Université Libre de Bruxelles
Faculté des Sciences appliquées

Année académique 2000-2001

Prototypage du Gestionnaire d'Evénements pour le système d'acquisition de données de l'expérience CMS auprès du collisionneur LHC du CERN

Directeur de Mémoire :
Prof. Esteban ZIMANYI
Co-Directrice :
Prof. Catherine VANDERVELDE

Travail de fin d'études présenté par Michaël Manalis en vue de l'obtention du grade d'Ingénieur Civil Informaticien

Remerciements

Qu'il me soit ici permis de remercier l'ensemble des personnes sans qui l'aboutissement de ce travail n'aurait pas été possible.

Le professeur Esteban Zimányi, pour ses commentaires toujours affûtés et son aide précieuse

La professeur Catherine Vandervelde, pour m'avoir ouvert les portes de la physique des hautes énergies et du CERN

Wim Beaumont, pour sa supervision attentive et ses conseils

Pascal Vanlaer, pour son infinie patience et les nombreuses heures passées à améliorer ce travail

et enfin, ma famille, pour ne pas avoir oublié mon existence durant ces longs mois d'absence

Table des matières

1 Le collisionneur LHC et l'expérience CMS 1

2 Les systèmes d'acquisition de données et de tri sélectif de l'expérience CMS 4

2.1 Le système d'acquisition de données 4

2.1.1 Le gestionnaire d'événements 7

3 Le gestionnaire d'événements en action 8

3.1 Réception des déclenchements de niveau primaire 8

3.2 Inhibition des déclenchements 9

3.3 Transfert des données des détecteurs 9

3.4 Attribution des événements aux unités de formation des événements 9

3.5 Contrôle de cohérence 10

3.6 Recyclage des identificateurs d'événements 10

3.7 Transfert de données, deux approches 10

3.7.1 Centralisation maximum 10

3.7.2 Centralisation minimum 11

4 XDAQ, une infrastructure de programmation distribuée pour les systèmes d'acquisition de données 12

4.1 L'architecture I2O 12

4.2 XDAQ, une implémentation basée sur I2O 14

4.3 Les modules de XDAQ 16

5 Le gestionnaire d'événements : description du code existant 17

5.1 Les fonctions membres spécifiques au protocole I2O 18
5.2 Les attributs du Readout Manager 18
5.3 Les fonctions membres du Readout Manager 19
5.4 Les attributs du Builder Manager 20
5.5 Les fonctions membres du Builder Manager 21
5.6 Les fonctions membres du First Level Trigger 22
5.7 Un séquence de fonctionnement type 23

6 Réalisation d'un gestionnaire d'événements pour le dispositif de test X5 26

6.1 Le dispositif de test en faisceau X5 26
6.2 Les spécifications du dispositif de test en faisceau X5 27
6.3 Un gestionnaire d'événements pour X5 27
6.3.1 Déclenchement de niveau primaire, description du code 28
6.3.2 Discussions sur l'implémentation 29
6.3.3 Gestion des erreurs 30
6.3.4 Améliorations possibles 30

7 Evaluation de performances 32

7.1 Conditions expérimentales 32
7.1.1 Modifications du code 32
7.1.2 Plateforme d'exécution 33
7.1.3 Code complémentaire 33
7.2 Résultats 34
7.3 Commentaires 35

8 Conclusion 36

A Glossaire 38

B Configuration XML pour JXDAQ 40

C Sources 42

C.1 FLT 42
C.1.1 i2oFLT.h 42
C.1.2 i2oFLT.cc 44
C.1.3 hwfltmessage.h 46
C.2 RM 47
C.2.1 i2oRM.h 47
C.2.2 i2oRM.cc 50
C.3 BM 54
C.3.1 i2oBM.h 54
C.3.2 i2oBM.cc 57
C.4 Test de rapidité en lecture 63
C.4.1 driverspeed.cc 63

Bibliographie **65**

Chapitre 1

Le collisionneur LHC et l'expérience CMS

Le CERN, le laboratoire européen pour la recherche en physique des particules, est le plus grand centre mondial de recherche dans ce domaine. Il a pour vocation l'étude des constituants de la matière et de leurs interactions.

Pour sonder le coeur de la matière, le principe utilisé est de projeter des particules à très haute énergie les unes contre les autres au moyen d'un accélérateur de particules appelé collisionneur. La structure de la matière est alors testée avec une résolution spatiale $\lambda = \frac{hc}{E}$, où E est l'énergie disponible dans le centre de masse de la collision. En enregistrant ce qui se passe lors de ces réactions au moyen de détecteurs de particules placés autour des points de collision, on retrouve la nature des particules formées ainsi que leurs propriétés physiques. La connaissance de ces données permet de démêler le processus microscopique de réaction et de connaître plus précisément la constitution et les propriétés des particules fondamentales, ainsi que d'en découvrir de nouvelles.

Si le principe de ces expérimentations peut sembler simple, la réalisation constitue l'un des plus grands défis scientifiques et technologiques de l'histoire. Le CERN joue ainsi le rôle de pionnier dans de nombreux domaines et participe à l'évolution de l'ensemble de nos technologies en constituant un important banc d'essai des technologies industrielles de demain. En effet, les outils du laboratoire, les accélérateurs et les détecteurs de particules, figurent parmi les instruments scientifiques les plus complexes du monde.

Les hautes énergies permettent d'atteindre des champs encore inexplorés en générant des particules toujours plus massives afin de compléter le portrait de famille des particules connues. A cette fin, le CERN entame actuellement la construction d'un collisionneur à protons, le LHC (Large Hadron

Collider), qui permettra de réaliser des collisions frontales de protons à une énergie de 14TeV (tera électron-volt) soit 1TeV pour les interactions entre quark-quark, gluon-gluon ou quark-gluon.

Plusieurs expériences et leurs détecteurs associés prendront place sur ce collisionneur. L'expérience CMS aura essentiellement deux objectifs : la découverte du boson de Higgs et l'exploration de nouveaux domaines d'énergie où pourraient se manifester des phénomènes qui ne cadrent pas avec le Modèle Standard, modèle qui peut décrire actuellement toutes les particules et interactions connues, exceptée la gravité.

Le boson de Higgs étant justement l'élément manquant du Modèle Standard, il permet d'expliquer l'origine de la masse des particules. La théorie ne prédisant pas la masse de cet objet, il convient d'explorer un large champ d'énergies pour tenter de l'observer et de faire apparaître ses propriétés.

Le détecteur associé à cette expérience est constitué d'un ensemble de couches cylindriques de détecteurs différents spécialisés suivant la nature des particules pouvant émerger des interactions. Chacun de ces modules constitue un système physique et électronique complexe. Au centre, le trajectographe a pour but de mesurer de façon non destructive la quantité de mouvement et la trajectoire des particules chargées. Ensuite sont disposés les calorimètres, qui absorbent toutes les particules sauf les muons et les neutrinos, et mesurent leur énergie. Enfin, les chambres à muons mesurent la quantité de mouvement des muons. Chacun de ces détecteurs comporte un grand nombre de canaux de lecture (près de 15 millions de voies pour l'ensemble).

Les paquets de protons des faisceaux du LHC se croisent dans le détecteur à une cadence de 40MHz. Environ une collision sur seulement 10^9 trahit des phénomènes physiques intéressants, nouveaux ou mal connus. De plus, les signaux des détecteurs représentent une quantité de données d'environ un méga byte (MB) par événement. Le traitement complet d'un événement nécessite un temps de calcul de l'ordre de la minute sur un processeur actuel de type Pentium III. Il est donc indispensable d'effectuer un tri en ligne des événements potentiellement intéressants pour éviter de mobiliser inutilement des ressources et décharger le système de flux de données trop importants.

Le cadre de ce travail est le système de prise de données et de tri en ligne de l'expérience CMS. En particulier, nous étudions le logiciel qui assure la coordination du transfert des données relatives à un même événement, depuis l'électronique de lecture des détecteurs jusqu'aux systèmes de tri en ligne. Ce logiciel porte le nom de gestionnaire d'événements, ou Event Manager (EVM[1]) en anglais.

[1] Un glossaire reprenant les acronymes utilisés se trouve en annexe, page 38

Ce travail se base sur la librairie XDAQ, développée au CERN pour implémenter le système de prise de données. Il consiste plus particulièrement à fournir une adaptation du gestionnaire d'événements spécifique à un dispositif de test de détecteurs dans un faisceaux de particules, le faisceau X5 au CERN.

Le système de prises de données du dispositif X5 est un banc d'essai idéal pour le système de prises de données de CMS et pour la librairie XDAQ.

Au chapitre 2 nous découvrirons le fonctionnement du système d'acquisition de données. Le chapitre 3 abordera les différentes fonctionnalités requises pour le gestionnaire d'événements. Le chapitre 4 s'attache à la description de la librairie XDAQ. Après quoi, le chapitre 5 décrit la structure et le fonctionnement du code du gestionnaire d'événements tel qu'il est compris dans XDAQ. La description des spécifications du dispositif X5 et de la réalisation du gestionnaire d'événements adapté suit au chapitre 6. Enfin, un évaluation des performances terminera le présent travail en chapitre 7.

Chapitre 2

Les systèmes d'acquisition de données et de tri sélectif de l'expérience CMS

Dans ce chapitre, nous décrirons le système d'acquisition de données de l'expérience CMS, ou Data AcQuisition (DAQ) en anglais, destiné à acheminer les données enregistrées par l'électronique de lecture des détecteurs vers les systèmes d'archivage et le système de tri sélectif des événements (ou trigger en anglais), dont le rôle est d'effectuer la sélection en ligne des événements potentiellement intéressants.

Pour comprendre l'organisation du système d'acquisition de données, il faut préciser les contraintes auxquels ce système doit faire face : le débit initial de données sortant du détecteur est estimé à 10^9 interactions par seconde, chaque interaction représentant un volume d'à peu près un méga byte de données. Ce débit doit être réduit à une centaine d'événements sélectionnés à chaque seconde pour le stockage et les analyses ultérieures. On a donc un facteur de tri en temps réel de 10^7 sur un débit initial de données de l'ordre du peta byte par seconde.

Ces quelques chiffres donnent la mesure du défi qui est posé par la réalisation de ce système, tant sur la largeur de la bande passante que sur la nécessaire rapidité d'exécution des critères de tri.

2.1 Le système d'acquisition de données

L'architecture du DAQ est représentée à la figure 2.1. Elle comprend l'électronique de lecture des détecteurs, ou Detector Dependent Units (DDU) en anglais, les mémoires tampons regroupant les données de 50 à 100 lec-

teurs, ou Readout Units (RU), le système d'interconnexion servant à transférer les données stockées dans les mémoires-tampons (Switch), les fermes de processeurs[1] destinées à regrouper les données relatives à un même événement, ou Builder Units (BU), et à effectuer une partie de la sélection en ligne, ou Filter Units (FU).

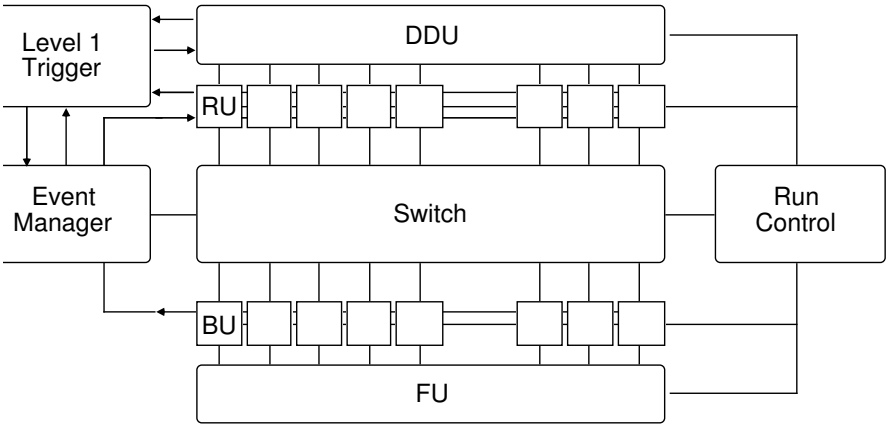

FIG. 2.1 – le DAQ

Un premier tri des données est réalisé au niveau de l'électronique de lecture des détecteurs. Lorsque un événement passe ce niveau de sélection, un signal de déclenchement primaire (Level 1 Trigger) est transmis aux détecteurs afin qu'ils transfèrent leurs signaux aux mémoires-tampons. Ce signal est également transmis au gestionnaire d'événements, qui gère le transfert de données depuis les mémoires-tampons jusqu'aux unités de formation des événements. L'interface entre l'utilisateur et le système de prise de données est réalisée par le système de contrôle de prises de données (ou Run Control).

Le gestionnaire d'événements fait plus précisément l'objet de ce travail et sera décrit en détail au chapitre 3. Voyons à présent plus en détail les éléments principaux du système de prise de données.

Les détecteurs et leur électronique de lecture

Les Detector Dependent Units (DDU) constituent l'électronique des différents parties du détecteur de particules. Chaque détecteur qui enregistre le passage d'une particule stocke de manière momentanée, dans un buffer (FiFo), les informations y attenant.

[1] On nomme ainsi un grand ensemble de PC interconnectés destiné à fournir une importante puissance de calcul à faible prix en exécutant du code distribué

Le système de déclenchement et de sélection primaire

Le déclencheur de niveau primaire (Level 1 trigger) est le module responsable du premier niveau de tri. C'est lui qui décide si les informations présentes dans les DDU et relatives à un même événement sont susceptibles de contenir des informations intéressantes, et donc d'être acheminées plus loin dans la chaîne de traitement, ou peuvent être directement supprimées des mémoires. Ce module, entièrement électronique pour des raisons de rapidité (maximum 3μs pour une décision), devrait réduire le flux de données d'événements à 100kHz pour le reste du système qui peut alors être constitué de systèmes informatiques commerciaux. Il implante au moyen de circuits intégrés spécifiques (Application Specific Integrated Circuits, ASICS) et de circuits logiques programmables (Fast Programmable Gate Array, FPGA), 128 algorithmes de sélection sur des critères de base pouvant indiquer un événement intéressant.

Les unités de lecture des détecteurs

Les Readout Units (RU) sont de modules contenant une mémoire à accès aléatoire (RAM). Quand un événement a reçu l'aval du déclencheur de niveau 1, les informations sont transférées des DDU vers les RU correspondants. Chaque RU regroupe les données de 50 à 100 DDU.

Le système d'interconnexion

Ce commutateur est destiné à connecter les 500 RU aux 500 BU. Il s'agira d'un switch actif devant fournir un débit de minimum 2Gb/s par ligne. La technologie utilisée n'est pas encore fixée mais des essais en ont montré la faisabilité. Des RU vers les BU, un réseau de marque Myrinet ou Gigabit Ethernet assurerait le transfert des données, tandis que, dans l'autre sens, un réseau de type Fast Ethernet serait suffisant au transfert des signaux de requêtes des BU vers les RU.

Les unités de formation des événements

Ces modules (Builder Units, BU) centralisent les signaux relatifs à un événement qui sont répartis sous forme de fragments dans les RU en les regroupant dans une mémoire RAM. Ils servent ensuite les unités de sélection (FU).

Les unités de filtrage des événements

Les Filter Units (FU) constituent l'interface avec l'installation de calcul et de stockage (plusieurs milliers de PC inter-connectés). C'est à ce niveau que tournent les programmes de reconstruction et de sélection qui permettent de garder ou de jeter un événement sur base de la détection de signatures physiques caractéristiques.

2.1.1 Le gestionnaire d'événements

Ce module a pour but de gérer le transfert des données des RU vers leur BU correspondant. Le présent travail a pour but la mise en place et l'adaptation de ce module à un dispositif de test en faisceau au CERN (X5). Une description plus complète est donnée au chapitre suivant.

Chapitre 3

Le gestionnaire d'événements en action

Le gestionnaire d'événements contrôle les flux de données DDU→RU et RU→BU. Il est responsable de la cohérence des données dans le DAQ : des fragments dans les RU jusqu'aux données réunifiées dans les BU, les événements simultanément traités doivent rester strictement indépendants et complets.

Il connaît à tout moment la disponibilité des BU, les événements en cours de traitement et ceux en attente. Il est averti des déclenchements de premier niveau et commande le transfert des fragments des DDU vers les RU.

Le présent chapitre reprend les spécifications du gestionnaire d'événements en décrivant dans un ordre chronologique la succession d'actions à prendre pour assurer la bonne marche du DAQ.

3.1 Réception des déclenchements de niveau primaire

Le gestionnaire d'événements maintient en interne une liste d'identificateurs d'événements qu'il attribue aux fragments d'événements. A chaque déclenchement transmis par le déclencheur de niveau 1, un numéro d'événement (EVT ID) est réservé. Il est ensuite associé aux données qui apparaissent en tête des piles FIFO des DDU lors du transfert de celles-ci vers les mémoires-tampons RU.

3.2 Inhibition des déclenchements

En situation de surcharge, lorsque l'occupation des RU dépasse un plafond prédéterminé, le gestionnaire d'événements a pour charge d'avertir le déclencheur de niveau 1 afin de cesser le flux de données jusqu'à désengorgement du système.

3.3 Transfert des données des détecteurs

Le gestionnaire d'événements est séparé en deux parties : un gestionnaire de lecture des données, le Readout Manager (RM) en anglais, et un gestionnaire de la formation des événements, le Builder Manager (BM), en charge des messages vers les RU et les BU.

Lorsque un déclenchement primaire a été reçu et qu'un identificateur lui a été associé, le gestionnaire d'événements diffuse un message (via le RM) informant les RU de la présence des fragments d'un événement intéressant dans les DDU et donnant l'identificateur associé qui a été choisi. Les RU ont alors la charge du transfert des données correspondantes dans leur mémoire RAM et de l'association de l'identificateur à ces données.

3.4 Attribution des événements aux unités de formation des événements

Les BU informent régulièrement le gestionnaire d'événements de leur disponibilité. Celui-ci décide en conséquence d'attribuer un événement à traiter à un BU. Le choix des unités de calcul doit se faire suivant une tournante. Lorsqu'un BU se voit attribuer le traitement d'un événement, une procédure de tri supplémentaire se met en place. Le BU peut rapatrier tout ou partie des fragments de l'événement (voir ci-dessous). Le tri de second niveau est effectué par le FU. Sur base du traitement de ces données, si l'événement passe ce niveau de tri, l'ensemble des fragments éventuellement non-encore rapatriés est transféré et l'événement reconstitué est envoyé pour stockage. Si le résultat du tri est négatif, un message est envoyé aux RU pour les en informer et provoquer l'effacement des fragments relatifs à cet événement.

Si les premières spécifications prévoyaient un transfert des fragments en plusieurs étapes, pour ne pas surcharger le système d'interconnexion, l'évolution des performances des switches laisse entrevoir la possibilité d'effectuer systématiquement une reconstitution des événements des RU vers les BU en une seule étape.

3.5 Contrôle de cohérence

Les déclenchements reçus par le gestionnaire d'événements comportent une information de numéro de croisement de faisceau. Il s'agit d'une référence fournie par l'accélérateur permettant d'identifier de manière univoque l'événement. Cette référence est aussi reprise dans le traitement des données jusqu'aux RU. Lorsque le gestionnaire d'événements transmet un EVT ID aux RU, il associe également cette information pour contrôle. Les DDU étant des FIFO et les déclenchements arrivant de manière séquentielle au gestionnaire d'événements, les fragments de chaque événement sont logiquement tous présents aux sommets des piles des DDU lorsque l'information de déclenchement correspondante arrive, relayée par le gestionnaire d'événements. Le numéro de croisement permet néanmoins de s'assurer qu'il n'y a pas eu de perte de séquentialité du premier niveau du DAQ (DDU et déclencheur de niveau primaire).

3.6 Recyclage des identificateurs d'événements

Dans tous les cas, le gestionnaire d'événements est informé de la fin du traitement d'un événement et son numéro peut alors être réintégré dans la liste des identificateurs disponibles pour un nouvel usage. Les EVT ID sont en nombre limité pour permettre de garder en permanence une trace du nombre d'événements simultanément présents dans le système. De plus, vu la quantité d'événements circulant dans le DAQ, il faudrait une codification sur un nombre trop important de bits pour une attribution unique d'identificateurs, ce qui nuirait à la rapidité du système.

3.7 Transfert de données, deux approches

Lorsqu'un événement réparti sur les RU a été attribué aux BU, deux approches sont possibles pour effectuer le transfert de données. Elles diffèrent par le degré de contrôle que réalise le gestionnaire d'événements.

3.7.1 Centralisation maximum

Les demandes de compléments de données des BU vers les RU sont envoyées à l'EVM qui les adresse ensuite vers les RU. Cette approche permet d'adopter une architecture d'interconnexion simple car à sens unique.

3.7.2 Centralisation minimum

Les BU adressent directement les RU pour recevoir des données après allocation d'un EVT ID. Cette approche évite de faire de l'EVM un goulot d'étranglement du système lorsque de nombreuses demandes simultanées des 500 BU surviendront. C'est cette option qui est privilégiée dans les implémentations existantes.

Chapitre 4

XDAQ, une infrastructure de programmation distribuée pour les systèmes d'acquisition de données

Afin de faciliter la programmation des modules du DAQ et d'uniformiser leur développement, une infrastructure de code distribué a été conçue au CERN. Dénommée XDAQ, il s'agit d'une implémentation basée sur le standard I2O (pour "Intelligent Input/Output" en anglais). XDAQ fournit une librairie de communication qui masque l'aspect distribué et multiplateforme, facilite l'intégration du code des différents modules en définissant les interfaces des classes C++ et permet la réutilisabilité. La librairie est conçue dans un souci de performance dicté par les contraintes du DAQ.

Le présent chapitre s'attache à la description de XDAQ et de son fonctionnement en commençant par une brève description du standard I2O.

4.1 L'architecture I2O

Les spécifications I2O ("Intelligent Input/Output"), dont la version 2.0 a été utilisée pour la création de XDAQ, définissent une architecture standard de gestion des entrées-sorties de haute performance propre à assurer d'importants flux de données dans un environnement en réseau.

Le choix de l'architecture I2O comme base pour le développement de l'environnement de programmation du système d'acquisition de données fut motivé par

– la nécessaire rapidité de fonctionnement d'un système faisant un usage intensif de communication de données
– l'aspect pratique d'une infrastructure permettant de gérer toutes les transactions entre modules de manière transparente via un système d'adressage uniformisé
– l'absence d'environnement de programmation d'objets distribués suffisamment performant
– la flexibilité imposée par un environnement changeant et multiplate-forme

L'idée conceptuelle de I2O consiste, premièrement, à dégager le processeur principal d'une plate-forme de la gestion des entrées/sorties en lui adjoignant un ou plusieurs processeurs dédiés à cet usage placés sur la carte d'extension. Ces processeurs, appelés "IOP", prennent également en charge le transfert de messages avec leurs pairs, que ce soit sur d'autres machines ou sur la même et ce sans intervention du processeur central. (fonctionnalités "Peer-to-peer")

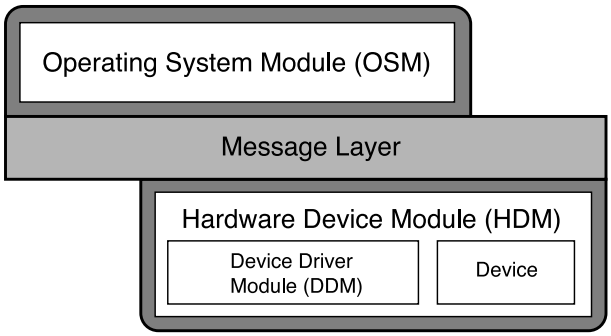

Fig. 4.1 – La conception modulaire d'un pilote I2O

Une autre avancée de l'architecture I2O consiste à séparer la conception des pilotes en deux parties : l'une liée au matériel, l'autre au système d'exploitation de la station recevant le code. Cette dualité permet d'éviter la réécriture d'un pilote neuf pour chaque association matériel-système possible puisqu'il suffit d'adapter les modules matériel et système appropriés. La figure 4.1 illustre cette séparation, le pilote est divisé en plusieurs parties :

– Une partie système (Operating System Module, OSM) spécifique au système d'exploitation mais totalement indépendant du matériel. Elle réside sur la station accueillant le processeur IOP et permet de présenter une interface standardisée aux applications voulant utiliser la carte d'extension I2O.
– Une partie matérielle (Hardware Device Module, HDM) chargée de traduire les messages reçus par l'OSM en actions sur la carte. L'HDM inclut du code chargé dynamiquement dans l'IOP afin de contrôler le

fonctionnement de la carte. Ce code porte le nom de Device Driver Module, DDM. Bien qu'embarqué sur la carte de communication et intimement lié à celle-ci, ce code ne peut la contrôler que par certaines fonctions clairement définies par le standard I2O.
– Une couche de communication standardisée (Messaging Layer) entre les modules système et matériel permet d'effectuer le lien entre n'importe quel OSM et HDM. Elle établit la communication entre ces modules en définissant un format de messages I2O standards reconnu par tout OSM et HDM.

Les DDM qui résident sur un IOP peuvent communiquer avec d'autres DDM sur d'autres IOP à tout moment sans interventions de l'OSM ou de la station-hôte (communication *Peer-to-peer*). Cette aptitude permet la conception d'unités de traitement de données communiquant entre eux à grande vitesse, via une infrastructure réseau, ce qui rend cette architecture particulièrement intéressante dans le cadre du DAQ. L'adressage des IOP s'effectue de manière uniformisée et transparente : chaque IOP se voit attribué un numéro unique lors de sa configuration, masquant ainsi à l'utilisateur les spécificités du réseau et des adresses physiques des cartes. Ceci rajoute encore à la flexibilité du standard I2O.

L'exécution du code des DDM I2O suit un schéma à déclenchement : l'envoi de messages constitue le déclencheur qui provoque l'exécution du code associé fourni par le programmeur sur l'unité visée. En conséquence, tout code de carte I2O doit implémenter une série de fonctions de contrôle et de configuration propres à I2O. Nous verrons quelques exemples lors de l'analyse du code du gestionnaire d'événements, au chapitre 5.

4.2 XDAQ, une implémentation basée sur I2O

XDAQ constitue un environnement de programmation distribuée basé sur I2O qui permet de masquer les détails d'implémentation du standard I20 tout en fournissant des fonctionnalités propres à des systèmes de code distribué.

Le standard I2O étant assez souple pour autoriser la définition de nouveaux messages et du code associé, une infrastructure de code distribué a pu être construite sur cette base. Le modèle est celui des systèmes à *Proxy* : les classes des objets destinés à être répartis sur un réseau sont définies en plusieurs classes. Une classe mère définit l'interface de toutes les méthodes de l'objet sans apporter d'implémentation. Deux classes héritent de cette interface : une classe *proxy* et une classe *adapter*. La classe *proxy* permet d'encapsuler les appels à l'objet, elle est instanciée dans tout noeud suscep-

tible d'avoir besoin de ce type d'objet. Lorsqu'un appel est lancé vers cet objet, c'est en fait le *proxy* qui est appelé, de manière transparente puisqu'il présente exactement la même interface que l'objet voulu. Le *proxy* se charge alors d'encapsuler l'appel et ses paramètres éventuels dans un message I2O. Ce message est ensuite envoyé vers l'hôte où réside le code de l'objet. Là, une instanciation de la classe *adapter* est activée par l'arrivée du message, elle a pour charge de déballer le message I2O et de faire appel à une troisième classe, la classe concrète de l'objet, en lui passant les paramètres. L'objet reçoit ainsi l'appel malgré la distance qui le sépare de l'objet appelant.

Les fonctions correspondant à chaque message peuvent être définies dynamiquement en téléchargeant le code compilé correspondant dans les modules I2O avant exécution.

Afin de permettre l'exécution de code sur des stations et non uniquement sur des cartes d'extensions I2O, XDAQ fournit une application prenant en charge les fonctionnalités *peer-to-peer*, le formatage, l'envoi et la réception de messages I2O. Cette application, appelée *executive*, s'intègre à l'architecture I2O existante et permet l'exécution de code DDM sur des stations commerciales (PC-Pentium ou SUN par exemple) non pourvues de périphériques I2O. L'*executive* doit être lancé sur toute station prenant part au système d'acquisition de données et peut ensuite se voir attribuer une configuration d'un ou plusieurs IOP émulés. On peut ainsi intégrer un ensemble hétérogène de systèmes I2O ou non.

Comme cadre de développement, XDAQ fournit les prototypes de classes de l'ensemble des modules impliqués dans le développement du DAQ. Chaque module (BU, EVM,...) est ainsi l'instanciation d'une classe XDAQ pour laquelle les programmeurs doivent fournir le code correspondant aux fonctionnalités requises.

Chaque classe hérite de classes constitutives de l'architecture I2O et de XDAQ et reçoit ainsi une série de fonctionnalités utiles. Comme le montre la figure 4.2, le module pris pour exemple (RM) hérite d'une interface (*i2oRMInterface*) définissant le prototype de toutes les classes et dont hérite également une classe *i2ORMProxy* pour l'exécution de code distribué, comme expliqué précédemment. La classe *i2oRMAdapter*, qui hérite de fonctionnalités I2O pour l'exécution par déclenchement sur une arrivée de message, est le correspondant du *proxy*, elle reçoit les messages emballés par le *proxy*, les déballe et fait appel à la fonction adéquate du RM. Apparaissent également des classes permettant d'ajouter des fonctionnalités SOAP (Simple Object Access Protocol) pour l'échange d'informations (de configuration ou d'état) au format XML (eXtensible Markup Language). La classe *i2oUtilAdapter* définit les fonctionnalités de configuration propres à I2O. Les classes *TriggerListener* sont spécifiques au RM.

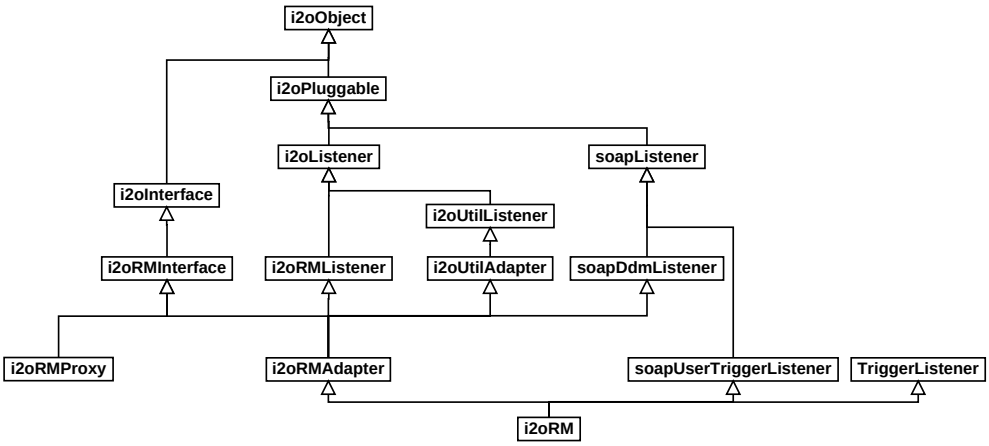

FIG. 4.2 – Diagramme de classes pour un module type

Les objets ayant reçu ces fonctionnalités sont alors compilés et le code objet peut être téléchargé dynamiquement sur l'*executive* voulu. La classe *i2oPluggable* permet alors d'enregistrer le code objet au sein de l'architecture I2O en lui fournissant un numéro d'identification et en l'informant de la présence d'autres modules. Le code est alors fonctionnel.

4.3 Les modules de XDAQ

Des modules génériques, bases de code remplissant les fonctionnalités minimales pour le développement du DAQ, sont compris avec XDAQ. Les différents modules présents correspondent aux éléments matériels du DAQ :

- Readout Unit (RU) : divisé en deux parties, RUI et RUO pour la gestion respectivement des entrées et des sorties
- Builder Unit (BU) : module en une seule classe
- Event Manager (EVM) : en deux parties, le BM et le RM, voir plus loin
- Filter Unit (FU) : encore à l'état embryonnaire (aucun algorithme de reconstruction ou de tri n'est implanté), également séparé en RUI et RUO

La configuration et le contrôle de chaque *executive* et des modules qui lui sont assignés se fait via une interface graphique en Java : JXDAQ. Le paramétrage des modules et des *executives* est contenu dans un fichier XML préalablement complété suivant les besoins de l'utilisateur.

Chapitre 5

Le gestionnaire d'événements : description du code existant

Le gestionnaire d'événements est séparé en deux parties (voir figure 5.1) : le "Readout Manager" (RM) qui prend en charge la gestion des communications avec les RU et le déclencheur de niveau primaire, et le "Builder Manager" (BM) qui s'occupe de la gestion des interactions avec les BU. Ces modules sont respectivement connectés au RCN ("Readout Control Network") reliant le gestionnaire d'événements aux RU et au BCN ("Builder Control Network") reliant le gestionnaire d'événements aux BU.

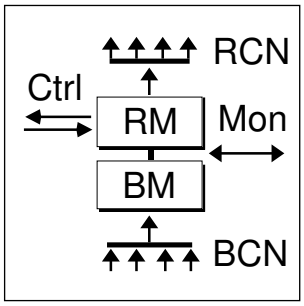

FIG. 5.1 – Le gestionnaire d'événements, ses deux modules RM et BM, les liens vers les RCN et BCN ainsi que les raccords vers les systèmes de contrôle et de monitoring (Ctrl et Mon)

A ces deux parties s'ajoute une classe particulière, étroitement liée au code du gestionnaire d'événements et destinée à gérer l'interface avec le déclencheur de niveau primaire, qui est un module matériel. C'est la classe

"flt" pour "First Level Trigger".

5.1 Les fonctions membres spécifiques au protocole I2O

Une série de fonctions sont imposées par les spécifications I2O et offrent des services de configuration et de contrôle des modules. Une brève description de ces fonctions suit[1] :

UtilParamsSet et UtilParamsGet
Mise en place ou demande de l'état des paramètres par l'utilisateur.

DdmSystemChange
Passage de l'état d'arrêt à l'état actif du module en chargeant les paramètres.

DdmSystemEnable
Idem, mais sans relecture des paramètres.

DdmSystemHalt
Spécifique au BM, cette fonction permet de placer ce module en mode d'arrêt, les appels éventuellement reçus par la suite sont ignorés.

DdmDeviceSuspend et DdmDeviceResume
Suspension temporaire et reprise de l'activité. Ces fonctions ne sont pas implémentées pour l'instant.

5.2 Les attributs du Readout Manager

flt_
Pointeur sur le code du first level trigger.

identifiersq_
Liste des identificateurs d'événements disponibles, les identificateurs sont des entiers de type long positif.

noEIDs_
Nombre total d'identificateurs disponibles, fixé par l'utilisateur dans le fichier XML de configuration.

counterTrigger_ et counterLost_
Compteurs de déclenchements reçus et éventuellement perdus.

builderDisable_ et enable_

[1]Les messages propres à I2O suivent une convention de dénomination du type (classe)(nom)(verbe) où "classe" est la catégorie de message, "nom" est l'abréviation ou le nom de l'objet et "verbe" l'action à effectuer.

Variable fixée par l'utilisateur et indiquant si le RM doit travailler indépendamment du BM. Permet d'effectuer des tests sur les RU sans avoir à se soucier de la présence de BU disponibles.

pending_
Indique si des événements sont en attente de traitement suite à une pénurie d'identificateurs.

mutex_
Variable d'exclusion mutuelle, protège certaines parties sensibles du code (comme l'attribution des identificateurs d'événements) en ne permettant à chaque instant l'accès qu'à une seule fonction.

5.3 Les fonctions membres du Readout Manager

SystemConfigure

void SystemConfigure()

Dans le code fourni par XDAQ, les déclenchements sont simulés de manière autonome par la classe "flt" (voir plus loin). Cette fonction a pour objectif d'activer le déclencheur en appelant sa fonction "BusyOff" et de placer le gestionnaire d'événements en état de réception en mettant sa variable "enable_" à vrai. C'est également ici qu'un vecteur de numéros d'événements est instancié.

trigger

void trigger(**TriggerInfo** *info*)

Cette fonction est appelée par "flt" ou par *UserTrigger* et provoque l'exécution de la suite de commandes nécessaires au traitement d'un déclenchement : un appel de la fonction *readout_and_remove* est lancé à l'ensemble des RU pour leur indiquer la présence d'un événement à lire, du numéro d'événement associé et du numéro de croisement faisceau qui a été reçu par le *TriggerInfo*. Ensuite, suivant l'état de la variable *builderDisable_*, le numéro d'événement est soit recyclé, soit envoyé au BM pour traitement supplémentaire.

UserTrigger

DOM_Node UserTrigger(**DOM_Node node**)

Cette fonction déclenchée par l'utilisateur directement depuis JXDAQ simule l'arrivée d'un déclenchement de niveau un. Elle est utilisée pour le

débogage et les tests. Le type *DOM_Node* est une structure en liste liée utilisée pour transmettre des paramètres I2O.

readyToRead

void readyToRead (**EventIdentifier*** *eid*)

Cette fonction permet le recyclage des identificateurs. Elle est appelée par le Builder Manager qui passe en même temps un identificateur d'événement dont le traitement a été achevé. Elle relance également le processus de déclenchement suivant l'état de la variable *pending_*.

idnotempty, idget et idput

Ces fonctions privées servent en interne à la gestion de la liste des identificateurs.

5.4 Les attributs du Builder Manager

triggersq_
Liste des numéros d'identificateurs passés par le RM et pouvant être attribués à un BU.

urgentq_
Liste de requêtes émanant des BU en attente de données. Les BU signifient leur disponibilité au BM. Ces requêtes sont alors stockées dans une structure *UrgentRequest* contenant un information de destination (l'indice du BU, *dest*) et une information du nombre d'événements qui peuvent y être traités (*neids*).

noEIDs
Nombre d'identificateurs gérés par le gestionnaire d'événements. Fixé par l'utilisateur.

eida_
Variable temporaire, pointeur sur identificateur, réfère les identificateurs entre la sortie de la liste d'attente et l'attribution à un BU.

triggerDisable_
Variable donnée par l'utilisateur. Permet au BM de fonctionner de manière indépendante du RM, en l'absence de déclenchements et pour le test des BU. Le BM génère alors en interne sa propre liste d'identificateurs et les recycle après usage.

alloc_

Variable d'exclusion mutuelle, permet de protéger les files d'attente d'identificateurs et de BU en empêchant leur accès simultané par les fonctions *commit* et *SystemConfigure*.

5.5 Les fonctions membres du Builder Manager

SystemConfigure

void SystemConfigure()

Cette fonction permet d'initialiser les files d'attente d'identificateurs et de BU. Et ce aussi bien dans le cas où ces listes n'étaient pas encore instanciées que dans le cas où elles existaient déjà (reconfiguration en cours de fonctionnement).

readyToSend

void readyToSend (**EventIdentifier*** *eid*)

Fonction appelée par le RM pour informer le BM de la présence d'un événement en attente de transfert dans le RU. Un identificateur est passé et rajouté dans la liste des identificateurs en attente de lecture *triggersq_*. *commit* est ensuite appelée.

allocate

void allocate (**EventProfile*** *eprofile*, **U16** *events*, **U16** *instance*)

Fonction appelée par un BU pour donner son identité et sa disponibilité afin de faire ainsi la demande d'un ou plusieurs identificateurs. Les données reçues sont stockées dans la liste *urgentq_* puis la fonction *commit* est appelée. La valeur *EventProfile* n'est pas utilisée mais devrait permettre de spécifier un type d'événement requis.

commit

inline void commit()

Cette fonction permet de regrouper l'attribution de plusieurs identificateurs à une BU en une étape. Le premier élément de la liste *urgentq_* est inspecté, pour connaître la disponibilité du BU qui lui correspond. Si suffisamment d'identificateurs en attente se trouvent dans la liste *triggersq_*,

ils sont attribués au BU correspondant (fonction *confirm* du BU, le pointeur *eida_* est utilisé pour le transfert), sinon le traitement est remis à un prochain appel, en attendant un nombre d'identificateurs suffisants.

clear

void clear (**EventIdentifier*** *eid*, **int** *neids*)

Appelée par les BU qui ont achevé leur traitement et acheminé l'événement correspondant en dehors du DAQ, cette fonction permet la réutilisation d'un identificateur qui est passé en paramètre. Suivant l'état de la variable *triggerDisable_*, l'identificateur reçu est soit recyclé localement (situation de test des BU) ou la fonction *readyToRead* du RM est appelée pour informer le RM de la réutilisation possible de l'identificateur (fonctionnement normal).

triggernotempty, triggerget, triggerput & triggernum

Fonctions privées de gestion de la file d'attente des identificateurs.

urgentput, urgentpop, urgentfront & urgentnotempty

Fonctions privées de gestion de la file d'attente des BU demandeurs.

retrieve et allocateRetrieve

void retrieve (**unsigned long*, unsigned long*, DestinationCookie***)
void allocateRetrieve (**unsigned long*, unsigned long*, Destination-Cookie***)

Ces deux interfaces, leurs fonctionnalités ne sont actuellement pas implémentées, sont présentes pour répondre à l'éventualité d'une approche de centralisation maximum sur le gestionnaire d'événements. Elle permettraient à un BU de faire la demande de fragments de données de RU en faisant passer la demande par le gestionnaire d'événements.

5.6 Les fonctions membres du First Level Trigger

Le code générique de cette classe permet de simuler des déclenchements en l'absence de tout système physique connecté. Ce code n'a que peu d'importance dans le sens où le but de ce travail est d'adapter le gestionnaire d'événements à un dispositif de test concret. Néanmoins, en voici une brève description.

addListener

void addListener(**TriggerListener** * listener)

Cette fonction a pour but de donner au code du déclencheur de niveau primaire la référence d'un module RM qu'il doit appeler lors d'un déclenchement. La classe *TriggerListener* dont hérite le code du RM permet d'encapsuler les spécificités.

BusyOff

void BusyOff()

Cette fonction permet de relancer l'horloge. Dans la version générique, les déclenchements sont simulés par une horloge.

timeElapsed

void timeElapsed(**Timer** * *timer*)

Dans la même idée de simulateur de déclenchement, c'est cette fonction qui est appelée par l'horloge à intervalles réguliers.

5.7 Un séquence de fonctionnement type

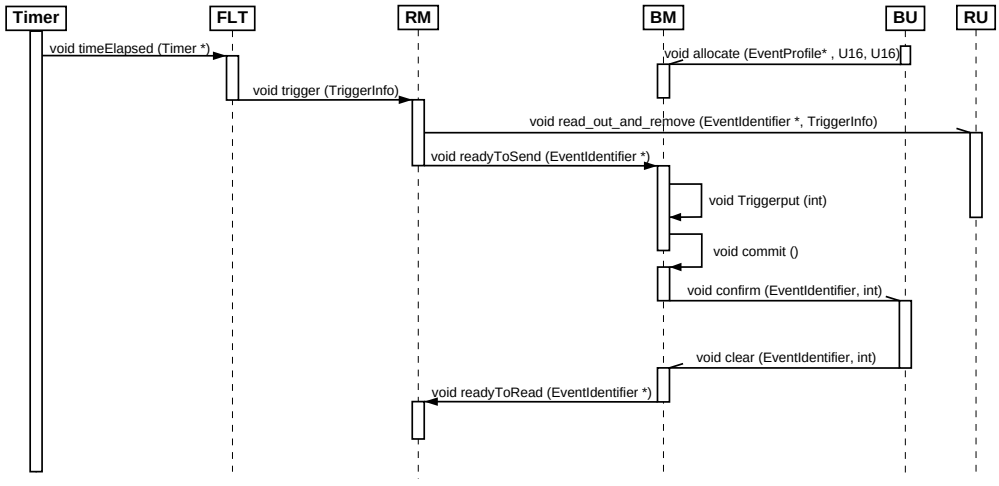

FIG. 5.2 – Diagramme de séquence des classes constitutives du gestionnaire d'événements

La figure 5.2 reprend un diagramme de séquence du fonctionnement normal du gestionnaire d'événements au sein du DAQ. La séquentialité des appels est la suivante.

- Le Timer arrive périodiquement à échéance et fait appel à la fonction *timeElapsed* de la classe FLT.
- Cette fonction a pour effet de provoquer un déclenchement du gestionnaire d'événements en appelant la fonction *trigger* du RM. Une information concernant le numéro de croisement de faisceau est passée en même temps.
- Le RM qui est en charge de la gestion des RU leur envoie alors un message pour les autoriser à transférer dans leur mémoire les données relatives à l'événement accepté par le déclencheur de niveau primaire. Ce message est asynchrone car il transite via la couche I2O, le RM peut reprendre son fonctionnement dès le départ du message, sans attendre son arrivée ou le retour d'une réponse. Un identificateur d'événement, extrait de la liste des identificateurs disponibles, est passé lors de cet appel. Il revient aux RU d'associer cet identificateur aux fragments d'événements qu'ils reçoivent afin de permettre l'identification de toutes les données en transit dans le DAQ. Le numéro de croisement de faisceau est également transmis pour vérification de la synchronisation des différentes parties du DAQ avec le faisceau.
- Le RM transmet ensuite l'identificateur utilisé au BM pour l'informer d'un événement en attente de lecture. Ceci est fait par l'envoi d'un message *readyToSend* qui contient en paramètre le numéro d'identificateur.
- Le BM prend alors en charge la suite de opérations : l'identificateur reçu est placé dans sa liste des événements en attente de lecture par l'appel de la fonction privée *Triggerput*. Puis un appel à *commit* est lancé. Cette fonction centralise la gestion des attributions d'événements aux BU qui en ont fait la demande.
- Cette demande a été préalablement formulée par l'appel de la fonction *allocate* avec les passages d'un identificateur propre de destination et du nombre d'événements pouvant être reçus.
- Prenons la situation où un BU est en attente d'événements. Dans ce cas, dès que le nombre d'événements demandés est réuni, la fonction *commit* attribue ces événements au BU. Cette opération est effectuée par l'appel de la fonction *confirm* du BU avec le passage en paramètre d'une liste d'identificateurs et d'un entier contenant le nombre de ces identificateurs. Cet appel est également asynchrone.
- Lorsqu'un événement a été traité par un BU (transmis pour stockage ou détruit), l'identificateur peut être réutilisé. Les BU informent le BM de la fin de traitement d'un ou plusieurs événements en faisant appel à la fonction *clear* et en transmettant une liste d'identificateurs

libérés ainsi que leur nombre.
- Le BM retransmet ces informations au RM. Pour ce faire, il effectue autant d'appels à la fonction *readyToRead* du BM qu'il y a d'identificateurs libérés. A chaque appel est transmis un numéro d'identificateur en paramètre.
- Le RM récupère ainsi des numéros d'identificateurs qui peuvent à nouveau servir. La boucle est bouclée.

Chapitre 6

Réalisation d'un gestionnaire d'événements pour le dispositif de test X5

L'objectif final du présent travail est la création d'un gestionnaire d'événements fonctionnel répondant aux spécifications et intégré à l'architecture XDAQ pour le système de test en faisceau X5 du CERN.

Le présent chapitre décrit les spécificités de X5, le code du gestionnaire d'événements réalisé, son fonctionnement et son intégration dans XDAQ.

6.1 Le dispositif de test en faisceau X5

Pour un projet aussi complexe et avant-gardiste que l'expérience CMS, il convient de contrôler et de valider régulièrement en tout point les choix technologiques. Chaque partie du détecteur, que ce soit un module physique, informatique ou logiciel doit ainsi faire l'objet de tests destinés à vérifier l'adéquation des solutions adoptées aux prérequis.

Le CERN dispose à cet effet d'une série de faisceaux de particules secondaires utilisés pour les tests et calibrations des détecteurs et des systèmes d'acquisitions de données. X5 est le nom d'un de ces faisceaux. Il est plus particulièrement destiné au test de modules électroniques comme le déclencheur de niveau primaire et les RU.

6.2 Les spécifications du dispositif de test en faisceau X5

La création d'un déclencheur de niveau primaire est en cours en ce moment pour les essais sur X5. Ce déclencheur devrait prendre la forme, en fin de chaîne, d'une carte de type PCI prenant place dans une station accueillant le code du gestionnaire d'événements. Cette carte ne répond pas aux spécifications I2O dans le sens où elle n'accueille pas d'IOP.

La sortie du déclencheur est la suivante : une pile stocke les informations des déclenchements acceptés au terme de la sélection de premier niveau. Cette pile comporte 256 entrées. Chaque entrée contient des informations sur le déclenchement, dont un numéro de croisement du faisceau pour le contrôle de la cohérence dans le DAQ et un numéro indiquant le nombre de déclenchements encore en attente dans la pile.

Le gestionnaire d'événements doit lire les informations présentes dans cette pile le plus rapidement possible et lancer le traitement des événements correspondant dans le DAQ. Pour minimiser le nombre de requêtes PCI dans la machine hôte et ainsi augmenter la rapidité du traitement des événements, il convient de lire en une étape l'ensemble des déclenchements présents dans la pile dans le cas où celle-ci en contient plus d'un.

Le déclencheur de niveau primaire n'étant pas encore finalisé, un pilote a été écrit pour simuler ce module en redirigeant les requêtes de lecture sur le port parallèle du PC utilisé pour le développement du code du gestionnaire d'événements. Le pilote retourne plusieurs informations à chaque déclenchement simulé reçu via le port parallèle :

- un compteur d'événements ;
- un numéro de croisement de faisceau ;
- deux indications du temps, l'un en secondes, l'autre en micro-secondes ;
- un nombre indiquant la quantité de déclenchements en attente de lecture dans la file ;
- un numéro d'interruption ;
- le nombre d'interruptions lancées.

6.3 Un gestionnaire d'événements pour X5

Pour adapter un gestionnaire d'événements répondant aux spécifications de XDAQ, l'organisation des classes du module générique a été conservée. Les classes des BM et RM ont été conservées tandis que la classe "FLT" a été entièrement réécrite pour s'adapter au déclencheur de niveau primaire de X5. En effet, le code du FLT fourni avec XDAQ n'est destiné qu'aux tests, il ne

fait appel à aucun dispositif matériel pour la réception des déclenchements mais uniquement à l'activation par une horloge arrivant périodiquement à échéance.

6.3.1 Déclenchement de niveau primaire, description du code

La classe FLT[1] permet de masquer au gestionnaire d'événements les spécificités concernant la liaison avec le système (matériel) de déclenchement primaire. Le but de cette classe est de lire les informations d'un déclenchement lorsqu'il survient, le plus rapidement possible, pour ensuite relayer ces informations en un appel à la fonction *trigger* du RM.

Devant être en permanence à l'écoute de déclencheur, la classe FLT implémente ses fonctionnalités sous forme d'une tâche. Elle fonctionne ainsi sur une branche d'exécution indépendante du reste du DAQ. Cette contrainte est dictée par le fait que le déclencheur n'est pas de type I2O, auquel cas, l'arrivée d'un déclenchement aurait pu être associée à un message personnalisé de type I2O provoquant l'exécution du code correspondant.

Le constructeur se charge de l'activation de la tâche en appelant la fonction *activate* héritée des fonctionnalités d'une classe *Task*.

La fonction *addListener* est inchangée, elle reçoit en paramètre un pointeur de type *TriggerListener*, il s'agit d'une référence au RM. De cette manière, le FLT peut directement appeler le RM. Il s'agit ici d'appel tout à fait conventionnel et direct, cet appel ne fait pas usage de fonctionnalités I2O.

La fonction *svc* est héritée de la classe *Task*. C'est cette fonction qui est appelée lorsque la tâche est activée. Elle contient une boucle infinie qui englobe les opérations de lecture des déclenchements. Ainsi, la tâche est continuellement à l'écoute des déclenchements surgissant sur la carte via le pilote. La lecture suit plusieurs étapes :

- un premier déclenchement est lu et est relayé vers le RM avec le numéro de croisement faisceau.
- le champ du nombre de déclenchements en attente est inspecté, si le nombre indiqué est un, il n'y avait qu'un seul déclenchement en attente, la boucle est reprise au début
- si le résultat est supérieur à un, on a plus d'un déclenchement dans la file au moment de la première lecture, une lecture de tous les déclenchements en attente est alors effectuée en une fois, dans une table de la taille de la file d'attente, une boucle est ensuite initiée pour envoyer autant d'appels à la fonction *trigger* du RM que de

[1] Les codes sources se trouvent en annexe C, page 42.

déclenchements lus.

Pour effectuer les lectures des déclenchements, une structure correspondant au format des données fournies par le pilote est déclaré. Les déclenchements sont lus en exécutant une instruction de lecture à laquelle sont passés la référence du port de lecture préalablement initialisé et un pointeur vers une variable au format des déclenchements.

Une fonction supplémentaire permet d'afficher à l'écran l'ensemble des champs d'un déclenchement reçu, à titre d'information pour un débogage par exemple.

6.3.2 Discussions sur l'implémentation

Les lectures sont de type bloquantes. Lors de l'arrivée d'un déclenchement en file d'attente de la carte du déclencheur de niveau primaire, une interruption est lancée à destination du pilote de la carte. Le pilote est alors activé et le déclenchement est envoyé au FLT via l'instruction *read*. L'instruction *read* attend indéfiniment, jusqu'à l'arrivée d'un déclenchement sur la carte.

Des réserves initiales existaient quant à l'usage de ce type de lecture, qui bloque l'exécution du programme tant qu'il n'y a pas de données à lire. Elles sont levées par le fait que le FLT est une tâche, s'exécutant parallèlement aux autres modules du gestionnaire d'événements. Le FLT peut donc être bloqué sans que cela stoppe le fonctionnement des autres parties du code. Ceci évite également de mettre en place un mécanisme plus complexe de lecture non bloquante.

Une autre question concernait la synchronisation du FLT lors des appels vers le RM. Il importe que le gestionnaire d'événements donne les garanties qu'aucun déclenchement ne puisse être perdu lors de son traitement. Dans la situation présente, le FLT étant une tâche séparée, peut-on avoir des appels vers le RM alors que celui-ci n'est pas en situation d'y répondre (le traitement de l'appel précédent n'étant pas achevé par exemple) ? La solution est apportée par la nature du lien entre FLT et RM : étant donné qu'il s'agit d'un appel de fonction tout à fait classique entre objets, sans le découplage qui existe entre deux objets séparés par un système *proxy-adaptateur*, les appels sont synchrones. La tâche ne reprend donc la main que lorsque l'appel à la fonction *trigger* est terminé, à savoir lorsque le RM rend la main. Ceci exclut toute perte d'information.

L'usage d'un boucle infinie laissait supposer un problème potentiel pour terminer l'exécution du code de la tâche. Les spécifications de la classe *Task* assurent néanmoins un mécanisme d'arrêt de la tâche lorsque le pointeur sur l'objet qui hérite de *Task* est supprimé. C'est le cas : on peut le voir clairement si on remonte dans les héritages à la classe SO (*Shared Object*)

qui crée les instances des FLT, RM et BM et les détruit en fin d'utilisation, c'est à dire lors de l'arrêt de XDAQ.

6.3.3 Gestion des erreurs

La classe FLT inclut quelques mécanismes de gestion des erreurs et de contrôle de cohérence :

- si l'ouverture du port de communication échoue, la tâche termine son exécution. Le gestionnaire d'événements n'est pas correctement chargé et il convient de le relancer ;
- toute lecture est accompagnée d'un contrôle de la taille des données recueillies pour le ou les déclenchements. Plusieurs cas d'erreurs sont possibles : les données réceptionnées sont incomplètes ou le chiffre du nombre de déclenchements en attente peut être erroné. Dans le premier cas, aucun appel n'est fait au RM. Dans le second, le nombre de déclenchements lus en une fois est réévalué à partir de la taille des données effectivement reçues ;
- le nombre de déclenchements en attente est également comparé à la taille de la file d'attente de la carte afin d'éviter toute lecture incohérente.

Ces choix sont forcément partiaux, ils conviennent bien au traitement des données issues du pilote de simulation (où les données sont toujours complètes mais le nombre de déclenchements en attente souvent erroné) mais ils ne rencontrent pas toutes les garanties requises de cohérence des données.

Les spécifications finales de la carte d'acquisition devraient lever certaines incertitudes :

- le nombre des déclenchements en attente peut-il être erroné ? Sinon, un contrôle peut être supprimé ;
- les données incomplètes doivent-elles être traitées comme un déclenchement mal transmis ou simplement ignorées ? Dans le premier cas, il conviendrait d'ajouter certaines informations au *TriggerInfo* par exemple, pour préserver l'événement tout en prévenant du risque de données faussées.

6.3.4 Améliorations possibles

En l'absence de spécifications précises de déclencheur de niveau primaire, le gestionnaire d'événements ainsi constitué n'inclut pas de mécanisme d'arrêt en cas de surcharge. Il est spécifié dans les prérequis du DAQ de l'expérience CMS qu'en cas de surcharge, lorsque le traitement simultané d'événements approche son maximum, une information doit pouvoir être envoyée au déclencheur

de niveau primaire pour lui demander un arrêt des déclenchements.

Chapitre 7

Evaluation de performances

Pour donner une idée de la rapidité des acquisitions de données du gestionnaire d'événements conçu, il convient de mettre en place des systèmes de test. La fréquence finale de traitement d'événements par le gestionnaire d'événements doit atteindre 100 kHz suivant les prérequis. Ce chapitre décrit les conditions expérimentales et les résultats obtenus sur des tests du gestionnaire d'événements.

7.1 Conditions expérimentales

7.1.1 Modifications du code

Pour permettre le test de la rapidité du gestionnaire d'événements seul, le code a été modifié afin d'en retirer les accès aux modules extérieurs (en l'occurence, RU et BU). Des macros de précompilation permettent de passer facilement du code normal au mode de test, moyennant une recompilation du gestionnaire d'événements.

Dans ce mode, les appels aux RU sont supprimés et les appels aux BU sont redirigés directement vers les fonctions *clear* et *allocate*, qui sont les fonctions appelées normalement par un BU en fin de traitement d'un événement. Ainsi, le gestionnaire d'événements fonctionne de manière autonome. La figure 7.1 illustre la séquence des appels suivant un déclenchement dans les cas d'un exécution en test de vitesse.

La fonction *gettimeofday (timeval*, NULL)* de la librairie *time.h* permet de récupérer le temps après l'exécution d'un nombre d'événements donné. La durée de traitement ainsi évaluée est ensuite affichée en secondes et microsecondes. Les sorties du RM et du BM sont toutes deux évaluées par ce moyen. L'évaluation concerne le nombre d'appels à la fonction *read_out_and_remove*

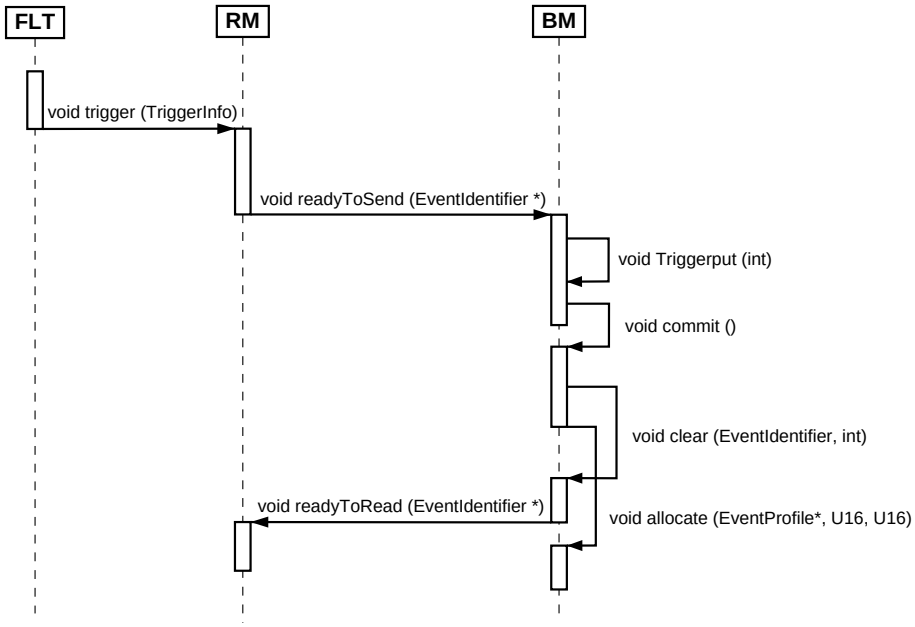

FIG. 7.1 – Diagramme de séquence du code modifié

pour le RM et les appels à le fonction *confirm* pour le BM.

7.1.2 Plateforme d'exécution

Les tests ont été effectués sur un PC équipé d'un processeur Pentium II cadencé à 400 MHz fonctionnant sous Linux (RedHat 6.2, kernel 2.2.14-5.0). La configuration complète incluait trois stations identiques reliées par un réseau Ethernet 10Gb en TCP/IP.

L'annexe B, page 40, donne un exemple de fichier XML de configuration en même temps que les paramètres pris pour les différents modules lors des tests.

7.1.3 Code complémentaire

Un programme complémentaire, écrit en C++ et utilisant les mêmes boucles que les procédures de test des BM et RM, effectue une lecture brute de déclenchement sur le pilote. Ce programme permet d'évaluer le temps de réponse de la simulation de déclenchements par le pilote sur port parallèle.

7.2 Résultats

L'image 7.2 présente une trace d'exécution du gestionnaire d'événements en test pour 17 paquets de 100 000 déclenchements. Les déclenchements sont comptés par 100 000 pour avoir un point de repère : en effet, selon les spécifications, le gestionnaire d'événements doit pouvoir gérer une fréquence de 100 000 déclenchements par seconde.

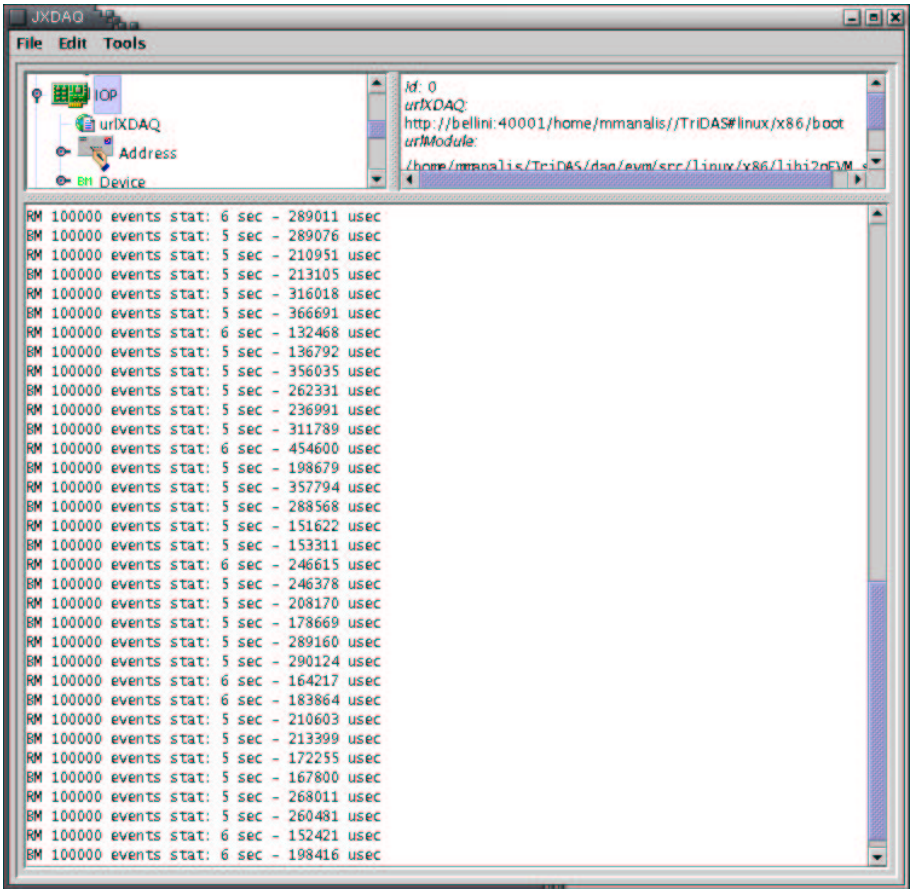

FIG. 7.2 – Trace d'exécution du gestionnaire d'événements en test dans l'interface JXDAQ

Les moyennes des temps d'exécution sont de 5,6 ± 0,5 secondes pour le RM et de 5,3 ± 0,3 pour le BM. Pour des mêmes jeux de 100 000 événements, le temps d'exécution moyen du programme comparatif de lecture seule est de 3,5 ± 0,4 secondes.

On obtient donc, en retranchant le temps de lecture, respectivement pour le RM et le BM 2,1 et 1,8 secondes. Soit un taux de traitement des

déclenchements de 47,6 kHz pour le RM et 55,6 kHz pour le BM.

7.3 Commentaires

Les ordre de grandeur atteints sont insuffisants puisqu'il faudrait une fréquence de traitement des déclenchements de 100 kHz.

Il convient de tempérer ces résultats par plusieurs facteurs d'erreurs :
- ces tests ne tiennent pas compte du temps d'emballage et d'envoi de messages I2O vers les RU et BU, ces messages n'étant pas émis à cause des modifications du code ;
- l'exécution se fait sur une machine linux relativement âgée et qui n'est pas configurée de manière optimale pour ce type d'opérations ;
- le port parallèle dispose d'une bande passante bien plus faible que celle d'un bus PCI.

Les résultats restent donc positifs, dans le sens où on peut espérer une grande réduction du temps pris pour les lectures avec une carte de déclenchement en électronique rapide sur bus PCI. En conséquence, les fréquences de lecture et de traitement des déclenchements devraient être facilement améliorés.

Chapitre 8

Conclusion

Ce travail nous a mené dans le monde de l'informatique appliquée à la recherche fondamentale en physique des particules. Plus précisément dans le cadre de l'expérience CMS auprès du futur accélérateur LHC du CERN. Nous avons vu comment se conçoit un système d'acquisitions de données pour un détecteur de particules. Avec la double contrainte de réaliser un système capable de traiter des flots de données immenses tout en assurant une fiabilité maximale.

L'approche empruntée par les concepteurs de ces systèmes d'acquisition relève d'un concept en apparence simple : multiplier les niveaux de tri pour réduire le nombre de données et organiser un système d'interconnexion permettant de rassembler les fragments issus des différents détecteurs vers des unités de filtrage et de stockage. Pour assurer la cohérence des données transitant dans ce système, le DAQ, le gestionnaire d'événements alloue des identificateurs et assigne les événements aux unités de traitement.

Car même dans l'informatique distribuée, on a encore du mal à se passer de chef d'orchestre, c'est le rôle du gestionnaire d'événements de s'assurer que, de la myriade de fragments de données jusqu'à l'événement reconstitué et filtré, les données deviennent information, ajoutant par là une dimension rigoureuse de classement, d'identification et de cohérence. Tout ceci afin qu'un jour, peut-être, l'information devienne savoir.

Mais le travail des chercheurs ne s'arrête pas à la conception des systèmes physiques. Il faut une structure logicielle cohérente et solide pour rencontrer toutes les contraintes inhérentes à l'acquisition de données. XDAQ, une librairie originale, développée à partir de l'architecture I2O, offre toutes les qualités requises. Elle offre un cadre pour le développement par objets distribués, masque les spécificités physiques des systèmes et optimise le traitement et le transport de l'information.

Ce travail consistait à développer le gestionnaire d'événements pour le système de test en faisceau X5, tout en respectant les principes de XDAQ. Pour cela, un premier travail de documentation a du être fait afin d'appréhender dans ses détails le fonctionnement de l'infrastructure XDAQ et du système d'acquisition auquel elle est intimement liée. Il fallait ensuite partir à la rencontre des spécifications de X5 pour enfin proposer une implémentation complète et fonctionnelle.

Finalement, les essais ont montré la validité de l'approche proposée, qui pour être avant-gardiste, n'en est pas moins réaliste. Les avancées technologiques devraient rendre bientôt possible le fonctionnement d'un gestionnaire d'événements répondant aux spécifications de CMS. Ce sera alors le début d'une nouvelle expérience.

Annexe A

Glossaire

- BCN : *Builder Control Network*
  Réseau reliant le BM aux BU.
- BM : *Builder Manager*
  Partie du gestionnaire d'événements en charge de la gestion de l'attribution d'événements aux BU.
- BU : *Builder Unit*
  Unités de reconstruction des événements, au nombre de 500, rassemblent les fragments de données dans une mémoire RAM.
- CERN : *Laboratoire européen pour la physique des particules*
- CMS : *Compact Muon Solenoid*
  Détecteur de particules prenant place dans le LHC pour l'étude du boson de Higgs et du modèle standard.
- DAQ : *Data AcQuisition*
  Système de tri et de rassemblement des données brutes extraites du détecteur de particules.
- DDM : *Device Driver Module*
  Logiciel chargé sur un périphérique I2O et destiné à le contrôler.
- DDU : *Detector Dependent Unit*
  Module électronique accueillant une partie des données du détecteur dans une pile FIFO à destination du DAQ.
- EVM : *Event Manager*
  Le gestionnaire d'événements est en charge de l'identification et de la répartition des données des événements dans le DAQ.
- EVT ID ou EID : *Event ID*
  Identificateur d'événement.
- FLT : *First Level Trigger*
  Déclencheur de niveau primaire, effectue le premier niveau de tri sur les données de CMS.

– FU : *Filter Unit*
  Unité de filtrage, reconstruit et trie les événements présents dans le BU associé.
– HDM : *Hardware Device Module*
  Partie spécifique au matériel dirigé dans un pilote I2O.
– I2O : *Intelligent I/O*
  Spécification de périphériques intelligents à pilote modulaire.
– IOP : *I/O Processor*
  Processeur de traitement des entrées-sorties sur un périphérique I2O.
– JXDAQ : *Java XDAQ*
  Interface Java pour la configuration et le contrôle de XDAQ.
– LHC : *Large Hadron Collider*
  Accélérateur en cours de construction au CERN pour la collision de protons à 14TeV.
– OSM : *Operating System Module*
  Partie spécifique au système d'exploitation dans une pilote I2O.
– RCN : *Readout Control Network*
  Réseaux reliant le RM au RU.
– RM : *Readout Manager*
  Partie du gestionnaire d'événements en charge de la gestion de identificateurs d'événements et de la communication avec les RU.
– RU : *Readout Unit*
  Unités de lecture des données, au nombre de 500, rendent les fragments d'événements disponibles dans une mémoire RAM pour le reste du DAQ.
– XDAQ : *Cross DAQ*
  Librairie de programmation répartie pour les modules du DAQ. Basée sur I2O.

Annexe B

Configuration XML pour JXDAQ

La configuration d'une infrastructure XDAQ peut être spécifiée dans un fichier XML lisible par JXDAQ. Ce fichier comporte le paramétrage et les indications pour tout module prenant part dans une simulation de système d'acquisition de données. A titre d'exemple, une description du paramétrage d'un gestionnaire d'événements suit, les autres modules sont configurés suivant le même canevas.

Premièrement il convient de définir un IOP comme cadre à un ou plusieurs modules, plusieurs informations doivent être fournies : un nom de station d'accueil, un numéro de port de communication et une référence à l'*exécutive* qui doit être accessible (sur un disque réseau, par exemple) depuis toutes les stations en fonctionnement.

```
<IOP id = "0">
  <urlXDAQ>http://bellini:40001/home/mmanalis//TriDAS#linux/
                                                x86/boot</urlXDAQ>
  <Address type="TCP">
    <IP>bellini</IP>
    <Port>50000</Port>
  </Address>
```

Ensuite peuvent être définis les modules actifs du DAQ. Chaque module se voit attribuer dès à présent une adresse unique sous forme d'un numéro d'identification. Ce numéro sera utilisé par le système I2O pour l'adressage de message à destination de ce module. Un numéro d'instance permet d'indiquer la présence de plusieurs modules de même nature (utile pour les RU et BU). La nature du protocole réseau est précisée, puis une liste de paramètres permet ensuite de préciser l'état voulu pour le module

en question. Dans l'exemple qui suit il s'agit d'un BM pour lequel doivent être spécifiés le nombre d'identificateurs d'événements (*noEIDs*) disponibles pour le DAQ et l'usage ou non de ce module en cas d'utilisation pour des tests (*triggerDisable*). Le choix d'un millier d'identificateurs disponibles permet de s'assurer d'une certaine latitude en fonctionnement, ce qui évite de ce retrouver à cours d'indentificateurs lors de pointes de déclenchements.

```
<Device Class="BM">
  <TargetAddr>9</TargetAddr>
  <Instance>0</Instance>
  <Transport>TCP</Transport>
  <Params>
     <triggerDisable>0</triggerDisable>
     <noEIDs>1000</noEIDs>
  </Params>
</Device>
```

Le BM est configuré en même temps et de la même manière, à ceci près que l'option *triggerDisable* est remplacée par *builderDisable*. Une référence à l'exécutable à télécharger est enfin passée. Ce qui termine la configuration du gestionnaire d'événements.

```
  <urlModule>/home/mmanalis/TriDAS/daq/evm/src/linux/x86/libi2oEVM.so
  </urlModule>
</IOP>
```

Annexe C

Sources

Le code source des différents modules du gestionnaire d'événements adapté au dispositif de test en faisceau X5 sont repris ici, ainsi que le code source du programme de test de vitesse du pilote.

C.1 FLT

C.1.1 i2oFLT.h

```
//First Level Trigger code for hardware read in X5 testbeam
//by Michael Manalis
//mmanalis@ulb.ac.be
//may 2001

#ifndef __i2oFLT_h__
#define __i2oFLT_h__

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>

#include "Task.h"
#include "i2oRMInterface.h"
#include "i2oFLTInterface.h"
#include "i2oExecutive.h"
#include "vxSys.h"
#include "TriggerListener.h"

#include "hwfltmessage.h"

class i2oFLT: public i2oFLTInterface, public Task
{
```

```
        public:

        i2oFLT();
        ~i2oFLT();

        void addListener(TriggerListener * listener );

        void printhwfltmessage(const hwfltmessage *tp);

        void BusyOff()
        {
            busy_ = FALSE;
            //flt restart, to implement
        };

        int svc(); //task starting code

        /*muon test beam code,
        void vetoOn();
        void vetoOff();
        void enableInterrupt();
        void disableInterrupt();
        */


        protected:

        BOOL busy_;

        //listener to be triggered by this code
        TriggerListener * listener_;

        //events pending into the acquisition card, read at once with this pointer
        hwfltmessage * pendingEvents_;

        //interface to the hardware flt card device
        int fltDevice_;
    };

    #endif
```

C.1.2 i2oFLT.cc

```
#include "i2oFLT.h"

#define FLT_MAX_MESSAGES 256

5

void i2oFLT::addListener(TriggerListener * l)
{
    listener_ = l; // direct invokation ( no i2o )
10 }


i2oFLT::i2oFLT()
{
15      pendingEvents_ = new hwfltmessage[FLT_MAX_MESSAGES];

        this->activate(); //starting the task, svc() is the entry point
}

20
i2oFLT::~i2oFLT()
{
        delete pendingEvents_;
        pendingEvents_ = 0;
25 }


int i2oFLT::svc()
{
30      //opening reading port for trigger information
        if ( ( fltDevice_=open("/dev/shortint",O_RDONLY)) < 1)
        {
            XDAQ_WARN(("can_not_open_device,_exiting_flt"));
            return -1;
35      }


        //pointer (in stack) to a flt triggering message
        hwfltmessage message;
40      const hwfltmessage * pFltMessage_ = &message;

        //readed lenght
        uint   length_ = 0;
        //normal flt trigger message lenght
45      const int MESSAGE_LENGTH = sizeof (hwfltmessage);


        for (;;)
        {
50
            //read message by message blocking read
            while (MESSAGE_LENGTH == (length_ =
                    read ( fltDevice_ , ( void*) pFltMessage_, MESSAGE_LENGTH)))
```

```
            {
55          XDAQ_DEBUG (("trigger_received"));

            TriggerInfo   tinfo = pFltMessage_->bx;
            listener_ ->trigger(tinfo);

60          //batch reading of pending events
            if (pFltMessage_->pendevents > 1)
            {
                int npend_ = pFltMessage_->pendevents - 1;
                //minus 1, the first have just been read
65
                if (npend_ > FLT_MAX_MESSAGES)
                {
                    //pending events number error, pending number is higher
                    //than the flt buffer size
70                  XDAQ_WARN(("pending_event_error:_%d,_should_be_%d",
                                npend_, FLT_MAX_MESSAGES));
                    npend_ = FLT_MAX_MESSAGES;
                }

75
                if ((MESSAGE_LENGTH * npend_) == (length_ =
                read(fltDevice_ , (void*) pendingEvents_, MESSAGE_LENGTH * npend_)))
                {
                    for (int i=0; i<npend_; i++)
80                  {
                        TriggerInfo   tinfo = pendingEvents_[i].bx;
                        listener_ ->trigger(tinfo);
                    }
                }
85              else
                //batch read error: readed length of all pending events is not what it should be
                {
                    XDAQ_WARN(("batch_return_length:_%d_,_should_be_%d\n",
                                length_ , MESSAGE_LENGTH * npend_));
90
                    //recovering by re-evaluation of the pending number
                    int realpend_ = length_ / MESSAGE_LENGTH;
                    for (int i=0; i<realpend_; i++)
                    {
95                      TriggerInfo   tinfo = pendingEvents_[i].bx;
                        listener_ ->trigger(tinfo);
                    }
                }

100         }
        }
        XDAQ_WARN(("return_length:_%d_,_should_be_%d\n",
                    length_ , MESSAGE_LENGTH));
    }
105
    delete pFltMessage_;
    pFltMessage_ = 0;
```

```
    return(0);
}


//debug purpose code, printout a trigger message

void i2oFLT::printhwfltmessage (const hwfltmessage *tp)
{
  printf("eventnr: \t%8u\n",tp->eventnr);
  printf("bunchcrossing\t%8u\n",tp->bx);
  printf("time [s]: \t%8u\n",tp->timeS);
  printf("time [us]: \t%8u\n",tp->timeUS);
  printf("pendingevnt: \t%8u\n",tp->pendevents);
  printf("interruptnr: \t%8u\n",tp->interruptnr);
  printf("nrofinterrupts: \t%8u\n",tp->nrofintr);
}
```

C.1.3 hwfltmessage.h

```
#ifndef __HWFLTMESSAGE
#define __HWFLTMESSAGE

//flt trigger message format

typedef struct {
    uint eventnr;       /* event number from the HW FLT interface */
    uint bx;            /*bunch crossing number from the HW FLT interface */
    uint timeS;         /* computer time (s) of the arival of the event ( for debugging) */
    uint timeUS;        /* computer time (us) of the arival of the event ( for debugging) */
    uint pendevents;    /* pending events from the HW FLT interface */
    uint interruptnr;   /* interrupt follow order number */
    uint nrofintr;      /* due the structure of top bottom half it could be that
                           there is more than one interrupt but this should not possible
                           with the hardware */
    uint ped2;          /* to get a 32 bit structure */

} hwfltmessage;

#endif
```

C.2 RM

C.2.1 i2oRM.h

```
//RM is the part of the Event Manager in charge of the triggers ID and RU control

#ifndef __i2oRM_h__
#define __i2oRM_h__

#include "vxSys.h"
#include "i2oRMAdapter.h"
#include "i2oEVMParam.h"
#include "i2oFLT.h"
#include "BSem.h"

#include "TriggerListener.h"

#include "soapUserTriggerListener.h"


#define EVM_TEST_SPEED
//uncomment this line to test the EVM code
//no call will be isued to BU or RU
//instead, speed value will be issued after 100 000 events
//should be beter than a second for final specs
//uncomment also in i2oBM.h


#ifdef EVM_TEST_SPEED
#include <sys/time.h>
#include <sys/types.h>
#endif


class i2oRM: public i2oRMAdapter
, public soapUserTriggerListener, public TriggerListener
{

//
//  Member Functions:
//

public:

    i2oRM(i2oFLTInterface *flt);

    //
    // Emulate FLT (commanded from RC)
    //
    DOM_Node UserTrigger(DOM_Node node ) {
        // force a trigger independently of busy state (unsafe of course if no triggers available)

        XDAQ_DEBUG((""));

```

```
            TriggerInfo  tinfo = 0;
            this->trigger(tinfo);

            DOM_Node reply;
            e_->soapReply(reply);

        }

        DOM_Node UtilParamsGet(DOM_Node funcNode);
        DOM_Node UtilParamsSet(DOM_Node funcNode);

        DOM_Node DdmSystemChange(DOM_Node node);
        DOM_Node DdmSystemEnable(DOM_Node node);
        DOM_Node DdmDeviceSuspend(DOM_Node node);
        DOM_Node DdmDeviceResume(DOM_Node node);

        void SystemConfigure();


        //Invoked by BM
        void readyToRead(EventIdentifier* eid);

        //Invoked by FLT
        void trigger( TriggerInfo  info );

        void spillOn(TriggerInfo  info) {}
        void spillOff( TriggerInfo  info) {}

    private:
        // ID Queue
        inline int idnotempty()
        {
            BOOL empty = identifiersq_.empty();

            return(! empty);
        }

        inline void idput(int id)
        {
             identifiersq_ .push_back(id);
        }

        inline int idget()
        {
            int tmpId = identifiersq_.front();
             identifiersq_ .pop_front();

            return(tmpId);
        }

    private:
        i2oFLTInterface * flt_ ;

        unsigned long counterTrigger_;
        unsigned long counterLost_;
```

```
        unsigned long noEIDs_;
        //EVTID list
         rlist <unsigned long> identifiersq_;

        unsigned long builderDisable_;
        unsigned long enable_;
        BSem * mutex_;
        BOOL pending_;

    #ifdef EVM_TEST_SPEED
        long cptr;
    #endif

    };

    #endif
```

C.2.2 i2oRM.cc

```
// i2oRM.cc
#include "i2oRM.h"

#ifdef EVM_TEST_SPEED
//test speed mode

#define RM_ITER 100000
//number of events before printing time used

//timing function
void RMbench (bool stop);

timeval RMtimebefore, RMtimeafter;
#endif


i2oRM::i2oRM( i2oFLTInterface *flt )
{
#ifdef EVM_TEST_SPEED
    cptr = -1;
#endif


    builderDisable_ = 0;
    flt_ = flt ; //interface to First Level Trigger adapter code

    pending_ = FALSE;
    enable_ = FALSE;
    counterTrigger_ = 0;
    counterLost_ = 0;
    noEIDs_ = 1024;
    mutex_ = new BSem(BSem::FULL);
    put("builderDisable",builderDisable_ );
    put("noEIDs",noEIDs_);

}

DOM_Node i2oRM::UtilParamsGet(DOM_Node funcNode)
{
    put("builderDisable",builderDisable_ );
    put("noEIDs",noEIDs_);
    return soapUtilAdapter::UtilParamsGet(funcNode);
}

DOM_Node i2oRM::UtilParamsSet(DOM_Node funcNode)
{
    soapUtilAdapter::UtilParamsSet(funcNode);

    get("builderDisable",&builderDisable_);
    get("noEIDs",&noEIDs_);

    XDAQ_DEBUG(("builderDisable: %d",builderDisable_));
    XDAQ_DEBUG(("noEIDs: %d",noEIDs_));
```

```
55      DOM_Node n;
        return n;
    }

    DOM_Node i2oRM::DdmSystemChange(DOM_Node node)
60  {
        get("builderDisable",&builderDisable_);
        get("noEIDs",&noEIDs_);


65      XDAQ_NOTE(("builderDisable: %d",builderDisable_));
        XDAQ_NOTE(("noEIDs: %d",noEIDs_));

        this->SystemConfigure();

70      DOM_Node replyNode;
        return replyNode;
    }

    DOM_Node i2oRM::DdmSystemEnable(DOM_Node node)
75  {
        this->SystemConfigure();

        DOM_Node replyNode;
        return replyNode;
80  }

    void i2oRM::SystemConfigure()
    {
         identifiersq_ . resize (noEIDs_+1);
85       identifiersq_ .setName("i2oRM/identifiersq_");

        for ( int i =0; i< noEIDs_; i++ ) idput(i);
        enable_ = TRUE;
         flt_ ->BusyOff();
90  }

    DOM_Node i2oRM::DdmDeviceSuspend(DOM_Node node)
    {
        DOM_Node n; return n;
95  }

    DOM_Node i2oRM::DdmDeviceResume(DOM_Node node)
    {
        DOM_Node n; return n;
100 }



    //Invoked by BM, return an EVT ID that have been totaly read by a BU
105 void i2oRM::readyToRead(EventIdentifier* eid)
    {
        mutex_->take();
        idput(*eid);
```

```
    if ( pending_ )
    { // re-enable trigger
        pending_ = FALSE;
        flt_->BusyOff();
        XDAQ_DEBUG(("Re-enable trigger again"));
    }
    mutex_->give();
}

//Invoked by FLT, TriggerInfo is the bunch crossing number
void i2oRM::trigger(TriggerInfo info)
{
    if ( ! enable_ )
    {
        XDAQ_DEBUG(("Trigger is not enable"));
        flt_->BusyOff();
        return;
    }

    XDAQ_DEBUG(("Triggering..."));

    if ( idnotempty() )
    {
        // IDs availables
        EventIdentifier eid = idget();

        for (int i=0; i< rui_.size(); i++ )
        {
            XDAQ_DEBUG(("ask for readout"))

#ifdef EVM_TEST_SPEED
            //testing output calls speed
            cptr++;
            if (cptr == RM_ITER)
            {
                RMbench(0);
                cptr = 0;
            }
            else
            {
                if (cptr == -1) RMbench(1);
            }

#else
            rui_[i]->readout_and_remove(eid, info);
#endif
        }

        if ( builderDisable_)
            idput(eid);
        else
        {
            XDAQ_DEBUG(("give eid to BM for reading (readyToSend)"));
            bm_->readyToSend(&eid);
        }
```

```
165         }
            else
            {
                // TRIGGER LOST
                cout << "*** Fatal Error: a trigger came witn no IDs available"
170                 << endl;
                sleep (1);
        #ifdef vxworks
                ::taskDelay(0);
        #endif
175         }

            mutex_->take();
            if ( idnotempty() )
            { //space available in RDPM
180             XDAQ_DEBUG(("Re-enable trigger now"));
                flt_ ->BusyOff();
            }
            else
            {
185             XDAQ_DEBUG(("Keep trigger disable"));
                //as soon as ID come in readyToRead
                pending_ = TRUE;
            }
            mutex_->give();
190
            return;
        }


195     #ifdef EVM_TEST_SPEED
        void RMbench (bool start)
        {
            if (start)
            {
200             gettimeofday (&RMtimebefore, NULL);
            }
            else
            {
                gettimeofday (&RMtimeafter, NULL);
205
                long diffsec = RMtimeafter.tv_sec - RMtimebefore.tv_sec;
                long diffusec = RMtimeafter.tv_usec - RMtimebefore.tv_usec;

                if ( diffusec < 0) {diffusec +=1000000;}
210
                XDAQ_NOTE(("RM %d events stat: %d sec - %d usec", RM_ITER, diffsec, diffusec));

                gettimeofday (&RMtimebefore, NULL);
            }
215     }

        #endif
        // End of file
```

C.3 BM

C.3.1 i2oBM.h

```
//BM is the part of the Event Manager in charge of controling event attributions to BU

#ifndef __i2oBM_h__
#define __i2oBM_h__

#include "vxSys.h"
#include "i2oBMAdapter.h"

#include "BSem.h"
#include "rlist.h"


#define EVM_TEST_SPEED
//uncomment this line to test the EVM code
//no call will be isued to BU or RU
//instead, speed value will be issued after 100 000 events
//should be beter than a second for final specs
//uncomment also in i2oRM.h


#ifdef EVM_TEST_SPEED
#include <sys/time.h>
#include <sys/types.h>
#endif

class i2oBM: public i2oBMAdapter {

//
//  Member Functions:
//

public:
    //request from a BU
    typedef struct {
        short dest;
        short neids;
    } UrgentRequest;

    i2oBM();


    DOM_Node UtilParamsGet(DOM_Node funcNode);
    DOM_Node UtilParamsSet(DOM_Node funcNode);

    DOM_Node DdmSystemChange(DOM_Node node);
    DOM_Node DdmSystemEnable(DOM_Node node);
    DOM_Node DdmDeviceSuspend(DOM_Node node);
    DOM_Node DdmDeviceResume(DOM_Node node);
    DOM_Node DdmSystemHalt(DOM_Node node);
```

```
        void allocate (EventProfile* eprofile , U16 events, U16 instance);
        void retrieve(unsigned long*, unsigned long*, DestinationCookie*);
        void allocateRetrieve(unsigned long*, unsigned long*, DestinationCookie*);
        void readyToSend(EventIdentifier* eid);
        void clear ( EventIdentifier * eid , int neids );

    private:
        void SystemConfigure();
        inline void commit();

        // Trigger Queue
        inline int triggernotempty()
        {
            //TS mutex_->take();
            BOOL empty = triggersq_.empty();
            //TS mutex_->give();
            return(! empty);
        }

        inline int triggernum() {
            return (triggersq_. size ());

        }

        inline void triggerput(int id)
        {
            //TS mutex_->take();
            triggersq_ .push_back(id);
            //TS mutex_->give();
        }
        inline int  triggerget ()
        {
            //TS mutex_->take();
            int tmpId = triggersq_.front ();
            triggersq_ .pop_front ();
            //TS mutex_->give();

            return(tmpId);
        }

        inline void urgentput(short id, short events)
        {
            UrgentRequest r;
            r.dest = id;
            r.neids = events;
            //TS mutex_->take();
            urgentq_.push_back(r);
            //TS mutex_->give();
        }

        inline UrgentRequest urgentfront()
        {
            return(urgentq_.front());
        }
```

```
        inline void urgentpop()
        {
            urgentq_.pop_front();
110     }

        inline int urgentnotempty()
        {
            //TS mutex_->take();
115         BOOL empty = urgentq_.empty();
            //TS mutex_->give();
            return(! empty);
        }

120 protected:

        //list of EVT ID waiting for a RU
        rlist <unsigned long> triggersq_;
        //list of BU waiting for EVT ID
125     rlist <UrgentRequest> urgentq_;

        BSem * mutex_;
        BSem * alloc_;

130     unsigned long triggerDisable_;
        unsigned long noEIDs_;

        EventIdentifier * eida_;

135 #ifdef EVM_TEST_SPEED
        long cptr;
    #endif

    };
140
    #endif
```

C.3.2 i2oBM.cc

```
// i2oBM.cc

#include "i2oBM.h"

#ifdef EVM_TEST_SPEED

#define BM_ITER 100000
//number of events before printing time used

//timing function
void BMbench (bool stop);

timeval BMtimebefore, BMtimeafter;
#endif


i2oBM::i2oBM()
{
#ifdef EVM_TEST_SPEED
    cptr = -1;
#endif

    state(Halted);

    triggerDisable_ = 0;
    noEIDs_ = 0;
    eida_ = 0;

    mutex_ = new BSem(BSem::FULL);
    alloc_ = new BSem(BSem::FULL);

    put("triggerDisable", triggerDisable_ );
    put("noEIDs",noEIDs_);
}


DOM_Node i2oBM::UtilParamsGet(DOM_Node funcNode)
{
    put("triggerDisable", triggerDisable_ );
    put("noEIDs",noEIDs_);
    return soapUtilAdapter::UtilParamsGet(funcNode);
}


DOM_Node i2oBM::UtilParamsSet(DOM_Node funcNode)
{
    soapUtilAdapter::UtilParamsSet(funcNode);

    get("triggerDisable",&triggerDisable_ );
    get("noEIDs",&noEIDs_);

    XDAQ_DEBUG(("triggerDisable:_%d",triggerDisable_));
```

```
    XDAQ_DEBUG(("noEIDs: %d",noEIDs_));

    DOM_Node replyNode;
    return replyNode;
}


DOM_Node i2oBM::DdmSystemChange(DOM_Node node)
{
    if ( state() == Enabled )
    {
        XDAQ_WARN(("BM_system_is_already_enabled_(ignore_request)"));
    }
    else
    {
        get("triggerDisable",&triggerDisable_);
        get("noEIDs",&noEIDs_);

        XDAQ_NOTE(("BM_instance_%d_parameter_list",instance_));
        XDAQ_NOTE(("triggerDisable: %d",triggerDisable_));
        XDAQ_NOTE(("noEIDs: %d",noEIDs_));

        this->SystemConfigure();

        state(Enabled);
    }
    DOM_Node replyNode;
    return replyNode;
}

DOM_Node i2oBM::DdmSystemEnable(DOM_Node node)
{
    if ( state() == Halted )
    {
        XDAQ_WARN(("BM_system_is_already_enabled_(ignore_request)"));
    }
    else
    {
        this->SystemConfigure();
        state(Enabled);
    }
    DOM_Node replyNode;
    return replyNode;
}


DOM_Node i2oBM::DdmDeviceSuspend(DOM_Node node)
{
    DOM_Node n; return n;
}


DOM_Node i2oBM::DdmDeviceResume(DOM_Node node)
{
    DOM_Node n; return n;
```

```
    }

    DOM_Node i2oBM::DdmSystemHalt (DOM_Node node)
    {
        if ( state() == Halted )
        {
            XDAQ_WARN(("BM_system_is_already_halted_(ignore_request)"));
        }
        else
        {
            state(Halted);

            // Nothing to be done here. All re- initialisation  is
            // done on the next DdmSystemChange
            sleep(1);
        }
        DOM_Node n;
        return n;
    }


    //invoked by BU to send disponibility
    void i2oBM::allocate(EventProfile* eprofile , U16 events, U16 instance)
    {
        if (state() != Enabled) return;

        XDAQ_DEBUG(("EVM_Allocating_%d_events_to_instance_%d",events,instance));
        urgentput(instance,events);

        //deadlock avoidance when called from "confirm"
    #ifndef EVM_TEST_SPEED
        commit();
    #endif
    }

    //invoked by RM to give a EVT ID that should be read by a BU
    void i2oBM::readyToSend(EventIdentifier* eid)
    {
        if (state() != Enabled) return;

        XDAQ_DEBUG(("event_id:_%d",*eid));

        triggerput(*eid);
        commit();
    }


    //Invoked by a BU to recycle EVT ID
    void i2oBM::clear (EventIdentifier* eid , int neids)
    {
        if (state() == Halted) return;

        XDAQ_DEBUG(("EVM_Clearing_%d_events", neids));
```

```
        if ( triggerDisable_ )
165     {
            for (int i =0; i < neids; i++)
            {
            triggerput(eid[i]); //recycle ID
            }
170     }
        else
        {
            for (int i =0; i < neids; i++)
            rm_->readyToRead(&eid[i]);
175     }
    }

    //EVM maximum centralisation
    void i2oBM::retrieve(unsigned long*, unsigned long*, DestinationCookie*) {}
180 void i2oBM::allocateRetrieve(unsigned long*, unsigned long*, DestinationCookie*) {}


    void i2oBM::SystemConfigure()
    {
185     // Don't conflict with the commit()
        alloc_->take();

        triggersq_.resize(noEIDs_+1);
        triggersq_.setName("i2oBM/triggersq_");
190     urgentq_.resize(10000);
        urgentq_.setName("i2oBM/urgentq_");

        // We empty all queues here in the configure to
        // ensure that there are no leftover entries from
195     // previous runs.

        // Empty trigger queue
        if ( triggerDisable_ )
        {
200         while (triggernotempty())
            {
                triggerget ();
            }
        }
205     else
        {
            // nothing, RM takes care of resetting triggers
        }

210     // Clear urgent queue
        while (urgentnotempty())
        {
            urgentpop();
        }
215
        if ( triggerDisable_ )
        {
            for ( int i =0; i< noEIDs_; i++ )
```

```
            triggerput(i); //No trigger
        }

        if (eida_ != 0) delete eida_;
        eida_ = new EventIdentifier[noEIDs_];

        alloc_->give();
    }

    //EVT ID to BU assignements
    inline void i2oBM::commit()
    {    // Commit must be serialized (thread safe)

        alloc_->take();

        XDAQ_DEBUG((""));

        if ( urgentnotempty() )
        { // requests pending

            UrgentRequest r = urgentfront();

            if ( triggernum() >= r.neids )
            {
                XDAQ_DEBUG((""));

                for ( int i = 0 ; i < r.neids; i ++ )
                {
                    eida_[i] = triggerget ();
                }
                XDAQ_DEBUG((""));

    #ifdef EVM_TEST_SPEED
                //bypass BU calls and count them
                this->clear(eida_, r.neids);
                this->allocate(NULL, r.neids, r.dest);

                cptr++;
                if (cptr == BM_ITER)
                {
                    BMbench(0);
                    cptr = 0;
                }
                else
                {
                    if (cptr == -1) BMbench(1);
                }

    #else
                bu_[r.dest]->confirm(eida_,r.neids);
    #endif

                urgentpop();
            }
        }
```

```
        alloc_->give();
}


#ifdef EVM_TEST_SPEED
void BMbench (bool start)
{
    if (start)
    {
        gettimeofday (&BMtimebefore, NULL);

    }
    else
    {
        gettimeofday (&BMtimeafter, NULL);

        long diffsec = BMtimeafter.tv_sec - BMtimebefore.tv_sec;
        long diffusec = BMtimeafter.tv_usec - BMtimebefore.tv_usec;

        if (diffusec < 0) {diffusec+=1000000;};

        XDAQ_NOTE(("BM_%d_events_stat:_%d_sec_-_%d_usec",
                                                BM_ITER, diffsec, diffusec));

        gettimeofday (&BMtimebefore, NULL);
    }
}

#endif

// End of file
```

C.4 Test de rapidité en lecture

C.4.1 driverspeed.cc

```
//This is a smallprogram designed to read as fast as possible triggers comming on the flt
//port, a timing is executed on a basis of ITER number of events read.
//Should provide an idea of the flt driver speed.
//by Michael Manalis, mmanalis @ulb.ac.be

#include <stdio.h>
#include <iostream>
#include <sys/types.h>
#include <sys/time.h>
#include <fcntl.h>
#include <errno.h>

#include "../hwfltmessage.h"

#define ITER 100000
#define FLT_MAX_MESSAGES 256

void bench (bool stop);

timeval before, after;

int main()
{
    long cptr = 0;

    //events pending into the acquisition card, read at once with this pointer
    hwfltmessage * pendingEvents_ = new hwfltmessage[FLT_MAX_MESSAGES];

    //pointer (in stack) to a flt triggering message
    hwfltmessage message;
    const hwfltmessage * pFltMessage_ = &message;

    uint  length_ = 0;
    const int MESSAGE_LENGTH = sizeof (hwfltmessage);


    //interface to the hardware flt card device
    int fltDevice_;

    //opening reading port for trigger information
    if ( ( fltDevice_ = open("/dev/shortint",O_RDONLY)) < 1)
    {
        perror("can_not_open_device_");
        return -1;
    }
    else cout<<"flt_device_loaded\n";


    bench(1);

```

```
        for (;;)
        {

            //read message by message blocking read
55          while (MESSAGE_LENGTH == (length_ =
                    read ( fltDevice_ , ( void*) pFltMessage_, MESSAGE_LENGTH)))
            {
                cptr++;
                if (cptr >= ITER)
60              {
                    bench(0);
                    cptr = 0;
                }

65              //batch reading of pending events
                if (pFltMessage_->pendevents != 0)
                {
                    int npend_ = pFltMessage_->pendevents - 1;
                    //minus 1 because the first has just been read
70
                    if (npend_ > FLT_MAX_MESSAGES)
                    {
                        //pending events number error
                        /*
75                      cout<<"pending number too high: "<<npend_
                        <<", should be max "<<FLT_MAX_MESSAGES<<endl;
                        */
                        npend_ = FLT_MAX_MESSAGES;
                    }
80

                    if ((MESSAGE_LENGTH * npend_) == (length_ =
                        read( fltDevice_ , ( void*) pendingEvents_, MESSAGE_LENGTH * npend_)))
                    {
85                      for (int i=0; i<npend_; i++)
                        {
                            cptr++;
                            if (cptr >= ITER)
                            {
90                              bench(0);
                                cptr = 0;
                            }

                        }
95                  }
                    else //batch read error: length of all pending events is not what it should be
                    {
                        //printf("batch return length: %d , should be %d\n",
                        //       length_ , MESSAGE_LENGTH * npend_);
100
                        //cout<<"recovering from pending error\n";

                        int realpend_ = length_ / MESSAGE_LENGTH;
                        for (int i=0; i<realpend_; i++)
105                     {
```

```
                                cptr++;
                                if (cptr >= ITER)
                                {
                                    bench(0);
110                                 cptr = 0;
                                }

                            }
                        }
115
                    }

                }
                printf("return_length:_%d_,_should_be_%d\n", length_, MESSAGE_LENGTH);
120         }
        }


125     void bench (bool start)
        {
            if (start)
            {
                gettimeofday (&before, NULL);
130         }
            else
            {
                gettimeofday (&after, NULL);

135             long diffsec = after.tv_sec - before.tv_sec;
                long diffusec = after.tv_usec - before.tv_usec;

                if (diffusec < 0) {diffusec+=1000000;};

140             printf("%d_events_stat:_%d_sec_-_%d_usec\n", ITER, diffsec, diffusec);

                gettimeofday (&before, NULL);
            }
        }
```

Bibliographie

[1] CERN. *CMS TriDAS DAQ Event Builder, System Design Description*, version 1.0α draft edition, August 1999. CMS_EVB_SDD.

[2] Claude Delannoy. *Programmer en langage C++*. Eurolles, 5e dition edition, 2000.

[3] R. Kwarciany M. Litmaath V.O'Dell K. Sumorok E. Barsotti, M. Bowden. *Preliminary Specifications for the CMS Event Manager*, draft, version 8.0 edition, November 1999.

[4] I2O Special Interest Group. *Intelligent I/O (I2O) - Architecture Specification*, 2.0 edition, March 1999.

[5] Johannes Gutleber. *Muon Testbeam, Trigger and Readout, Interface Requirements Specification.* CERN, version 0.9, draft edition, July 2000. CMS_DAQ_IRS_02.

[6] Johannes Gutleber. *XDAQ Configuration and Control, Interface Requirements Specification.* CERN, November 2000. CMS_DAQ_IRS_03.

[7] Johannes Gutleber. *DAQ-DCS, Interface Requirements Specification.* CERN, version 1, revision 1 edition, March 2001. CMS_DAQ_IRS_04.

[8] Johannes Gutleber. *XDAQ Event Builder, User Manual.* CERN, May 2001. CMS-TriDAS/00_009/eng/doc April2001R1.

[9] L. Orsini J. Gutleber. *DCS-XDAQ, Interface Design Description.* CERN, version 1 revision 2 edition, March 2001. CMS_DAQ_IDD_0004.

[10] S. Cittolin F. Meijers L. Orsini D. Samyn J. Gutleber, E. Cano. Architectural software support for processing clusters. In *IEEE International Conference on Cluster Computing*, CERN, 1211 Geneva 23, Switzerland, 2000.

[11] Luciano B. Orsini. *CMS TriDAS XDAQ Framework, Software Release Notes.* CERN, April 2001. APRIL2001R1.

[12] Luciano B. Orsini. *XDAQ Configuration and Control, Interface Design Description.* CERN, version 0 revision 2 edition, January 2001. CMS_DAQ_IDD_0001.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addisson-Wesley, third edition edition, 1999.