clears the partial event,i.e. deletes all objects in the partial event but keeping its structure.

After the reconstruction of each event, the HEvent::Clear function must be called in order to make place for the new data.

D.1.16.12. void clearAll(Int_t level)

See HEvent

D.1.16.13. void setRecLevel(Int_t aRecLevel)

Sets the reconstruction level of the event, this is useful to know which categories in the event are already reconstructed.

D.1.16.14. Int_t getRecLevel(void)

Returns the reconstruction level of the partial event.

D.1.16.15. void Browse(TBrowser *b)

Method used to browse a partial event with a Root browser.

This function will be called by a Root browser, not by the user

- Autor: Manuel Sanchez

D.1.17. class HRaIndexNode

- Base classes: public HRaNode
- protected methods
  - HRaIndexNode HRaIndexNode()
- public methods
  - HRaIndexNode HRaIndexNode(Int_t size)
  - HRaIndexNode HRaIndexNode(HRaIndexNode&)
  - virtual void ~HRaIndexNode()
  - TClass* Class()
  - virtual void clear()

- TObject* getCell(Int_t i)
- virtual TClass* IsA()
- void setCell(TObject* obj, Int_t i)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

- Data Members
  - protected:
    - TObjArray fCells:

D.1.17.1. Class Description

```
HRaTree, HRaNode, HRaIndexNode

 HRaTree stands for "Hades Random Access Tree". This class implements
random access to the objects in any category using a tree; the only condition
is that the data objects in the category inherit from HLocatedDataObject, so
the random access implemented by the HRaTree corresponds to the indexes
in the data objects rather than the natural indexing in the category.

 As said, the HRaTree is a tree of HRaNode objects. The lower level nodes
are HRaIndexNode and store pointers to the actual data objects being accesed
with the HRaTree.
```

D.1.17.2. void clear(void)

```
Clears the pointers to the data objects being accessed throught the HRaTree
```

- Autor: Manuel Sanchez

D.1.18. class HRaNode

- Base classes: public TObject
- protected methods
  - HRaNode HRaNode()
- public methods
  - HRaNode HRaNode(Int_t size)
  - HRaNode HRaNode(HRaNode&)
  - virtual void ~HRaNode()

- void addSubNode(HRaNode* n)
- TClass* Class()
- virtual void clear()
- HRaNode* getSubNode(Int_t idx)
- virtual TClass* IsA()
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

- Data Members
  - protected:
    - TObjArray* fSubNodes:

D.1.18.1. Class Description

```
HRaTree, HRaNode, HRaIndexNode

 HRaTree stands for "Hades Random Access Tree". This class implements
random access to the objects in any category using a tree; the only condition
is that the data objects in the category inherit from HLocatedDataObject, so
the random access implemented by the HRaTree corresponds to the indexes
in the data objects rather than the natural indexing in the category.

 As said, the HRaTree is a tree of HRaNode objects. The lower level nodes
are HRaIndexNode and store pointers to the actual data objects being accesed
with the HRaTree.
```

D.1.18.2. void clear(void)

```
Calls "Clear" for each of the subnodes
```

- Autor: Manuel Sanchez Manuel Sanchez

D.1.19. class HRaTree

- Base classes: public TObject
- protected methods
  - Bool_t addObject(HLocatedDataObject* obj)
  - HRaNode* buildNode(TArrayI* sizes, Int_t lvl)
  - Bool_t buildTree(TArrayI* sizes)

- public methods
  - HRaTree HRaTree()
  - HRaTree HRaTree(HCategory* cat, TArrayI* sizes)
  - HRaTree HRaTree(HCategory* cat, HLocation& aLoc, TArrayI* sizes)
  - HRaTree HRaTree(HRaTree&)
  - virtual void ~HRaTree()
  - TClass* Class()
  - TObject* getObject(HLocation& aLoc)
  - TObject* getObject(Int_t i1 = -1, Int_t i2 = -1, Int_t i3 = -1, Int_t i4 = -1, Int_t i5 = -1, Int_t i6 = -1, Int_t i7 = -1, Int_t i8 = -1, Int_t i9 = -1)
  - HRaNode* getRoot()
  - virtual TClass* IsA()
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
  - Bool_t update()
- Data Members
  - protected:
    - HRaNode* fRoot: *index tree's root node*
    - HCategory* fSourceCategory: *! Category actually holding data*
    - Int_t fDepth: *Tree's depth*
    - HLocation fLoc: *Location for this tree.*
    - HIterator* fIter: *! Iterator on data objects in the category*
    - Int_t fLowerLevel:

D.1.19.1. Class Description

```
HRaTree, HRaNode, HRaIndexNode

 HRaTree stands for "Hades Random Access Tree". This class implements
random access to the objects in any category using a tree; the only condition
is that the data objects in the category inherit from HLocatedDataObject, so
the random access implemented by the HRaTree corresponds to the indexes
in the data objects rather than the natural indexing in the category.

 As said, the HRaTree is a tree of HRaNode objects. The lower level nodes
are HRaIndexNode and store pointers to the actual data objects being accesed
with the HRaTree.
```

D.1.19.2. HRaTree(void)

```
Constructor
```

D.1.19.3. HRaTree(HCategory *cat,TArrayI *sizes)

```
Constructor for a tree accesing the data in "cat". "sizes" gives the
max size of each level in the tree.
```

D.1.19.4. HRaTree(HCategory *cat,HLocation &loc,TArrayI *sizes)

```
Constructor like the one before but accessing only those objects in the
category which correspond to the location "loc"
```

D.1.19.5. HRaNode* buildNode(TArrayI *sizes,Int_t lvl)

```
Recursive function used to instantiate the tree.
```

D.1.19.6. Bool_t buildTree(TArrayI *sizes)

```
Instantiates the tree using buildNode()
```

D.1.19.7. ~HRaTree(void)

```
Destructor
```

D.1.19.8. Bool_t addObject(HLocatedDataObject *obj)

```
Adds the pointer "obj" to the right place in the tree so it can be accesed
after.
```

D.1.19.9. Bool_t update(void)

```
Updates the contents of the tree with the data in the accessed category.
```

D.1.19.10. TObject* getObject(HLocation &aLoc)

```
Returns the object corresponding to the location "loc"
Warning: for the sake of speed no index checking is done here.
```

D.1.19.11. TObject* getObject(Int_t i1,Int_t i2,Int_t i3,Int_t i4, Int_t i5,Int_t i6,Int_t i7,Int_t i8, Int_t i9)

```
Returns the object corresponding to the location with indexes:
"i1","i2" up to "i9"
```

- Autor: Manuel Sanchez

D.1.20. class HRecEvent

- Base classes: public HEvent
- public methods
  - HRecEvent HRecEvent()
  - HRecEvent HRecEvent(HRecEvent&)
  - virtual void ~HRecEvent()
  - virtual void activateBranch(TTree* tree, Int_t splitLevel)
  - virtual Bool_t addCategory(Cat_t aCat, HCategory* cat, Option_t* opt)
  - HPartialEvent* addPartialEvent(Cat_t eventCat, Text_t* name, Text_t* title)
  - void addTrack(HTrack& aTrack)
  - virtual void Browse(TBrowser* b)
  - TClass* Class()
  - virtual void Clear()
  - virtual void clearAll(Int_t level)
  - void clearTracks()
  - virtual HCategory* getCategory(Cat_t aCat)
  - virtual HEventHeader* getHeader()
  - HPartialEvent* getPartialEvent(Int_t idx)
  - Int_t getRecLevel()
  - HTrack* getTrack(UInt_t aId)
  - virtual TClass* IsA()
  - virtual Bool_t IsFolder()
  - virtual void makeBranch(TBranch* parent)
  - HTrack* newTrack()
  - void setRecLevel(Int_t aRecLevel)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)

- Data Members
  - protected:
    - Int_t fRecLevel: *Reconstruction level for this event*
    - Int_t fNTracks: *Number of reconstructed tracks in this event*
    - HEventHeader fHeader: *Event header*
    - TObjArray* fPartialEvs: - *Partial events for this event*
    - TClonesArray* fTracks: - *Reconstructed tracks for this event*

D.1.20.1. Class Description

```
HRecEvent

 A HRecEvent is an event under reconstruction, and ,in particular, a
reconstructed event. These events can be in different reconstruction levels,
the reconstruction level of a HRecEvent is controlled by setRecLevel() and
getRecLevel(); the reconstruction levels are identified by global constants
(rlRaw, rlHit ...)

 As for the data, a HRecEvent holds reconstructed tracks, an event header
and a list of HPartialEvent objects.
```

D.1.20.2. HRecEvent(void) : HEvent("Event","Event under Reconstruction")

```
Allocates a new HRecEvent
```

D.1.20.3. ~HRecEvent(void)

```
Cleans the HRecEvent for its deallocation
```

D.1.20.4. void activateBranch(TTree *tree,Int_t splitLevel)

```
 see HEvent::activateBranch
```

D.1.20.5. void makeBranch(TBranch *parent)

```
see HEvent::makeBranch
```

D.1.20.6. void Clear(void)

Clears the data in the event (i.e. clears the internal buffers...)

D.1.20.7. void clearAll(Int_t level)

Clears the data in the event and the event structure (list of subevents...)

D.1.20.8. HTrack* newTrack(void)

Returns an pointer to a new HTrack object

D.1.20.9. void addTrack(HTrack &aTrack)

Adds the track aTrack to the list of reconstructed tracks

D.1.20.10. HTrack* getTrack(UInt_t aId)

Returns the track identified by aId (the position in the track list)

D.1.20.11. void clearTracks(void)

Clears the track list

D.1.20.12. Int_t getRecLevel(void)

Returns the reconstruction level for this event

D.1.20.13. void setRecLevel(Int_t aRecLevel)

Sets the reconstruction level for the event

D.1.20.14. HPartialEvent* getPartialEvent(Int_t idx)

Returns a pointer to the partial event with number idx.

D.1.20.15. HCategory* getCategory(Cat_t aCat)

Returns the category identified by aCat in the correct Partial event

D.1.20.16. Bool_t addCategory(Cat_t aCat,HCategory *cat,Option_t opt[])

```
 Adds a new category to the event. The partial event it belongs to is
determined by aCat; if this partial event doesn't exist, one is created
with the name given in opt.
```

D.1.20.17. HPartialEvent* addPartialEvent(Cat_t eventCat, const Text_t *name, const Text_t *title)

```
 Adds a new HPartialEvent to the list of HPartialEvent objects in the
HRecEvent

Input:
 eventCat  ---> Base category for the event (i.e. for Mdc it is catMdc)
 name      ---> Name of the new partial event (used to build Root trees)
 title     ---> Title of the new partial event
```

D.1.20.18. void Browse(TBrowser *b)

```
 Event browser.
```

- Autor: Manuel Sanchez

D.1.21. class HReconstructor

- Base classes: public HTask
- public methods
  - virtual void ~HReconstructor()
  - TClass* Class()
  - virtual Bool_t connectTask(HTask* task, Int_t n = 0)
  - virtual Int_t execute()
  - virtual TClass* IsA()
  - virtual HTask* next(Int_t& errCode)
  - void setActive(Bool_t state)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:

- TList* fHistograms: *List of histograms generated by this reconstructor.*
- Bool_t fActive: *Active flag*
- TObjArray fOutputs:

D.1.21.1. Class Description

```
HReconstructor

  Base class for any reconstruction method or algoritm. See HTask for
more information.
```

D.1.21.2. ~HReconstructor(void)

```
Destructor
```

D.1.21.3. HTask* next(Int_t &errCode)

```
Returns next task to be performed according to the return value of the
execute() function
```

D.1.21.4. Bool_t connectTask(HTask *task,Int_t n)

```
Connects the return value "n" to the task "task"
```

- Autor: Manuel Sanchez

D.1.22. class HRootSource

- Base classes: public HDataSource
- public methods
  - HRootSource HRootSource()
  - HRootSource HRootSource(HRootSource&)
  - virtual void ~HRootSource()
  - TClass* Class()
  - void Clear()
  - void deactivateBranch(Text_t* branchName)
  - Bool_t getEvent(Int_t eventN)
  - virtual EDsState getNextEvent()

- Int_t getSplitLevel()
- TTree* getTree()
- virtual Bool_t init()
- virtual TClass* IsA()
- Bool_t setInput(Text_t* fileName, Text_t* treeName)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

- Data Members
  - private:
    - TTree* fInputTree: *TTree to be read.*
    - TFile* fInputFile: *Input file.*
    - Int_t fCursor: *Number of next event.*
    - Int_t fSplitLevel: *Split level of input tree*

D.1.22.1. Class Description

```
HRootSource

 The HRootSource class reads all or part of the data stored in a Root file
using a TTree.
```

D.1.22.2. HRootSource(void)

D.1.22.3. ~HRootSource(void)

D.1.22.4. Bool_t init(void)

D.1.22.5. EDsState getNextEvent(void)

D.1.22.6. Bool_t getEvent(Int_t eventN)

D.1.22.7. void Clear(void)

D.1.22.8. Bool_t setInput(Text_t *fileName,Text_t *treeName)

D.1.22.9. void deactivateBranch(Text_t *branchName)

D.1.22.10. TTree* getTree(void)

- Autor: Manuel Sanchez

D.1.23. class HRuntimeDb

- public methods
  - HRuntimeDb HRuntimeDb()
  - HRuntimeDb HRuntimeDb(HRuntimeDb&)
  - virtual void ~HRuntimeDb()
  - Bool_t addContainer(HParSet* container)
  - void addEventFile(Text_t* name, Text_t* refFile)
  - TClass* Class()
  - void clearContainerList()
  - void clearEventFileList()
  - void closeFirstInput()
  - void closeOutput()
  - void closeSecondInput()
  - Int_t findOutputVersion(HParSet* cont)
  - HParSet* getContainer(Text_t* name)
  - HEventFile* getCurrentEventFile()
  - HEventFile* getEventFile(Int_t n = 0)
  - HEventFile* getEventFile(Text_t* name)
  - TList* getEventFileList()
  - Int_t getEventFilePosition(Text_t* name)
  - HParIo* getFirstInput()
  - HEventFile* getNextEventFile()
  - Int_t getNumEventFiles()
  - HParIo* getOutput()
  - HParIo* getSecondInput()
  - Bool_t initContainers()
  - void insertEventFile(Text_t* name, Int_t position, Text_t* refFile)
  - virtual TClass* IsA()
  - void print()
  - Bool_t readAll()
  - void removeContainer(Text_t* name)
  - void removeEventFile(Int_t n = 0)
  - void removeEventFile(Text_t* name)
  - void resetAllVersions()

- void resetInputVersions()
- void resetOutputVersions()
- void saveOutput()
- void setContainersStatic(Bool_t f = kTRUE)
- void setCurrentEventFile(Int_t n = -1)
- Bool_t setFirstInput(HParIo* inp1)
- Bool_t setInputVersion(Text_t* eventFileName, Text_t* container, Int_t version, Int_t inputNumber)
- Bool_t setOutput(HParIo* output)
- Bool_t setRootOutputVersion(Text_t* eventFileName, Text_t* container, Int_t version)
- Bool_t setSecondInput(HParIo* inp2)
- void setVersionsChanged(Bool_t f = kTRUE)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)
- Bool_t writeContainer(HParSet* cont, HEventFile* file)
- Bool_t writeContainers()
- void writeSetup()
- void writeVersions()

■ Data Members

- protected:
  - TList* containerList: *list of parameter containers*
  - TList* eventFiles: *list of event files*
  - HParIo* firstInput: *first (prefered) input for parameters*
  - HParIo* secondInput: *second input (used of not found in first input)*
  - HParIo* output: *output*
  - Int_t currentFilePos: *position of current event file in the list*
  - Bool_t versionsChanged: *flag for write of list eventFiles (set kTRUE by each write)*

D.1.23.1. Class Description

```
HRuntimeDb

  Administration class for parameter input/output
```

D.1.23.2. HRuntimeDb()

```
constructor creates an empty list for parameter containers
and an empty list of event files for the version management
```

D.1.23.3. ~HRuntimeDb()

```
destructor
deletes the list of event files and all containers
```

D.1.23.4. Bool_t addContainer(HParSet* container)

```
adds a container to the list of containers
```

D.1.23.5. HParSet* getContainer(Text_t* name)

```
returns a pointer to the container called by name
```

D.1.23.6. void removeContainer(Text_t* name)

```
removes the container from the list and deletes it
```

D.1.23.7. void clearContainerList()

D.1.23.8. void addEventFile(Text_t* name,Text_t* refFile=)

```
adds an event file at the end of the list of event files
```

D.1.23.9. void insertEventFile(Text_t* name,Int_t pos, Text_t* refFile=)

```
inserts an event file in the list of event files at position pos
```

D.1.23.10. Int_t getEventFilePosition(Text_t* name)

```
returns the position of the event file in the list of event files
returns -1 if not found
```

D.1.23.11. Int_t getNumEventFiles()

```
returns number of event files
```

D.1.23.12. TList* getEventFileList()

D.1.23.13. void clearEventFileList()

D.1.23.14. void setCurrentEventFile(Int_t n=-1)

```
sets the current event file pointer to position n
ATTENTION: This number must be 1 less then the file to be
analyzed because the function getNextEventFile() increments
the position by 1 internally!
```

D.1.23.15. HEventFile* getCurrentEventFile()

```
returns a pointer to the current event file
(without calling the init() funtions of the containers)
```

D.1.23.16. HEventFile* getEventFile(Int_t n=0)

```
returns a pointer to the event file position n in list
```

D.1.23.17. HEventFile* getEventFile(Text_t* name)

```
returns a pointer to the event file called by name
```

D.1.23.18. void removeEventFile(Text_t* name)

```
removes the event file from the list and deletes it
```

D.1.23.19. void removeEventFile(Int_t n=0)

```
removes the event file at position n from the list and deletes it
```

D.1.23.20. void writeSetup()

```
writes the setup to the output ever time the setup has changed
```

D.1.23.21. void writeVersions()

```
writes the event file versions to the output
```

D.1.23.22. Bool_t writeContainers()

```
writes all containers to the output
loops over the list of containers and calls for each the
function writeContainer(...)
```

D.1.23.23. Int_t findOutputVersion(HParSet* cont)

D.1.23.24. Bool_t writeContainer(HParSet* cont, HEventFile* file)

```
writes a container to the output if the containers has changed
The output might be suppressed if the changes is due an
initialisation from a ROOT file which serves also as output
or if it was already written
```

D.1.23.25. Bool_t initContainers()

```
loops over the list of containers and calls the init()
function of each container if it is not static
```

D.1.23.26. void setContainersStatic(Bool_t f)

```
sets the status flag in all containers
flag kTRUE sets the all 'static'
flag kFALSE sets the all not 'static'
```

D.1.23.27. HEventFile* getNextEventFile()

```
checks first if the containers have changed during the analysis
of the last event file; the containers which have changed are
written to the output
Each container is reinitialized if necessary with the version
needed for the next event file.
return a pointer to this 'event file'.
```

D.1.23.28. Bool_t readAll()

```
reads all containers with all versions for all event files and
writes the containers to the output
```

D.1.23.29. Bool_t setInputVersion(Text_t* eventFileName,Text_t* container, Int_t version,Int_t inp)

```
sets the input version of a container defined by its name and an
event file defined by its name taken from input with inputNumber inp
(1 for first input and 2 for second input)
```

D.1.23.30. Bool_t setRootOutputVersion(Text_t* eventFileName,Text_t* container, Int_t version)

```
sets the Root file output version of a container defined by its name
and an event file defined by its name
should only be used after initialization 'by hand' on the interpreter level
```

D.1.23.31. void print()

```
prints the list of the actuzal containers, the list of the
event files/versions and information about input/output
```

D.1.23.32. void resetInputVersions()

```
resets all input versions in the event file list and in all
containers which are not static
is called each time a new input is set
```

D.1.23.33. void resetOutputVersions()

```
resets all output versions in the event file list
is called each time a new output is set
is called also each time a new input is set which is not
identical with the output
```

D.1.23.34. void resetAllVersions()

```
resets all input and output versions in the event file list
and in all containers which are not static
```

D.1.23.35. Bool_t setFirstInput(HParIo* inp1)

```
sets the first input pointer
```

D.1.23.36. Bool_t setSecondInput(HParIo* inp2)

```
sets the second input pointer
```

D.1.23.37. Bool_t setOutput(HParIo* op)

```
sets the output pointer
```

D.1.23.38. HParIo* getFirstInput()

```
return a pointer to the first input
```

D.1.23.39. HParIo* getSecondInput()

```
return a pointer to the second input
```

D.1.23.40. HParIo* getOutput()

```
return a pointer to the output
```

D.1.23.41. void closeFirstInput()

D.1.23.42. void closeSecondInput()

D.1.23.43. void saveOutput()

```
writes the setup, the versions and the containers (if not
yet written out)
without the setup and version information the containers cannot
be read again!
```

D.1.23.44. void closeOutput()

```
calls saveOutput() and deletes then the output
```

D.1.23.45. HEventFile* getNextEventFile()

D.1.24. class HSpectrometer

- Base classes: public TNamed
- public methods

- HSpectrometer HSpectrometer()
- HSpectrometer HSpectrometer(HSpectrometer&)
- virtual void ~HSpectrometer()
- void activateParIo(HParIo* io)
- void addDetector(HDetector* det)
- TClass* Class()
- HDetector* getDetector(Text_t* name)
- Bool_t hasChanged()
- Bool_t init(Text_t* level = raw)
- virtual TClass* IsA()
- void print()
- void setChanged(Bool_t f = kTRUE)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)
- Bool_t write()
- Bool_t write(HParIo* io)

■ Data Members

- protected:
  - TList* detectors: *List of detectors*
  - Bool_t changed: *! flag is kTRUE if the setup changes*

D.1.24.1. Class Description

```
HSpectrometer

  class for the actual setup of the HADES spectrometer used
  in the analysis.
  It contains the list of defined detectors.
```

D.1.24.2. HSpectrometer(void) : TNamed("Hades","The Hades spectrometer")

```
 constructor creates an empty list of detectors
```

D.1.24.3. ~HSpectrometer(void)

```
 destructor
```

D.1.24.4. void addDetector(HDetector* det)

```
adds a detector to the list of detectors
```

D.1.24.5. HDetector* getDetector(Text_t* name)

```
returns a pointer to the detector called by name
```

D.1.24.6. Bool_t init(Text_t* level=raw")

```
Calls the init function of each detector in the spectrometer.
The string level indicates the start level of the analysis
Each detector (as e.g. MDC) which needs special initialization
before creating the tasks must have an init(...)-function
```

D.1.24.7. void activateParIo(HParIo* io)

```
loops over the list of detectors and activates the
corresponding detector I/Os
```

D.1.24.8. Bool_t write()

```
writes the actual setup to the output defined in the
runtime database
```

D.1.24.9. Bool_t write(HParIo* io)

```
writes the actual setup to the output
```

D.1.24.10. void print()

```
prints the actual setup
```

D.1.25. class HTask

- Base classes: public TNamed
- public methods
  - virtual void ~HTask()
  - TClass* Class()
  - virtual Bool_t connectTask(HTask* task, Int_t n)

- virtual Bool_t finalize()
- virtual Bool_t init()
- virtual TClass* IsA()
- virtual HTask* next(Int_t& errCode)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

D.1.25.1. Class Description

```
HTask

 This is the base class for a task (i.e. a transform data, take a decision
depending on some parameters ...). Examples of tasks are the HReconstructor
class which stands for a algorithm used to process event's data; the
HTaskSet class which corresponds to a set of generic tasks.

 The task is expected to be initialized using the init() function before the
first call to the next() function. This last method (next()) performs the
actual task and retrieves a pointer to the next task to be executed (or
NULL if there is no next task).

 The finalize() method does some clean-up and in general will be called in
the destructor. However it is provided as a separate function allowing the
posibility to reset the task (two consecutive calls to finalize() and init())
without destructing the object.

 The connectTask(aTask,n) function connects the task "aTask" as the next task
to be performed. "n" is an optional integer and its meaning can be defined
by each subclass.
```

- Autor: Manuel Sanchez

D.1.26. class HTaskSet

- Base classes: public HTask
- protected methods
  - virtual Bool_t connectTask(HTask*, Int_t n)
- public methods
  - HTaskSet HTaskSet()
  - HTaskSet HTaskSet(HTaskSet& ts)

- HTaskSet HTaskSet(Text_t* name, Text_t* title)
- virtual void ~HTaskSet()
- TClass* Class()
- void Clear()
- Bool_t connect(HTask*)
- Bool_t connect(HTask*, HTask*, Int_t n = 0)
- Bool_t connect(HTask*, Text_t* where, Int_t n = 0)
- Bool_t connect(Text_t* task, Text_t* where, Int_t n = 0)
- virtual Bool_t finalize()
- virtual Bool_t init()
- virtual TClass* IsA()
- virtual HTask* next(Int_t& errCode)
- HTask* operator()(Int_t& errCode)
- virtual void ShowMembers(TMemberInspector& insp, char* parent)
- virtual void Streamer(TBuffer& b)

▪ Data Members

- protected:
  - THashTable fTasks: *Tasks in this set.*
  - HTask* fNextTask:
  - HTask* fFirstTask:

D.1.26.1. Class Description

```
HTaskSet

 This HTask is in fact a set of tasks arbitrarily connected among them.

 The tasks are connected among the so when a task is finished a pointer to
the next task is retrieved; note that in general the tasks are not connected
in a linear fashion.
```

D.1.26.2. HTaskSet(Text_t name[],Text_t title[]) : HTask(name,title)

```
Constructor
```

D.1.26.3. HTaskSet(void)

```
Default constructor
```

D.1.26.4. HTaskSet(HTaskSet &ts)

Copy constructor.

D.1.26.5. ~HTaskSet(void)

Destructor.

D.1.26.6. Bool_t connect(HTask *task)

Connects "task" as the first task to be performed in the set.

D.1.26.7. Bool_t connect(HTask *task,HTask *where,Int_t n)

connects task "task" to the task "where" with parameter "n"

D.1.26.8. Bool_t connect(HTask *task,Text_t *where,Int_t n=0)

Connects task "task" to the task with name "where" using "n" as parameter.

D.1.26.9. Bool_t connect(Text_t task[],Text_t where[],Int_t n=0)

Connects the task named "task" to the one named "where" using "n" as paramter.

D.1.26.10. void Clear(void)

D.1.26.11. Bool_t init(void)

Calls the init function for each of the tasks in the task set.

D.1.26.12. Bool_t finalize(void)

Calls the finalize function for each of the tasks in the set

D.1.26.13. HTask* next(Int_t &)

Iterates throught the task set. When the iteration is finished it returns a pointer to the next task as set by connectTask()

D.1.26.14. Bool_t connectTask(HTask *task,Int_t)

```
Connects "task" as the next task to be performed; n is ignored.
```

- Autor: Manuel Sanchez

D.1.27. class Hades

- Base classes: public TObject
- public methods
  - Hades Hades()
  - Hades Hades(Hades&)
  - virtual void ~Hades()
  - void activateTree(TTree* tree)
  - virtual void Browse(TBrowser* b)
  - TClass* Class()
  - Int_t eventLoop(Int_t nEvents = kMaxInt, Int_t startEvent = 0)
  - HEvent*& getCurrentEvent()
  - TFile* getOutputFile()
  - HRuntimeDb* getRuntimeDb()
  - HSpectrometer* getSetup()
  - Int_t getSplitLevel()
  - HTaskSet* getTask()
  - HTree* getTree()
  - Bool_t init()
  - virtual TClass* IsA()
  - virtual Bool_t IsFolder()
  - Bool_t makeTree()
  - Int_t setAlgorithmLayout(Text_t* fileName)
  - Int_t setConfig(Text_t* fileName)
  - void setDataSource(HDataSource* dataS)
  - void setEvent(HEvent* ev)
  - Int_t setEventLayout(Text_t* fileName)
  - Bool_t setOutputFile(Text_t* name, Option_t* opt, Text_t* title, Int_t comp)
  - void setSplitLevel(Int_t splitLevel)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)

    - virtual void Streamer(TBuffer& b)

- Data Members
    - protected:
        - Bool_t fSplitLevel: *Indicates the split level (0,1,2)*
        - HDataSource* fDataSource: *! Data source where data are taken from*
        - HEvent* fCurrentEvent: *Event under reconstruction*
        - HTaskSet* fTask: *Task for each event.*
        - TFile* fOutputFile: *!File used to store the output tree*
        - HSpectrometer* setup: *! Spectrometer s setup.*
        - HTree* fTree: *Output tree*
        - HRuntimeDb* rtdb: *! Runtime database of reconstruction parameters.*

D.1.27.1. Class Description

```
Hades Begin_Html <a href="Hades_class.gif">(UML Diagram)</a> End_Html

  Hades is the main control class for the reconstruction program.

  There must be one and only one object of this class in the program. This
  object will be accesible anywhere in the reconstruction program or the
  external macros through the global pointer gHades.

  See the initialization section in the HOORUS manual for more information.
```

D.1.27.2. Hades(void)

```
Default constructor for Hades. It sets the global pointer gHades.
This constructor also creates one HDebug object if none exists.
```

D.1.27.3. ~Hades(void)

```
Closes the reconstruction program. If there are unsaved data, it saves them
When using Hades within a macro don't forget to call the destructor,
or data won't be saved.
```

D.1.27.4. Int_t setAlgorithmLayout(Text_t *fileName)

```
This method is used to build the reconstructor's graph. For that purpose
the macro in the file "fileName" is executed.

Returns 0 if successfull.
```

D.1.27.5. Int_t setEventLayout(Text_t *fileName)

This method is intended to build the event's structure. For that purpose the macro in the file "fileName" is executed.

Returns 0 if succesfull.

D.1.27.6. Int_t setConfig(Text_t *fileName)

This method is used to interpret a "configuration macro". Within this macro all the information to run the analisys is given. This is an alternative way of initialization without using setEventLayout(), setAlgorithmLayout()...

For more information on initialization see the initialization section in the HOORUS manual.

Within this macro the user is responsible of calling the Hades::init() function before the call to Hades::makeTree();

D.1.27.7. Bool_t init(void)

This function initializes Hades. This includes initialization of the reconstructors, the data source and the detectors.

```
if (!setup->init()) return kFALSE;          // what ??????????
```

D.1.27.8. void setEvent(HEvent *ev)

This function sets the event layout to that given by "ev". The user first builds the event layout by creating an HEvent object and then call this function; which sets the data member fCurrentEvent.

This function is tipically called from the macro invoked in setEventLayout()

D.1.27.9. void setDataSource(HDataSource *dataS)

Method used to establish the data source where data are taken from. For that purpose an object of a class derived from HDataSource is created and then the setDataSource() method is called giving a pointer to this object as a parameter

This function should be called typically from the macro invoked with setConfig()

D.1.27.10. void setSplitLevel(Int_t splitLevel)

```
Method used to control the shape of the output Root tree for the events.
Three split levels are supported:

  splitLevel=0 ---> An only branch is created for the event, which is
                    stored as a whole (i.e. the whole HEvent is stored as
                    one piece).

  splitLevel=1 ---> The partial events are stored as a whole, meanwhile the
                    top of the event class (tracks, header ...) is stored
creating one TBranch per data member

  splitLevel=2 ---> One branch per data member is created (with little
exceptions, see Root automatic split rules). However
the categories (see HCategory) still can decide how the
split is done (by creating their own branches). This
                    is the default value set by the Hades constructor
```

D.1.27.11. Int_t getSplitLevel(void)

```
Returns the current splitLevel (see setSplitLevel() )
```

D.1.27.12. Bool_t setOutputFile(Text_t *name, Option_t *opt, Text_t *title, Int_t comp)

```
Sets the output file, giving its name, compresion level...
For more information on the parameters see the constructor of TFile

  This method allocates the file indicated taking care of saving the
current file if any. If the file does not exist or the user wants it to be
overwritten, then opt="RECREATE"; if the file already exists and is to be
updated then opt="UPDATE"...
```

D.1.27.13. HTree* getTree(void)

```
Returns a pointer to the current output Root tree of events
```

D.1.27.14. void activateTree(TTree *tree)

```
 Sets the right branch address and branch status (=1) for all the branches
in tree which correspond to the branches which would eventually be
generated for the output Root tree of events if the function makeTree() is
```

```
called.

 This mechanism allows to only read those branches in "tree" which are
neccesary to fill an event as defined in setEventLayout()
```

D.1.27.15. Bool_t makeTree(void)

```
Creates an output tree from the information in the event structure
and according to the splitLevel (see setSplitLevel() )
```

D.1.27.16. Int_t eventLoop(Int_t nEvents, Int_t)

```
  Executes the event loop;

  For each new event:
  First, the current event is cleared.
  Second, a new event is read from the data source (see HDataSource)
  Third, the reconstruction of this event is launched (see HReconstructor)
  Fourth, if a tree was created (see makeTree() ) then it is filled.

  This function returns the number of events processed. A negative value
  corresponds to an error
```

D.1.27.17. Bool_t IsFolder(void)

```
Returns true. This tells the Root browser to show Hades as a folder
holding other objects
```

D.1.27.18. void Browse(TBrowser *b)

```
Used to browse the reconstructor's tree, a particular event or the
reconstructor's histograms...

 This function is called by Root when browsing gHades with the Root browser
```

D.1.27.19. void Streamer(TBuffer &R__b)

- Autor: Manuel Sanchez Manuel Sanchez

D.1.28. class HldSource

- Base classes: public HDataSource

- protected methods
  - void decodeHeader()
- public methods
  - virtual void ~HldSource()
  - void addUnpacker(HldUnpack* unpacker)
  - TClass* Class()
  - virtual EDsState getNextEvent()
  - Bool_t initUnpacker()
  - virtual TClass* IsA()
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - TList* fUnpackerList: *! List of the unpackers used to extract data*
    - HldEvt* fReadEvent: *! Buffer where the data is first read.*

D.1.28.1. Class Description

```
HLdSource

  Base class for a "data source" which delivery LMD data.
  The derived class HldFileSource reads the data from a file on disk and
  the derived class HldRemoteSource getrs the data directly from the
  Data Adquisition.

  The data in the file need to be unpacked. The user must specify the list
  of unpackers (see HldUnpack) to use, this way the user can select which
  parts of the data will be retrieved. That also give support to the case of
  LMD files not containing all the possible types of data (Rich data,
  Mdc data...)

  An example would be:
  HLmdFileSource *lmdSource=new HLmdFileSource();
  lmdSource->addUnpacker(new HRichUnpacker);

  This way you only get the Rich data in the files specified in the runtime
  database
```

D.1.28.2. ~HldSource(void)

```
Destructor for a LMD data source
```

D.1.28.3. void addUnpacker(HldUnpack *unpacker)

```
 adds an unpacker to the list of unpackers for a LMD source
```

D.1.28.4. Bool_t initUnpacker(void)

```
Calls the init() function for each unpacker.
```

D.1.28.5. void decodeHeader(void)

```
Decodes the event's header
```

- Autor: Manuel Sanchez

D.1.29. class HldUnpack

- Base classes: public TObject
- public methods
  - virtual void ~HldUnpack()
  - TClass* Class()
  - virtual Int_t execute()
  - HldSubEvt* *const getpSubEvt()
  - virtual int getSubEvtId()
  - virtual Bool_t init()
  - virtual TClass* IsA()
  - void setCategory(HCategory* aCat)
  - virtual void ShowMembers(TMemberInspector& insp, char* parent)
  - virtual void Streamer(TBuffer& b)
- Data Members
  - protected:
    - HldSubEvt* pSubEvt: *! Buffer where data are read from*
    - HCategory* pRawCat: *The category where data will be stored;*

D.1.29.1. Class Description

```
HldUnpack

  ABC for the different unpackers.

  The unpackers are used to unpack the data from Lmd files and put them
into the HEvent structure.

  The job of an unpacker starts when its execute function is called within
HLmdSource::getNextEvent(), in this function the unpacker reads data from
an HldEvt (corresponding to an event as stored by the data adquisition
system) and extracts the info to the category pRawCat.
```

D.1.29.2. ~HldUnpack(void)

```
Destructor
```

D.1.29.3. int execute(void)

```
Default execute function. It just gives some info about the readed subevent
 This function is overriden by derived classes.
```

D.1.29.4. void setCategory(HCategory *aCat)

```
Sets the category where the unpacked data go to.
```

- Autor: Walter Karig