KOTO実験のためのDAQバックエンドシステムの開発とその評価

大阪大学大学院 理学研究科 物理学専攻
山中卓研究室 博士前期課程2年

中谷 洋一

February 10, 2011

概 要

茨城県東海村にある大強度陽子加速器施設（J-PARC）で 2012 年から開始される予定の J-PARC E14 KOTO 実験では、長寿命中性 K 中間子の稀崩壊イベント $K_L \to \pi^0\nu\overline{\nu}$ の観測、分岐比の精密測定を目指す。

実験で用いる検出器のすべての信号は Flash ADC (FADC) で波形を記録し、16 台の Level2 Trigger Board から Ethernet で PC farm へ転送したのちに処理される。

この論文では、PC farm 上で処理を行う Data acquisition (DAQ) backend system の開発成果について、動作及び性能評価の結果を述べる。

目 次

図 目 次

表 目 次

リスト一覧

第1章　序論

茨城県東海村にある大強度陽子加速器施設（J-PARC）で行われる KOTO 実験では、長寿命中性 K 中間子の稀崩壊モード $K_L \to \pi^0\nu\overline{\nu}$ の分岐比を測定する。

本論文では、この KOTO 実験で必要となる Data acquisition (DAQ) backend-system の開発成果について説明し、その性能を評価する。

1.1　$K_L \to \pi^0\nu\overline{\nu}$ の物理

弱い相互作用の固有状態は $|K_0\rangle$ と $|\overline{K_0}\rangle$ であり、質量固有状態 $|K_L\rangle$ と $|K_S\rangle$ は、$|K_0\rangle$ と $|\overline{K_0}\rangle$ を用いて、

$$\begin{cases} |K_L\rangle = 1 + 1 = 2! \\ |K_S\rangle = 1 - 1 = 0! \end{cases} \tag{1.1}$$

まだ

1.2　KOTO 実験

KOTO 実験は、長寿命中性 K 中間子 K_L の稀崩壊モード $K_L \to \pi^0\nu\overline{\nu}$ の崩壊分岐比（BR: Branching Ratio）を測定することで、弱い相互作用による CP の破れの大きさに関するパラメータ η を直接測定し、標準理論の検証を行うことを目的とする。

$K_L \to \pi^0\nu\overline{\nu}$ の崩壊分岐比は、標準理論によると 2.8×10^{-11} [1] と予言されている。よって、崩壊事象の観測には $O(10^{11})$ の統計数を期待できる大強度の K_L ビームが必要であり、そのため、茨城県東海村の大強度陽子加速器施設（J-PARC）で実験を行う。また、パイルアップ事象の影響が無視できないので [?]、検出器の全ての信号は Flash ADC（FADC）を用いて波形読み出しをする。

1.2.1　検出器

図 1.1 は検出器の全体配置である。$K_L \to \pi^0\nu\overline{\nu}$ 崩壊モードで生成される π^0 はすぐに 2 本の γ へ崩壊する。また、ν は検出できない。KOTO 実験では、2 本の γ

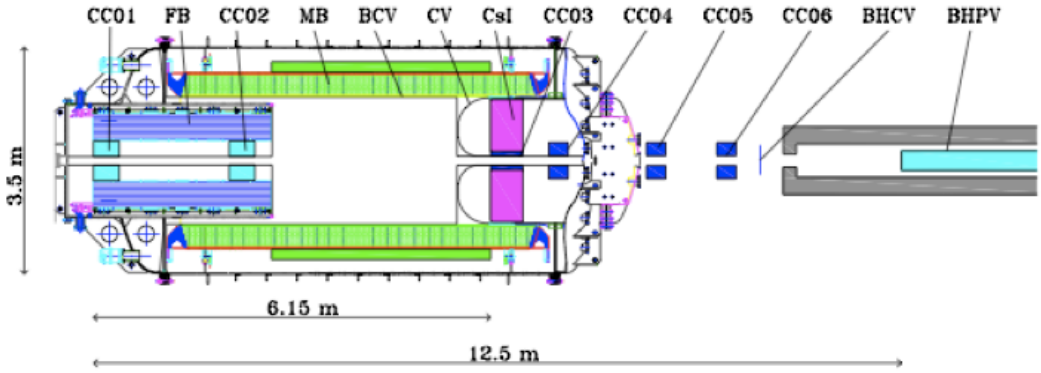


図 1.1: KOTO 実験の検出器。左から入射した K_L が中央の領域で崩壊する事象を領域の全立体角を覆う検出器で観測する。

図 1.2: Flash ADC ボード

を CsI 電磁カロリメーターで検出し、他の崩壊領域を囲む検出器にはなにも検出されないということを確認することで $K_L \to \pi^0\nu\bar{\nu}$ を同定する。

γ を検出するための CsI 電磁カロリメーターは約 2700 本の CsI 結晶で構成され、結晶 1 本づつに PMT を取り付けて信号を読み出す。KOTO 実験の検出器ごとの信号の数を表 1.1 に示す。他の検出器からのものも合わせると、扱う信号の数は約 3000 チャンネルにもなる。

1.3 DAQ

KOTO 実験では、検出器からの約 3000 チャンネルの信号はすべて、フィルターを通した後に Flash ADC を用いて波形を記録する。記録した波形を用いてハードウェアによる Level 1 トリガー、Level 2 トリガー決定が行われる。Level 2 トリガーされたイベントは PC ファームへイーサネット経由で転送されて、ソフトウェアによる Level 3 トリガー処理がなされる。

1.3.1 DAQ システム

Flash ADC ボード

検出器の信号はフィルターを通した後、シカゴ大学が開発を行った 16ch 125MHz 14bit Flash ADC（図 1.2）を使用してその波形を記録する。

表 1.1: 検出器ごとの信号の数、及び使用する FADC の条件

Detector	Channel No.	Readout, # of ADC boards
NCC	48	125-MHz ADC × 3
FB	32	125-MHz ADC × 2
BCV	64	125-MHz ADC × 4
MB	128	125-MHz ADC × 8
MB extra	64	125-MHz ADC × 4
CV	64	500-MHz ADC × 16
LCV	not many	125-MHz ADC × 1 or 2
CC03	32	125-MHz ADC × 2
CC04	48 to 80	125-MHz ADC × 5
CsI	2716	125-MHz ADC × 140
CC05	48 to 80	125-MHz ADC × 5
CC06	48 to 80	125-MHz ADC × 5
BHCV	16	500-MHz ADC × 4
BHPV	25 or 55	500-MHz ADC × 14

Level 1 トリガーボード

Level 1 トリガー決定を行うボード。登場場面は今後ないが、Level 1 トリガーが E_T と Veto 検出器の信号を組み合わせたトリガー決定を行う（未定）ことを明記する？

Level 1 トリガーを発行したことを Level 2 トリガーボードへ伝える。

Level 2 トリガーボード

Level 2 トリガー決定を行うボードであり、PC ファームへ波形情報を送るボード。FADC でサンプリングされた波形情報は、オプティカルファイバーを通して FADC から Level 2 トリガーボードへ送られる。波形情報をもとに Level 2 トリガーを発行されたイベントは、Level 2 トリガーボード上に搭載されたメモリーに格納される。

1 スピルの間にメモリーに格納されたイベント情報は、次のスピルが始まると、PC へ送信され始める。メモリーは 2 つ搭載されていて、スピルごとに切り替えて使用するようになっているので、イベント収集と PC への転送が並行して行える仕様となっている。

Level 2 トリガーボードから PC へのイベント情報の転送は、Ethernet を通して UDP プロトコル形式で送られる。図 1.3 は Level 2 トリガーボードが送るデータ

図 1.3: Level 2 トリガーボードから PC へ送られてくるイベント情報の構造

図 1.4: 各検出器からの信号の伝送の様子。Flash ADC で digitize された信号は Level 2 トリガーボードへ、さらに PC へと転送される。

の構造である。送られるデータは 1 つのイベントの情報の断片であり、断片には、それを識別するための情報である、. . . が記されている。

PC ファーム

複数の Level 2 トリガーボードから送られるイベント情報の断片からイベント情報を再構築（イベントビルド）し、Level 3 トリガー決定を行い、トリガーを発行されたイベント情報をストレージへ保存する。

1.3.2 DAQ フロントエンドでのデータ伝送

検出器からの信号を入力する Flash ADC ボードは検出器ごとに 1 つの VME クレートに挿入される。1 つの VME クレートには

1.3.3 トリガーレートの見積もり

このようなセットアップで実験を行うとき、1 スピル当たりのトリガー数がどれくらいになるかについて述べる。

トリガーと成り得る K_L の崩壊事象

K_L の崩壊モードを表 1.2 に示す。

表 1.2: K_L の主要な崩壊モード

崩壊モード	分岐比
$K_L \to \pi^{\pm} e^{\mp} \nu_e$	(40.55 ± 0.12) %
$K_L \to \pi^{\pm} \nu^{\mp} \nu_\nu$	(27.04 ± 0.07) %
. . .	. . .

シミュレーションによる見積もり

Geant 4[1] を用いたシミュレーションの結果 [?] によると、E_T による Level 1 トリガーのみでは、スピルあたりに期待されるイベント数は...。

Zero Suppression の適用、Level 2 トリガーボートに搭載されるメモリーの容量による制限

前の小々節で述べたスピルあたりのイベント数は、さらに、DAQ システムのハードウェアの仕様により制約される。

1 つのトリガーにより作られた 1 イベントを構成する約 3000 チャンネルの信号のうち、実際にアクティビティのあるチャンネルはそれほど多くない。Flash ADC には、アクティビティの無かったチャンネルの波形情報を除くことで情報の大きさを小さくする、ゼロサプレッション機能が実装されていて、これにより、どういうサプレッション条件の下では、大きさを何パーセントにすることができる。

このゼロサプレスされた情報は、Level 2 トリガー条件を満たすと、Level 2 トリガーボードに搭載されたメモリーへ格納される。

1.3.4 最低要件？

チャンネル数＝大。複数のクレートを使用。push 型 → DAQ ハードウェアの簡略化、Ethernet をフルに使用。L2 16 台からばらばらになったイベントデータが送信されてくる。

また、現在のところ、Level2 trigger board × 16 台が 16Gbps でデータを送ってくるのに対して、KEK へ送る回線の帯域が現状では 2Gbps であり、イベントデータの削減が必須である。

1.4 本研究の意義

そうゆうわけで、そうゆうソフトウェアの開発を行う必要がありましたとさ。

[1] Geant4: http://geant4.cern.ch/　　粒子と物質の相互作用をシミュレートするソフト

第2章　DAQバックエンドの処理

KOTO 実験では、前の章で述べた要請を満たす DAQ バックエンドソフトウェアの開発が必要であった。今回開発を行ったバックエンドソフトウェアは、PC ファーム上で、Level 2 トリガーボードから転送されるイベント情報の受信と同時並行でのイベントビルド、イベント選択、保存という、3 段階の処理を行う。

この章では、これらの処理を順に説明していく。

2.1　全体のフロー図

開発したバックエンドソフトウェア上で行われる処理を、イベント情報の受け渡しに沿って図示したのが 図 2.1 である。図に示したとおり、Level 2 トリガーボードから送られた情報は、受信部、イベント選択部、保存部の順に処理される。

2.2　受信部

受信部では、Level 2 トリガーボードからイーサネットを通して送信されるイベント情報の断片を受信して統合し、そのあとの処理へと情報を受け渡す。

イーサネットを経由した受信操作は OS の提供する機能であり、第 2.2.1 小節で簡単に説明する。

また、前の章で述べたような、16 台の Level 2 トリガーボードから転送されてくるイベント情報の断片を 1 つに統合するイベントビルド処理もこの受信部で行うが、その具体的な手順を第 2.2.2 小節で説明する。

2.2.1　ソケット関数

ソケット関数とは、OS（Operating System: オペレーティングシステム）が提供するネットワーク通信のための機能の 1 つであり、Linux, Windows, Mac OS といった一般的な OS に標準で提供されている。

本開発では、この関数を利用して受信を行うこととした。その動機としては、

- 標準的に利用される機能であり、ドキュメンテーションや解説が充実している

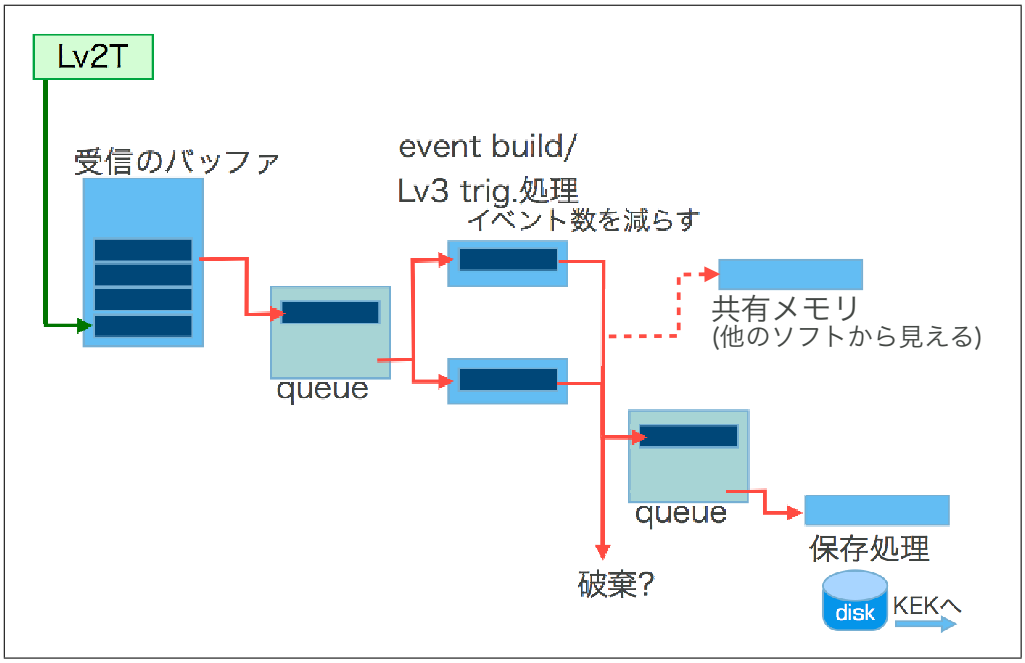


図 2.1: バックエンドソフトウェアの全体像
情報が左（受信部）から右（保存部）へ渡されていく様子を表す。

- OS でバッファリングが行われるので、高頻度で到着するデータを受け取り損ねることがない

が挙げられる。

ソケット関数は、実際にはいくつかの関数の集合である。例として、受信を行う手順は、

1. socket() 関数を呼び出し、ファイルディスクリプタ fd を取得。
2. bind(fd, ...) 関数を呼び出し、どのポート番号宛に来たデータを受信するのかを設定。
3. recv(fd, ...)/recvfrom(fd, ...) 関数などを呼び出し、データを受け取る。

となる（例: リスト 2.1）。

リスト 2.1: socket() 関数を利用した受信の例

```
1  #include <sys/socket.h>
2
3  // create socket.
4  int fd;
5  fd = ::socket(PF_INET, SOCK_DGRAM, 0);
6  // bind address to the socket.
```

```
7   sockaddr_in addr;
8   ::memset(&addr, 0, sizeof(addr));
9   addr.sin_family = AF_INET;
10  inet_aton("192.168.1.1", &addr.sin_addr);
11  addr.sin_port = htons(12345);
12  ::bind(fd, (sockaddr_in*) &addr, sizeof(addr));
13  // receive data.
14  while (1) {
15    char buf[65536];
16    ssize_t recvlen = ::recv(fd, buf, sizeof(buf), 0);
17  }
```

さて、socket() 関数を使用した場合の OS 内の動作から、実験前に必要な OS の設定について述べる。

受信手順にしたがい、まず、socket() を呼び出してネットワーク通信に利用する「ソケット」を 1 つ作成する（図 2.2(a)）。ソフトウェアはこのソケットを介して通信機能にアクセスすることになる。ソケットは「ソケットバッファ」というバッファを持ち、OS が Ethernet から PC へ送られてきたデータをバッファに次々と詰めていくので（図 2.2(b)）、ソフトウェアは recv() 関数を繰り返し呼び出してこのデータを受け取る（図 2.2(c)）。しかし、recv() と次の recv() の間に多量のデータがやってくると、ソケットバッファが溢れてしまい、その溢れたデータを失うことになる（図 2.2(d)）ので、バッファのサイズは十分大きくある必要がある。

バッファサイズはプログラムから

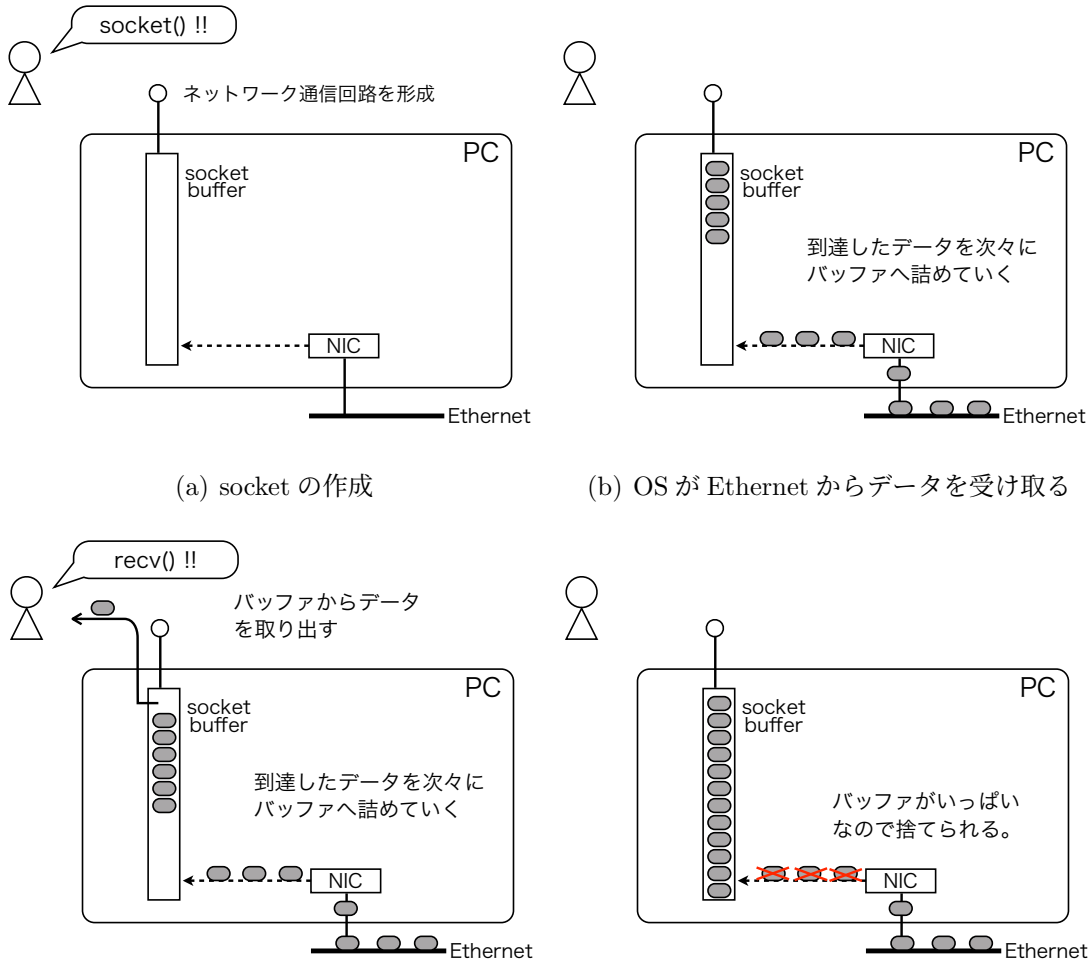
```
1  int fd; // file descriptor for open socket.
2  int val, len = sizeof(val);
3  val = 16777216; //set the size to 16MB.
4  ::setsockopt(fd, SOL_SOCKET, SO_RCVBUF, &val, &len);
```

というように設定できるのだが、Linux ではサイズの最大値に制限が設けられていて、その値より大きな値を設定できないようになっている。KOTO 実験でのデータ転送では、予め設定されている最大値では不十分なことが起こりうるので、制限値をより大きな値に変更する。16MB まで拡張可能にしたければ、端末から、スーパーユーザーになってから

```
# echo 16777216 > /proc/sys/net/core/rmem_max
```

というコマンドを実行する（注: 仕様により実際には倍の 32MB まで拡張可能となる）。確認は以下のとおり。

```
# cat /proc/sys/net/core/rmem_max
```



(a) socket の作成　(b) OS が Ethernet からデータを受け取る

(c) recv() で受信　(d) ソケットバッファ溢れ

図 2.2: ソケット関数を利用して受信する場合の OS 内部の挙動

2.2.2 受信イベントバッファ

受信イベントバッファは、受信したイベント情報を保持するためのテーブル兼保持場所である。このバッファに受信したイベント情報の断片を記録していくとイベントビルドが達成されるのだが、その具体的な仕組みを以下で説明する。

受信イベントバッファの構造は、図 2.3 のような、多数の決まった数の「エントリー」をもつ「ブロック」を複数個保持するものである。

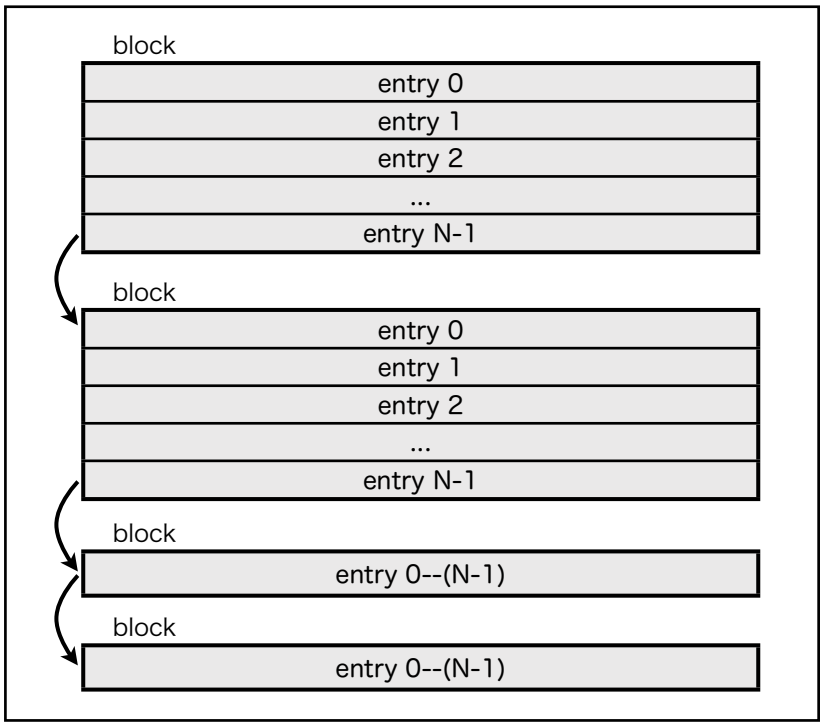


図 2.3: 受信イベントバッファの概略図。N 個のエントリーを持つブロックを連結した構造。

Level 2 トリガーボードから送られてくるイベント情報の断片には、イベントのスピル番号やトリガー番号、タイムスタンプ、どの FADC の情報か、という各種の識別情報が記録されている。この識別情報のうち、スピル番号とトリガー番号から絶対エントリー番号を求める。絶対エントリー番号からは、（ブロック番号、相対エントリー番号）が容易に求まり、これによって、受信した断片の書き込み位置がどのブロックのどのエントリーになるかが一意に決まる。これを、受信した断片に対して順次に行うと、最後には全ての断片が同じところに集まることとなって、イベントビルドが達成されるのである。

なお、実際の実装においては、受信データそのものをエントリーに書きこむのではなく、データ専用の場所へ書き込み、その書き込んだ位置（アドレス）をエントリーに記録するようにした（図 2.4 参照）。データとテーブルを分離する実装をすることで、受信される情報の大半にゼロサプレッションが適用されて情報サ

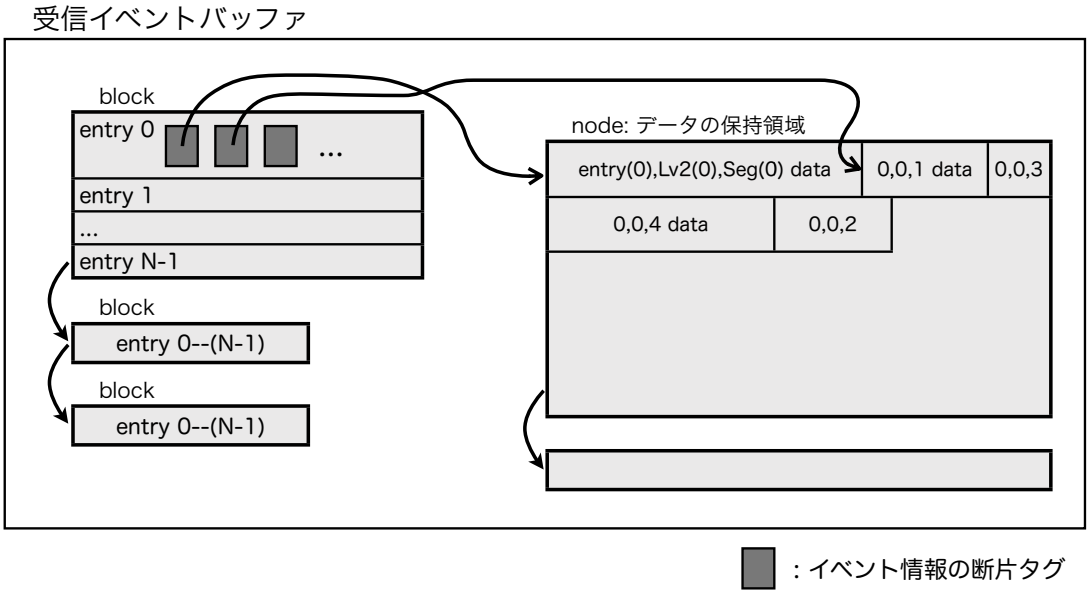


図 2.4: もう少し詳細なデータ保持の方法。
断片のデータ自体は node の上に受信順に前から記録していき、
block の entry にはその所在地を記録する。

イズが小さくなるような高レートな状況においても、領域の無駄が生じるのを避けることが出来る。（分離しない場合、そのような状況下では、テーブルに設けたデータ書き込み用領域が使用されないで無駄になることになる。）

このあとに続くイベント選択処理、保存処理は、このブロックを単位として行う。

以下に該当するブロック、

1. 受信バッファが保持するブロック数が、ある数以上になるとき最も古いブロック
2. 最新のブロックのブロック番号からある数ブロックより古いブロック

がイベント処理部へ送出され、続く処理が始まる。

2.2.3 イベント選択処理待ちキュー

受信部とイベント処理部の間でブロックを受け渡すために存在するのが、このイベント選択待ちキュー（待ち行列、FIFO）である。送出されるブロックはこのキューへ入れられる。

キューを利用することで、受信部がブロックを次へ渡そうとするときにイベント処理部が前のブロック処理の最中でも、終わるまで待つようなことにならずに済む。イベント処理部は、取り掛かっていたブロックの処理が終わった後に、このキューから次のブロックを取り出す。

2.3 イベント選択部

受信部より送出されたブロックのエントリーに記されたイベント情報から、情報の品質や物理に基づいてイベントセレクションを行うのが、このイベント選択部の仕事である。

ソフトウェアの用意する前段階的な処理として、

- 受信したイベント情報の整合性チェック
- サンプリング波形情報を扱いやすいように整列（第?? 節を参照）

を実装してある。これについて、第 2.3.1 小節、第 2.3.2 小節で説明する。

こうして処理された情報を用いて、イベントセレクション処理を行うのであるが、具体的な処理内容は本開発の趣旨でないので省く。

ブロックのエントリー１つ１つに対して、イベント選択処理が行われ、その結果をエントリーに記録していく。全てのエントリーが処理されたら、ブロックは、次の保存部へ送出される。

2.3.1 イベント情報の整合性チェック

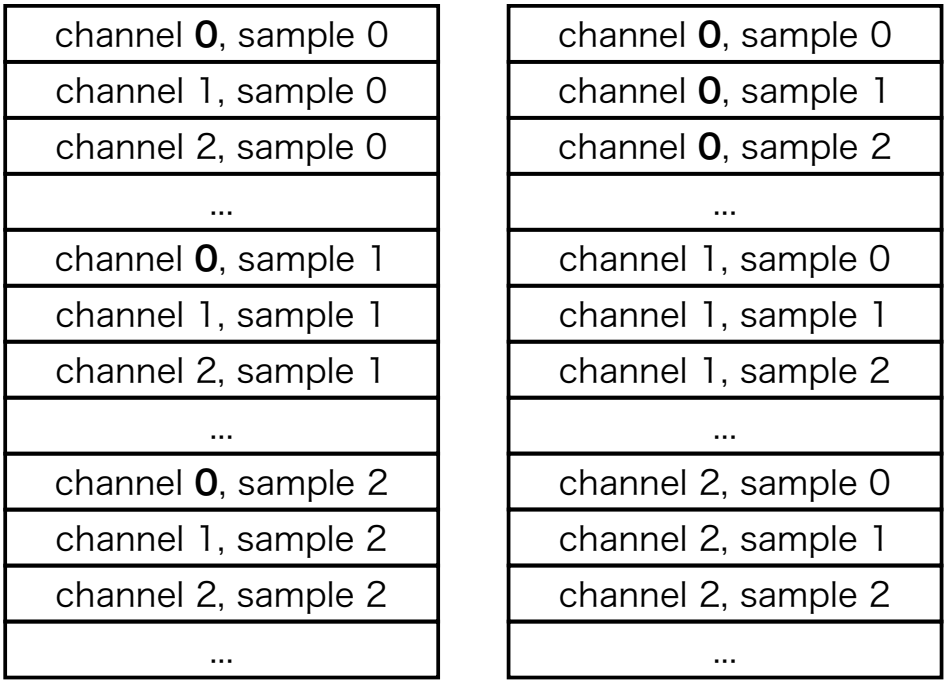
整合性チェックとしては、

- イベント情報を構成する全ての断片を受信できているかを確認
- イベント情報が正しくビルドされているかを、イベントを構成する全ての断片のスピル番号、トリガー番号、タイムスタンプを比較して確認
- イベント情報の断片のサイズ、構造といった、断片自体の正当性確認

などが挙げられる。これらをチェックし、いずれかの項目に失敗したなら、そのイベント（エントリー）に異常マークを付けて、使用しないようにする。

2.3.2 波形情報の整列

Level 2 トリガーボードから送られる情報に記録された FADC のサンプリング波形情報は、図 2.5(a) のような構造をもつ。これでは、チャンネル 0 の波形情報を取得したい場合などに手間なので、図 2.5(b) のように波形情報が連続して記録された構造へ変換する。



(a) 整列前のデータ構造　　(b) 整列後

図 2.5: 整列によってチャンネルデータにアクセスしやすくなる。

2.3.3 保存処理待ちキュー

第 2.2.3 小節のイベント選択処理待ちキューと同様に、イベント選択処理が終わったブロックは保存処理待ちキューを介して保存部へ送出される。

2.4 保存部

イベントセレクション処理がされたイベント情報をファイルへ書き出すのが保存部の仕事である。

保存部の処理は、保存処理待ちキューからイベントセレクション処理がされたブロックを 1 つ取り出すことから始まる。手順としては、

1. 保存処理待ちキューから、1 つブロックを取り出す
2. ブロックの各エントリーに対して、
   (a) イベント選択処理の結果に基づき、イベント情報の一部、あるいは全部を書きだす、あるいは全く書き出さない。
   たとえば、結果が「Accept」ならイベント情報のすべてを書き出す、結果が「Reject」ならイベント情報を書き出さない、という具合。

というようになる。

DAQ システムが正しく動いているかを確認する段階においては、すべての情報を書き出すようにすることになるだろう。このあたりの実装は、今後の状況に応じて変えていくことになるだろう。

2.4.1 書き出しファイル形式

本開発の実装は現在のところ、ランごとにファイルが 1 つ作成されるように実装されている。ファイル構造の概要図を図 2.6 に示す。ファイルの先頭にヘッダー、末尾にフッターを持つ。また、書き出しはブロック単位で行われるので、ブロック 1 つの書き出しのたびに、ブロックヘッダー/フッターが書き込まれる。エントリーの情報はそれらの間に書かれる。

こうした構造を持つことで、書き出し中に、もしバックエンドソフトウェアが予期せずに異常終了した場合でも、直前までに書き込んだデータの復旧が可能となる。

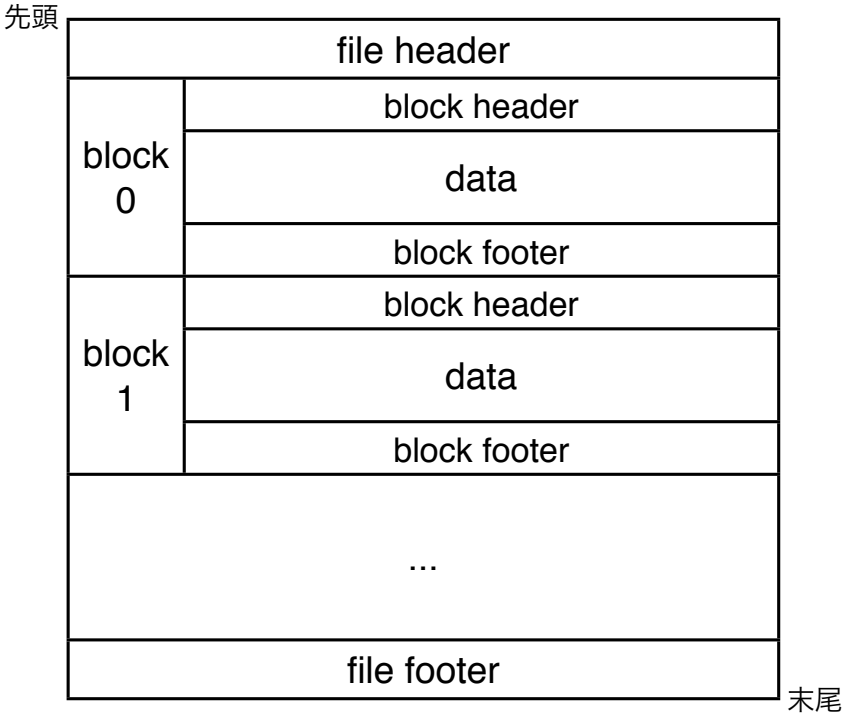


図 2.6: 書き出し形式の概要図

2.4.2 書き出しデータの圧縮

KEK への転送量の軽減、ディスク領域の節約、ディスク処理の負荷軽減などを目的として、書き出されるデータは、情報を失わずにサイズを小さくする圧縮処理を行ったのちにファイルに書き出される。

圧縮処理を行うプログラムはいくつかあり、zip ファイルなどで利用される deflate 方式で圧縮する gzip や、deflate に比べて圧縮率は高くないが処理が軽量な LZO 方式で圧縮を行う lzop というソフトウェアが挙げられる。

また、自分で実装を行った圧縮方法が比較的軽量であったので、これを使うこととする。詳細は以下。

実際にどのような方法を採用するかは、KOTO 実験のビーム強度やイベント選択処理の負荷などとの兼ね合いで選択することにする。

独自実装の圧縮方法

図 2.7 に示すように、波形情報はサンプリング点の個数の 16bit 整数の配列である（14bit の値を 16bit で保持）。

これは、実際には、いつも 16bit の幅を必要とするわけではない。ファイルへ保存される波形情報の大部分は、サンプリング点の高さは 3–5 ビット程度である。こ

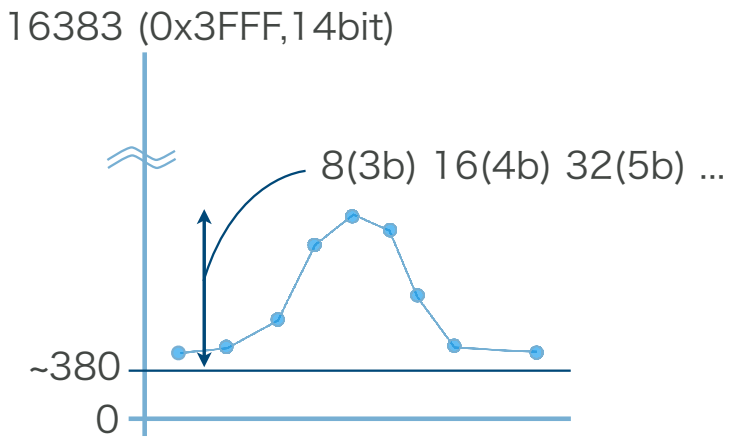


図 2.7: サンプリングされた波形情報の例

の特徴を利用して圧縮を行う。実際に、2010 年 10 月の Run で取られたデータを用いたところでは、約 1/3 程度のサイズまで圧縮できた。

外部アプリケーションでの圧縮

圧縮コマンドの実行には「パイプ」を利用する。パイプを利用することで、あるプログラムの出力を、別のプログラムの入力へつなぐことができる。

まず、端末上でパイプを利用して、テキスト hoge.txt を圧縮コマンド gzip で圧縮してみる。2 つのコマンドを | で繋ぐことで、cat コマンドが出力する hoge.txt の内容を、gzip へ入力して圧縮を行わせることができる。

```
# cat hoge.txt | gzip
```

これと同等のことをソフトウェアで行うコードがリスト 2.2 である。1–11 行目でパイプで繋がった gzip の生成がなされ、19 行目 `write()` でデータを gzip に渡している。

リスト 2.2: ソフトウェアからパイプを利用する

```
1  int fd[2];
2  ::pipe(fd);
3
4  pid_t child_pid;
5  if ((child_pid = ::fork()) == 0)
6  {
7    // Here is in child process.
8    ::close(fd[1]);
9    ::dup2(fd[0], 0);
```

```
10
11      ::execlp("gzip", "gzip", NULL);
12    }
13    else
14    {
15      // Here is in parent process.
16      ::close(fd[0]);
17
18      const char data[] = "test-data desu...";
19      ::write(fd[1], data, strlen(data));
20
21      ::close(fd[1]);
22    }
```

なお、パイプは、パイプで繋がった先のプログラムへ write() をしたときにデータを「パイプバッファ」へメモリーコピーする点から言えば、比較的重い処理かもしれない。

deflate などの圧縮方式を利用する必要が生じた場合は、gzopen() といった zlib の関数を直接利用するべきかもしれない。

2.5 その他の追記

2.5.1 boost ライブラリ

boost[1] という C++ライブラリを利用して開発を行った。これにより、OS の提供する関数の面倒さを隠蔽して開発の負担を軽減できる。

2.5.2 ソフトウェアで利用している技術（？）

書いてはみたが、どうも、必要に思えない。削除するかも。

スレッド

スレッドを利用することで、1 つのプロセスのなかで複数のコードを同時に走らせることが出来る。例えば、リスト 2.3 は、5 行目から A を表示し続ける main_A() 関数が開始され（スレッド A）、6 行目から B を表示し続ける main_B() 関数が開始され（スレッド B）、スレッド A と B を開始した main() は 8 行目で C を表示し続ける結果、A と B と C が順序なくひたすら表示されるプログラムとなる。

[1] http://www.boost.org　2011/01/17 現在の最新版である Version 1.45.0 を使用

リスト 2.3: スレッドの利用例

```
1  #include <boost/thread.hpp>
2
3  int main()
4  {
5    boost::thread thA(main_A);
6    boost::thread thB(main_B);
7
8    while (1) { printf("C"); }
9  }
10
11 void main_A()
12 {
13   while (1) { printf("A"); }
14 }
15 void main_B()
16 {
17   while (1) { printf("B"); }
18 }
```

バックエンドソフトウェアでは、スレッド 1 つに各処理 1 つを割り当てて使用している。

ミューテックス

ミューテックスはプログラムに排他制御を提供するもので、スレッドを複数利用する（マルチスレッド）プログラムで使用される。リスト 2.4 は使用例である。行 20–21 で作成したスレッドと行 22 が `increment_counter()` を呼び出して変数 `counter` を +1 する。あるスレッド T が行 10 の実行でミューテックスをロック状態にすると、このミューテックスに対して他のスレッドはロックすることが出来なくなり、ロック解除を待つ。スレッド T はロックの後の行 11–13 を実行し、行 14 でロック解除をする。こうして、行 11–13 は排他的に実行されることになる。

リスト 2.4: ミューテックスの使用例。行 10 と 14 に挟まれたコードは排他的に実行される。

```
1  #include <boost/thread/mutex.hpp>
2
3  void increment_counter_through_10()
4  {
5    static thread::mutex mtx;
6    static int counter = 0;
7
8    while (1)
9    {
```

```
10      mtx.lock();
11      counter++;
12      if (counter > 10)
13        counter = 0;
14      mtx.unlock();
15    }
16  }
17
18  int main()
19  {
20    boost::thread thA(increment_counter);
21    boost::thread thB(increment_counter);
22    increment_counter();
23  }
```

条件変数

条件変数はリスト 2.5 のように使用し、変数の変更を他のスレッドに通知するのに使用する。例では、メインスレッドは行 13–14 で待機状態（通知が来るまで待つ）になったあと、別スレッドが行 21 で counter の変更を通知し、待機状態を解除している。処理待ちキューで使用。

リスト 2.5: 条件変数の使用例。timer_ep() で counter を 1 にセットしたことを通知する。

```
1  #include <boost/thread/mutex.hpp>
2  #include <boost/thread/condition_variable.hpp>
3
4  boost::condition_variable cond;
5  boost::mutex mutex;
6  int counter = 0;
7
8  int main()
9  {
10   boost::thread thTimer(timer_ep);
11
12   boost::mutex::scoped_lock lock(mutex);
13   while (counter == 0)
14     cond.wait(lock);   // wait until cond.signal() is called.
15   printf("signaled!");
16 }
17 void timer_ep()
18 {
19   sleep(10); // sleep 10 seconds
20   counter = 1;
21   cond.signal(); // and signals.
```

```
22 }
```

共有メモリ

通常、あるプロセスから別のプロセスの仮想アドレス空間にアクセスすることはできない。例えば、プロセス 1 が

```
1  const char *str = "kono mojiretsu ni akusesu shitai.";
2  printf("%x", str);  // str のアドレスが 0x12345678 と表示された
```

であるときに、プロセス 2 から

```
3  const char *str = (const char *) 0x12345678;
4  printf(str); // SEGV
```

というアクセスは出来ない。

そこで、OS にはプロセス間通信機能が用意されている。前の章で利用していた pipe もその 1 種である。

共有メモリは名前通り、プロセス 1 のメモリ空間を他のプロセス 2 と共有する機能である。これを利用すると、プロセス 1 がメモリに書き込んだ内容をプロセス 2 が閲覧/更新できる。

第3章　試験、評価

パケットサイズを変えて。（ゼロサプレッションの程度）小さいパケットをたくさん、というのがスループット的には一番きつい。
イベントレートを変えて。（トリガーレート → ブロックのエントリー数の調整）

3.1　処理ごとの処理能力の評価

3.2　総合しての性能評価

3.2.1　2010年10月 Enginerring Run のデータを用いた評価

第4章　まとめ

「まとめ」だけすんなりここに入れちゃいました。

付 録A　dummy appendix

appendix はこのようになります。

謝辞

謝辞はこのようになります。

参考文献

[1] xxx

[2] W.-M. Yao, *et al.* [Particle Data Group], Journal of Physics G **33**, 1 (2006).