0　実習内容解説

このディレクトリには実習で行う内容が書いてあります。

ex01 から ex10 までで DAQ-Middleware を使う上で必要になる事項 (c++、実習で使うボードから送られてくるデータのフォーマット、デコード方法) を習得します。

ex11 から ex14 で DAQ-Middleware を使ったデータ収集システムの構築を実習します。

- ex01 実習環境確認
- ex02 C++ の簡単な復習 (クラス)
- ex03 ネットワークバイトオーダー
- ex04 char buffer からの数値の取り出し
- ex05 バイナリファイルの読みだし
- ex06 ファイルを読んでデコード
- ex07 ROOT を使ってグラフを書く
- ex08 ファイルを読みながらグラフを画面に表示する
- ex09 ネットワークからデータを読みデコードする
- ex10 nc コマンドでデータを読みグラフを画面に表示する
- ex11 DAQ-Middleware 付属サンプルコンポーネントを動かしてみる
- ex12 Web モードでシステムを動かす
- ex13 ボードを読むシステム (DAQ-Middleware 使用) を動かしてみる (Reader - Logger)
- ex14 ボードを読んでモニターするシステムを DAQ-Middleware で作る (Reader - Monitor)

1 (テーマ) コンパイル環境確認プログラム

1.1 実習内容

ファイルを sandbox 以下にコピーして make を実行し、実行ファイルを作成します。

```
% cd ~/daqmw-tc/sandbox
% cp -r ../ex/ex01 .（最後にカレントディレクトリを示すドット（.）があることに注意）
% cd ex01
% make
```

1.2 確認

- コンパイルでエラーがでないことを確認します。
- sample という実行ファイルができていることを確認します。

1.3 実行方法

```
% ./sample
```

とすると hello, world と画面に表示されるので試してみる。

Source Code: ex01/Makefile

```
1  CC = g++
2  PROG = sample
3  CXXFLAGS = -g -O2 -Wall
4  # LDLIBS += -L/usr/local/lib/ -lmylib
5
6  all: $(PROG)
7
8  clean:
9      rm -f *.o $(PROG)
```

Source Code: ex01/sample.cpp

```
1  #include <cstdio>
2  #include <fstream>
3  #include <iostream>
4
5  using namespace std;
6
7  int main(int argc, char *argv[])
8  {
9      cout << "hello, world" << endl;
10
11     return 0;
```

```
12 }
```

2 (テーマ) C++ の復習

クラスファイルを作りそれを利用するプログラムを作る

2.1 ファイル

- MyClass.h (クラス宣言)
- MyClass.cpp (実装)
- main.cpp (MyClass を使うプログラム)

C++ プログラムは main() 関数からスタートする。main() 関数は main.cpp に書いてある (かならず main.cpp というファイル名でなければいけないわけではない)。

2.2 実習内容

ファイルをコピーして make で実行ファイルを作る:

```
% cd ~/daqmw-tc/sandbox
% cp -r ../ex02 .
% make
```

実行前にコードを見て結果を予想する。次に 実行ファイル main を実行してみる:

```
% ./main
```

また Makefile の中身を見てみる。Makefile については以下を参照。

```
% make clean
```

で*.o ファイル、および実行ファイル (今の場合は main) が消されることも確認しておく。

2.3 コードの変更

- MyClass.h、MyClass.cpp にメンバー変数 m_z を追加し、set_z() メソッド、get_z() メソッドを追加する。
- main.cpp を変更し、set_z()、get_z() を使って値をセット、ゲットするプログラムを書く。

2.4 Makefile の確認

```
% g++ main.c MyClass.c -o main
```

のようにコマンドを投入してもよいが、コードの変更後は再びこのコマンドを投入することになるので Makefile を使って実行ファイル生成は make とだけ入力するとできるようにしておいたほうがよい。

Makefile 中での変数は

```
CC = g++
PROG = main
```

のように定義する。変数を参照するには変数名先頭に $ (dollar) を置き変数名を () でくくる。

ある変数に文字列を追加するには += で追加することができる。ここでは

```
OBJS += main.o
OBJS += MyClass.o
```

のように OBJS 変数に 2 個の文字列を追加している。

Makefile では

```
target: dependencies
    dependencies から target を作成するコマンド
```

という形式で実行するコマンドを書く (2 行目のコマンドを書く行の先頭はスペースではなくてタブである必要がある)。上のように書くと target ファイルと dependencies に書かれたファイル群のタイムスタンプを比較し、target ファイルのほうが古ければ (あるいは target ファイルがなければ) 2 行目に書いたコマンドを実行し target ファイルを作成する。

Makefile 中には複数のターゲットを書くことができる。引数なしに単に make と入力して実行すると Makefile 中の最初のターゲットを生成するコマンドが実行される。通常、最終目標物ターゲットを all として

```
all: program_name
```

と書いておく。

この Makefile を見ると

```
$(PROG): $(OBJS)
```

と書いてあり、 $(OBJS) から $(PROG) を作ることを示してあるが、実際に実行するコマンドは書いていない。これは make コマンドが*.o ファイルから実行ファイルを作成するコマンドをしっているからである。この機能を利用することで Makefile の記述を簡略化することができる。

その他 make についての詳細はコマンドラインから

```
info make
```

あるいは Web 上で http://www.gnu.org/software/make/manual/make.html から参照できる。

Source Code: ex02/Makefile

```
CC = g++
PROG = main
CXXFLAGS = -g -O2 -Wall
# LDLIBS += -L/usr/local/lib/ -lmylib

OBJS += main.o
OBJS += MyClass.o

all: $(PROG)
$(PROG): $(OBJS)

clean:
    rm -f *.o $(PROG)
```

Source Code: ex02/MyClass.h

```
#ifndef _MYCLASS
#define _MYCLASS

#include <iostream>

class MyClass
{
public:
    MyClass();
    MyClass(int x, int y);
    virtual ~MyClass();
    int set_x(int x);
    int set_y(int y);
    int get_x();
    int get_y();
private:
    int m_x;
    int m_y;

};

#endif
```

Source Code: ex02/MyClass.cpp

```
#include "MyClass.h"

// Default Constructor
MyClass::MyClass(): m_x(0), m_y(0)
{
    std::cerr << "MyClass default ctor()" << std::endl;
}

// Constructor with two arguments
MyClass::MyClass(int x, int y): m_x(x), m_y(y)
{
    std::cerr << "MyClass ctor(int, int)" << std::endl;
}

// Destructor
MyClass::~MyClass()
{
```

```
18      std::cerr << "MyClass dtor()" << std::endl;
19  }
20
21  int MyClass::set_x(int x)
22  {
23      m_x = x;
24      return 0;
25  }
26
27  int MyClass::set_y(int y)
28  {
29      m_y = y;
30      return 0;
31  }
32
33  int MyClass::get_x()
34  {
35      return m_x;
36  }
37
38  int MyClass::get_y()
39  {
40      return m_y;
41  }
```

Source Code: ex02/main.cpp

```
1   #include <cstdio>
2   #include <fstream>
3   #include <iostream>
4
5   #include "MyClass.h"
6   using namespace std;
7
8   int main(int argc, char *argv[])
9   {
10      MyClass a;
11      MyClass b(1, 2);
12
13      int x = b.get_x();
14      int y = b.get_y();
15      cerr << "b.m_x: " << x << endl;
16      cerr << "b.m_y: " << y << endl;
17
18      a.set_x(10);
19      a.set_y(20);
20      x = a.get_x();
21      y = a.get_y();
22      cerr << "a.m_x: " << x << endl;
23      cerr << "a.m_y: " << y << endl;
24
25      return 0;
26  }
```

3 (テーマ) ネットワークバイトオーダー

3.1 実習内容

ネットワークバイトオーダーを実感するプログラムを書く。

3.2 方針

整数 0x12345678 を

```
union my_num {
    int num;
    unsigned char buf[4];
};
my_num x.num = 0x12345678;
```

と union を使って定義し、バイトオーダを確認するプログラムを書く。

変換は

```
#include <apra/inet.h>

int x = 0x12345678;
int y = htonl(x);
int z = ntohl(y);
```

のように行う。

htonl() を使うために必要なインクルードファイルは man htonl すると SYNOPSIS のところに書いているとおりで、上の例のように

```
#include <apra/inet.h>
```

が必要。

union は上のように定義したときに x.num とすると int としてアクセスできる。これとおなじ内容で unsigned char としてアクセスするときには x.buf[0], x.buf[1], x.buf[2], x.buf[3] を使う。

出力例:

変数名: 変数アドレス: 配列インデックス: 値

```
x: 0x7fff78597440 0 0x78
x: 0x7fff78597441 1 0x56
```

```
x: 0x7fff78597442 2 0x34
x: 0x7fff78597443 3 0x12
y: 0x7fff78597430 0 0x12
y: 0x7fff78597431 1 0x34
y: 0x7fff78597432 2 0x56
y: 0x7fff78597433 3 0x78
```

サンプルコードはこのディレクトリにおいてあるのでなにを書いたらよいのかわからない場合はこれをコピーして学習する。

Source Code: ex03/Makefile

```
CC = g++
PROG = byte_order
CXXFLAGS = -g -O2 -Wall
# LDLIBS += -L/usr/local/lib/ -lmylib

all: $(PROG)

clean:
    rm -f *.o $(PROG)
```

Source Code: ex03/byte_order.cpp

```
#include <cstdio>
#include <fstream>
#include <iostream>

#include <arpa/inet.h>

union my_num {
    int num;
    unsigned char buf[4];
};

int main(int argc, char *argv[])
{

    my_num x, y;

    x.num = 0x12345678;
    y.num = htonl(x.num);

    printf("0x%x\n", x.num);

    for (unsigned int i = 0; i < sizeof(x.num); i++) {
        printf("x: %p %d 0x%x\n", &x.buf[i], i, x.buf[i]);
    }

    for (unsigned int i = 0; i < sizeof(y.num); i++) {
        printf("y: %p %d 0x%x\n", &y.buf[i], i, y.buf[i]);
    }


    return 0;
}
```

4 (テーマ)char buffer からの数値の取り出し

デコードのときに必要になるので char buf[1024] のようなバッファからの数値の取り出し方法を習得する。

下のようなバッファから int, short を取り出すには次のようなコードを書けばよい。

```
/*
 *  0      1      2      3      4      5      6      7
 * +------+------+------+------+------+------+------+------+
 * | 0x01 | 0x23 | 0x45 | 0x67 | 0x89 | 0xab | 0xcd | 0xef |
 * +------+------+------+------+------+------+------+------+
 * <--------------------------><------------><------------>
 *              int                  short         short
 */

unsigned char  buf[8];  // データが入っているとする
unsigned int   *int_p;  // int（4バイト整数へのポインタ）
unsigned short *short_p // short（2バイト整数へのポインタ）
int   x;
short y;

int_p = (unsigned int *)&buf[0]; // buf[0]のアドレスをセット。型が違うのでキャスト(unsigned int *)が必要
x = *int_p;                      // *を付けると値が取り出せるので x に代入している
                                 // ネットワークバイトオーダからホストオーダに変更する必要があるなら
                                 // x = ntohl(x); とする。

// short も同様
short_p = (unsigned short *)&buf[4]; // buf[4]のアドレスをセット
y = *short_p;                    // ネットワークバイトオーダからホストオーダに変更する必要があるなら
                                 // y = ntohs(y); とする。

// buf[6] buf[7] の short の取り出しも同様
```

ソースファイルはこのディレクトリにあるのでコードを読み適当に変更して試してみる。

Source Code: ex04/Makefile

```
CC = g++
PROG = extract_from_buf
CXXFLAGS = -g -O2 -Wall
# LDLIBS += -L/usr/local/lib -lmylib

all: $(PROG)
# OBJS += $(PROG).o
# $(PROG): $(OBJS)

clean:
    rm -f *.o $(PROG)
```

Source Code: ex04/extract_from_buf.cpp

```
/*
 *  0      1      2      3      4      5      6      7
 * +------+------+------+------+------+------+------+------+
 * | 0x01 | 0x23 | 0x45 | 0x67 | 0x89 | 0xab | 0xcd | 0xef |
 * +------+------+------+------+------+------+------+------+
 * <---------------------------><------------><------------>
 *              int                  short          short
 */

#include <sys/time.h>

#include <err.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    unsigned int   * int_p;
    unsigned short * short_p;
    unsigned int   x;
    unsigned short y;

    unsigned char buf[8];

    buf[0] = 0x01;
    buf[1] = 0x23;
    buf[2] = 0x45;
    buf[3] = 0x67;
    buf[4] = 0x89;
    buf[5] = 0xab;
    buf[6] = 0xcd;
    buf[7] = 0xef;

    int_p = (unsigned int *)&buf[0];
    x = *int_p;
    printf("x (buf[0, 3]): 0x%x (dec: %d)\n", x, x);

    short_p = (unsigned short *)&buf[4];
    y = *short_p;
    printf("y (buf[4, 5]): 0x%x (dec: %d)\n", y, y);

    short_p = (unsigned short *)&buf[6];
    y = *short_p;
    printf("y (buf[6, 7]): 0x%x (dec: %d)\n", y, y);

    return 0;
```

```
50 }
```

5 (テーマ) バイナリファイルの読みだし

バイナリファイルを読んでみる。

5.1 実習内容

バイナリファイルを読むプログラムを書く。バイナリファイルは bs/sample.dat にある (これは実習に使うボードから読んだデータ)。

5.2 方針

サンプル例は ~/daqmw-tc/bs/fread にあるのでわからない場合はこれを見て 学習する。

5.3 ファイルのオープン

バイナリファイルはテキストファイルと違って行の概念がないのでどこまで読めばよいのか自動できめることができない。読むバイト数を指定する必要がある。

ファイルを読む前に fopen() する必要がある (テキストファイルと同様)。

```
#include <stdio.h>

FILE *fp = fopen("filename", "r");
if (fp == NULL) {
    // エラー処理
}
```

エラー処理の方法はいろいろあるだろうが、ここでは err() 関数を使う。

```
err(1, "fopen");
```

とすると

```
progname: fopen: 失敗した理由
```

が標準エラー出力に出力され、終了ステータス 1 で終了する。

5.4 読みだし

open できたら fread() 関数を使って読む。

```
char buf[1024];
```

```
int n = fread(buf, 1 /* bytes */, 128, fp);
```

とすると、「1 バイトのオブジェクト」を「128 個」読むのを試みる。戻り値はよみとった「オブジェクト数」。 128 バイト読むには、他に

```
fread(buf, 8, 16, fp); // 8 バイトを 16 個読む
fread(buf, 16, 8, fp); // 16 バイトを 8 個読む
```

というのも考えられる。戻り値がそれぞれ違う (一番下の例では 128 バイト読めたときに 16 を返す)。通常何バイト読めたか知りたいことが多いと思うので第 2 引数は 1 を指定するのがよいかと思う。(オブジェクトバイト数, 読みだし回数) の組合せがいろいろある場合もあるが、効率よく読めるという組合せは特にないので、自分が欲しい戻り値で決めればよい。

ファイル終端などリクエストしたバイト数未満しか読めない場合はエラーとはならず読み取ったオブジェクト数を返す。指定したバイト数未満しか読めないのは異常を示していると思う場合にはそれなりにエラーメッセージを出したほうがよい。

fread() の戻り値として 0 が返ってきた場合の状態は

- エラー
- ファイル終端まで達した (EOF)

のふたつがあり、戻り値だけでは区別がつかない。man fread で RETRUN VALUE のところを読むこと。エラー、EOF の判定には ferror()、feof() 関数を使えと書いてある。

6 (テーマ) ファイルを読んでデコード

ローデータをデコードできるようにする。

6.1 実習内容

実習に使うボードからとったデータをデコードするルーチンを書く。できたデコードプログラムは最終的に DAQ-Middleware コンポーネントに組み込むことになる。

6.2 データフォーマット

まずデータフォーマットをしらなければならない。 データフォーマットは~/daqmw-tc/doc/raw-data-packet-format.pdf にある。Linux 上で PDF ファイルを読むには evince プログラムを使う:

```
% evince ~/daqmw-tc/doc/raw-data-packet-format.pdf
```

ヘッダ部と、データ部にわかれている。ヘッダ部の長さは 12 バイトで固定である。このなかに次に続くデータ部の長さが書いてある (バイトを単位)。その他ヘッダには

- データタイプ (0xf) (他のデータパケットと見分けるのが目的)
- Word length (2) (1 トリガ 1 チャンネル 1 ウインドウデータのバイトサイズ)
- number of Ch (16) (チャンネル数)
- データ長
- トリガーカウント (0 が最初。トリガーがかかるごとに 1 づつ増える)

の情報が入っている。複数バイトで構成される数値はいずれもネットワークバイトオーダになっているので、PC で扱うには変換が必要になる。

word length とチャンネル数とデータ長から、何ウインドウ分のデータが入っているのかがわかる:

ウインドウ数 = データ長 / ((word length)*(チャンネル数))

6.3 行う作業内容

プログラムは ~/daqmw-tc/bs/read_file_decode/ にあるのでこれをコピーして 使う:

```
% cd ~/daqmw-tc/sandbox
% cp -r ../ex/ex5 .
```

デコード部分のメソッドが書いてないのでこれを埋めること。

サンプルデータは ~/daqmw-tc/bs/sample.dat にある。このファイルのデータ部をデコードして

```
trg: XXX ch: XXX window: XXX data: XXX
```

と並べたものを ~/daqmw-tc/bs/ascii.sample としておいてあるので自分で書いたプログラムで OK かどうかはこれと同じフォーマットで出力するようにして比較することで可能である。比較にはたとえば diff プログラムを使うと機械的にできる。

```
% diff -u file_a file_b
```

違いがなければなにも出力されない。

ファイル

- Makefile
- RawDataPacket.h デコードルーチンクラスファイル
- RawDataPacket.cpp デコードルーチンクラス実装 (各メソッドが書いてないので埋める)
- read_file_decode.cpp fread() を使ってファイルを読む (このなかで RawDataPacket で実装したメソッドを使っている。main() はこのなかにある)。

実装するメソッド (ヘッダデータを読むところ)

- is_raw_data_packet()
- get_word_size()
- get_data_length()
- get_num_of_ch()
- get_window_size()

get_window_size() はデータ長/(ワードサイズ*チャンネル数) で求めた window 数を返すこと。

データ部を読むところ

- get_data_at(int ch, int window)

データ部は window ごとにまとまっていてひとつのチャンネルのデータが連続しているわけではない。デコードする際にはチャンネルごとのデータがほしいことが多いかと思うので、引数にチャンネル番号、window を指定することにした。

6.4 解答例

各メソッドを実装したものを~/daqmw-tc/bs/read_file_decode/ においてある。

Source Code: ex06/Makefile

```
CC = g++
PROG = read_file_decode
CXXFLAGS = -g -O2 -Wall
# LDLIBS += -L/usr/local/lib/ -lmylib

OBJS += $(PROG).o
OBJS += RawDataPacket.o

all: $(PROG)

$(PROG): $(OBJS)
RawDataPacket.o: RawDataPacket.h RawDataPacket.cpp

clean:
    rm -f *.o $(PROG)
```

Source Code: ex06/RawDataPacket.h

```
//
// Data Format
// 31                               0
// +---------------------------------+
// |  (*)  | n_ch | unused | unused |
// +---------------------------------+
// |     Data  Length (data only)    |
// +---------------------------------+
// |          Trigger Count          |
// +---------------------------------+  <-- Window # 0
// |   ADC ch # 0   |  ADC ch # 1    |
// +---------------------------------+
// |   ADC ch # 2   |  ADC ch # 3    |
// +---------------------------------+
// |   ADC ch # 4   |  ADC ch # 5    |
// +---------------------------------+
// |   ADC ch # 6   |  ADC ch # 7    |
// +---------------------------------+  <-- Window # 1
// |   ADC ch # 0   |  ADC ch # 1    |
// +---------------------------------+
// |   ADC ch # 2   |  ADC ch # 3    |
// +---------------------------------+
// |   ADC ch # 4   |  ADC ch # 5    |
// +---------------------------------+
// |   ADC ch # 6   |  ADC ch # 7    |
// +---------------------------------+
//
// (*) Bit 7 - 4: type: (0xf)
//     Bit 3 - 0: one event data byte size (2 bytes)
// ADC Data: Bit 15 - 12: channel number
//           Bit 11 -  0: Data

#ifndef _RAWDATAPACKET
#define _RAWDATAPACKET

#include <arpa/inet.h> // for ntohs(), ntohl()
#include <iostream>

class RawDataPacket
{
public:
    RawDataPacket();
    virtual ~RawDataPacket();
    int set_buf(const unsigned char *buf, int buf_len);
```

```
45      bool is_raw_data_packet();
46      int  get_word_size();
47      int  get_num_of_ch();
48      int  get_data_length();
49      int  get_window_size();
50      int  get_trigger_count();
51      int  get_buf_pos();
52      int  get_buf_len();
53      unsigned int get_data_at(int ch, int window);
54      int  reset_buf();
55      const static int HEADER_SIZE = 12;
56      const static int FORMAT_POS  = 0;
57      const static int N_CH_POS    = 1;
58      const static int LENGTH_POS  = 4;
59      const static int TRIGGER_POS = 8;
60
61  private:
62      const unsigned char *m_buf;
63      int m_buf_len;
64
65  };
66
67  #endif
```

Source Code: ex06/RawDataPacket.cpp

```
1   #include "RawDataPacket.h"
2
3   RawDataPacket::RawDataPacket(): m_buf(NULL), m_buf_len(-1)
4   {
5   // no code required.  leave this method empty
6   }
7
8   RawDataPacket::~RawDataPacket()
9   {
10  // no code required.  leave this method empty
11  }
12
13  int RawDataPacket::set_buf(const unsigned char *buf, const int buf_len)
14  {
15      m_buf = buf;
16      m_buf_len = buf_len;
17
18      return 0;
19  }
20
21  bool RawDataPacket::is_raw_data_packet()
22  {
23      // write this method
24  }
25
26  int RawDataPacket::get_word_size()
27  {
28      // write this method
29  }
30
31  int RawDataPacket::get_data_length()
32  {
33      // write this method
34  }
35
36  int RawDataPacket::get_trigger_count()
37  {
38      // write this method
39  }
40
41  int RawDataPacket::get_num_of_ch()
42  {
```

```
43      // write this method
44  }
45
46  unsigned int RawDataPacket::get_data_at(int ch, int window)
47  {
48      // write this method
49  }
50
51  int RawDataPacket::get_window_size()
52  {
53      // write this method
54  }
55
56  int RawDataPacket::reset_buf()
57  {
58      m_buf = NULL;
59      m_buf_len = -1;
60
61      return 0;
62  }
```

Source Code: ex06/read_file_decode.cpp

```
1   #include <err.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <iostream>
5
6   #include "RawDataPacket.h"
7
8   using namespace std;
9   int usage()
10  {
11      cerr << "Usage: readfile filename" << std::endl;
12
13      return 0;
14  }
15
16  int main(int argc, char *argv[])
17  {
18      FILE *fp;
19      RawDataPacket r;
20      char *filename;
21      unsigned char buf[1024*1024];
22      int n;
23
24      if (argc != 2) {
25          usage();
26          exit(1);
27      }
28
29      filename = argv[1];
30      fp = fopen(filename, "r");
31      if (fp == NULL) {
32          err(1, "fopen for %s", filename);
33      }
34
35      for ( ; ; ) {
36          // Read Header Part
37          n = fread(buf, 1, RawDataPacket::HEADER_SIZE, fp);
38          if (n == 0) {
39              if (feof(fp)) {
40                  break;;
41              }
42              else if (ferror(fp)) {
43                  exit(0);
44              }
45          }
```

```
46          else if (n != RawDataPacket::HEADER_SIZE) {
47              errx(1, "partial read %d bytes.  Should be %d bytes", n, RawDataPacket::HEADER_SIZE);
48          }
49
50          // Set header part to decode
51          r.set_buf(buf, n);
52          // Decode.  Verify Type
53          if (! r.is_raw_data_packet()) {
54              cout << "Not a RawDataPacket" << endl;
55              exit(1);
56          }
57
58          // Get Data length
59          int data_length   = r.get_data_length();
60          //cout << "data_length:   " << data_length   << endl;
61
62          // Read Data Part
63          n = fread(&buf[RawDataPacket::HEADER_SIZE], 1, data_length, fp);
64          if (n == 0) {
65              if (feof(fp)) {
66                  break;;
67              }
68              else if (ferror(fp)) {
69                  exit(0);
70              }
71          }
72          else if (n != data_length) {
73              errx(1, "partial read %d bytes.  Should be %d bytes", n, data_length);
74          }
75
76          // Get window size, trigger count, number of channels
77          int window_size   = r.get_window_size();
78          int trigger_count = r.get_trigger_count();
79          int n_ch          = r.get_num_of_ch();
80
81          //cout << "window_size:   " << window_size   << endl;
82          //cout << "trigger_count: " << trigger_count << endl;
83
84          // Decode data
85          for (int ch = 0; ch < n_ch; ch ++) {
86              for (int w = 0; w < window_size; w++) {
87                  unsigned short data = r.get_data_at(ch, w);
88                  cout << "trg: "     << trigger_count;
89                  cout << " ch: "      << ch;
90                  cout << " window: " << w;
91                  cout << " data: "   << data;
92                  cout << endl;
93              }
94          }
95          r.reset_buf();
96      }
97
98      return 0;
99  }
```

7 (テーマ) ROOT を使ってグラフを書く

7.1 実習内容

この実習では、ボードから読んだデータはヒストグラムではなくてグラフを書くことでモニターコンポーネントを作成する。そのためにここで適当なデータから ROOT でグラフを書けるようにする。

7.2 方針

素粒子原子核分野では ROOT http://root.cern.ch/ を使ってヒストグラム、グラフを書くことが多い。ヒストグラムを書く方法は周知と思うがグラフはヒストグラムよりは書かれる機会がないと思うのでここで取り上げる。

~/daqmw-tc/bs/draw_graph に放物線上の点を指定して書くプログラムをおいてある。コピー、コンパイルして動かし、コードを読みグラフの書きかたを習得すること。

ROOT では C++ プログラムをコンパイルするのに必要なインクルードファイル、ライブラリファイルの指定を簡略化するために root-config というコマンドを用意している。Makefile 中で以下のように指定しているので見ておく。

```
CPPFLAGS += -I$(shell ${ROOTSYS}/bin/root-config --incdir)
LDLIBS   += $(shell ${ROOTSYS}/bin/root-config --glibs)
```

8 (テーマ) ファイルを読みながらグラフを画面に表示する

8.1 実習内容

ファイルを読みながらグラフを画面に表示できるようにする。

8.2 方針

いままでの実習でファイルを読みデコードすること、および ROOT を使ってグラフを書くことができるようになったのでこのふたつを合体させて、ファイルを読みながらグラフを書くことができるようになった。ここでそのプログラムを書く。

1 トリガー分データを読んで、チャンネル 0 のデータのグラフを書き画面に表示するプログラムを書く。続いてキャンパスを分割してチャンネル数分だけグラフを書く。

8.3 解答例

プログラム例は ~/daqmw-tc/bs/read_file_graph にある (これはチャンネル 0 だけをグラフ化している)。

9 (テーマ) ネットワークからデータを読みデコードする

9.1 実習内容

- ネットワークから送られてくるデータを nc コマンドを使ってセーブする。
- 送られてきたデータを nc コマンドを使って読みデコードする。

9.2 ボードの説明

データフォーマットについてはすでに説明したとおり。

本来はなにかトリガーをかける回路を用意したいところであるが準備が大変なので、お手軽に UDP でパケットを送るとトリガーがかかるような FPGA ファームウェアを作った。

データは TCP で送られてくる。ボードの IP アドレスは 192.168.10.16、ポートは 24 である。

9.3 データを nc コマンドを使ってセーブする

TCP で送られてくるデータは Linux 上で nc コマンドで読むことが できる。

```
% nc 192.168.10.16 24 > data
```

としてボートと TCP で接続する。まだトリガーがかかっていないのでデータはやってきていない (別ターミナルを開いて ls -l data を実行しファイルサイズをみることで確認できる)。

トリガーをかけるために

```
% ~/daqmw-tc/trigger/trigger.py
```

を起動する。start ボタンを押すと UDP パケットが送られ、トリガーがかかる (トリガーレートのデフォルトは 10Hz にセットされている)。数秒後に stop ボタンを押す。data ファイルに読んだデータが入っている。バイナリなので cat コマンドで内容は表示することはできない。hexdump コマンドで 16 進数ダンプで表示することができる。

```
% hexdump -vC data
```

9.4 パイプで接続

すでにファイルを読んでデコードするプログラムを書いたのでこれを改造してパイプでつないでネットワークからデータを読んでデコードするように改造する。

```
% nc 192.168.10.16 24 | ./decode_program
```

として起動すると nc からの出力が decode_program に標準入力として入力される。標準入力のファイルハンドラはプログラム起動時にすでに stdin として使えるようになっている。

```
FILE *fp;
fp = fopen("data", "r");
```

となっていた部分 (および fopen() エラー処理部分) を

```
fp = stdin;
```

とすると標準入力を読むようになるのですでに書いたコードを変更し試してみる。

10 (テーマ) nc コマンドでデータを読みグラフを画面に表示する

10.1 実習内容

nc コマンドでデータを読んでグラフを画面に表示する。

10.2 方針

前の例題とおなじく、ファイルを読んでグラフを書くプログラムを改造すると nc コマンドでデータを読んでグラフを画面に表示できるようになる。

また tee コマンドを使って入力をファイルに分流するとデータのセーブとモニターの両方ができるシステムができる。

```
% nc 192.168.10.16 24 | tee data | ./graph_command
```

この簡易システムの問題点は

- パイプ中のバッファサイズを自分で決めることができないのでデータ速度が大きいと対応がむずかしい
- ストップに Control-c を使うのでデータが途中で切れてしまう

などである。

以上で DAQ-Middleware を使わないでデータをグラフ化するプロジェクトは終了です。

11 (テーマ) DAQ-Middleware 付属サンプルコンポーネントを動かしてみる

DAQ-Middleware でのデータ収集システムがどういうふうに動くのかかんじをつかむために付属のサンプルコンポーネントを動かしてみる。

以下のような簡単なシステムを試してみる

emulator - SampleReader - SampleMonitor

11.1 ソースコードのコピーとコンパイル

サンプルコンポーネントのソースコードは /usr/share/daqmw/examples ディレクトリ 以下に入っているのでそれをコピーする。

```
% mkdir ~/MyDaq
% cd ~/MyDaq
% cp -r /usr/share/daqmw/examples/SampleReader .
% cp -r /usr/share/daqmw/examples/SampleMonitor .
```

続いてコンパイルを行う

```
% cd SampleReader
% make
(SampleReaderComp というファイル名の実行形式ファイルができる)
% cd ../SampleMonitor
% make
(SampleMonitorComp というファイル名の実行形式ファイルができる)
% cd ..
```

11.2 コンフィギュレーションファイルの作成

SampleReader - SampleMonitor システムのコンフィギュレーションは/usr/share/daqmw/conf/sample.xml にあるのでこれをコピーする。

```
% cd ~/MyDaq
% cp /usr/share/daqmw/conf/sample.xml .
```

sample.xml 中に execPath(2 箇所ある) というタグで SampleReaderComp、SampleMonitorComp の実行形式ファイルがあるパスを指定しているので正しいパスかどうか確認する。

11.3 コンポーネントの起動

SampleReaderComp、SampleMonitorComp およびこれらを統括する DaqOperator を起動する。起動は run.py というコマンドを使う。run.py の引数にコンフィギュレーションファイルを指定する。run.py はコンフィギュレーションファイルを解析し、execPath タグで指定してあるファイルを起動し、/usr/libexec/daqmw/DaqOperator/DaqOperatorComp を起動する。

(註)

DaqOperatorComp は通常書き換える必要がないのですでにコンパイル済みのものが入っている。

```
% run.py -c -l sample.xml
```

あるいはオプションは、まとめることができるので

```
% run.py -cl sample.xml
```

オプションの意味は、-l: run.py を起動した PC で DAQ コンポーネントを起動する。 -c: コンソールモードでシステムを起動する である。

ターミナルに DaqOperator が出力するパネルがでる。

11.4 システム起動

数値入力待ちになっているので 0 を押しエンターキーを押すと confiugred になる (以下数字キーを押したあとは同様にエンターキーを押すこと)。1 を押すとラン番号を入力するようなながされるので適当に番号 (1 とか 2 とか) を入力する。ラン番号を入力するとデータ収集を開始する。FATAL エラーとなるが今の手順ではエミュレータを起動していないのでこれは正しい動作である。 2 を押すとデータ収集を終了する。

11.5 エミュレータの起動

ターミナルを新たに開いて

```
% daqmw-emulator
```

でデータ生成エミュレータを起動する。起動後このターミナルはさわらないのでじゃまならアイコン化しておくなどする。

11.6 データ収集再開

データ生成エミュレータが起動している状態で 1 を押すとデータ収集がされ、画面上にヒストグラムが表示される。

11.7 システム終了

2 を押してデータ収集システムを終了させ、3 を押し unconfigured 状態にしたあと control-c を押すとシステムが終了する。

12 (テーマ) Web モードでシステムを動かす

DAQ-Middleware を稼働させる方法にはコンソールモードと Web モードがある。コンソールモードで動かす方法はすでに試したので、次に Web モードで動かすのを試してみる。

まず apache が起動しているかどうかを確認する。

```
root# service httpd status
```

httpd (pid 12345) is running... とでる場合は httpd は起動しているので OK。

httpd is stopped とでた場合は起動していないので起動する:

```
root# service httpd start
```

OS 起動時に自動起動するようにするためには以下のコマンドを実行する:

```
root# chkconfig httpd on
```

次に DAQ-Middleware の起動の説明に移る。

Web モードで起動するには run.py を-c なしで起動する。

```
% run.py -l sample.xml
```

なにも表示されなければ正常に起動できている。 Web ブラウザを使って http://localhost/daqmw/OperatorPan にアクセスすると Web UI が起動する。

configure, begin, end, unconfigure のボタンがあるので適当にボタンを押す。状態により押しても無意味なボタンはグレーアウトしている。

ラン番号は自動で 1 づつ増えていくようになっている。

13 (テーマ) ボードを読むシステム (DAQ-Middleware 使用) を動かしてみる (Reader - Logger)

ここではボードからデータを読んでデータを保存するシステムを動かしてみる。作業の順番としては

1. RawDataReader コンポーネントの作成
2. RawDataLogger コンポーネントの作成
3. コンフィギュレーションファイルの作成
4. システム起動、ラン
5. trigger.py でボードにトリガーを送る
6. trigger 停止
7. データがセーブされていることを確認

8. 13.1 RawDataReader コンポーネントの作成

Reader はすでに daqmw/RawDataReader/ディレクトリにあるのでこれをコピーして使う。

```
% cd
% mkdir RawData（ホームディレクトリに RawData ディレクトリを作成。作るシステムはこの下にいれる）
% cd RawData
% cp -r ~/daqmw-tc/daqmw/RawDataReader .
% cd RawDataReader
% make
```

13.2 2. RawDataLogger コンポーネントの作成

ほとんどのシステムではデータをファイルに保存する DAQ コンポーネントは DAQ-Middleware に例題として付属している SampleLogger コンポーネントを使うことができる。SampleLogger コンポーネントは

- データをコンフィギュレーションファイルの dirName パラメータで指定したディレクトリに保存する
- ファイルが maxFileSizeInMegaByte パラメータで指定するバイト数を越えたら次のファイルに書く (イベント途中で次のファイルに移ることはない。Reader コンポーネントが 1 回に送ったデータを途中で分断することはしない)。
- 保存するファイル名は YYYYMMDDTHHMMSS_RRRRR_BBB.dat (RRRRR は run 番

号、BBB は 000 から始まる連番)。例:20110202T143748_000000_000.dat

- パラメータの isLogging が yes になっている場合に実際にデータを保存する。no になっている場合はデータは保存しない (テストに使ったりする)

と動作する。

コピーしてそのままの名前で使ってもよいが、少し改造する場合もある (たとえば「イベントデータのなかに時刻情報が入っていないので Logger コンポーネントが動いている PC から時刻情報を取得して時刻情報を追加して書くようにするなど)。その場合は動作仕様が変わってくるので名前を SampleLogger から変更したほうがよい。ここで Sample*コンポーネントの名前を変更する方法を学ぶ。SampleLogger から RawDataLogger に名前を変更することにする。

コード上では大文字小文字を以下のように使い分けている。かならずこのようにしなければならないわけではないが、習慣上そうなっているので引継が生じるなどの場面を考えると守っておいたほうがよい。

- samplelogger (全部小文字) InPort、OutPort の名前
- SampleLogger (大文字小文字ミックス。最初の文字、および単語の切れ目は大文字) ファイル名、コンポーネント名
- SAMPLELOGGER (全部大文字) インクルードガード

SampleLogger コンポーネントは /usr/share/daqmw/example/SampleLogger/ ディレクトリにある。これをコピーして、手で修正してもよいが間違いが生じる可能性が大きいので sed コマンドを使ったスクリプトを用意したのでこれを使う (現在の DAQ-Middleware にはまだないが今後追加する予定になっている)。

```
% cd ~/RawData
% cp -r /usr/share/daqmw/examples/SampleLogger .
% mv SampleLogger RawDataLogger（ディレクトリ名の書き換え）
% cd RawDataLogger
% cp ~/daqmw-tc/daqmw/utils/change-SampleLogger-name.sh .
% chmod +x change-SampleLogger-name.sh
(change-SampleLogger-name.sh のなかみを見てみる）
(RawDataLogger ではない名前に変更する場合は new_name_camel_case を変更する）
% ./change-SampleLogger-name.sh
% grep -i sample * などして Sample，sample の文字が残っていないかどうか確認
% make
```

これで RawDataLoggerComp という実行ファイルができる。

13.3 3. コンフィギュレーションファイルの作成

コンフィギュレーションファイルを作る。ほぼ完成された雛型を~/daqmw-tc/daqmw/reader-logger.xml に用意してあるのでそれを編集して使う。

```
% cd ~/RawData
% cp ~/daqmw-tc/daqmw/reader-logger.xml .
% reader-logger.xml の execPath を自分の環境にあわせて編集
```

13.4 4. システム起動、ラン

```
% run.py -cl reader-logger.xml
```

でシステムを起動し、0, 1 とキーを押してシステムを動かす。

13.5 5. trigger.py でボードにトリガーを送る

```
% ~/daqmw-tc/trigger/trigger.py
```

として start ボタンを押しトリガーを送る。読みだしシステムのイベントバイト数が増えるのを確認する。

13.6 6. trigger 停止、システム停止

trigger GUI の stop ボタンを押しトリガーを送るのを停止する。2 を押してシステムを止める。

13.7 7. データがセーブされていることを確認

/tmp/ディレクトリにデータが保存されているのを確認する。hexdump コマンド、あるいは以前作ったファイルを読んでデコードするプログラムで正当性を確認する。

14 (テーマ) DAQ-Middleware でモニターコンポーネントを開発する

ボードからデータを読んで保存できるようになったので、保存ではなくてオンラインでグラフを書くモニターコンポーネントを開発する。ex09 で行った nc コマンドで読んでパイプ経由でグラフを書いたシステムの DAQ-Middleware 化を行う。

作業の順番としては

1. RawDataReader - RawDataLogger コンポーネントを作ったディレクトリに RawDataMonitor を置く場所を作る
2. RawDataMonitor コンポーネントの作成
3. コンフィギュレーションファイルの作成
4. システム起動、ラン
5. trigger.py でボードにトリガーを送りグラフが更新されていることを確認する
6. trigger 停止
7. モニターの改造など

となる。

14.1 1. RawDataMonitor コンポーネントの場所の確保

通常、一から RawDataMonitor コンポーネントを書くのではなくてすでにある SampleMonitor をコピーして、データフォーマットを変更、自分の用途にあうように改良していく。

SampleMonitor コンポーネントは DAQ-Middleware をセットすると/usr/share/daqmw/examples/SampleMon に入っているのでまずこれをコピーする:

```
% cd ~/RawData
% cp -r /usr/share/daqmw/examples/SampleMonitor .
% cd SampleMonitor
% make（正常にコピーされたかどうか確認する）
```

最後の make コマンドは正常に終了し SampleReaderComp ができているはずである。できていたら OK なので、生成されたオブジェクト、実行ファイルを消して前回同様コンポーネント名を RawDataMonitor に変更する作業を行う。

```
% make clean
% cp ~/daqmw-tc/daqmw/utils/change-SampleMonitor-name.sh .
% sh change-SampleMonitor-name.sh
```

```
% cd ..
% mv SampleMonitor RawDataMonitor
% cd RawDataMonitor
% make（名前を変えただけなので正常にコンパイルできるはず）
```

最後の make でエラーが出ないことを確認する。

このままではロジックは SampleMonitor のままなので RawData フォーマットにあわせる、描画するものを変更するなどの作業が必要になる。

デコードルーチンは ex06 で書いたものを使うので RawDataPacket.h および RawDataPacket.cpp をコピーし、さらに Makefile で

```
SRCS += RawDataPacket.cpp
```

の行を追加する。これでコードを書く準備ができた。あとはロジックを書き直せばよい。

14.2 2. RawDataMonitor コンポーネントの作成

まずチャンネル 0 のデータのグラフを書くのを目標にし、正常動作したあと残りのチャンネルも表示するという順でやるのがよいかと思う。

以下変更を要する点のポイントを書いておく。

14.2.1 RawDataMonitor.h での変更点

- SampleMonitor ではヒストグラムを書いていたので TH1.h をインクルードし、TH1F *m_hist としてヒストグラムへのポインタを宣言していたが、RawDataMonitor では書くものはグラフなので TGraph.h をインクルードし、変数名、型もそれにあわせて変更する必要がある。
- SampleMonitor では上流からくるデータをいれるバッファとして

  ```
  unsigned char m_recv_data[4096];
  ```

  を使っている。これは SampleMonitor の上流コンポーネントから送られてくるデータ長が 4096 バイトの固定長だからである。RawDataMonitor ではフォーマット上はデータ長は固定長ではないので、大きめにバッファを確保する必要がある。たとえば

  ```
  const static unsigned int DATA_BUF_SIZE = 1024*1024; // 1MB
  unsigned char m_recv_data[DATA_BUF_SIZE];
  ```

  とする。

14.2.2 RawDataMonitor.cpp での変更点

状態遷移関数 (daq_configure(), daq_start() など) および特定の状態にある関数 (daq_run()) について状態遷移の順番にコードを修正していくのがよいかと思う。

■daq_configure() 特に変更はない。パラメータを増やしたときには daq_configure() から呼ばれている parse_params() を変更する。

■daq_start() SampleMonitor ではヒストグラムデータ (m_hist) を使っているが、今回はグラフで変数名も変えているはずなのでそれにあわせて変更する必要がある。

ここで行う処理はグラフデータがあれば、それを delete し、あらたに TGraph() オブジェクトを作成することである。

■daq_run()

1. 上流コンポーネントから送られてきたデータを読み
2. デコードしてグラフデータをセットする (m_graph->SetPoint() などを使う)
3. ときどきグラフを画面に書く

という処理をおこなう。

上流コンポーネントから送られてくるデータを読むところは変更する必要はない。

デコードしてグラフデータをセットする部分は daq_run() から呼ばれている fill_data() 関数を使う。SampleMonitor では fill_data() 関数内で SampleMonitor::decode_data() 関数を使ってデコードを行っている。今回実習で作成する RawDataMonitor ではすでに実装ずみのデコードルーチンを使うので decode_data() 関数は必要ではない (消すときは RawDataMonitor.cpp での実装部だけではなく RawDataMonitor.h の decode_data() 関数を宣言しているところも消す必要がある)。

ときどきグラフを書くというのは以下の部分である:

```
if (m_monitor_update_rate == 0) {
    m_monitor_update_rate = 1000;
} // m_monitor_update_rate が指定されていなかった場合に備えている

// daq_run() 内前半で上流からデータを読んでいるが
// 読めたらシーケンス番号を 1 づつ増やしているので
// その数字を使って m_monitor_update_rate 回に一回画面上の図を更新
// している
unsigned long sequence_num = get_sequence_num();
if ((sequence_num % m_monitor_update_rate) == 0) {
    m_hist->Draw();
```

```
    m_canvas->Update();
}
```

■daq_stop()　daq_stop() に以降した直前に読んだデータを使って画面に図を出す処理を書く。

■daq_unconfigure()　SampleReader ではヒストグラムデータ、RawDataMonitor ではグラフデータを delete する (daq_start() で new していたので delete しないと多数回 start/stop を繰り返したときにメモリーを浪費する)。

14.3　3. コンフィギュレーションファイルの作成

コンフィギュレーションファイルを作る。ほぼ完成された雛型を~/daqmw-tc/daqmw/reader-monitor.xml に用意してあるのでそれを編集して使う。

```
% cd ~/RawData
% cp ~/daqmw-tc/daqmw/reader-monitor.xml .
% reader-logger.xml の execPath を自分の環境にあわせて編集
```

14.4　4. システム起動、ラン

```
% run.py -cl reader-monitor.xml
```

でシステムを起動し、0, 1 とキーを押してシステムを動かす。

14.5　5. trigger.py でボードにトリガーを送る

```
% ~/daqmw-tc/trigger/trigger.py
```

として start ボタンを押しトリガーを送る。読みだしシステムのイベントバイト数が増えるのを確認する。

14.6　6. trigger 停止

trigger GUI の stop ボタンを押しトリガーを送るのを停止する。2 を押してシステムを止める。

14.7　7. モニターの改造

できあがったら全てのチャンネルを表示するようにする。

モニターしながらデータも保存するというシステムにするには既に試した RawDataLogger の他に Dispatcher コンポーネントが必要になる。Dispatcher のソースコード

は/usr/share/daqmw/examples/Dispatcher にあるのでコピーしてコンパイルする (Dispatcher を改造することはあまりないのでここでは名前の変更は行わない)

```
% cd ~/RawData
% cp -r /usr/share/daqmw/examples/Dispatcher .
% cd Dispatcher
% make
```

4 個の DAQ コンポーネントを使う場合のコンフィギュレーションファイルは~/daqmw-tc/daqmw/4comps.xml にあるのでコピーして変更して使う。