JAVA 言語を用いた
Data Acquisition System の開発

坂本 泰伸

山形大学理学研究科物理学専攻 15 番

Chapter 1　論文概要

本論文は、“JAVA 言語を用いた Data Acquisition System(DAQ)の開発”に関する研究結果をまとめたものである。この研究は、山形大学クォーク核物性研究室と、高エネルギー加速器研究機構(KEK)のオンライングループの間で 1996 年度から始められたもので、現在も引き続き研究が行われている。本論分では、1997 年度まで行われた研究内容とその結果について報告をする。

DAQ とは、高エネルギー物理学実験などで用いられる、各種の計測機器から送られてくる物理的事象(アナログシグナル)を、VME bus、CAMAC、TKO といった標準回路を通してデジタル化し、コンピュータに接続されているハードディスクや DAT に、データとして保存するためのシステムである(図 1.1)。一般的に、高エネルギー物理学実験で用いられる DAQ には、きわめて短時間に起こる大量の物理的事象を、正確にハードディスクや DAT といった記録媒体にデータとして保存することや、実験を常時モニターできることなどが求められる。現在、多くの物理実験グループでは多数のワークステーションを導入し、実験専用のハードウエアとソフトウエアを用いて DAQ を構築している。これらの DAQ は、特定の実験専用に作成されているため、非常に大量のデータを単位時間内収集する事ができる。しかし、これらの DAQ には、汎用性が低く、コストがかかるといった欠点がある。

そこで、汎用性が高く、低コストの DAQ の開発を行うこととなった。まず、低コストを実現させるために、PC/AT 互換機(PC)と呼ばれるパーソナルコンピュータに、Linux と呼ばれる UNIX OS を導入する。PC は、ワークステーションに比べると非常にコストが低い。この計算機に Linux OS を導入することによって、ワークステーションと同等の機能を持たせることが可能である。実際に、PC/Linux システムは、性能比較テストの結果、他のワークステーションと比較しても、遜色がないことが証明された。

DAQ は、JAVA と呼ばれるプログラミング言語と、HORB と呼ばれる JAVA

アプリケーションを用いて作成した。JAVA 言語は、1996 年に Sun Microsystems より公式にリリースされたプログラミング言語である。この言語は、オブジェクト指向のプログラミング言語で、分散処理対応、インタープリタ形式、プラットホーム非依存、マルチスレッド対応、豊富な GUI ツールを有する、という特徴を持っている。これらの JAVA 言語自体の持つ特徴は、DAQ を作成するにあたり非常に重要となってくる。また、HORB は、JAVA 言語の持つ機能を用いて、オブジェクト通信機能を行うアプリケーションである。このアプリケーションは、電総研の平野博士が作成した。この HORB の持つ機能は、JAVA 言語を用いて DAQ を作成する場合に、非常に有用であると考えられる(Chapter 2)。

次に、標準回路の１つである CAMAC モジュールを、PC/Linux システムからコントロールするため、CAMAC デバイスドライバ、及び、ライブラリを作成した。これは、CC7000/CC7700 と呼ばれる CAMAC クレートコントローラを、Linux OS 上のソフトウエアから操作するために必要となるものである。このデバイスドライバ及び、ライブラリの性能は、DAQ 全体の性能を左右する重要なパーツである。このため、この CC7000/CC7700 ライブラリーの性能テストもおこなった(Chapter 3)。

今回、作成した JAVA DAQ Prototype は、複数の独立したプロセスからなるシステムで、それぞれのプロセスは、Collector、Cache、Recorder、Analyzer、Commander と呼ばれる。これらのプロセスの働きは次のようになっている。

- Collector：CAMAC からのデータの収集を行う。CAMAC ライブラリが含まれている。
- Recorder：データを記録するプロセス。
- Analyzer：オンラインでデータを解析するプロセス。CERN ライブラリを含んでいる。
- Cache：Collector から送られるデータを一時的に蓄え、Recorder、Analyzer へ送る。
- Commander:各プロセスに対し命令を送信する。

また、これらのプロセスは複数の計算機の上で協調し動作するように開

発されている。常に１台の計算機だけを用いてデータを収集する必要はない。データを収集するプロセスと、データを解析するプロセスなどを分離して **DAQ** を実行することが可能である。これにより、データ収集プロセスが動作している計算機の、**CPU** 負担を軽減することもできる。また、これらのプロセスは、**JAVA** 言語の特徴上、計算機の種類や、計算機上で稼働している **OS** の種類にとらわれることなく動作する。ただし、**Collector**、**Analyzer** のように、計算機、及び **OS** に依存するライブラリを含むプロセスを、他の計算機上で実行する場合は、ライブラリを改造する必要が生じる **(Chapter 4)**。

そして、今回作成された **JAVA DAQ Prototype** は、**1997** 年 **11** 月末に **KEK** 田無分室で実際に使用された。この実験は、“**PWO** クリスタルの基本性能テスト”というテーマである。この実験の目的は、大きく分けて２つある。１つ目は、今回作成した **DAQ** の機能が正しく動作するかのチェックをすることである。２つ目は、**1GeV/c** 以下の運動量を持つ電子の入射に対する **PWO(**タングステン酸鉛**)**クリスタルのエネルギー分解能、及び、今回の実験で使用されたセットアップでの位置分解能を、系統的に測定することである。この実験内容と、解析結果についても報告をおこなう **(Chapter 5)**。

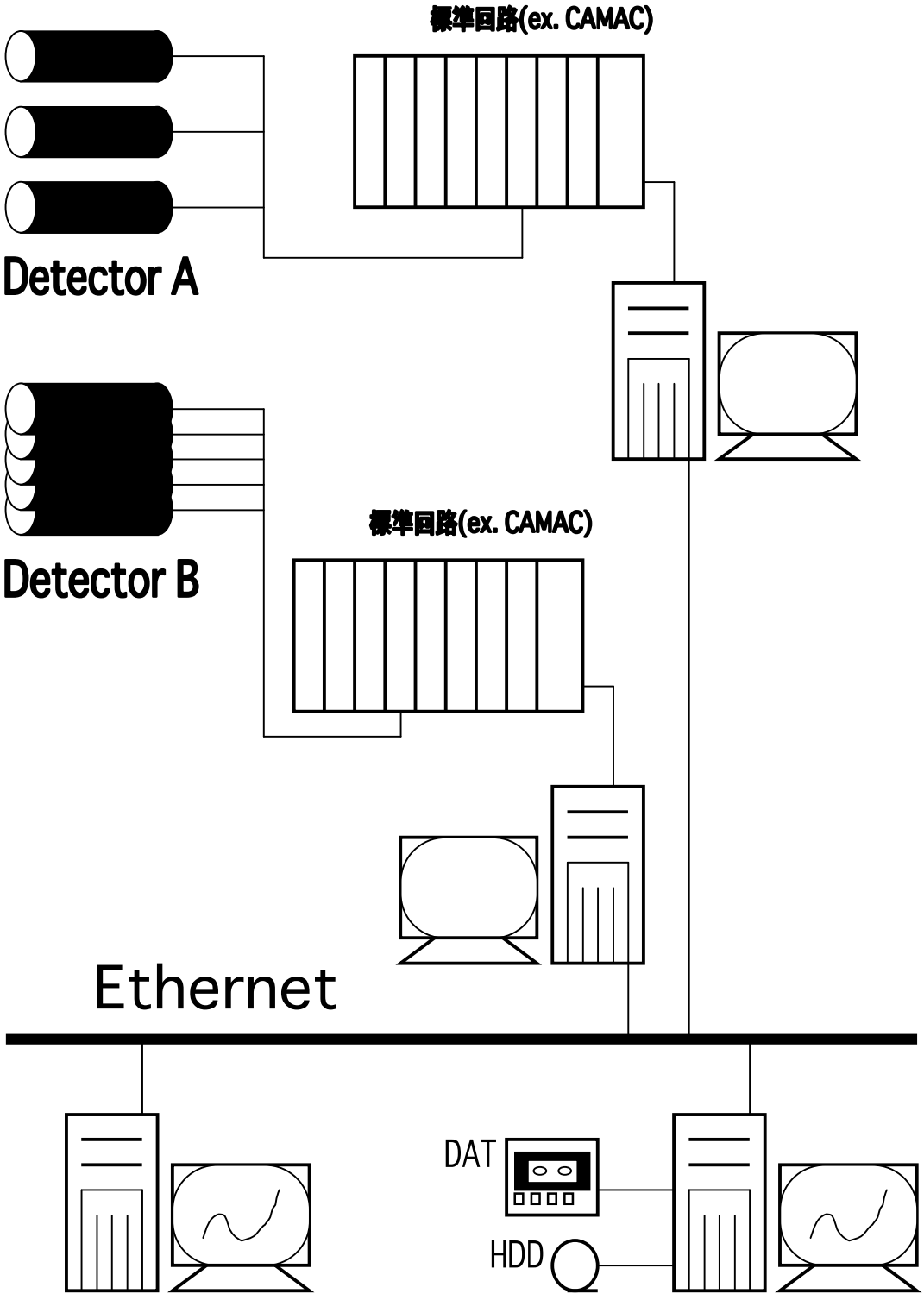


図 1.1 : Data Acquisition Model

Chapter 2　PC/Linux , JAVA and HORB

Section 2.1　PC/Linux

Linux は、ヘルシンキ大学の **Linus B.Torvalds** 氏が開発した **UNIX OS** である。**UNIX OS** には大きく分けて、**BSD** 系と、**System V** 系という、２つの系統が存在するが、この **OS** はそのどちらにも分類することはできない。**BSD** 系、**System V** 系の **UNIX OS** が混合された **OS** である。この **OS** は、**Linus** 氏を中心とする、世界中に広がるボランティアグループによってサポートされている **OS** で、この **OS** のソースプログラムは一般に公開されており、幅広い分野で利用されている。**Linux OS** を **PC** に導入する利点は、**UNIX** ワークステーション上で動作する **CERN** ライブラリを始め、多数の **UNIX** アプリケーションが **PC** 上で使用することができるようになることである。また、パーソナルコンピュータである **PC** は、ワークステーションに比べてコストが安いという点も長所として挙げられる。対して、**Linux OS** を物理学実験等で用いる場合の短所は、**OS** の使用に際して生じる問題点などが、まったくサポートを受けられないことである。全ての問題を、個人で処理を行わなくてはならない。このため、ネットワークなどを介して、**OS** に関する情報をこまめに収集する必要が生じる。また、バージョンアップの頻度が、市販されている **UNIX OS** と比較して非常にはやいため、対応が大変である。

DAQ を開発するにあたり、考えなくてはならないものは、その性能である。**DAQ** のソフトウエア部分の持つ様々な性能は、ただ単に **DAQ** のアルゴリズムのみに決定されるものではなく、計算機の能力、使用するプログラミング言語の特性、**DAQ** を走らせる **OS** の特性、といった様々な要素が絡み合って決定される。この章で報告するのは、これらの要素の中で、計算機に依存する性能、及び **OS** に依存する性能の評価である。また、この節に記した **PC/Linux** システムの性能テストの結果は、全て **DAQ-BENCH**[7] と呼ばれる性能評価プログラムを用いて測定した結果である。

プログラムは、計算機に対するさまざまな命令の集合体である。プログラムに含まれる命令の種類には、計算性能、データコピー性能、データ通信性能など、さまざまなものがある。**DAQ** にとって重要となってくる性能は、その特徴上、データコピー性能、データ通信性能などが挙げられる。まず、表 **2.1.1**と表 **2.1.2**は、**PC/Linux** システムのデータコピー性能テストの結果である。これは、メモリーからメモリーにデータをコピーするのに要する時間である。この測定では、**memcpy** 関数、**strcpy** 関数、及び **for** ループを用いた。この結果には、**memcpy** 関数、**strcpy** 関数を呼び出すのにかかる時間**(**オーバーヘッド**)**が含まれている。また、**for** ループを用いた測定では、ロングワード**(4** バイト**)**単位でデータをコピーした結果を記している。また、表 **2.1.3**は、**memcpy** を用いたデータコピー性能を、**3** 種類のワークステーションと比較した結果を表している。これより、**PC** が、ワークステーションと同等のデータコピー性能を有することがわかる。

つぎに、コンテキストスイッチに要する時間についても測定をおこなった。**UNIX OS** は、マルチタスク処理を実現している。**UNIX OS** 上で稼働している各ユーザプロセスは、非常に短い時間を **OS** から分配してもらい、その間のみ **CPU** 資源を消費できる。この **CPU** を占有できる時間を、各プロセスに対し次々と分配を行うことで、見かけ上、複数のプロセスが同時に動作しているように見える。このとき、ある１つのプロセスから、別のプロセスに実行権限が移る際にも時間が消費される。この時間が、コンテキストスッチに要する時間である。この時間は、**OS** の設計思想や、計算機の性能などによって特徴づけられるものである。この結果を記したのが、表 **2.1.4**である。この値から、**PC/Linux** は、ワークステーションと同等のコンテキストスイッチ性能を有することがわかる。

表 **2.1.5**と表 **2.1.6**、は、プロセス間通信に関連する性能を、２種類の **PC** を用いて評価した結果である。評価項目は、**FIFO(**名前付きパイプ**)**、メッセージキュー、**PIPE** と呼ばれる **UNIX OS** で実装されている、プロセス間データ通信にかかる時間、及び **Internet** ソケット、**UNIX domain** ソケットに対するデータの送受信にかかる時間である。これらの、プロセス間

通信の性能も、他のワークステーションの結果(表 2.1.7、表 2.1.8)と比較して遜色のない結果を得た。

これらの性能比較テストの結果は、我々の期待を十分満足させる結果である。これらの結果より、PC/Linux システムで DAQ の開発をおこなうことが可能であると判断した。

表 **2.1.1**：データコピー性能 **1**

P6-150MHz , Linux 2.0.0 , gcc 2.7.2

memcpy()	**40 MB/sec**
strcpy()	**37 MB/sec**
for-loop	**41 MB/sec**

表 **2.1.2**：データコピー性能 **2**

P5-120MHz , Linux 1.2.13 , gcc 2.7.0

memcpy()	**45 MB/sec**
strcpy()	**15 MB/sec**
for-loop	**36 MB/sec**

表 **2.1.3**：データコピー性能比較

memcpy 関数を使用

PC P5-120MHz Linux 1.2.13 (gcc 2.7.0)	**35 Mbyte/sec**
PC P5-120MHz Linux 1.2.13 (gcc 2.7.2)	**41 Mbyte/sec**
PC P6-150MHz Linux 2.0.0 (gcc 2.7.2)	**40 Mbyte/sec**
HP-735/HP-UX	**28 Mbyte/sec**
Alpha 200 4/233/OSF1	**29 Mbyte/sec**
Force SPARC5V/SOLARIS	**30 Mbyte/sec**

表 **2.1.4**：コンテキストスイッチ性能比較

PC P5-120MHz Linux 1.2.13 (gcc 2.7.0)	**31** μ **sec**
PC P5-120MHz Linux 1.2.13 (gcc 2.7.2)	**22** μ **sec**
PC P6-150MHz Linux 2.0.0 (gcc 2.7.2)	**2.3** μ **sec**
HP-735/HP-UX	**22** μ **sec**
Alpha 200 4/233/OSF1	**27** μ **sec**
Force SPARC5V/SOLARIS	**75** μ **sec**

表 2.1.5：プロセス間通信性能 1

P6-150MHz , Linux 2.0.0 , gcc 2.7.2

FIFO(名前付きパイプ)

Data size	Time
256 byte	16.8 μ sec
512 byte	23.4 μ sec
1024 byte	37.0 μ sec
2048 byte	64.5 μ sec

Message queue

Data size	Time
256 byte	20.9 μ sec
512 byte	27.6 μ sec
1024 byte	43.4 μ sec
2048 byte	71.3 μ sec

PIPE

Data size	Time
256 byte	16.7 μ sec
512 byte	23.5 μ sec
1024 byte	37.6 μ sec
2048 byte	65.7 μ sec

INTERNET SOCKET

Data size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	37.7(24.4) μ sec	37.7(13.1) μ sec
512 byte	52.4(35.5) μ sec	52.4(16.7) μ sec
1024 byte	82.5(57.6) μ sec	82.5(24.7) μ sec
2048 byte	134.9(88.1) μ sec	134.9(46.6) μ sec

UNIX domain SOCKET

Date size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	36.0(20.6) μ sec	36.0(15.4) μ sec
512 byte	39.8(19.7) μ sec	39.8(20.1) μ sec
1024 byte	60.8(30.5) μ sec	60.9(30.3) μ sec
2048 byte	96.3(47.3) μ sec	96.3(48.9) μ sec

表 2.1.6：プロセス間通信性能 2

P5-120MHz , Linux 1.2.13 , gcc 2.7.20

FIFO(名前付きパイプ)

Data size	Time
256 byte	17.5 μ sec
512 byte	25.0 μ sec
1024 byte	40.2 μ sec
2048 byte	69.5 μ sec

Message queue

Data size	Time
256 byte	20.8 μ sec
512 byte	31.8 μ sec
1024 byte	41.9 μ sec
2048 byte	76.9 μ sec

PIPE

Data size	Time
256 byte	16.9 μ sec
512 byte	24.6 μ sec
1024 byte	39.8 μ sec
2048 byte	70.1 μ sec

UNIX domain SOCKET

Date size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	35.9(18.4) μ sec	35.9(17.5) μ sec
512 byte	51.5(26.7) μ sec	51.6(24.8) μ sec
1024 byte	78.1(45.2) μ sec	78.1(32.9) μ sec
2048 byte	127.1(72.0) μ sec	127.1(55.1) μ sec

表 **2.1.7**：プロセス間通信性能 **3**

HP-735/HP-UX

FIFO(名前付きパイプ**)**

Data size	Time
256 byte	71.1 μ sec
512 byte	79.7 μ sec
1024 byte	94.1 μ sec
2048 byte	123.3 μ sec

Message queue

Data size	Time
256 byte	39.2 μ sec
512 byte	46.1 μ sec
1024 byte	83.7 μ sec
2048 byte	93.0 μ sec

PIPE

Data size	Time
256 byte	73.6 μ sec
512 byte	81.8 μ sec
1024 byte	93.5 μ sec
2048 byte	120.1 μ sec

INTERNET SOCKET

Data size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	84.0(41.5) μ sec	84.0(42.5) μ sec
512 byte	117.2(46.0) μ sec	117.3(71.1) μ sec
1024 byte	182.8(67.7) μ sec	182.9(115.0) μ sec
2048 byte	193.0(106.9) μ sec	193.0(84.0) μ sec

UNIX domain SOCKET

Date size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	98.4(52.1) μ sec	98.4(45.6) μ sec
512 byte	116.3(60.0) μ sec	116.3(56.2) μ sec
1024 byte	148.3(78.2) μ sec	148.3(69.5) μ sec

表 **2.1.8**：プロセス間通信性能 **4**

Alpha 200 4/233/OSF1

FIFO(名前付きパイプ**)**

Data size	Time
256 byte	33.8 μ sec
512 byte	45.0 μ sec
1024 byte	66.8 μ sec
2048 byte	112.0 μ sec

Message queue

Data size	Time
256 byte	27.3 μ sec
512 byte	40.7 μ sec
1024 byte	63.2 μ sec
2048 byte	104.0 μ sec

PIPE

Data size	Time
256 byte	35.0 μ sec
512 byte	46.8 μ sec
1024 byte	67.2 μ sec
2048 byte	112.8 μ sec

UNIX domain SOCKET

Date size	Send Elapse time (CPU time)	Receive Elapse time (CPU time)
256 byte	49.3(33.2) μ sec	49.3(15.8) μ sec

Section 2.2 JAVA

次世代の DAQ に求められるであろうものを考えると、ネットワーク機能、WEB 機能を持つ DAQ クライアント、オブジェクト指向プログラミング、分散データベース、プラットフォーム非依存といったキーワードが挙げられる[6]。これらの機能を、より簡単に DAQ に実装するためのプログラミング言語に、JAVA 言語を選択した。

JAVA 言語は 1997 年に、Sun Microsystems より Java Developer's Kit(JDK)として公式にリリースされた、オブジェクト指向言語のプログラミング言語で、分散処理対応、インタープリタ形式、プラットホーム非依存、マルチスレッド対応、ネットワークプログラミング対応、豊富な GUI ツールを有する、という特徴を持っている。また、JAVA 言語は、単にネットワーク対応のプログラミング言語なのではない。従来の C 言語といったプログラミング言語と比較して、ネットワークプログラミングが非常に簡単、かつ簡潔に記述できるのである(表 2.2.1)。これは、GUI についても同じである。これらの長所は、先に述べた、次世代の DAQ に対する要求を実現するのに非常に有用である。

JAVA 言語によって記述されたプログラムは、大きく分けて２つに分類することができる。JAVA アプリケーションと JAVA アプレットである。JAVA アプリケーションは、JAVA インタープリタによってそのまま実行可能である。JAVA アプレットは、Netscape や Appletviewer といったビュアーに読み込まれて始めて実行される。

JAVA 言語のプログラムは、まず、コンパイラーによって中間コードと呼ばれるものに変換される。このコードは、どの計算機上の JAVA コンパイラーによって再生されても、全て同じものである。次に、生成されたコードは、JAVA インタープリタによって、実行する計算機用のコードに変換され実行される。JAVA 言語を用いて DAQ を作成する時に、問題となるのは、JAVA 言語がインタープリタ形式で実行される点である。一般に、インタープリタ形式の言語で記述されたプログラムは、毎回１行ずつ解釈され実行される。解釈と実行が交互におこなわれているので、プログラムのスピードが遅いという欠点がある。この JAVA 言語の欠点を回避するために、ネイティブコンパイラーを使用する。ネイティブコンパイラー

の働きは、C コンパイラーの動作と似ている。JAVA プログラムソースや、JAVA コンパイラーによって生成された中間コードから、プログラムを実行する計算機専用の命令を生成するものである。

ネイティブコンパイラーの性能測定について説明する。まず、図 2.2.1 のような、ノードが 1000 個含まれるリンクリストを作成する。各データは、前後のデータを指し示す要素(ポインター)を持っている(JAVA 言語には、ポインターと呼ばれるものは表面上存在しないが、データの格納位置を示すために内部的に使われている。このため、このようなデータ構造の作成も可能である)。このリンクリストに対し、次のような操作を実行する。

- リンクリストの先頭からデータを挿入し、先頭から削除
- リンクリストの先頭からデータを挿入し、最後から削除
- リンクリストの最後からデータを挿入し、先頭から削除
- リンクリストの最後からデータを挿入し、最後から削除

データの挿入と削除は、次のような操作である。

- リストに１つずつデータを挿入し、最終的に 1000 データ挿入する。
- リストから１つずつデータを削除し、最終的に 1000 データ削除する。

この、一連の操作を 1000 回実行するのに要した時間を、表 2.2.2に記す。

この表では、JDK1.0.2、JDK1.1.1、及び C コンパイラーとの性能比較結果を表している。javac ; java というのは、JDK で提供される JAVA コンパイラーとインタープリターの組み合わせである。TowerJ は JAVA ネイティブコンパイラーである。kaffe は、ランタイムコンパイラーと呼ばれるもので、実行時に中間コードを一挙に変換し実行するものである。これらに、オプティマイズ(プログラム構成の最適化)オプションを付け測定をおこなった。

この結果から、インタープリタ言語である JAVA でも、ネイティブコンパイラーを用いることで、スピード性能を上がることが可能である。

表 2.2.1：JAVA を用いた TCP/IP プログラミング例 [6]より転載

(1) Server program.

```
ServerSocket server = new ServerSocket(port);
Socket client = null;
client = serevr.accept();
DataInputStream in = new DataInputStream(client.getInputStream());
for(int i=0;i<loop;I++){
    realcount = in.read(bufr,o,length);
         :
         :
}
```

(2) Client program.

```
Socket client = new Socket(host,port);
DataOutputStream out = new DataOutputStream(client.getOutputStream());
for(int i=0;i<loop;i++){
    out.write(bufw,0,length);
        :
        :
}
```

表 **2.2.2**：**JAVA** 性能評価

JDK 1.0.2

javac ; java	**40.5 sec**
javac –O ; java	**40.5 sec**
TowerJ	**3.41 sec**
TowerJ –optimize	**0.81 sec**

JDK 1.1.1

javac ; java	**14.0 sec**
javac –O ; java	**14.0 sec**
javac ; kaffe	**3.4 sec**
javac –O ; kaffe	**3.4 sec**

C compiler

gcc	**1.22 sec**
gcc –O	**0.56 sec**

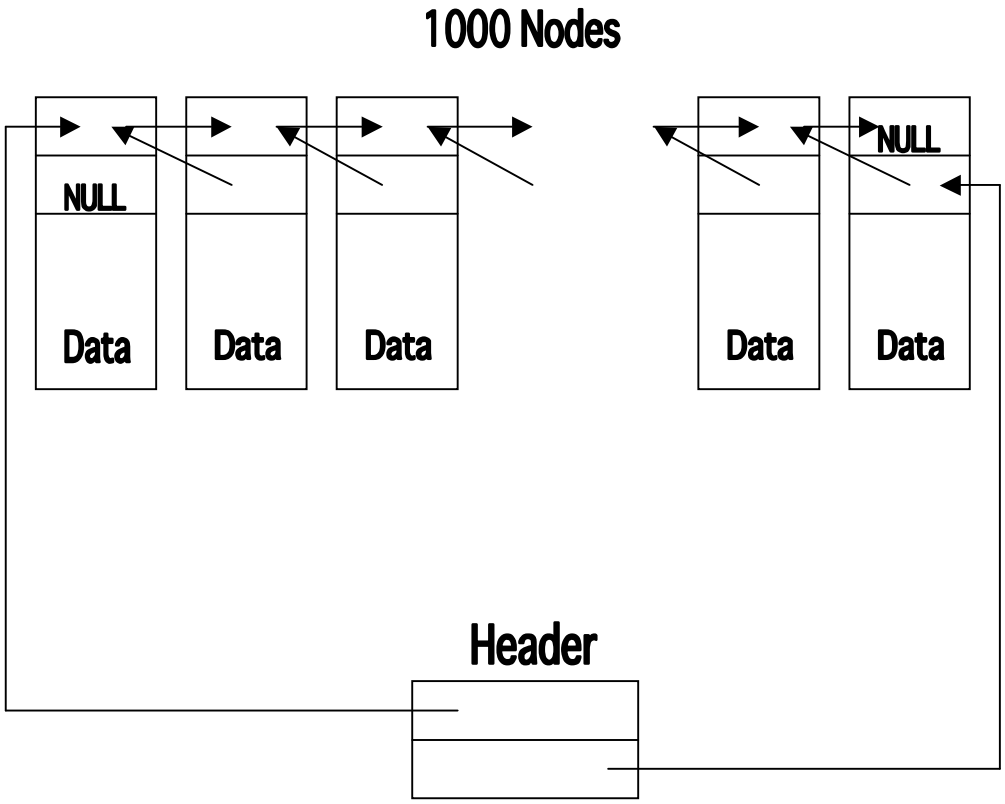


図 2.2.1 : Linked list data

Section 2.3 HORB

HORB は、JAVA 言語を用いてオブジェクト通信機構をおこなうためのアプリケーションである。このアプリケーションは、Object Request Broker(ORB) と HORB コンパイラーから構成される。ORB は、異なるホスト上に存在するオブジェクト間の、基本的な通信機能を提供するプロセスである。HORB コンパイラーは、JAVA 言語で記述されたプログラムをコンパイルし、Skeleton、Proxy オブジェクトを生成するプロセスである。JAVA のクラスは、HORB コンパイラーによって生成された、Skeleton、Proxy と呼ばれるオブジェクトを使用することで、オブジェクト通信実現している。また、HORB は完全に、JAVA 言語と互換である。このため、HORB コンパイラーによって生成されたプログラムは、JAVA インタープリタで実行することができる。すなわち、JAVA が動作する環境であればどの計算機でも稼働することができるのである。

サーバ・クライアントモデルを用いて、HORB の持つ機能の一部を説明する(図 2.3.1)。クライアントクラスは、サーバクラスのメソッドを呼び出し、両クラスは HORB コンパイラーによってコンパイルされているものとする。JAVA 言語では、クライアントクラスを実行すると、まず、クライアントクラスのインスタンス(クラスに含まれる、実装のコピーのようなもの)が生成される。このインスタンスは、HORB コンパイラーによって生成されたクライアント Proxy に対して、サーバー上のクラスにあるメソッドの要求をおこなう。サーバー側では、ORB を通じた Proxy からの要求により、サーバクラスのインスタンスと Skeleton オブジェクトが生成され、必要なメソッドが実行される。実行されたメソッドの返値は、Skeleton オブジェクトにより、ORB を通じて Proxy オブジェクトに渡され、必要の無くなったサーバインスタンスと Skeleton は消滅する。そして、最後に、クライアントインスタンスは、Proxy オブジェクトからメソッドの値を受け取ることができる。このように、他のマシン上にあるメソッドを実行する機能を、リモートメソッドコールと呼ぶ。

リモートメソッドコールには、大きく分けて２つの方法が存在する。

インスタンスの生成を伴うリモートメソッドコール(図 2.3.2上段)と、デーモンオブジェクトに対するリモートメソッドコール(図 2.3.2下段)である。

インスタンスの生成を伴うリモートメソッドコールの働きは、前述した通りである。このとき、クライアントの数が増えれば生成されるインスタンスの数も増加する。そして、サーバのクラスとインスタンスは、クライアントからのアクセスが無くなり次第、HORB の機能により消去される。JAVA には、クラス変数(クラスが管理するデータ)と、インスタンス変数(インスタンスが管理データ)と呼ばれる２種類のデータが存在する。インスタンスの生成を伴うリモートメソッドコールでは、クラス変数をクライアント間で共有し、インスタンス変数は各クライアント専用のものとなる。インスタンスの生成を伴うリモートメソッドコールは、JDK1.1.1 ではサポートされおらず、HORB 独自の機能である。また、このリモートメソッドコールは、DAQ モニターなどの、一時的に必要となったときのみに実行する処理をおこなうのに適しているといえる。

デーモンオブジェクトに対するリモートメソッドコールは、あらかじめサーバ上で起動した JAVA のクラス(デーモンオブジェクト)に対するアクセスである。この方法では、サーバインスタンスはただ１つしか生成されることはない。クライアントに対する処理は、インスタンス内部のスレッドと呼ばれる実行単位でおこなわれる。このときに注意しなくてはならないのは、インスタンス変数が、クライアント間で共有される点である。そのため、インスタンス変数に対する操作を、排他的におこなわなくてはならない。サーバ上で起動されたデーモンオブジェクトは、クライアントがある、ない、に関わらず常に動作を続ける。インスタンスの生成を伴うプロセスでは、クライアントがアクセスする度に、クラスオブジェクトやインスタンスの生成、消滅が発生する。すなわち、インスタンス(クラス)変数は、クライアントのアクセスが(始めて)発生した時点で初期化されてしまう。デーモンオブジェクトに対するリモートメソッドコールでは、このようなことはない。プログラムが、サーバ上で起動した時点で、クラス変数とインスタンス変数の初期化がおこなわれる。この機能は、データ収集プロセスや、データ記録プロセスのよう

に、アクセスするクライアントのある、ない、に関わらず実行を続ける処理をおこなうのに適しているといえる。

これらの、**HORB** の持つ機能は、**Command flow**、**Message flow**、**Data flow**、を作成するにあたり非常に有用であるといえる。また、**HORB** は **JAVA** と完全互換である**(HORB** 自体が **JAVA** 言語で作成されている**)**ため、**Section 2.2** で記した、**JAVA** ネイティブコンパイラーや、ランタイムコンパイラーを実行することも可能である。これらの点は、**DAQ** 作成に非常に有用である。

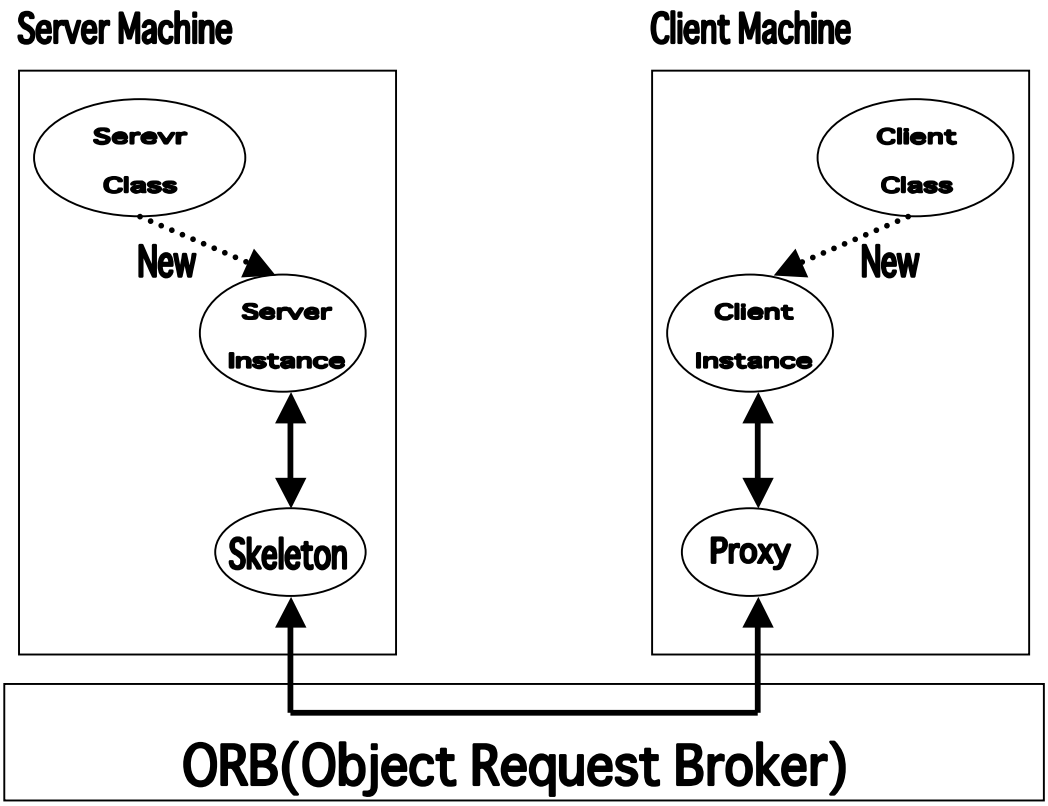


図 2.3.1：HORB アーキテクチャー

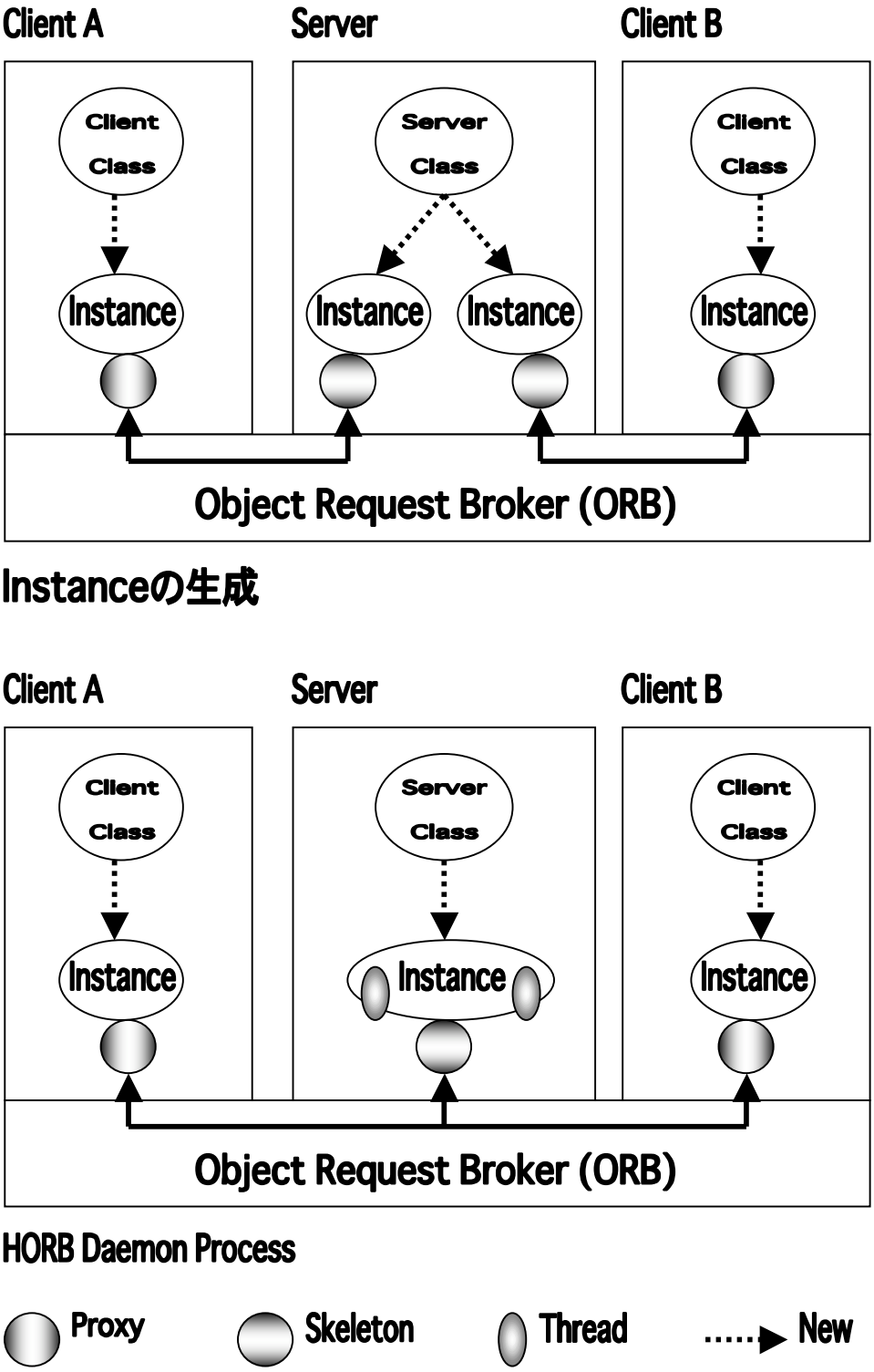


図 2.3.2：Remote Object の生成と Daemon Object

Chapter 3　CAMAC driver and library

Section 3.1　Crate controller and interface Card.

物理学実験では、観測される物理的事象、すなわち、さまざまな計測機器からの情報(デジタルシグナル)は、何らかの形でデジタル化され、コンピュータ上にデータとして保存される。高エネルギー物理学実験では、標準回路として CAMAC と呼ばれる規格のモジュールが使われる機会が多い。この CAMAC 規格のモジュールには、計測機器からの情報をデジタル化したり、NIM 規格のデジタルシグナルを出力したり、さまざまな働きをするものがある。これらの CAMAC モジュールを、CAMAC ファンクションを用いてコントロールするものが、CAMAC クレートコントローラと呼ばれるもので、現在、数社から何種類か製品化されている。今回、我々が用いたものは、CC7000 及び CC7700 と呼ばれている CAMAC クレートコントローラと PC 互換機用の ISA/CC7000、ISA/CC7700、PCI/CC7700 インターフェイスカード(インターフェイスボード)である(1998 年 1 月現在、市販されている CC7000、CC7700 用インターフェイスカードは，CC7000 用では ISA バス版が、CC7700 用には ISA バス用、及び、PCI バス用が存在する)。CC7000、CC7700 クレートコントローラの特徴を簡単に次に示す。

- CC7000 : インターフェイスカード(ISA/CC7000)とのデータの受渡しには 50 ピンフラットケーブルを用いる。このケーブルの長さはの制限が 2m 前後と短い。CAMAC 規格の 2 幅モジュールである。
- CC7700 : インターフェイスカード(ISA/CC7700、PCI/CC7700)とのデータの受渡しには 40 ピンフラットケーブルを用いる。ケーブルの長さは 50m まで延長可能。CAMAC 規格の２幅モジュール。CC7000 の後発機種。

これらのクレートコントローラは、対応するインターフェイスカードからのシグナルによって制御される。

インターフェイスカードは、ホストコンピュータの拡張バススロットに接続され、ホストコンピュータ上のアプリケーションと I/O ポートを

通して、データを受け渡すことができる。また、インターフェイスカードは、CAMAC クレートコントローラと物理的にケーブルで接続されている。このため、ホストコンピュータ上で動作しているアプリケーションは、I/O ポートを通して、CAMAC クレートコントローラの制御が行うことができる。

また、クレートコントローラは、CAMAC モジュールによって出された Look At Me(LAM)シグナルを関知し、インターフェイスボードに LAM が発生したことを伝えることができる。この時、インターフェイスボードは、割り込み信号を使うことによって、コンピュータに対し LAM が発生したことを知らせることができる。ISA バス用のインターフェイスカードでは、この割り込み信号の番号と、I/O ポートアドレスはカード上のディップスイッチを用いて設定する仕様であるが、PCI バス用インターフェイスカードでは、この設定はソフトウエアからおこなわれる。

このような、CAMAC クレートコントローラの制御を、PC/Linux 用にライブラリ化、デバイスドライバ化したものが、CAMAC ライブラリと、CAMAC ドライバである。CAMAC ライブラリと、CAMAC ドライバについては、Section 3.2、Section 3.3で記す。

Section 3.2 CAMAC Driver

UNIX 上のプロセスは、カーネルモードかユーザモードのどちらか１種類のモードで実行される。UNIX のプロセスは通常、優先度の低いユーザーモードで実行され、カーネルの機能が必要になったときに、カーネルモードで実行される。データ収集で必要となってくる、割り込み処理や、DMA 操作などは、カーネルに何らかの形で組み込まれた、デバイスドライバの中でしか行うことができない。割り込み処理や DMA 処理を必要でないのなら、デバイスドライバは必ずしも必要ではない。しかし、カーネルモードは、ユーザーモードに比べて非常に優先度が高いモードである。このことを利用して、複数の CAMAC アクションをデバイスドライバで実行させる、CAMAC List 処理機能[1]といった手法もある。

今回作成した CAMAC ドライバは、CC7000+ISA/CC7000、CC7700+ISA/CC7700、CC7700+PCI/CC7700 の３種類である。主な CAMAC ドライバの働きは、デバイスドライバで補足する割り込み信号の設定(ISA)、インターフェイスカードで使用する I/O ポートの設定(PCI 版)、割り込みの信号の処理(ISA)、CAMAC ライブラリの初期化、などである。通常 UNIX では、デバイスドライバはカーネルの一部として作成され、簡単にカーネルから切り離すことはできない。しかし、Linux OS には、ローダブルモジュールと呼ばれる機能がある。この機能は、デバイスドライバが必要になった時にロードを行い、不必要になったときにそれを取り除くことができる機能である。PC/Linux 用の CAMAC デバイスドライバは、ローダブルモジュール形式として作成された。

物理学実験を行う上で重要となるのは、LAM が生じてから、デバイスドライバで割り込みを検出して、しかるべき処理を行いユーザープロセス(CAMAC ライブラリルーチン)まで戻ってくるまでにかかる時間(Interrupt task response time)である。この時間は、CPU の負荷状態などで変わってくるもので、今回、作成した CC7700 用ドライバでは、３つの状態で Interrupt Task Response Time を測定した(表 3.2.1)。

３つの状態は、それぞれ、

- CPU の負荷が少ない状態
- CPU consumer プロセスを同時に実行している状態

- **I/O consumer** プロセスを同時に実行している状態

である。

ここで、**1** つ目の状態は、“**CPU** の負荷の少ない状態”であって、“**CPU** の負荷のない状態”ではない。最低限の、**Linux OS** を維持していくためのデーモンプロセスや、**X**、ウインドウマネージャーなどを起動している状態である。**2** つ目の状態は、**CUP consumer** プロセス**(**表 **3.2.2)**を測定中に同時に実行させる。そして、**3** つ目の状態は、**I/O consumer** プロセス**(**表 **3.2.3)** を、測定中に同時に実行させることで計測を行った。

PCI/CC7700 ドライバは、ソフトウエア上からインターフェイスカードの初期化**(I/O** ポートアドレスの設定、割り込みシグナルの設定**)**を行っているが、空いている **I/O** ポートアドレスや割り込みシグナルを自動的に検出し、設定するようには設計されていない。使用する人間は、デバイスドライバのコンパイル時に、そのアドレスなどを指定しなければならない。また、割り込み処理については、サポートされていない。

今後の改良点は、**CAMAC List** 処理機能を持たせること、及び、**PCI/CC7700** デバイスドライバに、プラグアンドプレイのような自動検出機能と、割り込み処理機能を持たせることなどが挙げられる。

表 3.2.1：Interrupt task response time.

CC7700 CAMAC driver for Linux OS.

Linux OS version 1.2.13.

CPU：P5-120MHz , Memory：48 Mbytes , Cache：256 Kbytes)

CPU consumer process	Mean Time
None	45.1 μ sec
CPU consumer	50.5 msec
IO consumer	998 msec

表 3.2.2：CPU consumer process. [2]より転載

```
void main(void){
        int i = 0;
        while(1){
                i++;
        }
}
```

表 3.2.3：I/O consumer process. [2]より転載

```
#!/bin/csh
while(1)
        ls -lR >& /dev/null
done
```

Section 3.3 CAMAC library

CAMAC ライブラリの実装を、表 3.3.1で示す。O 印の付いている関数は、ライブラリで実装されていることを示す。このライブラリ中の関数の名前、引数、返値といった仕様は、全て、KEK スタンダードの他のライブラリと同じである。これにより、他の KEK スタンダードのライブラリと、互換性が保たれている。CC7000/ISA、CC7700/ISA、CC7700/PCI の３つのインターフェイスカードには DMA 機能がないので、このライブラリは、擬似的に DMA 機能をソフトウエア上で実現している。CC7700/PCI ドライバは、Section 3.2で前述したように、割り込み処理はサポートされていない。従って、CC7700/PCI CAMAC ライブラリでも、割り込み処理に関連する関数はサポートしていない。

CAMAC モジュールにアクセスするための関数は、CAMAC、CAMACW、CDMAL、CDMAW と呼ばれる関数である。CAMAC、CAMACW を CAMAC シングルアクションと呼び、CDMAL、CDMAW を CAMAC ブロックアクションと呼ぶ。PC/Linux 用の CAMAC ライブラリでは、ブロックアクションが、疑似 DMA 操作となっている。この関数は、Ignore-Q、Q-Stop、Q-Repeat、Address-Scan という４つのモードで、連続的に CAMAC モジュールに対して操作を行うことができる。

DAQ に重要になってくるのは、CAMAC モジュールをどれくらいの早さでアクセスできるかということである。この性能については、表 3.3.2で示す。この表のデータは、全て 16 ビットデータサイズで CAMAC モジュールに対してアクセスをしたものである。この結果、オーバーヘッド(関数呼び出しにかかる時間)も考慮に入れても、２回以上の連続した CAMAC アクセスについては、ブロックアクションを用いた方がデータ転送の割合が高いことが判った。

また、表 3.3.3は、KEK から公開されている、他のドライバの性能評価表である。CC7x00 CAMAC ライブラリでは、DMA 機能を擬似的にソフトウエアで表現している。このため、CAMAC ブロックアクションについては、他のドライバと比較するとデータ転送性能が低いことがわかる。しかし、

CAMAC シングルアクションを比較すると、短時間で実行可能であることがわかる。

表 3.3.1 : **Functions list of CAMAC library.**

Name	Contents	CC7000 (ISA)	CC7700 (ISA)	CC7700 (PCI)
COPEN	Open CAMAC crate	O	O	O
CCLOSE	Close CAMAC crate	O	O	O
CRESET	Reset CAMAC crate.	X	O	O
CSETCR	Set current CAMAC crate No.	O	O	O
CGENZ	Initialize CAMAC crate.	O	O	O
CGENC	Clear CAMAC crate.	O	O	O
CSETI	Set inhibit line.	O	O	O
CREMI	Remove inhibit line.	O	O	O
CAMAC	Execute CAMAC function. (24 bit data)	O	O	O
CAMACW	Execute CAMAC function. (16 bit data)	O	O	O
CDMAL	Execute CAMAC block function. (24 bit data)	O	O	O
CDMAW	Execute CAMAC block function. (16 bit data)	O	O	O
CENLAM	CAMAC enable LAM.	O	O	O
CDSLAM	CAMAC disable LAM.	O	O	O
WTLAM	Waiting LAM is occurred.	O	O	X

表 3.3.2：Performance of CAMAC access. CC7x00.

[CC7000/ISA , CC7700/ISA]

Linux 1.2.13 , P5-120MHz , Memory : 48 Mbytes , Cache : 256 Kbytes

[CC7700/PCI]

Linux 2.0.29 , P5-166MHz , Memory : 64 Mbytes , Cache 256Kbytes

Machine			P5-120MHz		P5 166MHz
OS			Linux 1.2.13		Linux 2.0.29
Crate Controller			CC7000 (ISA)	CC7700 (ISA)	CC7700 (PCI)
Single Action (μ sec)		Read	7.8	11.2	
		Write	7.9	11.3	
		NDT	6.2	8.1	
Block Action	Read	Overhead(μ sec)	8.2	9.5	
		Speed(Kbyte/sec)	444	250	
	Write	Overhead(μ sec)	9.4	9.7	
		Speed(Kbyte/sec)	526	250	
Interrupt Handling (μ sec)			---	45.1	

表 3.3.3：Performance of CAMAC access. Kinetic 2917. [5]より転載

Machine			HP9000 /743	DEC3000/400	Sparc2
OS			HP-RT V2.0	OSF1 V1.3	Sun OS 4.1.2
Single Action (μ sec)		Read	22	96	125
		Write	20	90	130
		NDT	15	70	120
Block Action	Read	Overhead(μ sec)	95	370	720
		Speed(Kbyte/sec)	920	1000	1020
	Write	Overhead(μ sec)	95	380	720
		Speed(Kbyte/sec)	850	530	810
Interrupt Handling (μ sec)			70	200	---

Chapter 4　JAVA DAQ

Section 4.1　Members of JAVA DAQ

JAVA DAQ は、次の６つのプロセスから構成されている。

- **Collector :** データ収集プロセス
- **Cache :** データバッファ
- **Recorder :** データセーブプロセス
- **Analyzer :** データ解析プロセス
- **Commander : DAQ** コントローラ
- **Monitor + GUI : DAQ** モニター

これらのプロセスは、**Data flow** に **JAVA TCP/IP** ソケットを用い、**Command flow**、及び、**Message(Status) flow** に **HORB** を用いている**(**図 **4.1.1)**。このため、基本的に全てのプロセスは、コンピュータの種類や、**OS** の種類にとらわれることなく、ネットワーク上の全ての **JAVA** アプリケーションが動作可能なマシンで実行可能となっている。ただし、**Collector** や、**Analyzer** のように、計算機に依存するライブラリを、ネイティブメソッドとして使用しているものについては、その部分を使用する計算機や **OS** に合わせて作成しなくてはならない。

JAVA DAQ の、**Collector**、**Cache**、**Recorder,**、**Analyzer**、**Commander** の５つのプロセスは、自分自身を **HORB** デーモンオブジェクトとして **ORB** に登録をおこない、デーモン化して動作をおこなう。**DAQ** が動作している間、これらのプロセスは、**Initial**、**Neutral**、**Busy**、**Dead** という４種類の状態のうち、１つの状態を保存している。起動した時点では、**Initial** 状態であるが、**Commander** プロセスからの、**Begin**、**Resume**、**Pause**、**End**、**Shutdown** という５種類の命令を受けて、状態遷移を引き起こすよう設計されている**(**図 **4.1.2)**。これらの、各プロセスの実装の内容は、**Section 4.2**から**Section 4.7**で記す。

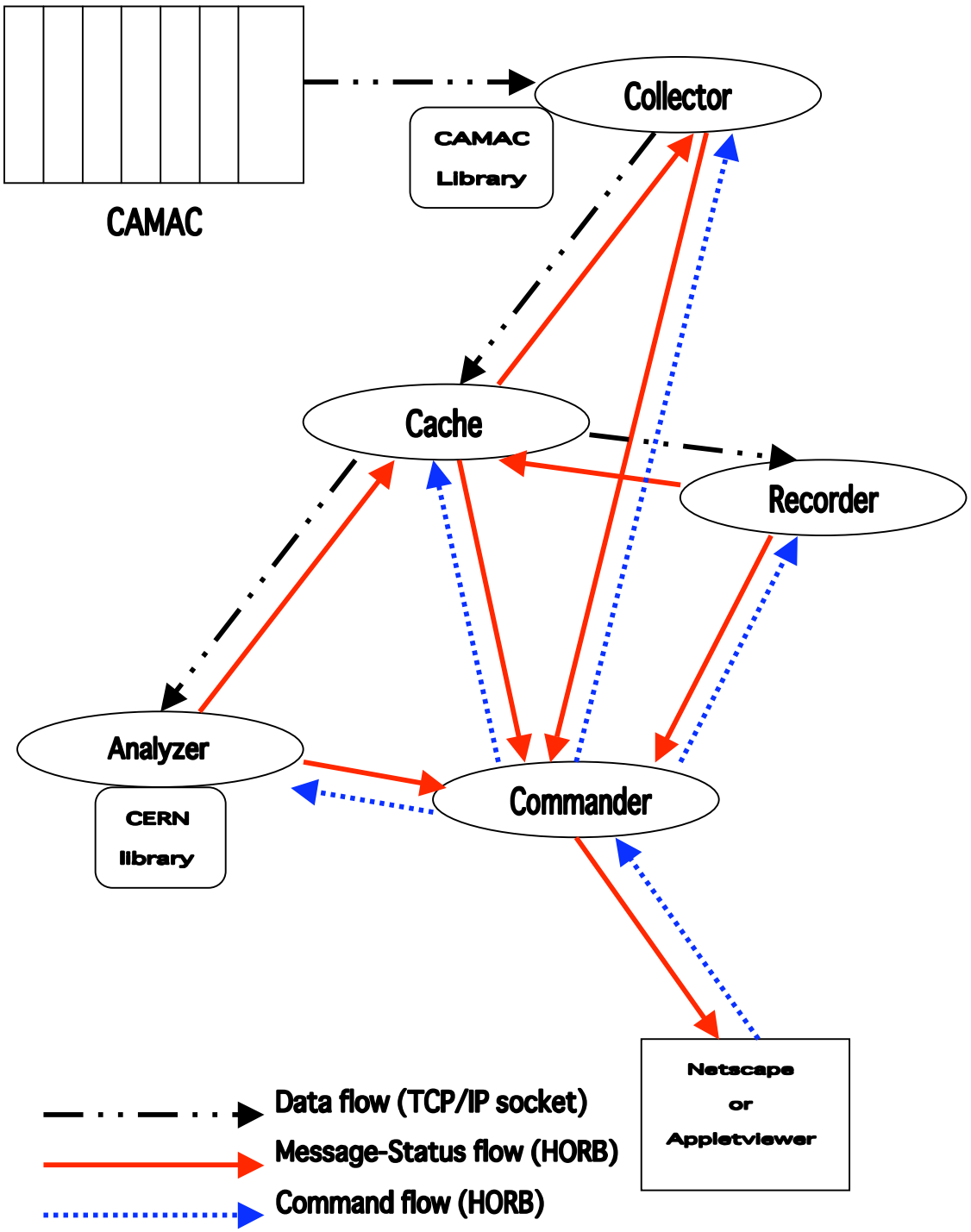


図 4.1.1 : Java DAQ Model

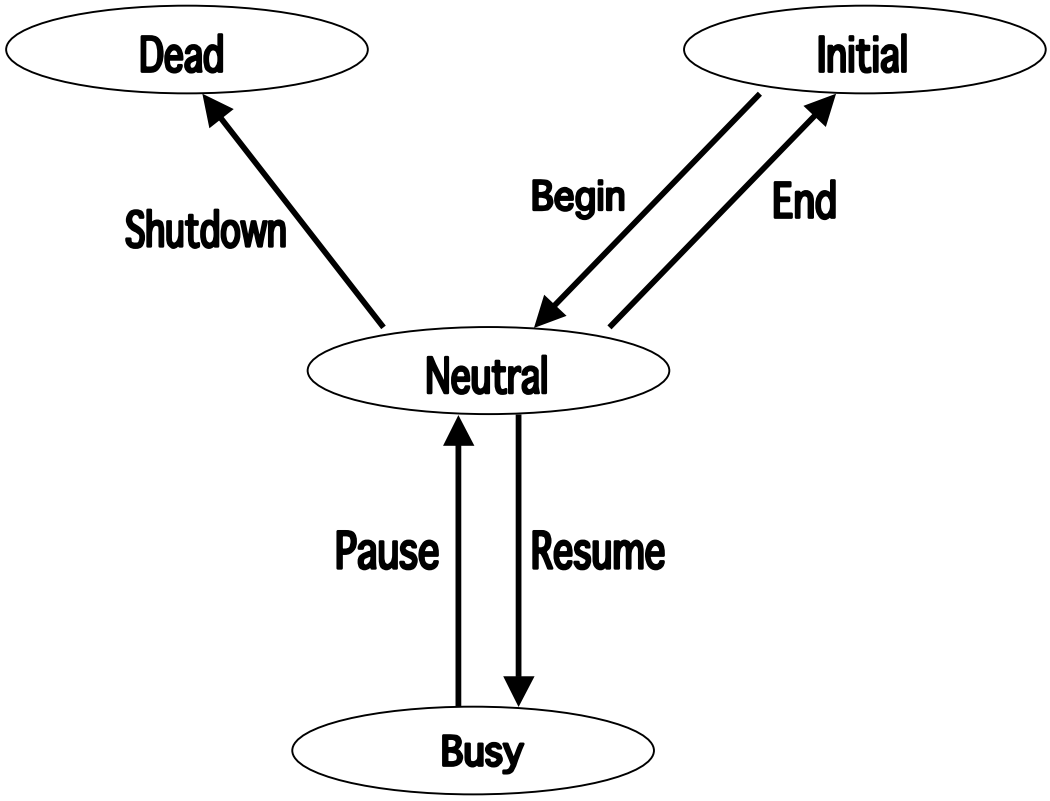


図 **4.1.2**：**JAVA DAQ** 状態遷移図

Section 4.2 Collector process

Collector プロセスは、データを収集するプロセスである。このプロセスは、Chapter 3で記した、CAMAC ライブラリを、Native メソッドとして取り込んである。CAMAC ライブラリは、C 言語用のライブラリであるので、JAVA 言語から呼び出すことができるような形式にしなければならない。考えられる手法はいくつか存在する(図 4.2.1)が、今回は、CAMAC ライブラリにラッパー関数をかぶせ、この関数と CAMAC ライブラリを一体化したネイティブメソッドを組み込む手法を用いた。この手法の利点は、１回の関数呼び出しで、必要な CAMAC モジュールからのデータを全て読むようにラッパー関数を記述することができること、及び、CAMAC ライブラリを改造することなくそのまま使用できることである。

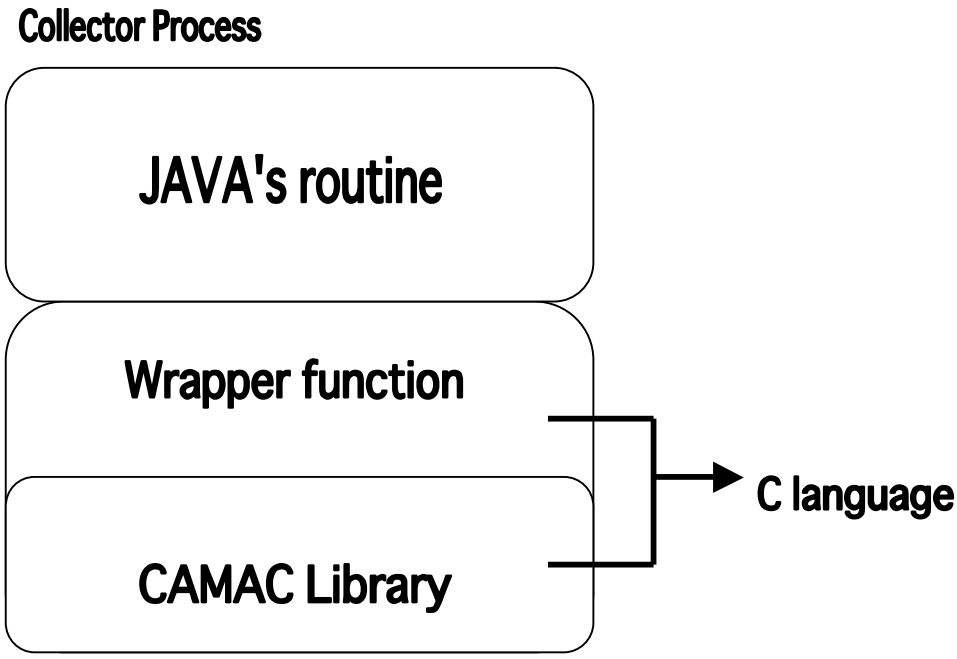
Collector プロセスは、起動時に自分自身をデーモンオブジクトとして ORB に登録し、Cache プロセスとのデータ通信に用いるソケットを準備する。起動直後のプロセスの状態は、Initial である。この状態は、Begin、Resume コマンドにより Busy 状態に遷移する。Begin コマンドには、ラン番号、収集するデータの最大イベント数、などの情報が含まれいる。Collector プロセスは、これに従ってデータを収集の準備をおこなう。

Collector プロセスは、Resume コマンドによって Busy 状態に遷移すると、実際にデータの収集を開始する。Collector によって収集されたデータは、イベント単位でソケットを通して Cache プロセスに送られる。その後、Collector プロセスは、HORB の機能を用いて、Cache プロセス内にある内部変数を加算する。この内部変数は、Collector プロセスが何イベント(バイト数ではない)のデータをソケットに対して送出したかを示す値で、Cache プロセスがソケットからイベントデータを受け取るときに参照される。この、収集したデータを Cache プロセスに渡す一連の流れは、Begin コマンドとともに渡された最大イベント数と、収集したデータイベント数が等しくなるまでか、Pause コマンドが送られてくるまで続けられる。最大イベント数と収集したイベント数が等しくなって、データ収集が止まっても Collector プロセスは Busy 状態を保つ。ただ単にデータ収集がストップす

るだけである。

Pause コマンドにより、**Collector** プロセスは **Neutral** に遷移する。この状態で、**end** コマンドを受けると、**Initial(**新たなランデータの収集開始待ち**)**状態に遷移する。また、**Shutdown** コマンドを受けると、**Collector** プロセスは、自分自身の **ORB** への登録を取り消し、プロセスを終了させる。

(1) CAMAC library に Wrapper 関数をかぶせる。



(２) CAMAC library の関数を JAVA に合うようにする。

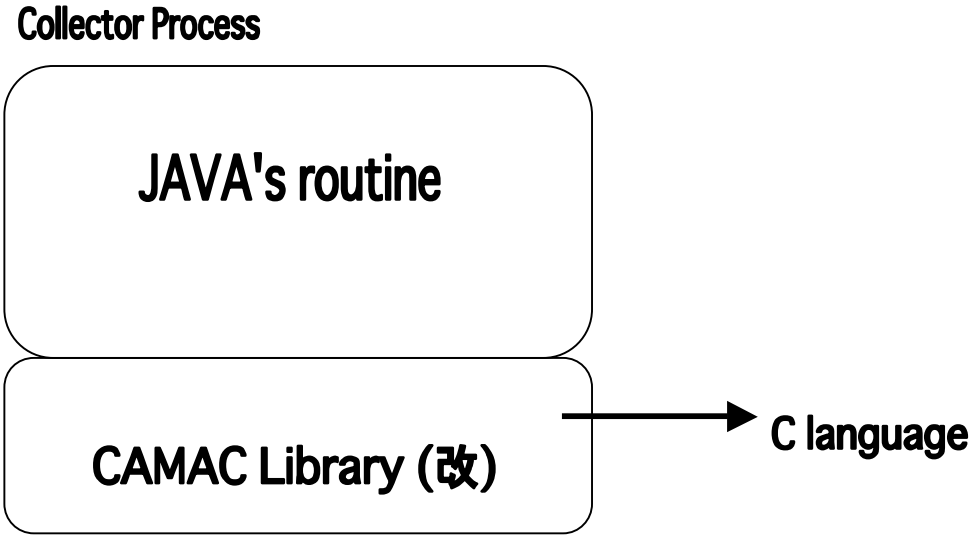


図 4.2.1 : Native method of collector process.

Section 4.3 Cache process

Collector によって収集されたイベントデータは、**Cache** プロセスによって **Analyzer**、**Recorder** プロセスに渡される。**Cache** プロセスは、データの流れの分岐を管理するプロセスである。

Cache プロセスは **Collector** プロセスと同じく、起動時は **Initial** 状態で始まる。起動すると、自分自身を **ORB** に登録しデーモンオブジェクト化する。そして、**Collector** プロセスの作成したソケットに接続をおこない、次に、**Recorder**、**Analyzer** プロセスとデータ通信をおこなうためのソケットを、それぞれ作成する。このため、**Cache** プロセスの起動は、**Collector** プロセスが起動した後からおこなわなくてはならない。

Initial 状態にある **Cache** プロセスは、**Begin**、**Resume** コマンドによって **Busy** 状態に遷移する。**Busy** 状態に遷移した **Cache** プロセスは、**Collector** プロセスによって加算された内部変数の参照を開始する。この値がゼロでない間、**Collector** によって送られたイベントデータをソケットから読み込む。イベントデータをソケットから読み込む度に、**Cache** プロセスは内部変数を減算する。このような仕組みで、常に、この内部変数の値は、ソケット内に何イベントのデータが残っているか示している。読み込まれたイベントデータは、ソケット を通して **Recorder** プロセスに送られる。その後、**Cache** プロセスは、**HORB** の機能を用いて、**Recorder** プロセス内にある内部変数を加算する。また、**Cache** プロセスは、**HORB** を用いて **Analyzer** プロセスの内部状態を調べる。そして、**Analyzer** プロセスがデータを必要としている状態の場合のみ、ソケットを通して **Analyzer** にイベントデータを送る。イベントデータを送った後は、**Recorder** プロセスに対する処理と同様に、**Analyzer** プロセスの内部変数を加算する。

Busy 状態にある **Cache** プロセスは、**Pause** コマンドによって **Neutral** 状態に遷移する。**Neutral** 状態に遷移する前に、**Cache** プロセスは必ず、**Collector** から送られてきたソケット内のイベントデータを全て読み込み、**Recorder**、**Analyzer** プロセスに送るように設計してある。これによって、**Collector** プロセスから送られてきたイベントデータは、ソケットを通して全て **Recorder**

に渡される。

Neural 状態にある **Cache** プロセスは、**Collector** プロセスと同じく、**end** コマンドを受けると、**Initial** 状態に遷移する。また、**Shutdown** コマンドを受けると、自分自身の **ORB** への登録を取り消し、プロセスを終了させる

Section 4.4 Recorder process

Recorder プロセスは、**Cache** プロセスから送られてきたイベントデータを記録するプロセスである。イベントデータを記録する媒体は、**HDD**、**DAT** などが可能である。このとき **JAVA** 言語では、データはビッグインディアント**(**数値の上位桁からの順**)**で保存される。一方、今回用いた計算機のハードウエア的な構造上、**C** 言語などでは、データはリトルインディアント**(**数値の下位桁からの順**)**で保存される。このため、C言語などの解析用プログラムを作成する場合、データコンバートをおこなう必要が生じる。

Recorder プロセスも、他のプロセスと同じく、起動時は **Initial** 状態で始まる。起動すると、自分自身を **ORB** に登録しデーモンオブジェクト化し、**Cache** プロセスの作成したソケットに接続をおこなう。このため、**Recorder** プロセスの起動は、**Cache** プロセスが起動した後からおこなわなくてはならない。

Recorder プロセスは、**Begin** コマンドによって、**Initial** 状態から **Neutral** 状態に状態遷移をする。このとき **Recorder** プロセスは、与えられたラン番号を基にした名前で、データファイルを書き込み用に作成する。そして、**Commander** プロセスから送られた、ラン番号、最大イベント数、ランに関するコメントをデータファイルのヘッダに書き込む。これらの情報は、後で解析をおこなうときに参照することができる。

Resume コマンドによって状態は、**Busy** に遷移する。**Busy** 状態になった **Recorder** プロセスは、内部変数を参照する。この値は、イベントデータがソケットに送られる度に、**Cache** プロセスによって加算される値である。**Recorder** プロセスは、この値がゼロではない間、イベントデータをソケットから読み込み、値の減算をおこなう。

Busy 状態は、**Pause** コマンドによって **Neutral** 状態に遷移する。**Neutral** 状態に遷移する前に、**Recorder** プロセスは必ず、**Cache** プロセスから送られてきたソケット内のイベントデータを全て読み込みデータファイルに記録する。これによって、**Cache** プロセスから送られてきたイベントデータは、全てデータファイルに記録される。

Neural 状態にある **Recorder** プロセスは、他のプロセスと同じく、**end** コ

マンドを受けると、**Initial** 状態に遷移する。このときにデータファイルはクローズされる。また、**Shutdown** コマンドを受けると、自分自身の **ORB** への登録を取り消し、プロセスを終了させる。

Section 4.5 Analyzer process

Analyzer プロセスは、オンラインアナリシスをおこなうプロセスである。オンラインアナリシスとは、DAQ を用いてデータ収集しつつ、そのデータを表示、及び解析をおこなうことである。これにより、DAQ で収集をおこなっているデータが、本当に目的とするデータなのかその場でチェックをおこなうことができる。また、収集したデータをその場で視覚的に表示できるため、CAMAC モジュールや検出器の故障などの問題点が、容易に発見できる。

Analyzer プロセスを実現する手法は、さまざまな方法を選択できる。

まず１つ目としては、Analyzer プロセスを JAVA アプレットで作成する方法がある(図 4.5.1)。この方法では、データの解析部分を自分で作成する必要がある。データの表示には、Netscape やアプレットビュアーといった既存のアプリケーションを用いることができる。図 4.5.2は、JAVA アプレットを用いて、CAMAC のスケーラーモジュールのデータを表示しているだけであるが、実際には、解析用のルーチンなどを加えて、Netscape などのビュアーから解析が行えるようにする事も可能である。２つ目は、CERN ライブラリといった、既存の解析用プログラムを用いる方法である。この方法の長所は、複雑な解析プログラムを作成する必要がなく、解析ツールの機能を用いるだけでよい点である。欠点としては、解析ツールが実行可能な環境で Analyzer プロセスを実行しなくてはならないということである。他には、解析ツールとビュアーを自分で作成する方法もあるが、これは非常に手間のかかる方法である。

JAVA-DAQ Prototype の Analyzer プロセスは、CERN ライブラリを用いる方法で作成した(図 4.5.3)。CERN ライブラリには、PAW、PAW++といった解析ツールがあり、このツールに対してフォートランを用いて記述されたプログラムから、データを渡すことが可能である。そこで、Analyzer プロセスでは、Fortran で記述された CERN ライブラリとのインターフェイス部分を、ネイティブメソッドとして取り込むことにした。Analyzer プロセスは、実際には解析をおこなわず、CERN ライブラリーに対してデータを渡すだけの、インターフェイスの働きしかしない。解析をおこなうのは、あくまで CERN ライブラリであり、ユーザは PAW、PAW++といった解析ツー

ルを使用するだけで、**Analyzer** がどのように動作しているかを、特に意識する必要はない。

Analyzer プロセスは、**Cache** プロセスから送られてきたイベントデータを **CERN** ライブラリへ渡すプロセスである。**Analyzer** プロセスも、他のプロセスと同じく、起動時は **Initial** 状態で始まる。起動すると、自分自身を **ORB** に登録しデーモンオブジェクト化し、**Cache** プロセスの作成したソケットに接続をおこなう。このため、**Analyzer** プロセスの起動も、**Cache** プロセスが起動した後からおこなわなくてはならない。

Analyzer プロセスは、他のプロセスとは異なり、モニターリングを抑制するためのフラグを内部に持っている。このフラグによって、モニターリングの **ON・OFF** が切り替えることが可能である。このフラグが **OFF** である間は、**Cache** プロセスからデータが送られてくることはないし、**Analyzer** プロセスもデータを読み込もうとはしない。この、モニターリングフラグによって、**Cache** プロセスの負荷を低減さえることができ、また、１台のコンピュータで **DAQ** を動かすような場合に、計算機の負荷も押さえることが可能である。モニターリングフラグは、**Commander** プロセスによって変化させられる。

Analyzer プロセスは、**Begin** コマンドによって、**Initial** 状態から **Neutral** 状態に状態遷移をする。このとき **Analyzer** プロセスは、**CERN** ライブラリのヒストグラムデータを初期化する。

Resume コマンドによって状態は、**Busy** に遷移する。**Busy** 状態になった **Analyzer** プロセスは、内部変数を参照する。この値は、イベントデータがソケットに送られる度に、**Cache** プロセスによって加算される値である。**Analyzer** プロセスもまた、この値がゼロではない間、イベントデータをソケットから読み込み、値の減算をおこなう。ソケットから得たデータは、随時 **CERN** ライブラリに送られる。

Busy 状態は、**Pause** コマンドによって **Neutral** 状態に遷移する。**Neutral** 状態に遷移する前に、**Analyzer** プロセスは必ず、**Cache** プロセスから送られてきたソケット内のイベントデータを全て読み込み **CERN** ライブラリにデータを渡す。これによって、**Cache** プロセスから送られてきたイベント

データは、全て **CERN** ライブラリに渡される。

Neural 状態にある **Analyzer** プロセスは、他のプロセスと同じく、**end** コマンドを受けると、**Initial** 状態に遷移する。また、**Shutdown** コマンドを受けると、自分自身の **ORB** への登録を取り消し、プロセスを終了させる。

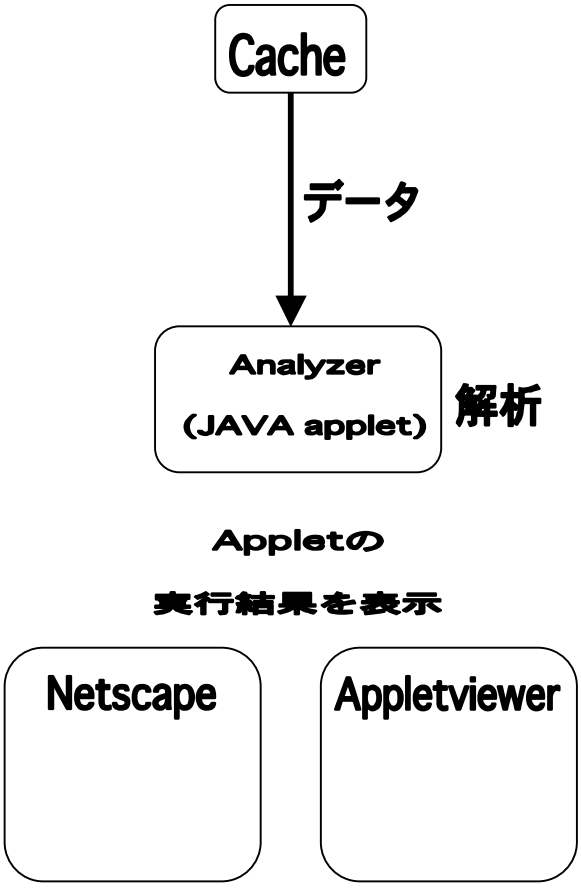


図 4.5.1：Analyzer(JAVA アプレットを用いる)

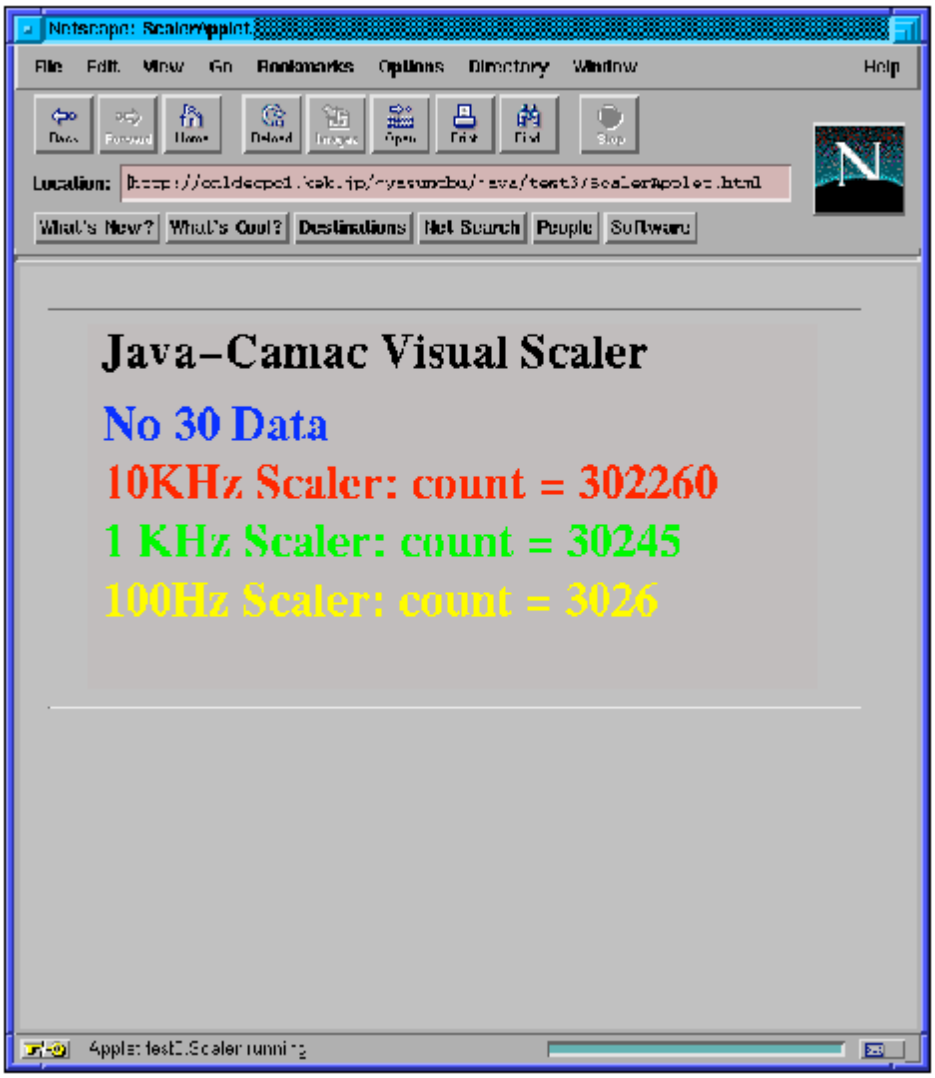


図 **4.5.2**：**Netscape** を用いたデータビュアー

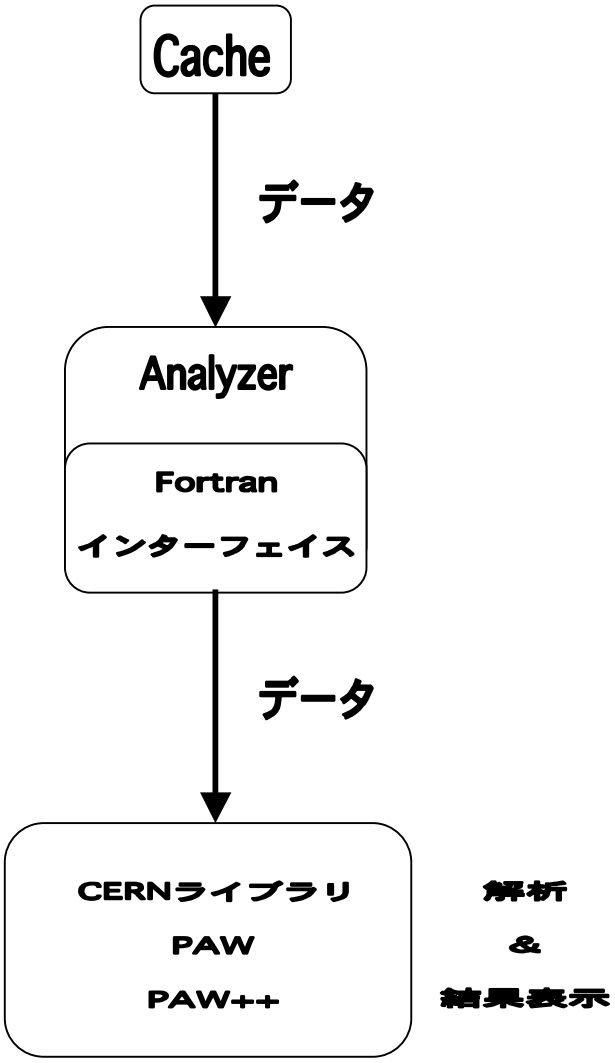


図 4.5.3：CERN ライブラリをネイティブメソッドとする

Section 4.6 Commander process

Commander プロセスは、**Collector** プロセス、**Cache** プロセス、**Recorder** プロセス、**Analyzer** プロセスの状態を遷移させるプロセスである。**Commander** プロセスは、**HORB** を用いて、各プロセスが保存している内部変数を変更している。この内部変数は、各プロセスの状態を表す変数である。また、**Commander** プロセスは、各プロセスの内部状態のチェックをおこなうことが可能である。これらの機能は、**Commander** プロセスのメソッドとして定義されている。このため、**HORB** を用いて他のプロセスから、これらのメソッドを呼び出すことで、各プロセスの状態遷移を引き起こすことが可能である。

Commander プロセスも、他のプロセスと同じく起動時は **Initial** 状態である。また、起動時には自分自身を **ORB** に登録し、デーモンオブジェクト化する。その後、**Commander** プロセスは、他のプロセスの状態が全て **Initial** 状態かどうかをチェックする。このために、**Commander** プロセスの起動は、全てのプロセスを起動した後でなくてはならない。他のプロセスの状態が、全て **Initial** となっていることを確認すると、自分自身も **Initial** 状態に設定する。

モニタープロセスなどの他のプロセスから状態遷移のメソッドが呼び出されると、**Commander** プロセスは、自分自身の状態をチェックする。そして、与えられたコマンドが、現在の状態で受け付けることができるときのみ、各プロセスの状態を変化させる。各プロセスの状態を変化させた後は、各プロセスが全て新しい状態に遷移したことを確認し、**Commander** プロセスは、自分自身のプロセスの状態を変化させる。**Commander** プロセスが、あるプロセスの状態を遷移させ、そのプロセスが実際に遷移するには時間差が生じる。このため、再確認をおこなわないで終了してしまうと、まだ全てのプロセスが遷移していないにも関わらず、新しいコマンドを受け付けてしまう。こうなってしまうと、**DAQ** の各プロセスは協調的に動作しなくなる。

Begin コマンドは、ラン番号、収集最大イベント、**Run** に関するコメン

トといった引数が必要となる。この引数をもとに、**Collector** プロセスや、**Recorder** プロセスは自分自身の初期状態を設定する。

Shutdown コマンドは、**DAQ** の全ての機能を停止させるコマンドであるので、実行に際しては注意しなくてはならない。そのため、**Shutdown** コマンドには必ずパスワードが必要となるように設計している。**Commander** プロセスは、**Shutdown** コマンドを受け取ると、まず、パスワードが正規のものであるかチェックする。その後、自分自身の状態をチェックし、**Shutdown** が可能であれば他のプロセスに対し、**Shutdown** コマンドを送る。コマンドを送った後、**Commander** プロセスは、**ORB** の登録を取り消し、プロセスが終了となる。

Section 4.7 Monitor process

Monitor プロセスの働きは、**HORB** を用いて **Commander** プロセスのメソッドを呼び出すだけである。**Commander** プロセスのメソッドが呼び出すことさえできればいいので、**JAVA** アプリケーションでも、**JAVA** アプレットのどちらでもかまわない。

図 **4.7.1**は、**JAVA** アプレットを用いた **Monitor** プロセスの１例である。ビュアーには **Appletviewer** を用いている。

画面内、上２段の白い部分がテキストウインドウと呼ばれる部分で、ラン番号、最大イベント、**Run** に関するコメントを書き込む部分である。この部分に必要事項を書き込み、**Begin** ボタン**(**３段目、左から１番目のボタン**)**を押すと、**Begin** コマンドがテキストウインドウの内容と共に **Commander** プロセスに送られる。

３段目の４つのボタンは、それぞれ **Begin**、**Resume**、**Pause**、**End** コマンドを **Commander** プロセスに送信するコマンドである。

４段目の３つのボタンは、**Analyzer** のモニターリング状態を制御するためのボタンである。左側から２つのボタンでモニターリングの **ON・OFF** を切り替える。３つ目のボタンは、解析プログラムに渡したデータを初期化し、新たに送られてくるデータで解析をおこなうためのボタンである。５段目には、そのときに設定されている **Analyzer** プロセスのモニターリング状態が表示される。

５段目の幅の広い黒い部分は、**DAQ** 全体の情報を表示する部分である。この部分の上半分には、テキストウインドウで設定された各種情報が表示され、下半分には各プロセスの情報が表示される。下半分で表示される内容は、

- **Collector : Collector** プロセスで収集され、**Cache** プロセスに送られたイベントデータ数。
- **Cache : Cache** プロセスが、**Collector** プロセスから他のプロセスへ送ったイベントデータ数。
- **Recorder : Recorder** プロセスが実際に記録媒体に保存したイベントデー

タ数

- **Analyzer : Analyzer**プロセスが、解析プログラムに送ったイベントデータ数と、モニターリングの状態**(ON・OFF)**。

であり、これらの情報は **10** 秒前後で更新される。

１番下の段は、**Shutdown** コマンドを送信するためのものである。**Shutdown** コマンドは、**DAQ** の全ての機能を停止させるコマンドであるので、パスワードを記入するためのテキストウインドウを追加してある。このテキストウインドウに記述されたパスワードは、**Shutdown** コマンドと共に **Commander** プロセスに送られる。

１例として、**Monitor** プロセスの機能を上に記したが、これらの機能は必ず必要なものではない。実験の目的などによって、ユーザが機能を追加、削除していく必要がある。

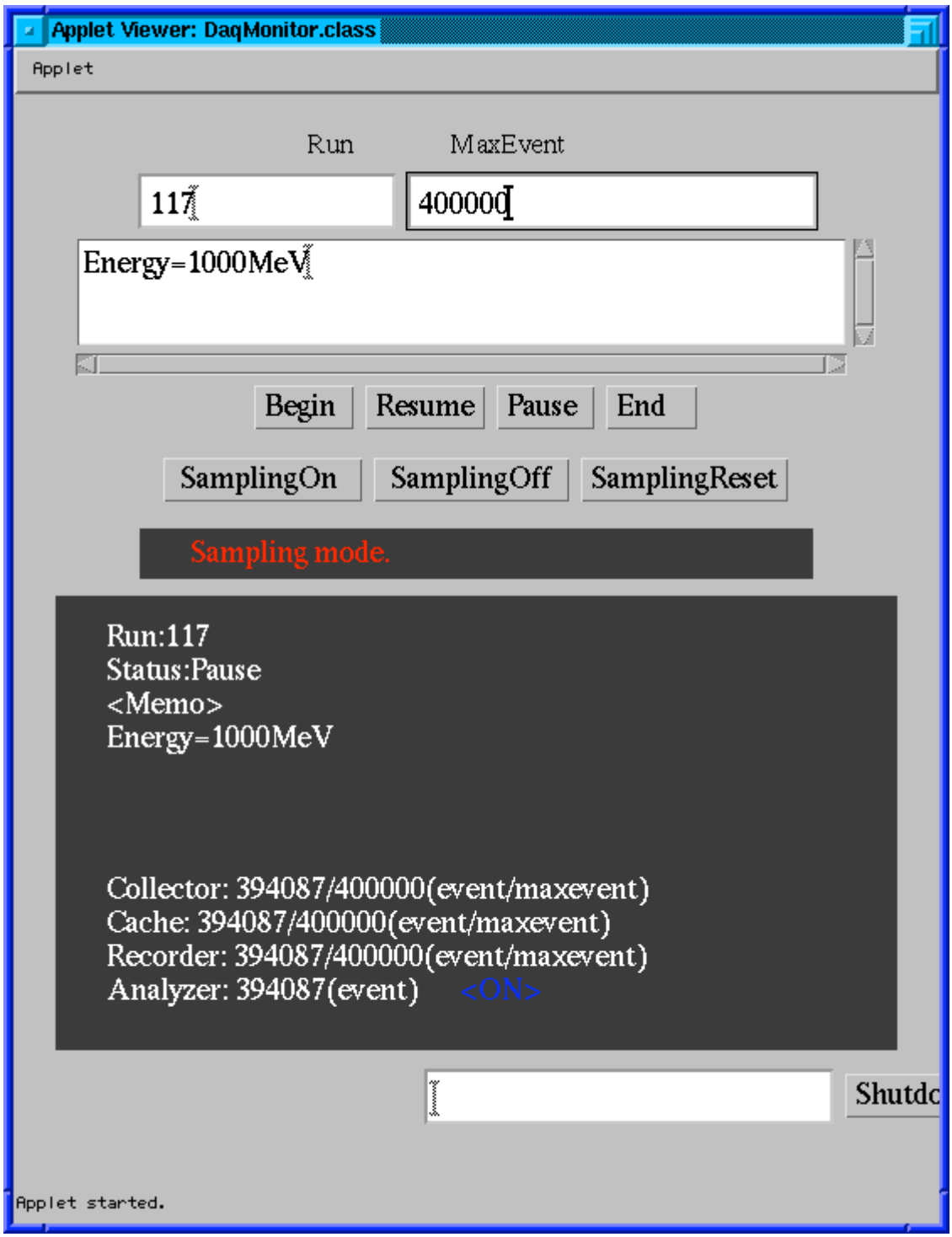


図 **4.7.1** : **JAVA** アプレットによる **Monitor** プロセス

Chapter 5　PWO クリスタル基本性能実験

Section 5.1　実験目的

この章では、1997 年 11 月に KEK 田無分室のγ３ビームラインに於いて行われた、“PWO クリスタルの基本性能実験”、について報告する。この実験の目的は大きく分けて２つある。

まず１つの目的は、開発した JAVA DAQ Prototype の動作確認である。今回の実験では、Analyzer プロセスでオンライン解析を正しく行うことができるか、Monitor プロセスの使用に際し不便な部分はないか、DAQ の実装に加えるべきものはなにか、などというユーザインターフェイスに関する部分の確認をおこなう。

PWO($PbWO_4$：タングステン酸鉛)クリスタルは、密度の高い結晶で、電磁カロリーメータとして用いられることができる。この結晶は、高エネルギー物理学実験や、原始核実験などで古くから用いられている NaI:Tl(Tl 添加ヨウ化ナトリウム)などと比較して、生成された光の減衰時間が短いという特徴を持っている(NaI:Tl の 4%程度)。反面、得られる光量が少ない(NaI:Tl の 0.3%程度)。このことは、PWO クリスタルが、NaI:Tl に比べ、大強度の粒子の入射に対して使用が可能である反面、エネルギー分解能、位置分解能が低い傾向にあることを意味している。過去に、1GeV 以上の電子ビームを用いた PWO クリスタルの、エネルギー分解能、及び、位置分解能について実験が行われている[4]。しかし、1GeV 以下のエネルギーでの、系統的なデータは得られていない。２つ目の目的は、PWO クリスタルの、1GeV 以下の電子ビームの入射に対するエネルギー分解能、及び、本実験で用いたセットアップでの位置分解能を系統的に測定することにある。

Section 5.2　実験セットアップ

本実験でのセットアップについて説明をする。各カウンタは、ビームの入射方向を Z 軸とし、先頭から、前方カウンタ(Ｆ)、トリガーホドスコープ(THX-THY)、PWO クリスタル BOX、という順で設置される(図 5.2.1)。前方カウンタ(Ｆ)は、厚さ 5mm のプラスティックシンチレーションカウンターである。このカウンタと、その後部にあるトリガーホドスコープ(THX-THY)とのシグナルの論理積を取ることによって入射電子の検出を行い、これをトリガーとする。THX-THY は、幅 2mm 厚さ 3mm のプラスティックシンチレーションカウンターを８枚重ね合わせたものである(図 5.2.2)。各シンチレーションカウンターには、光ファイバーが接続されている。この光ファイバーからの光を、ポジションセンシティブな光電子増倍管で観測することによって、X 方向、Y 方向にそれぞれ 2mm の精度で、入射した電子の位置を検出することが可能である。そして、その後ろに、PWO クリスタルが内部に納められたボックスを設置する。PWO クリスタルは、20mm×20mm×200mm の形状のクリスタルを９本使用している(図 5.2.3)。このとき、THX-THY の中心と、PWO クリスタル No.5 の中心は一致するように設置する。これらのクリスタルを、No1(1,1)から No9(3,3)とする。各クリスタルには、光電子増倍管が取り付けられており、各クリスタルでのシンチレーション光を測定することができるようになっている。そして、５番の PWO クリスタルに任意の運動量の電子を入射させることで、PWO クリスタルのエネルギー分解能及び位置分解能の測定をおこなう。なお、本実験は、KEK 田無分室のγ３ビームラインに於いて行われたが、γ３ビームラインでは、1.2GeV/c までの運動量の電子を使用することが可能である。

データ収集に用いた回路図を、

図 **5.2.4**に記す。クリスタルに入射する電子は、フロントカウンター（F）及び **THX**、**THY** を通過する。**THX**、**THY** からのシグナルは、論理和回路を通して、**X**、**Y** 方向にそれぞれ１つずつのシグナルとなり、さらにフロントカウンターとの論理積回路へ入力される。この論理積回路の出力が、データ収集のトリガとなる。トリガが発生すると、**Gate and Delay Generator(GDG)**からゲートシグナルが生成され、その後２つに分岐する。一方のゲートシグナルは、**Analog to Digital Converter(ADC)**レジスタ、及びコインシデンスレジスタに送られる。これらのレジスタは、ゲートシグナルを受け、**THX-THY** のシグナル及び、**PWO** クリスタルに接続された光電子増倍管からのシグナルを、デジタル化してモジュール内に取り込む。もう一方のゲートシグナルは、ディレイモジュールを通してインターラプトレジスタに送られる。ディレイモジュールでゲートシグナルが遅延される時間は、コインシデンスレジスタ、及び **ADC** レジスタが、データをデジタル化するのに要する時間より長く設定してある。これにより、インターラプトレジスタにシグナルが送られたときには、コインシデンスレジスタ、及び **ADC** レジスタでのデジタル化は終了している。インターラプトレジスタは、シグナルを受け取るとクレートコントローラを通じて、コンピュータにデータが **CAMAC** モジュールに蓄えられたことを知らせる。**Collector** プロセスは、このシグナルを受け取るとデータを収集し、アウトプットレジスタにデータを出力して、各 **CAMAC** モジュールや、**GDG** の初期化をおこなう。これにより、新たなイベントデータの収集が可能となる。回路図に記されているスケーラは、フロントカウンタの入射する電子の数や、発生したトリガ、**DAQ** で収集したイベント数をカウントするのに使用している。

本実験では、以上のセットアップを用いて、**PWO** クリスタルに **200MeV/c** から **200MeV/c** おきに **1000MeV/c** までの電子ビームを、５番のクリスタル入射させ、エネルギー分解能及び位置分解能の測定をおこなった。

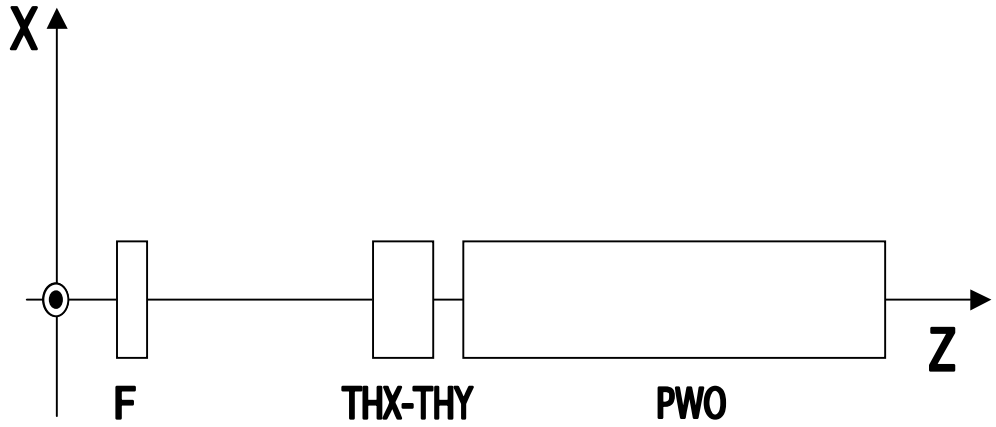


図 **5.2.1**：実験セットアップ

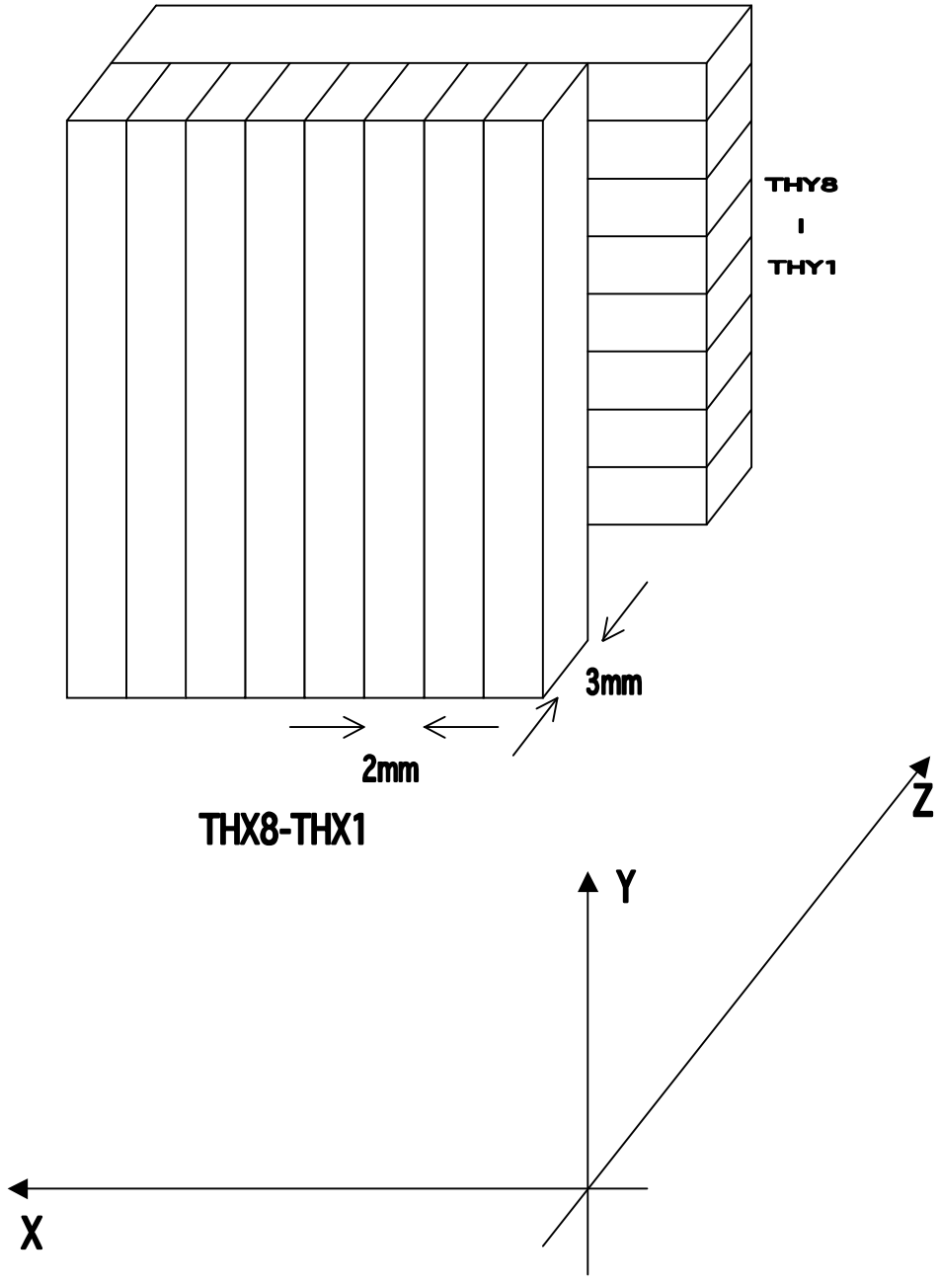


図 5.2.2 : THX - THY

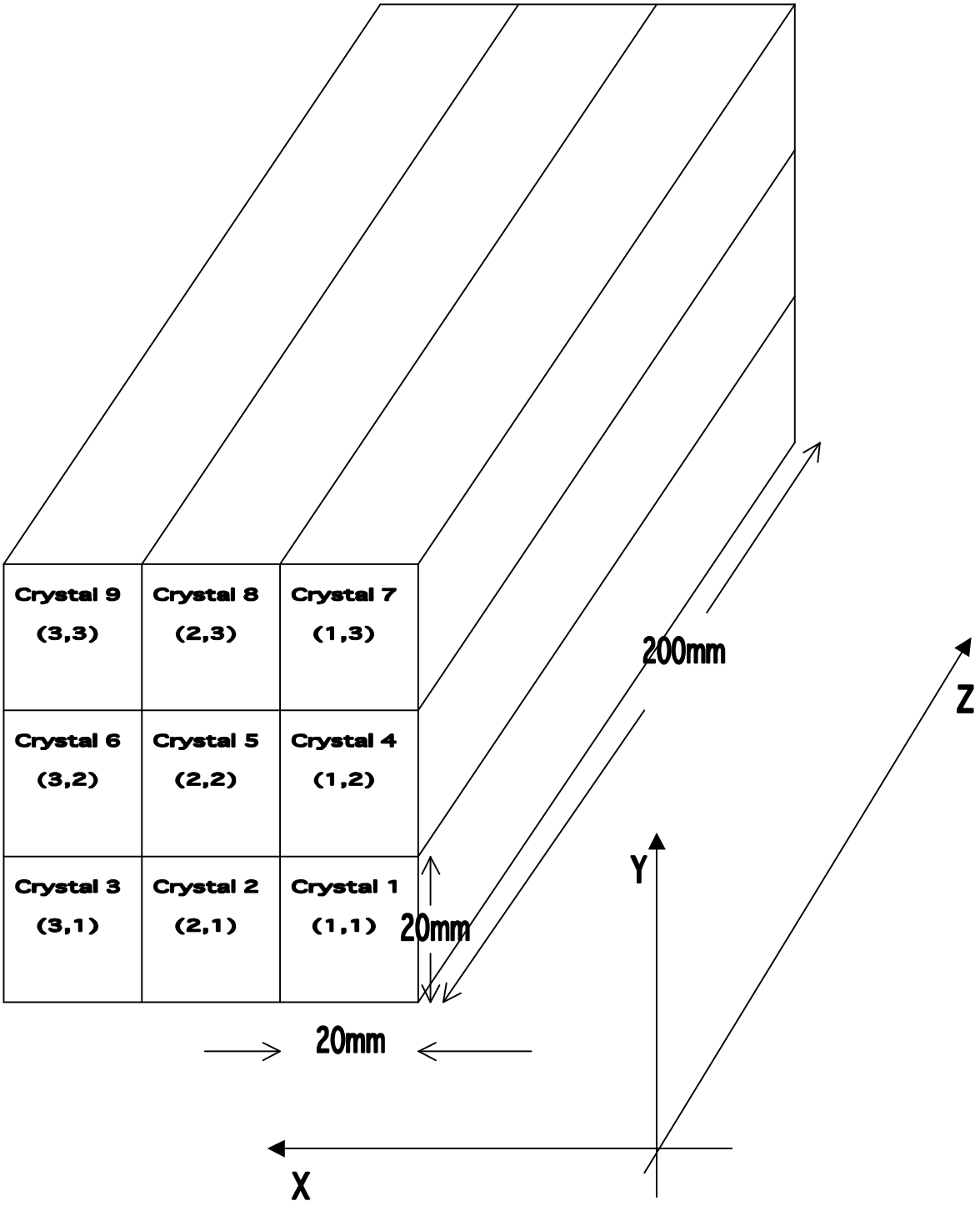


図 5.2.3：PWO クリスタル

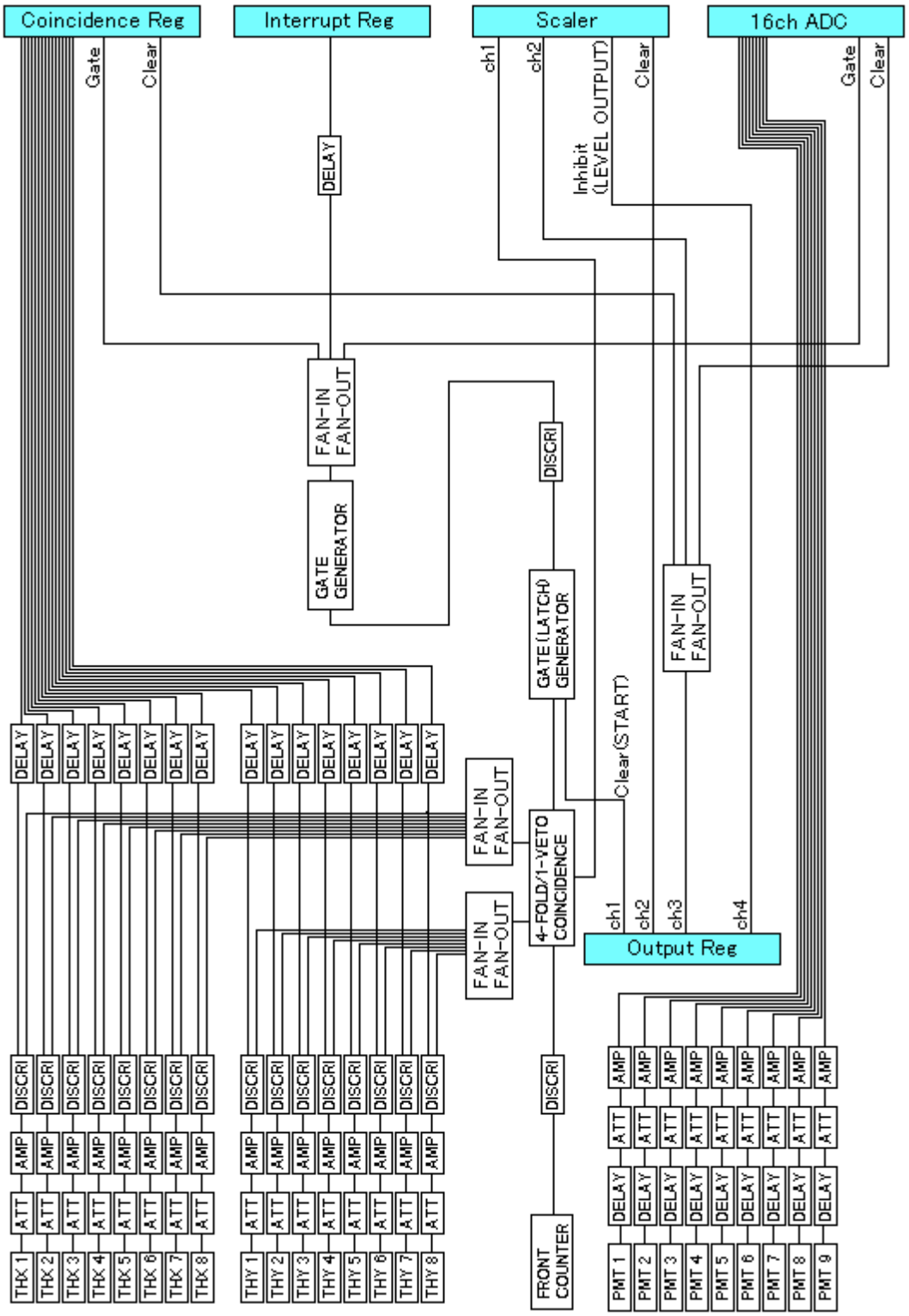

Coincidence Reg
Interrupt Reg
Scaler
16ch ADC
Gate
Clear
ch1
ch2
Inhibit (LEVEL OUTPUT)
Clear
Gate
Clear
DELAY
FAN-IN FAN-OUT
GATE GENERATOR
DISCRI
GATE (LATCH) GENERATOR
FAN-IN FAN-OUT
Clear (START)
4-FOLD/1-VETO COINCIDENCE
FAN-IN FAN-OUT
FAN-IN FAN-OUT
ch1
ch2
ch3
ch4
Output Reg
DELAY
DISCRI
AMP
ATT
THX 1
THX 2
THX 3
THX 4
THX 5
THX 6
THX 7
THX 8
THY 1
THY 2
THY 3
THY 4
THY 5
THY 6
THY 7
THY 8
FRONT COUNTER
PMT 1
PMT 2
PMT 3
PMT 4
PMT 5
PMT 6
PMT 7
PMT 8
PMT 9

図 **5.2.4**：測定回路図

Section 5.3　解析結果

1GeV/c の電子を No.5 の PWO クリスタルに入射させたときに、ADC で得られたデータについて記す(図 5.3.2)。中心のクリスタル以外のクリスタルでも、シンチレーション光が測定できていることが判る。これらの ADC で測定されたデータは、ペデスタル(図 5.3.1)を引かれ、エネルギーキャリブレーションが施されている。図 5.3.2のグラフは、横軸に PWO で検出したエネルギーをとり、縦軸にはイベント数をとる。それぞれのクリスタルのデータは、入射電子が 600MeV/c のときのデータでイテレーションがおこなわれている。このときのデータトリガーは、X方向で THX1～THX8 を、Y方向で THY1～THY8 を用いた。各クリスタルで検出したエネルギーの総和を求めることは、９本の PWO クリスタルを用いて、入射電子のエネルギーを求めることと同等である。この結果は、図 5.3.3で記す。この結果より、９本の PWO クリスタルを用いて、1GeV/c の電子を入射させた場合のエネルギー分解能($\sigma_E/E_{INCIDENT}$)が 3%程度であることが判る。これは、過去に行われた PWO クリスタルの実験結果から外れる値ではない。これより、本実験が正しくおこなわれていることが確認できる。同様に、800MeV/c、600MeV/c、400MeV/c、200MeV/c の電子でも測定をおこなった。図 5.3.4は、入射電子のエネルギーと、ADC で得られたピークエネルギーの関係を示している。横軸に、入射電子の運動量、縦軸に ADC で得られたピーク運動量を取る。

図 5.3.5のグラフは、それぞれ 1000MeV/c、800MeV/c、600MeV/c、400MeV/c、200MeV/c の電子を用いた PWO クリスタルのエネルギー分解能($\sigma_E/E_{INCIDENT}$)である。横軸に入射電子の運動量を取り、縦軸はエネルギー分解能をとる。低エネルギー側では、光量が少ないため分解能は悪くなる。入射電子が 200MeV/c では、9%のエネルギー分解能となることがわかる。

図 5.3.6のグラフは、THX5、THY2～THY7 のトリガーのデータに対して重心法を用いた結果である。横軸は、重心法を用いて測定した電子の入射位置で、縦軸はイベント数である。このデータは THX5、THX2～THX7 を

トリガーとしてデータを収集しているので、**THX-THY** より得ることのできる電子の入射位置は、**X** 軸方向のみに注目すると、**0mm≦X≦2mm** の範囲となる。実際の電子は、**0mm≦X≦2mm** の範囲を通過するわけであるが、これを簡略化し、**X=1mm(THX5** の中心**)**を通過したものとする。

重心法による、電子の仮想入射位置の求め方は次式で与えられる。

$$Xg = \frac{\sum_{i=0}^{9} EiXi}{\sum_{i=0}^{9} Ei}$$

ここで、E_i は各クリスタルで検出したエネルギー、X_i は各クリスタルの中心の **X** 座標である。

図 **5.3.7**のグラフは、**1GeV/c** の電子を用いた、**THX-THY** より求められる位置 **X** と、重心法より求められる位置 X_g の関係を示している。**THX1**～**THX8** だけでは、－**7mm≦X≦7mm** 間での範囲しか測定できない。そこで、**THX-THY** と **PWO** クリスタルの相対位置を **4mm** ずらして、トリガーを選択することによって、－**11mm≦X≦7mm** の範囲について測定をおこなった。また、これらのデータは、物理的に奇関数で与えられるものであることが容易に推測される。そのため、今回は次式で与えられる関数でフィッティングをおこなった。

$$F(X_g) = P_1(X-P_4)^5 + P_2(X-P_4) + P_3$$

この関数は、$F(X_g) = P_1X^5 + P_2X$ を平行移動したもので、P_4 、P_3 という値が **X** 軸、**Y** 軸方向の移動距離を与える。この、P_4 、P_3 という値は、**THX-THY** の中心と、**PWO** クリスタルの軸との位置のずれによって生じる値である。この関数よりデータを原点に平行移動し、位置分解能を求める。**THX** の **X** 座標位置を α とする。ここを通過した電子の **X** 座標位置が重心法によって、$F(\alpha)\pm\sigma$ であると得られたとする。このとき、

$$F(\alpha)+\sigma = P_1X^5 + P_2X$$

を満たす **X** を X_1、

$$F(\alpha)-\sigma = P_1X^5 + P_2X$$

を満たす **X** を X_2、とすると、**THX** の **X** 座標位置での誤差が、

$$\sigma_x = (X_1 + X_2)/2$$

で求められる。

図 5.3.8のグラフは、1GeV/c の電子を用いた PWO クリスタルの位置分解能を示している。このグラフは上で示したように、図 5.3.7のグラフに P_4 、P_3 用いた補正をおこない求めることができる。中心の PWO クリスタルへ入射した電子は、PWO クリスタルを通過中にエネルギーを失う。このエネルギーが、シンチレーション光に変換され、最終的に光電子増倍管で検出される。このとき、入射した電子の位置がクリスタルの中心から外れるほど、隣のクリスタルで検出できる光量が増加する。左右のクリスタルで検出できるエネルギーの差が大きいと、重心法で求めた入射位置は、精度が良くなる傾向にある。この反面、中心付近に入射した電子では、左右のクリスタルで検出できるエネルギーの差が小さいので、精度は悪くなる。本実験のセットアップを用いた場合、PWO クリスタルの 1GeV/c の電子による位置分解能は、クリスタル中心部分で 3mm 程度となった。600MeV/c、200MeV/c での位置分解能は、図 5.3.9から図 5.3.12で示す。

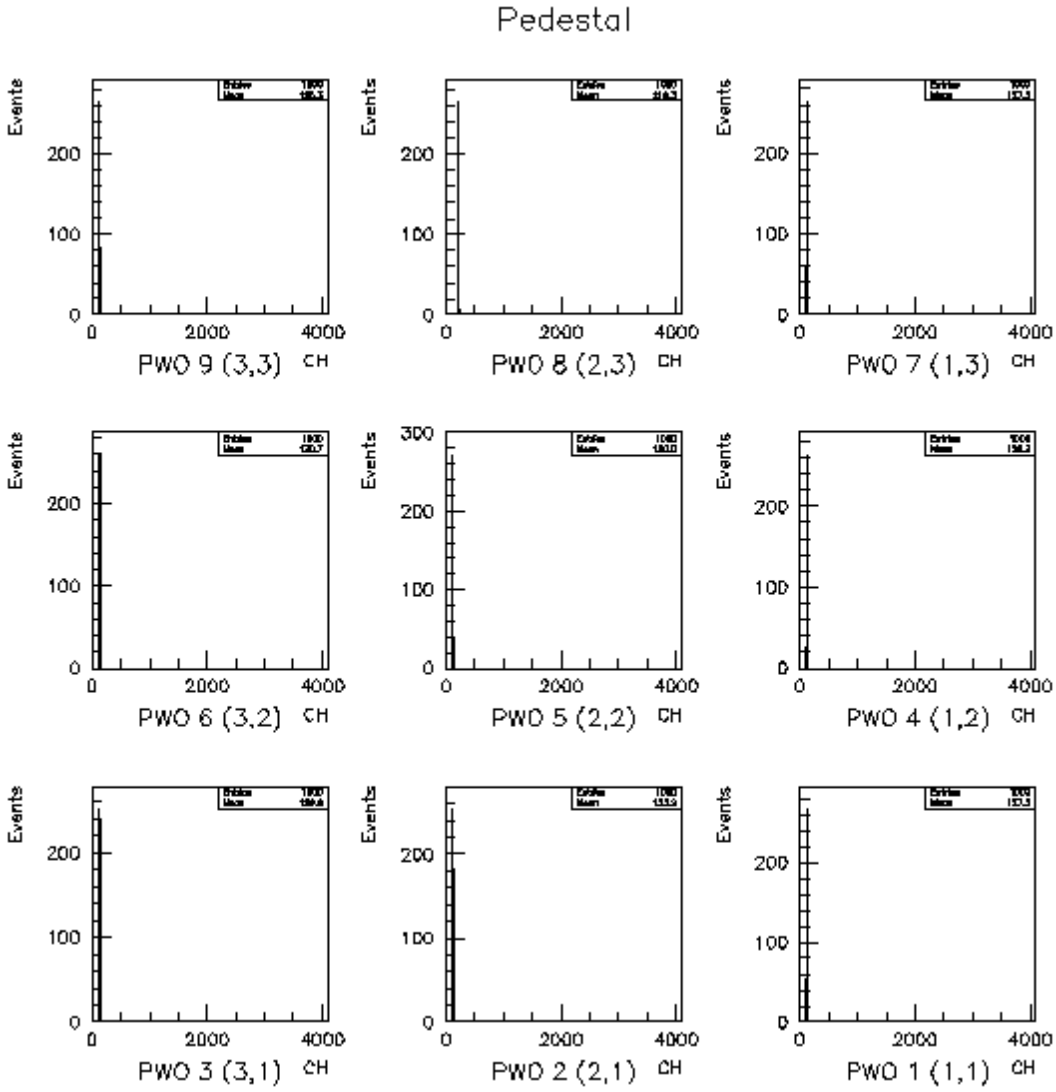


図 5.3.1 : ADC pedestal

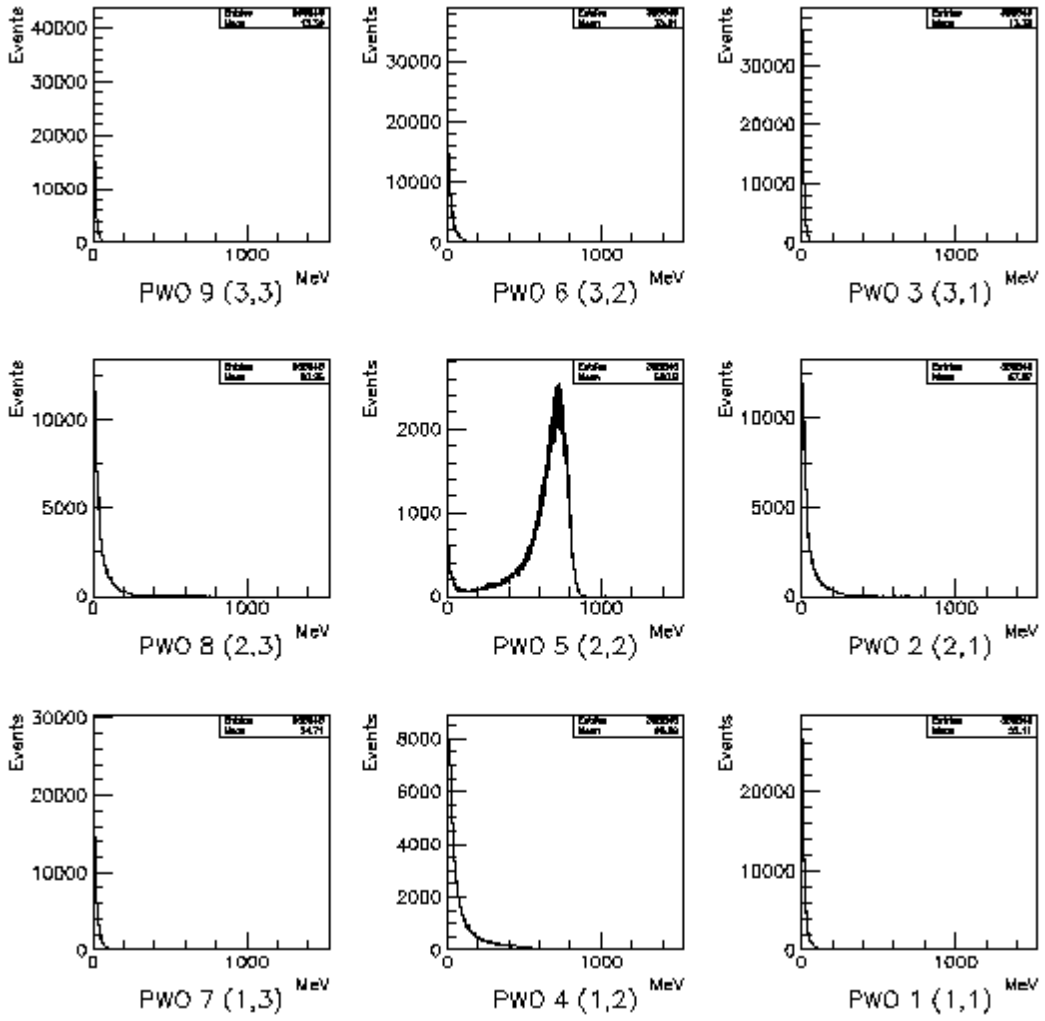


図 5.3.2 : ADC data at 1GeV/c

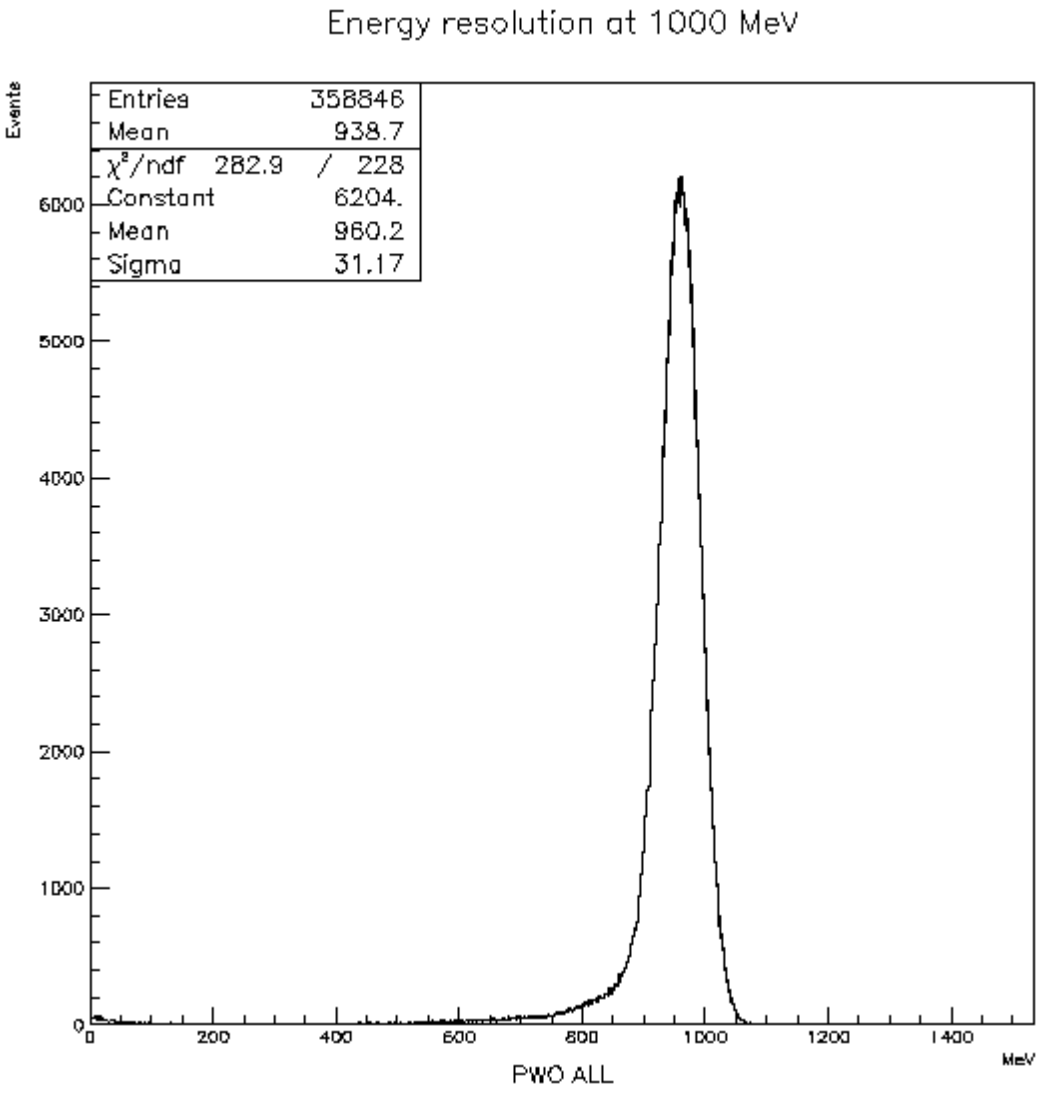


図 5.3.3 : Energy resolution at 1GeV/c

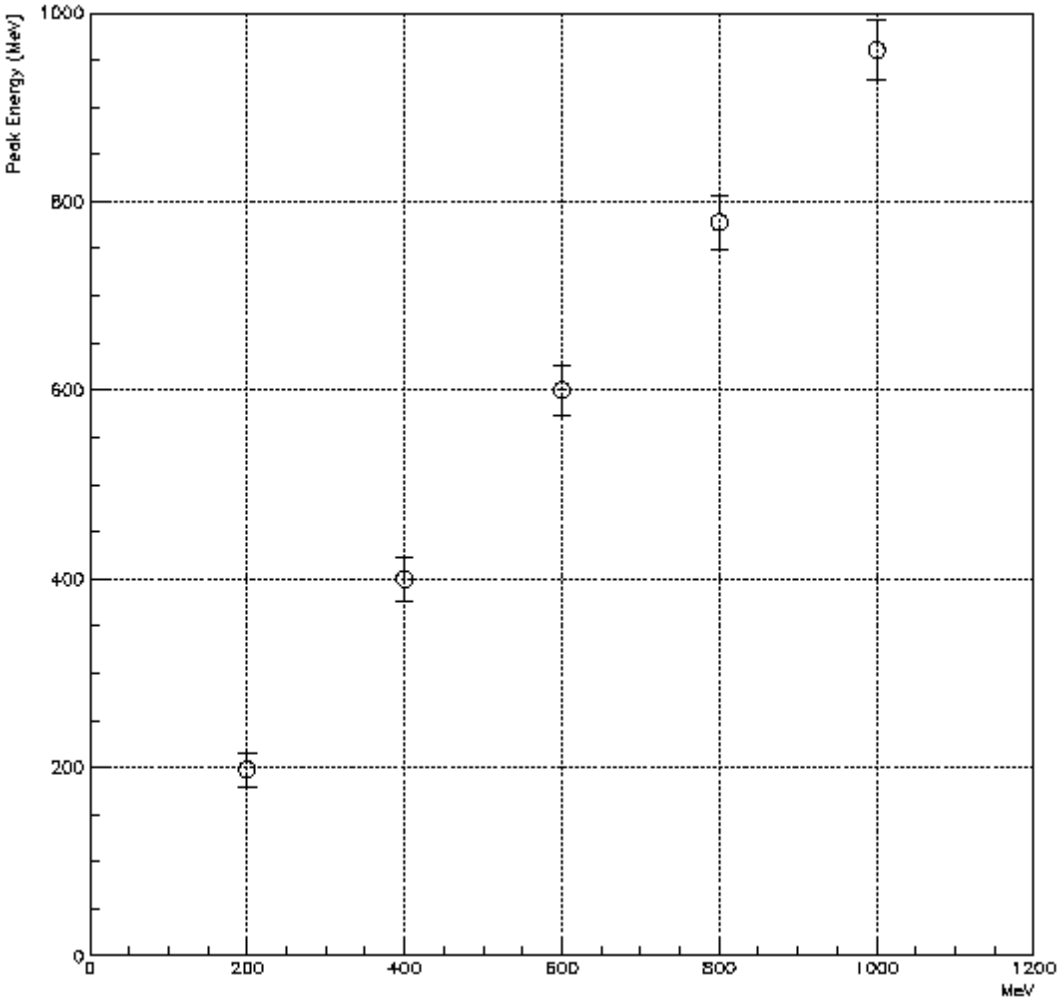


図 5.3.4 : ADC Peak

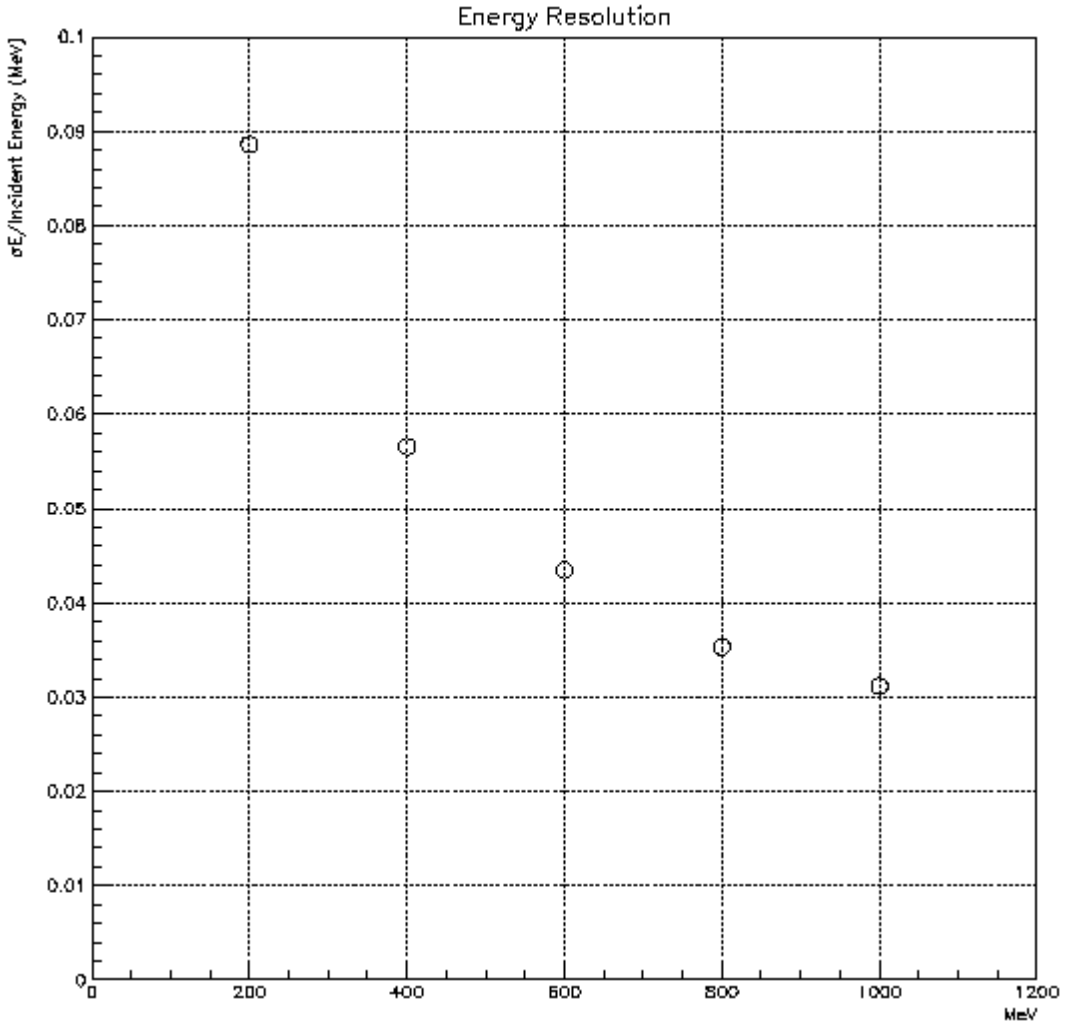


図 **5.3.5**：**PWO** エネルギー分解能

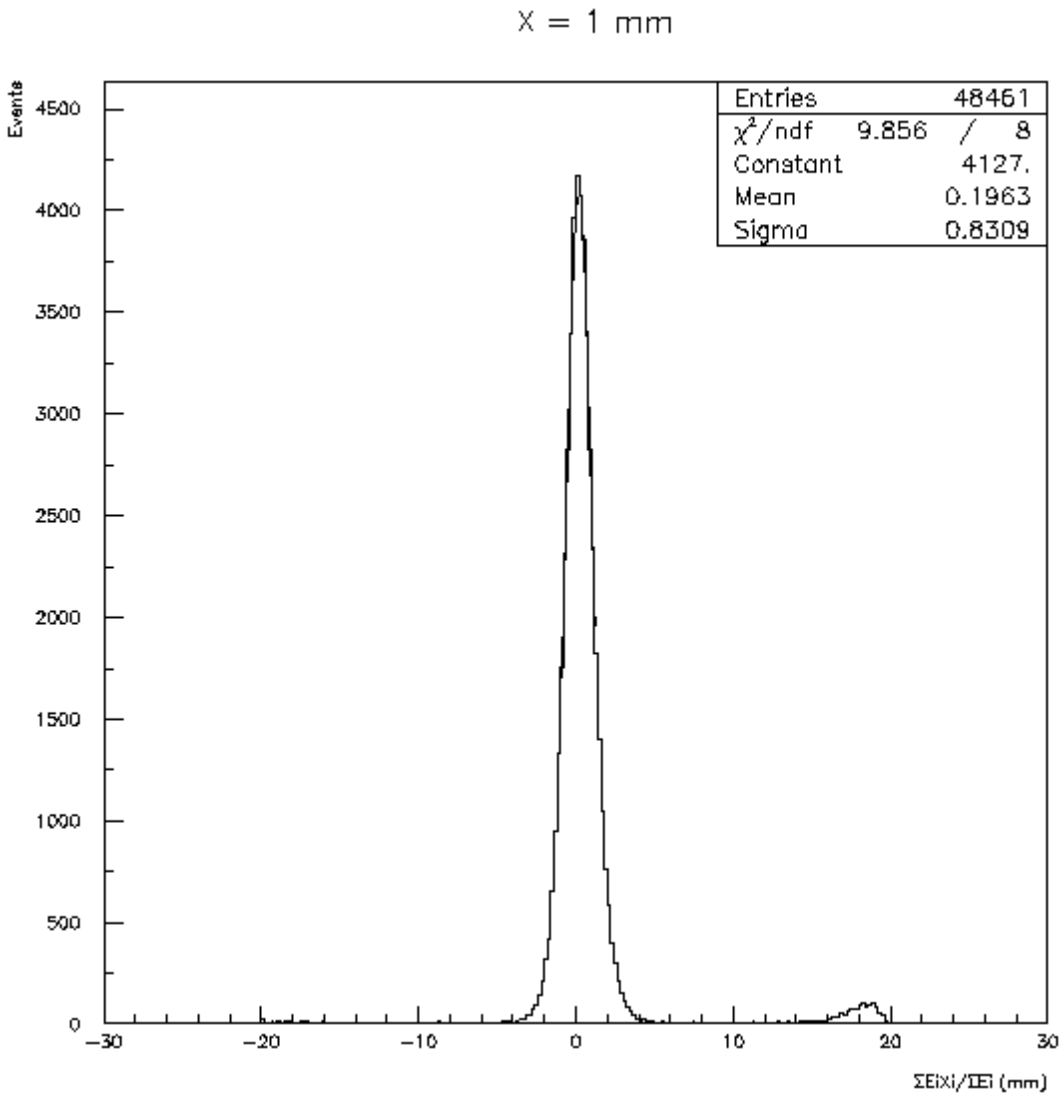


図 5.3.6 : Trigger THX=5 (X=1mm) at 1000 MeV/c

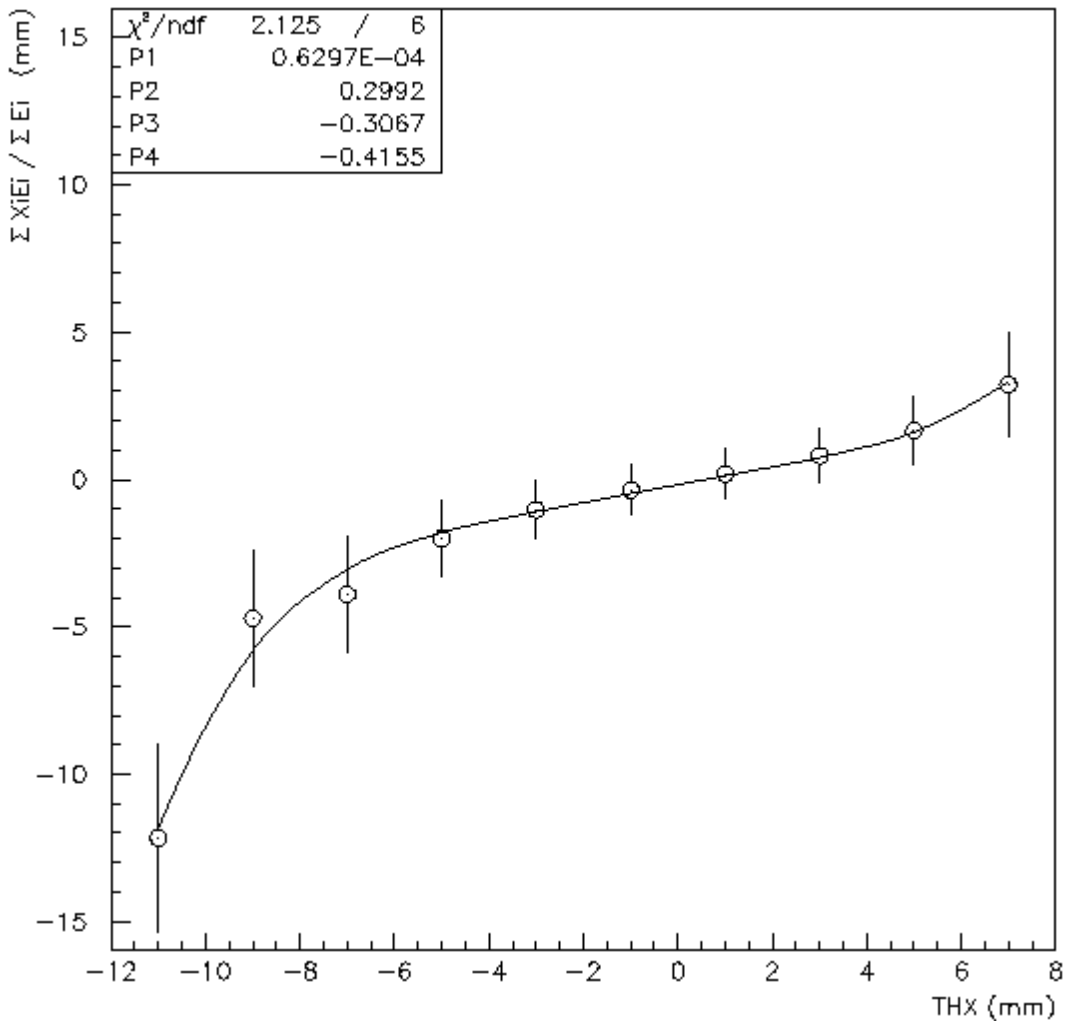


図 5.3.7 : Xg at 1GeV/c

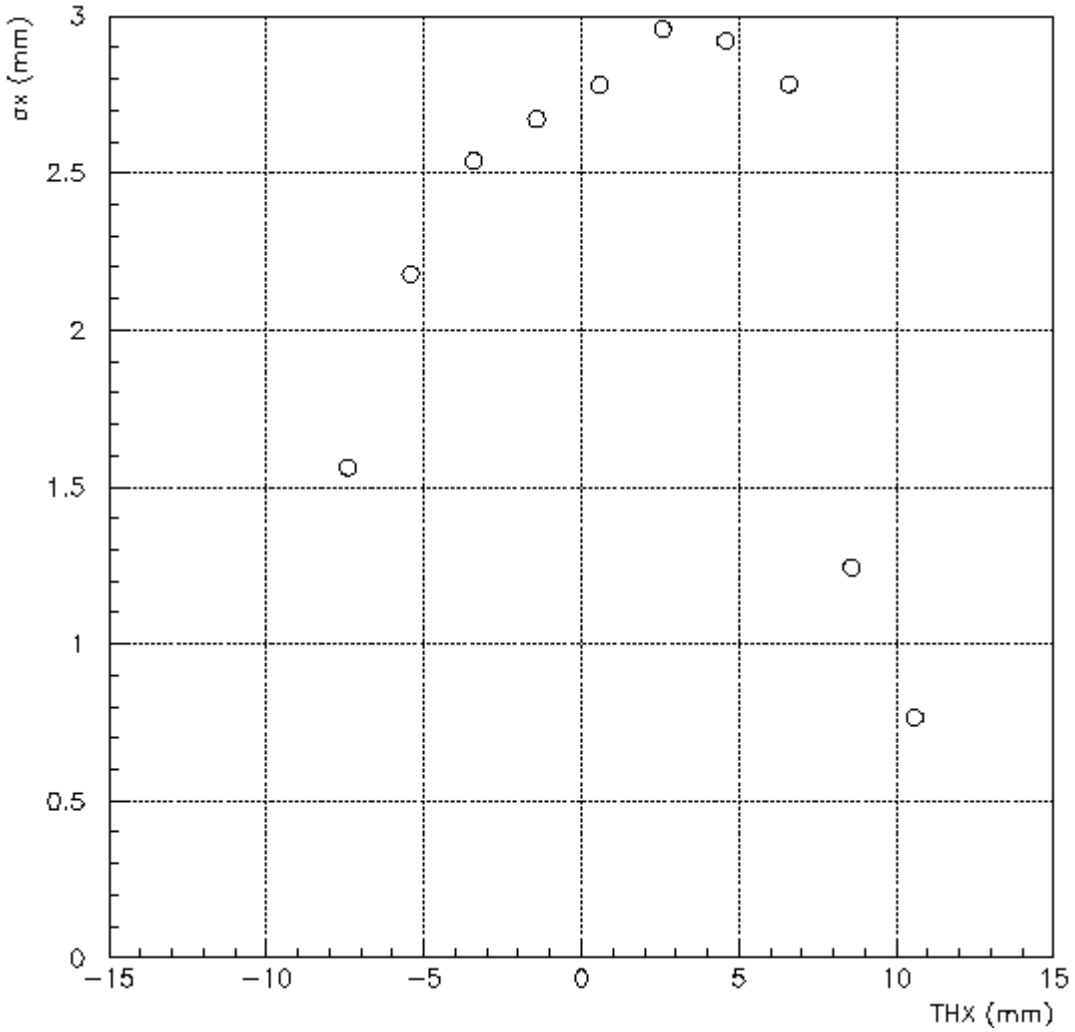


図 5.3.8 : Position resolution at 1GeV/c

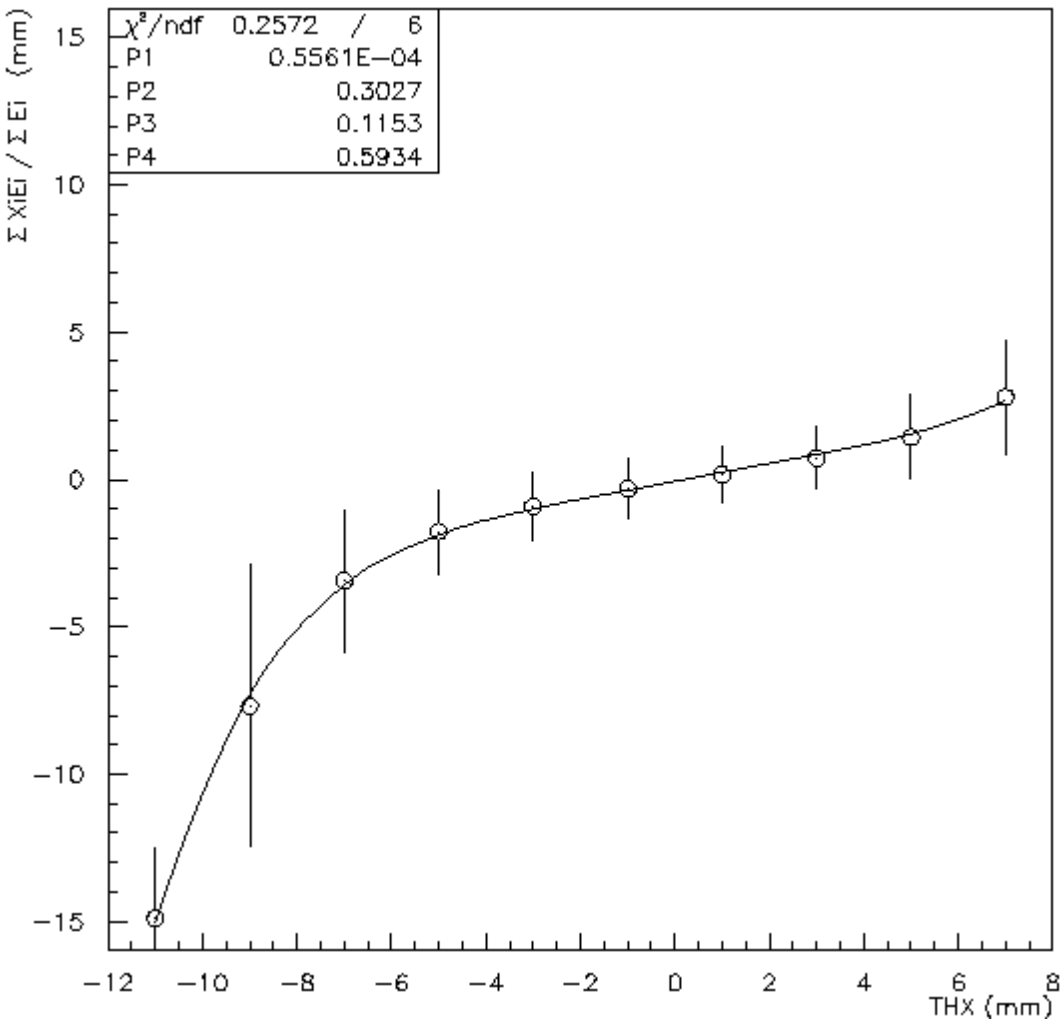


図 5.3.9 : Xg at 600MeV/c

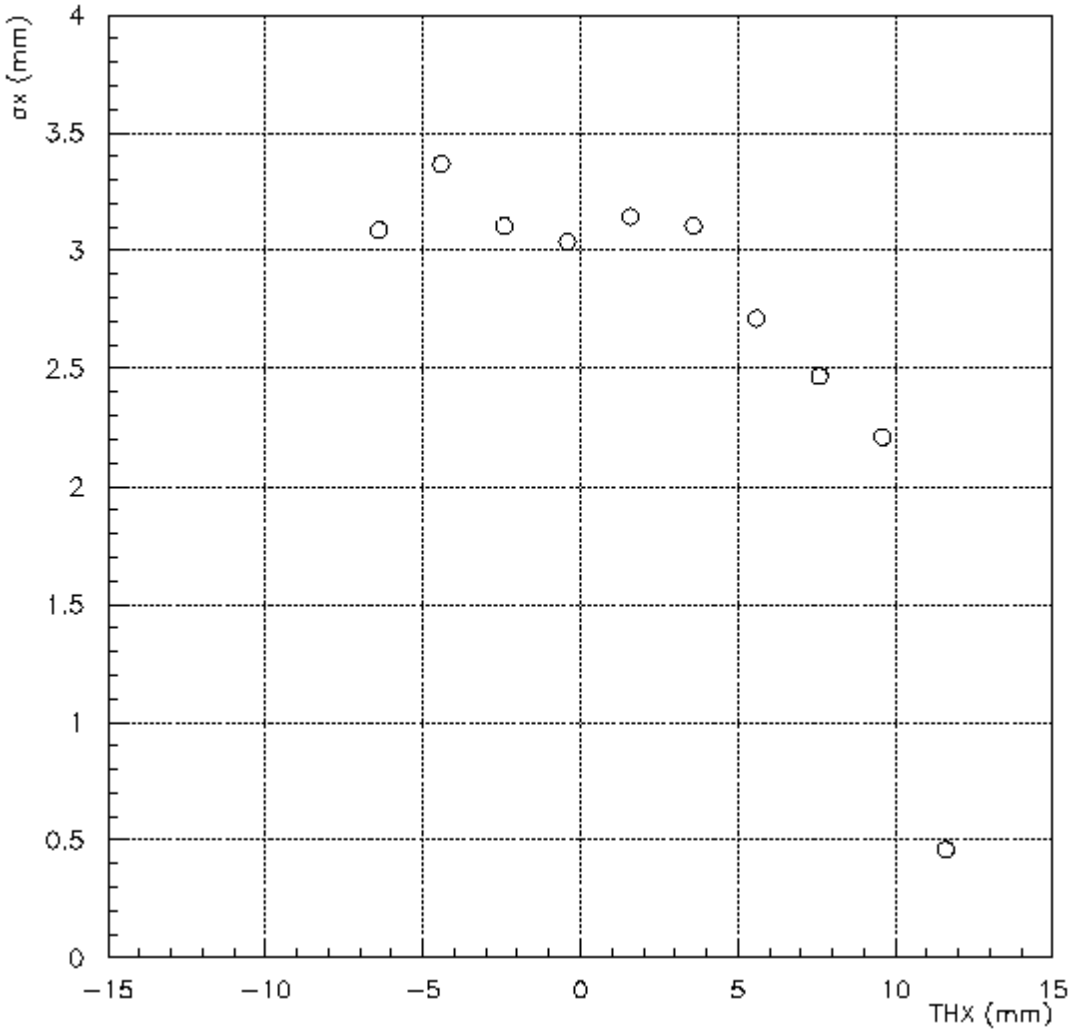


図 5.3.10 : Position resolution at 600MeV/c

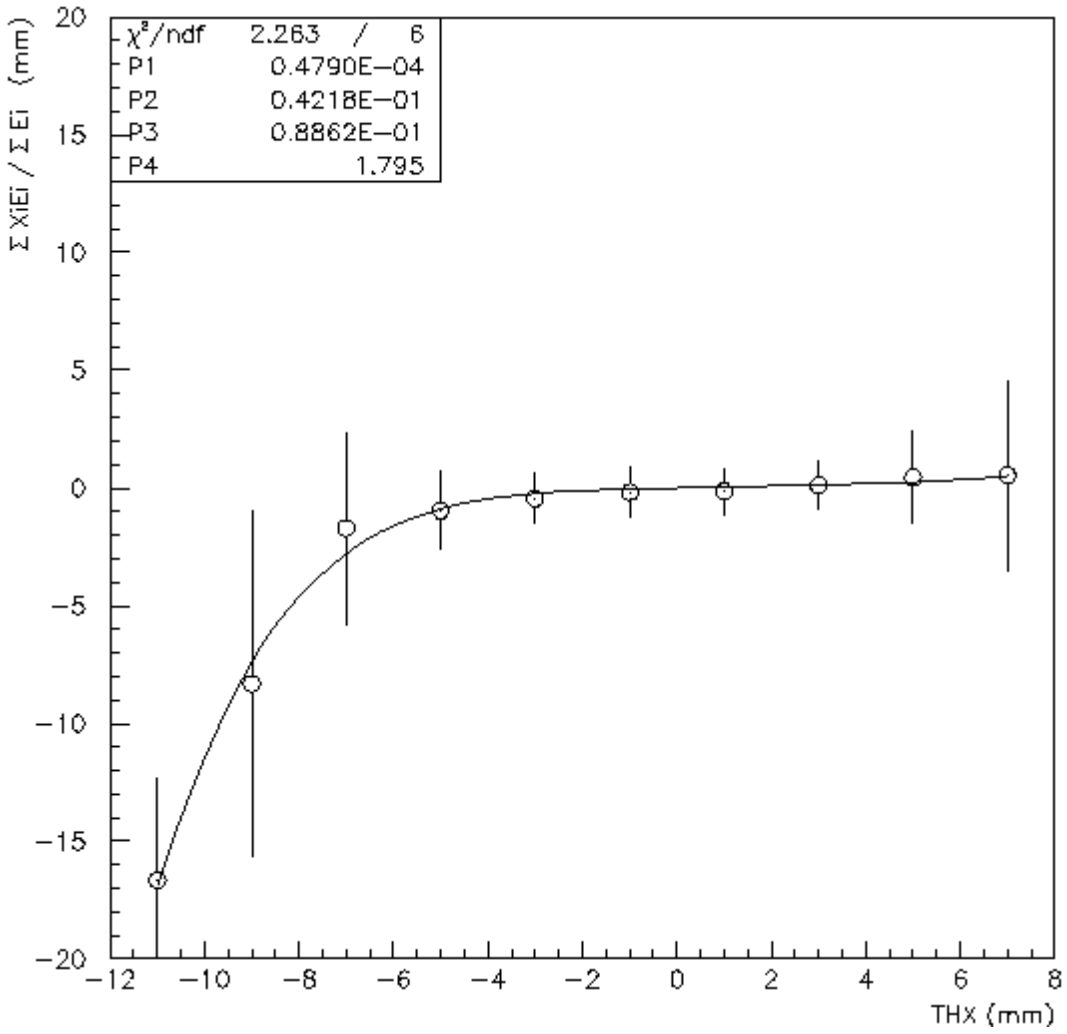


図 5.3.11 : Xg at 200 MeV/c

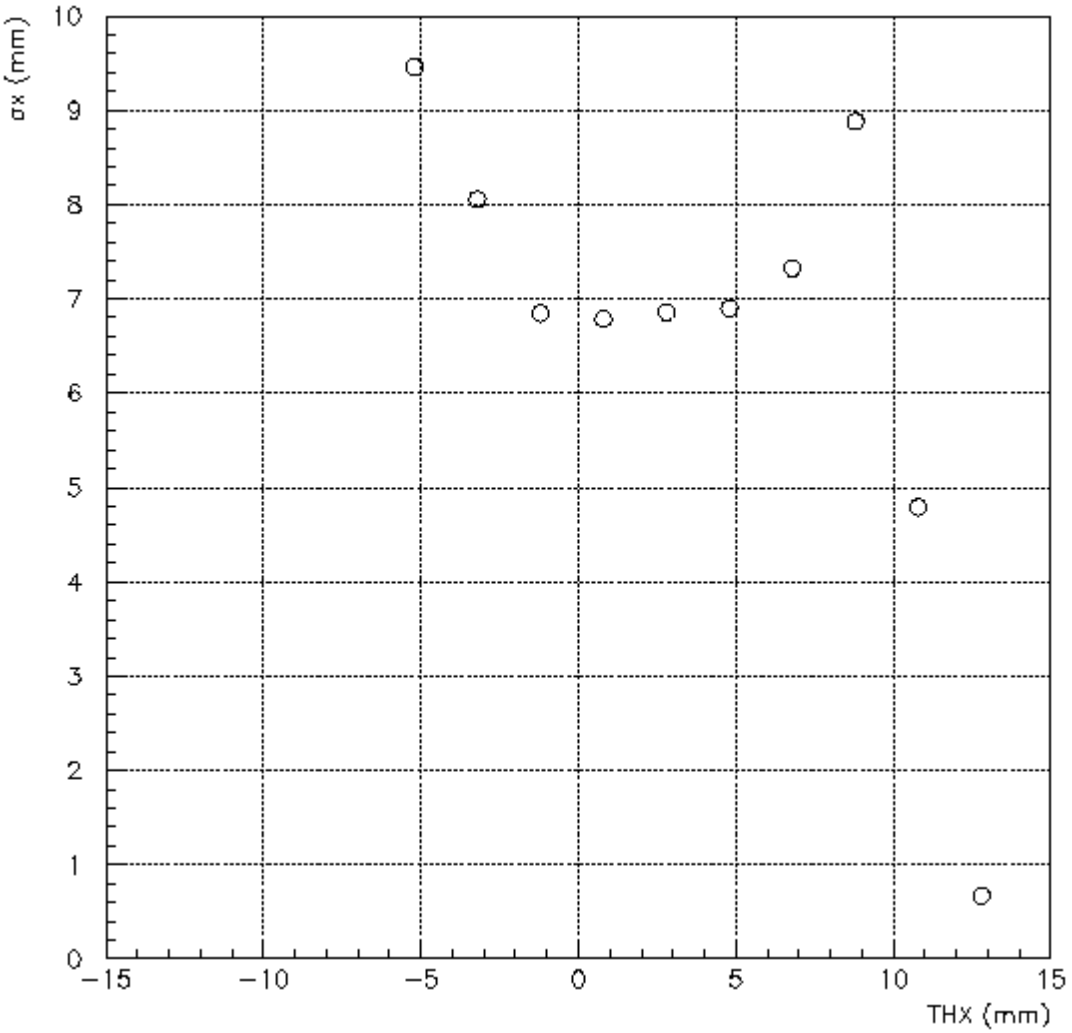


図 5.3.12 : Position resolution at 200MeV/c

Chapter 6　まとめ

JAVA DAQ Prototype は、**1997** 年 **11** 月に **KEK** 田無分室でおこなわれた“**PWO** の基本性能実験”にデータ収集システムとして導入された。この実験では、**JAVA DAQ** には非常に満足のゆく動作をし、要求されるデータを収集することができた。**DAQ** の各プロセスは、協調し動作をおこない、データ収集、オンラインデータ解析、**Monitor** プロセスが正常に並行動作することが確認できた**(**図 **6.1)**。実際、オンラインデータ解析と **Monitor** プロセスの働きによって、早期に、**CAMAC** モジュールの故障や、ケーブルの断線、**DAQ** システムの異常状態を検出することができた。これは、オンラインデータ解析、**Monitor** プロセスが、**GUI** を含むプロセスであった結果である。また、これは予期しない収穫であった。今回 **DAQ** は、１台の計算機上で実行したが、複数の計算機で動かすことも可能である。

JAVA DAQ Prototype は、現在も開発中で、今後も改良を重ねる必要がある。まず、プロセスの生成、消滅に関して自由度を持たせることが挙げられる。現在の **DAQ** では、プロセスの起動順番や、**DAQ** を構成するプロセスの数や種類が固定されている。これは改善しなければならない。また、現在の **DAQ** では、ラン番号や、**DAQ** の状態などは **Commander** プロセスから各プロセスへ送られる。各プロセスは、これらのデータを内部に保存している。今後は、これらのデータを集中的に管理するプロセスを作成するべきである。他にも永続オブジェクトを用いた **DAQ** 状態データの管理といった課題が、残っている。これらの課題について今後も開発をおこなわなくてはならない。

現在のバージョンの **CAMAC Driver/Library** は、**root** 権限を持ったプロセスしか、クレートコントローラを操作することが出来ない。この問題点についての改良は現在も行われており、次のバージョンでは **user** 権限のプロセスから、クレートコントローラがアクセス出来るようになるはずである。

今回作成した **Prototype DAQ** は、修士課程在学中に行われた一連の研究、

及び、開発の結果を集約したものである。とくに、**PC/Linux CC/7x00** 用に開発した **CC/7x00** ドライバ、及びライブラリは、現在、**KEK** のオンライングループのサーバコンピュータから、ネットワークを通じて公開されるている。

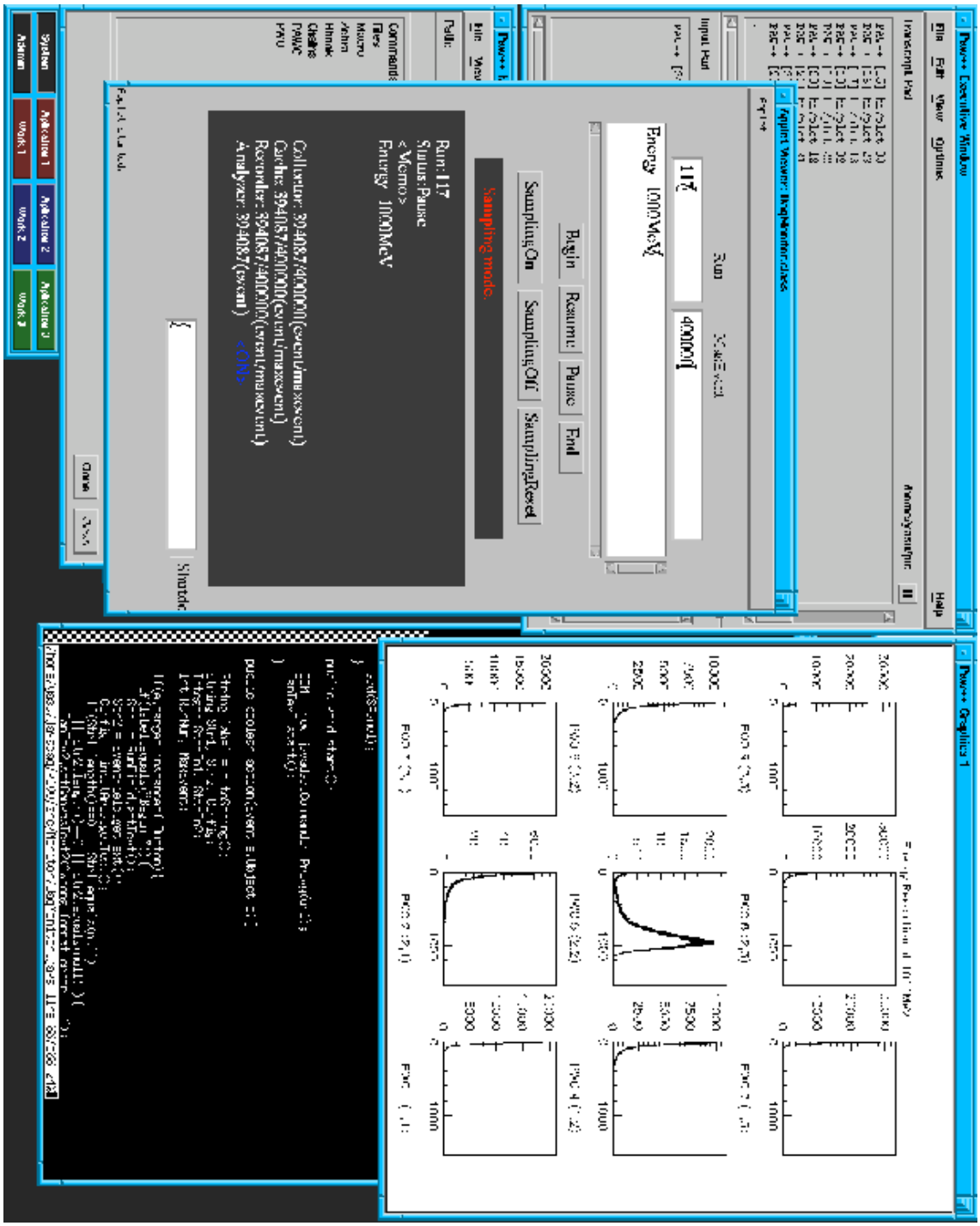


図 6.1 : Prototype of JAVA Data Acquisition System

Appendix

- オブジェクト：操作の対象となるもの。対象とその動作。オブジェクト指向言語では、C 言語のように、データと関数といった区別はない。強いて言うならば、C 言語で指す、関数とデータがオブジェクトである。
- オブジェクト指向：操作を中心として考える「手続き的」に対して、操作の対象となるデータの機能や意味を中心として考えること。ソフトウエアの作成・保守がしやすい。
- ネイティブメソッド：C 言語などで作成された、使用する計算機専用のコードを含むライブラリなどの関数のこと。
- プラットフォーム非依存：計算機やハードウエア等の実行環境に依存しないこと。
- 分散データベース：ネットワーク上にデータベースを分散するシステム。
- メソッド：関数や手続きといった実際の操作。強いて言うならば、C 言語で指す、関数にあたる。
- 分散処理：データ処理を１台のコンピュータで行うのではなく、複数のコンピュータを用いて、そのデータの発生場所などで処理をしていく方法。
- **GUI**：**Graphical User Interface.** グラフィックを用いた、ユーザインターフェイスのこと。
- デバイスドライバ：周辺機器（デバイス）を制御するためのソフトウエア。
- スレッド：１つの起動中のタスクに存在する、更に細かい起動可能なプログラムの実行単位。
- プロセス：プログラムの実行単位。

謝辞

本論分は、多くの方々の協力によって作成することができました。特に、高エネルギー加速器研究機構(KEK)の安芳次氏には、山形大学理学研究科修士課程に進学してから卒業するまでの２年間、出来の悪い学生であった私に諦めることなく、計算機に関する指導をしていただきました。氏の御指導が無ければ、**CAMAC Driver/Library** や、**JAVA DAQ Prototype** の開発は出来ませんでした。本当にありがとうございます。

山形大学理学研究科では、加藤静吾教授、清水肇教授、吉田浩司講師、木梨徹助手からは、高エネルギー物理学・物理学実験に必要な知識をはじめとして、多くのことを授かりました。阿部敬一君、橋本朋幸君、松本哲也君には、**KEK** 田無分室における実験の準備や実験後の解析などを手伝って頂きました。特に、**JAVA DAQ Prototype** の **Monitor** プロセスは、松本君の援助により完成することが出来ました。

高エネルギー加速器研究機構で作業をするに当たり、東北大サイクロの田島靖久助手には計算機、**CAMAC** などの器機についてのご指導などをしていただきました。また、**KEK** での生活全般についても御世話になりました。ありがとうございます。中山博士氏、堀川壮介氏には幅広い分野で多くのことを教えて頂き、また、森本巌君、佐藤圭太君には、様々な部分で援助をしていただきました。

最後に、高エネルギー加速器研究機構田無分室の奥野英城助教授には山形大学理学部に在籍中の実験でお世話になっただけでなく、今回の実験についても多大なる援助をしていただきました。

みなさん本当にありがとうございました。

参考文献

[1] 安芳次　竹内康雄　**UNIX** ワークステーションによるリアルタイムデータ収集．日本物理学会誌 **Vol.48, No.11, 1993** 年

[2] Y.Yasu, M.Nomachi, Y.Nagasaka, R.Ball, Y.Tajima and C.Timmermans. UNIDAQ, Real-time Response of the System. KEK Preprint 95-84, KEK, 1995.

[3] Y.Yasu, H.Fujii, E.Inoue, H.Kodama and Y.Sakamoto. A Study of Network-based Data Acquisition System. KEK Preprint 97-6, KEK, 1997.

[4] S.Inaba, M.Kobayashi, M.Nakagawa, T.Nakagawa, H.Shimizu, K.Takamatsu, T.Tsuru, Y.Yasu. Beam test of a calorimeter prototype of PWO crystals at energies betoween 0.5 and 2.5 GeV. Nuclear Instruments and Methodes in Physics Research A 359,1995.

[5] Y.Yasu, Y.Tajima, Lynx-HP and DAQ-System. KEK-Preprint 94-191, KEK, 1995.

[6] Yoshiji Yasu, Hirofumi Fujii, Eiji Inoue, Hideyo Kodama and Yasunobu Sakamoto. Prototype performance of Distributed DAQ using HORB based on JAVA. Rt97, 1997.

[7] 安芳次、田島靖久　**DAQBENCH** の開発　**1994** 年秋の物理学会**(**山形大学**)**

[8] 坂本泰伸、安芳次、吉田浩司、田島靖久、長谷野雅哉、藤井啓文、能町正治　**PC-UNIX** を用いたデータ収集システムの開発　**1997** 年秋の物理学会**(**佐賀大学**)**

[9] 坂本泰伸、安芳次、吉田浩司、田島靖久　**JAVA** 言語を用いた分散データ収集の試み　**1998** 年春の物理学会**(**名古屋大学**)**

[10] 坂本泰伸、安芳次、吉田浩司、田島靖久、藤井啓文、児玉英世、井上英二　分散オブジェクト指向言語を用いたデータ収集プロトタイプの作成　**1998** 年秋の物理学会**(**東京都立大学**)**