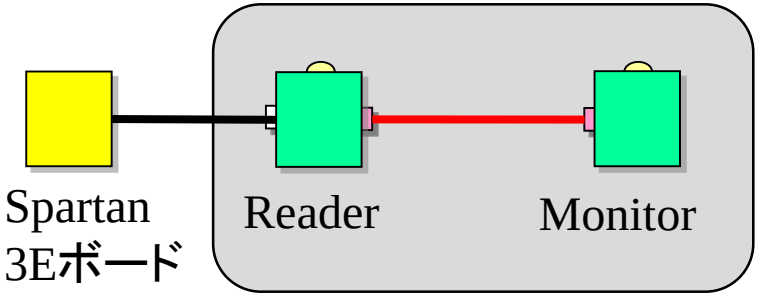
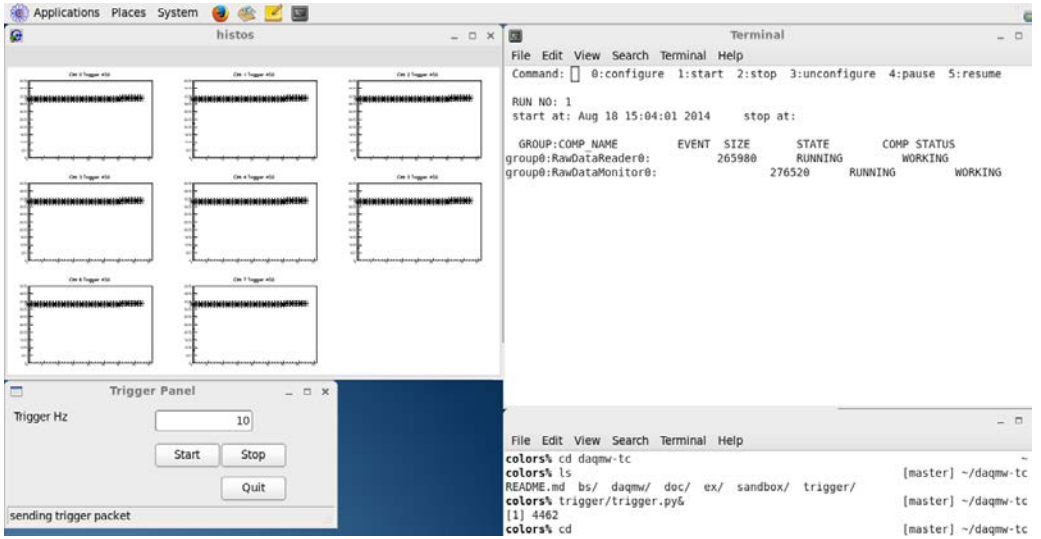
DAQ-Middleware トレーニングコース実習

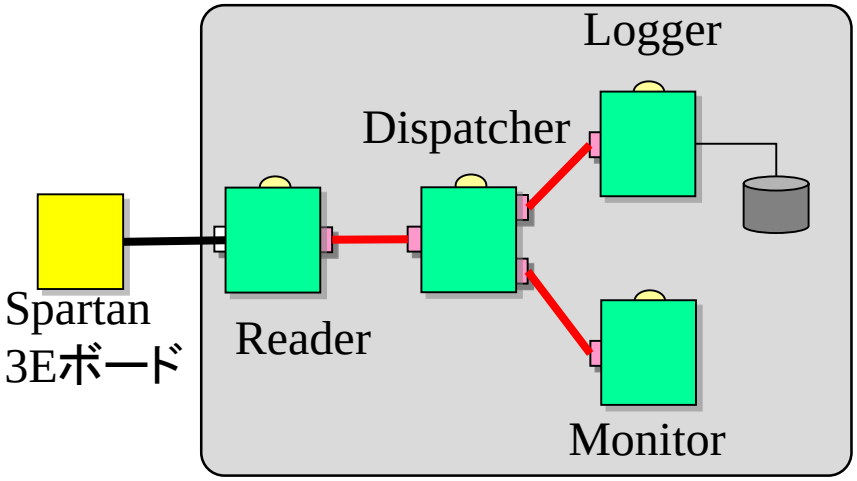
濱田英太郎
高エネルギー加速器研究機構
素粒子原子核研究所

実習最終目標

Spartan 3Eボードからデータを読んでグラフを画面に表示するシステムをDAQ-MWで作る







実習で行う事項

- ex01 DAQ-Middleware付属サンプルコンポーネントを動かしてみる
- ex02 Webモードでシステムを動かす
- ex03 ログの確認（状態遷移の確認）
- ex04 コンフィグレーションファイルの編集（コンポーネント構成）
- ex05パラメータ取得
- ex06 コンポーネント間のデータについて
- ex07ボードを読むシステムを動かしてみる(Reader - Logger)
- ex08 DAQ-Middlewareでモニターコンポーネントを開発する
- ex09 Mergerを利用して複数台のネットワークノードからデータを収集する

実習環境確認

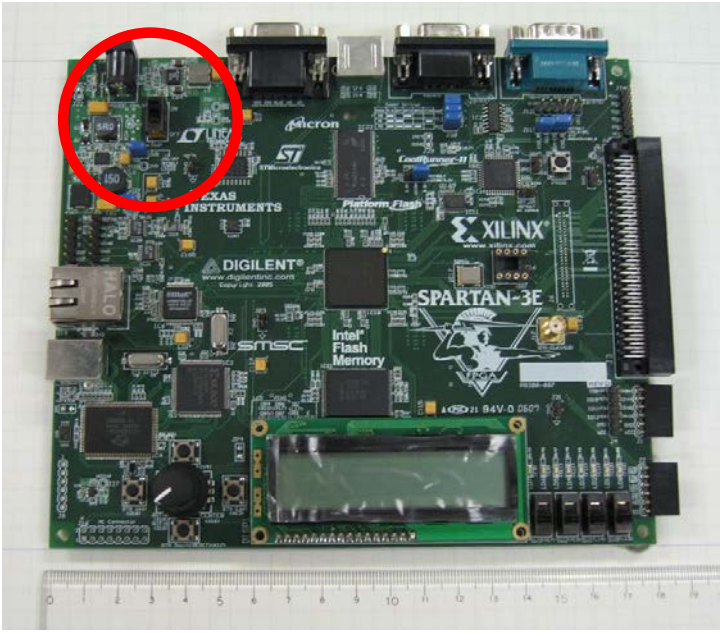
- VirtualBoxのセットアップ

以下のコマンドを実行して、インターネットに接続できることを確認してください。

```
% ping www.yahoo.co.jp
```

- Spartan 3Eの配布

ACアダプタ、LANケーブルをさすだけ。
電源スイッチはACアダプタコネクタそば



以下のコマンドを実行して、ボードに接続できることを確認してください。

```
% ping 192.168.10.16
```

実習ファイルダウンロード

- 実習ファイルダウンロード

（下記はwebページに記載されています。）

```
%  cd
%  git clone https://github.com/e-hamada/daqmw-tc2.git
```

ホームディレクトリに「daqmw-tc2」というディレクトリが追加されます。

実習ファイル　中身の説明

- ex

  実習で行う項目の解説、一部のコード

- sandbox

  特に使わない

- doc

  Spartan 3Eが送ってくるデータのデータフォーマットを説明する資料がある

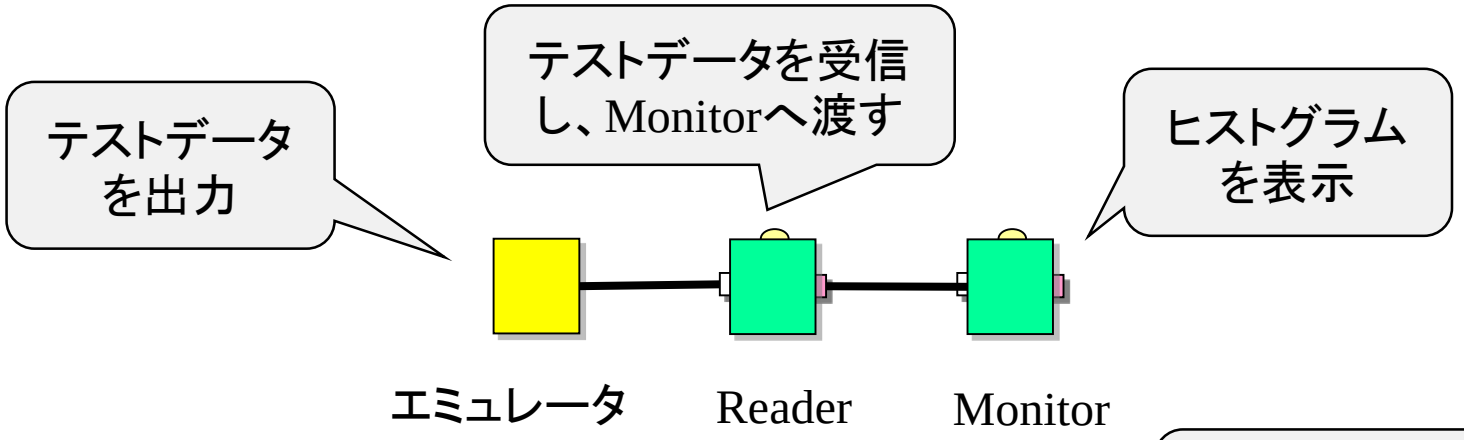
- trigger

  Spartan 3Eにトリガー信号を送るプログラム

- daqmw

  DAQコンポーネントの答え（できるだけ見ないでください）

ex01 DAQ-Middleware付属 サンプルコンポーネントを動かしてみる



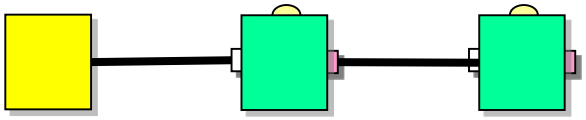
手順

1. ソースコードのコピーとコンパイル
2. コンフィギュレーションファイルの作成
3. コンポーネントの起動
4. システム起動
5. エミュレータの起動
6. データ収集再開
7. システム終了

各コンポーネントごとにプログラムを用意

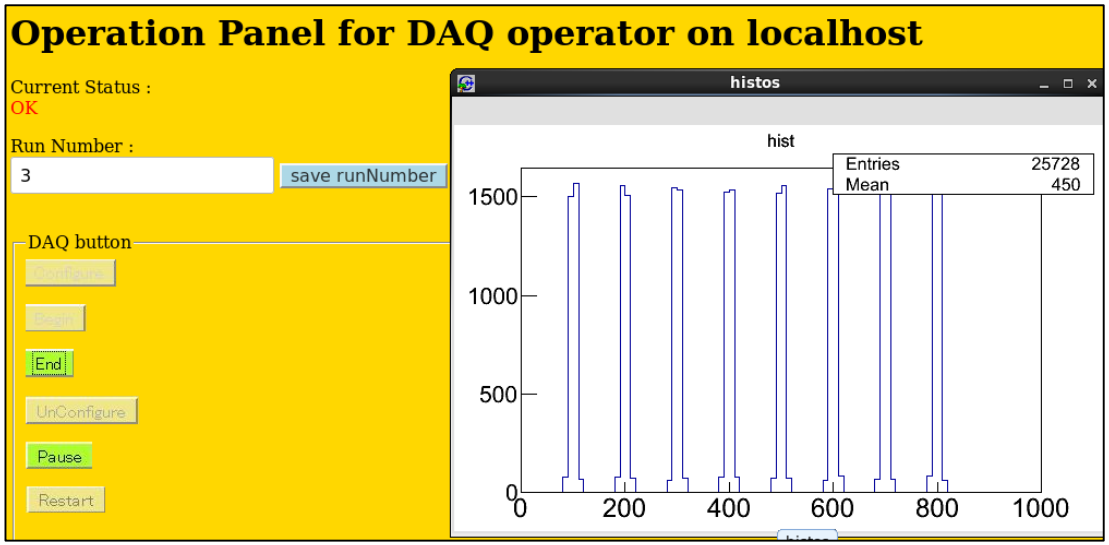
設定ファイル（コンポーネントとその接続情報等）

ex02 Webモードでシステムを動かす



エミュレータ　Reader　Monitor

ex01と同様、ReaderとMonitorから構成されるシステムを起動



webブラウザに表示

手順

1. apache （webサーバ）を起動
2. DAQ-Middlwareをコンソールモードを抜かして起動
3. webブラウザで確認

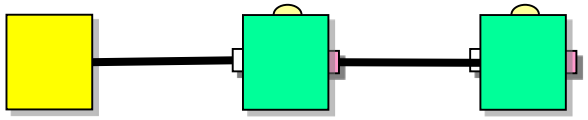
ex03 ログの確認（状態遷移の確認）

ログの出力方法
　　プログラム内にて

```
std::cerr << "◯◯◯" << std::endl;
```

ログはコンポーネントごとに作成され、
/tmp/daqmw/log.(コンポーネント名)Compに置かれる

目的
　プログラム内の各関数にログを記載するよう編集し、状態遷移の確認を行う。



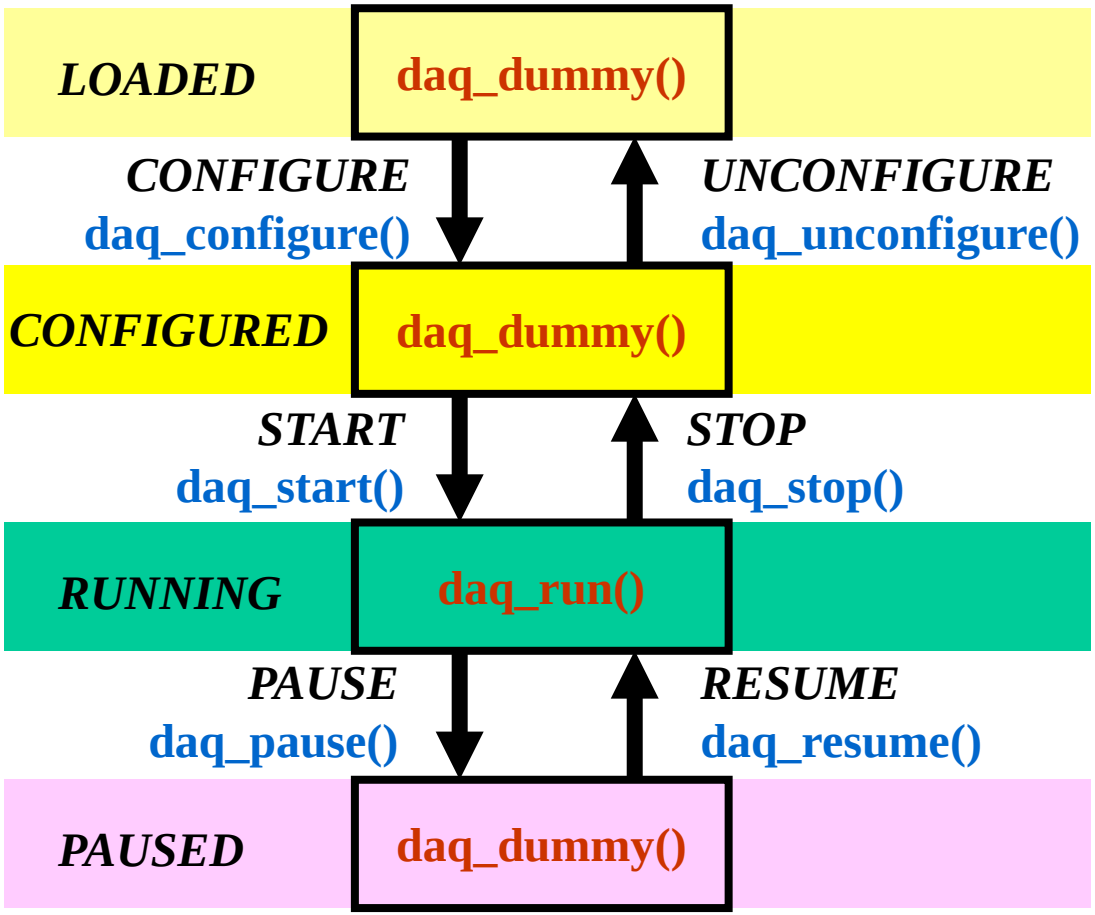
エミュレータ　Reader　Monitor

手順
1. SampleReaderのconfigureのときに、出力されるログを確認
2. SampleReaderのrun状態のときにログが出力されるよう、プログラムを編集
3. SampleReaderのrun状態のときに、出力されるログを確認

ex01と同様、ReaderとMonitorから構成されるシステムを起動

コンポーネント状態遷移

技術解説書15-17ページ



各状態(LOADED, CONFIGURED, RUNNING, PAUSED)にある間、対応する関数が繰り返し呼ばれる。

状態遷移するときは状態遷移関数が呼ばれる。

状態遷移できるようにするためには、daq_run()等は永遠にそのなかでブロックしてはだめ。
（例：Gathererのソケットプログラムでtimeoutつきにする必要がある）

各関数を実装することでDAQコンポーネントを完成させる。

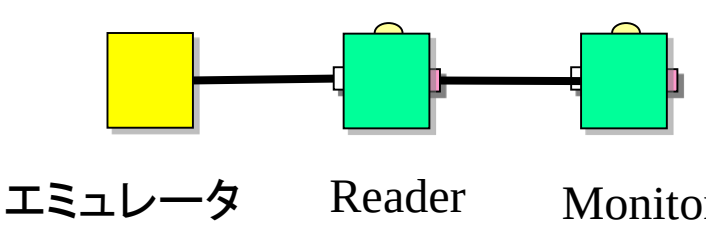
ex04　コンフィグレーションファイルの編集（コンポーネント構成）

目的

コンフィグレーションファイルの編集方法を学ぶ。

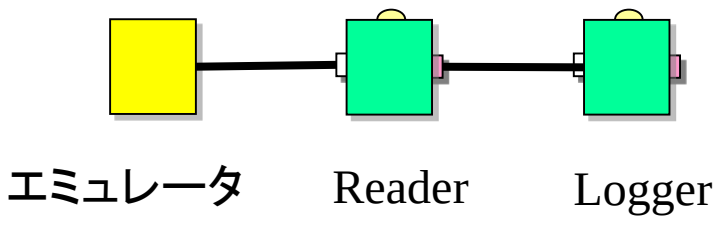
行うこと

コンフィグレーションファイルを編集し、使用するコンポーネントを変更する。



ex01～ex03
コンフィグレーションファイルは
sample.xml

Monitorから
Loggerに変更



ex04
コンフィグレーションファイルは
sample.xmlを編集した
reader-logger.xml

コンフィグレーションファイル

sample.xml

```
<configInfo>
  <daqOperator>
    <hostAddr>127.0.0.1</hostAddr>
  </daqOperator>
  <daqGroups>
    <daqGroup gid="group0">
      <components>
        <component cid="SampleReader0">
          SampleReaderの情報（詳細は次ページ)
        </component>
        <component cid="SampleMonitor0">
          SampleMonitorの情報（詳細は2ページ後）
        </component>
      </components>
    </daqGroup>
  </daqGroups>
</configInfo>
```

DAQ-Operatorのアドレス
= ローカルホスト

コンポーネントID

コンポーネントの情報

コンフィグレーションファイル(Reader)

```
<hostAddr>127.0.0.1</hostAddr>
<hostPort>50000</hostPort>
<instName>SampleReader0.rtc</instName>
<execPath>(省略)</execPath>
<confFile>/tmp/daqmw/rtc.conf</confFile>
    <startOrd>2</startOrd>
    <inPorts>
    </inPorts>
    <outPorts>
        <outPort>samplereader_out</outPort>
    </outPorts>
    <params>
        <param pid="srcAddr">127.0.0.1</param>
        <param pid="srcPort">2222</param>
    </params>
```

コンポーネントがおいてあるPCのIPとポート番号

コンポーネントのインスタンス名

コンポーネントの実行形式ファイルのパス

コンポーネントのスタートコマンド投入の際の順番

inportの設定(readerの場合はない)

outpoerの設定

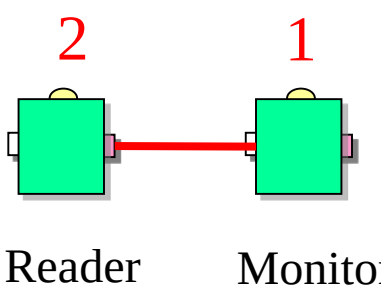
初期パラメータ

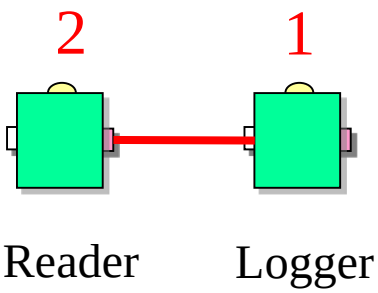
Reader Monitor

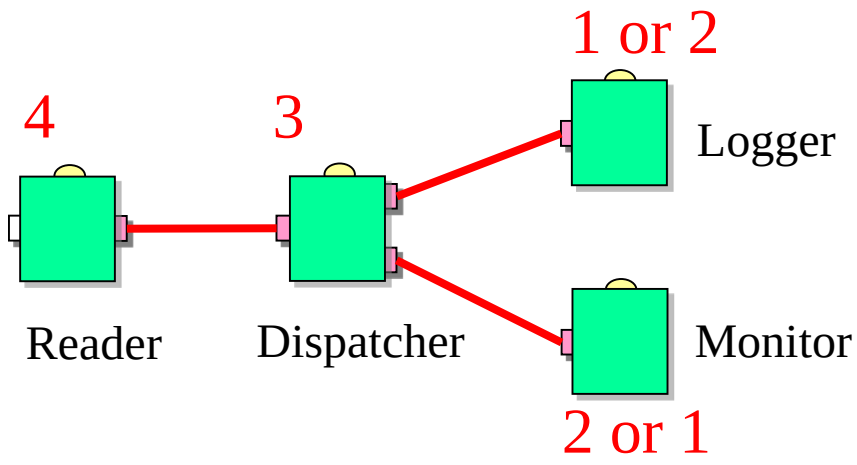
コンフィグレーションファイル（スタートコマンド投入の際の順番）

DAQコンポーネント起動の順序は下流から起動を開始させる。

例







LoggerとMonitorは
どちらが先でもよい

コンフィグレーションファイル(Monitor)

```
<hostAddr>127.0.0.1</hostAddr>
<hostPort>50000</hostPort>
<instName>SampleMonitor0.rtc</instName>
<execPath>(省略)</execPath>
<confFile>/tmp/daqmw/rtc.conf</confFile>
<startOrd>1</startOrd>
<inPorts>
  <inPort from="SampleReader0:samplereader_out">samplemonitor_in</inPort>
</inPorts>
<outPorts>
</outPorts>
<params>
  <param pid="monitorUpdateRate">20</param>
</params>
```

コンポーネントがおいてあるPCのIPとポート番号

コンポーネントのインスタンス名

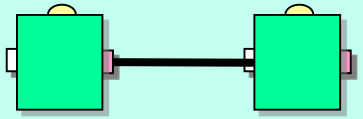
コンポーネントの実行形式ファイルのパス

コンポーネントのスタートコマンド投入の際の順番

inportの設定

outpoerの設定(Monitorの場合はない)

初期パラメータ



Emulatorの仕様

- daqmw-emulator [-t tx_bytes/s] [-b buf_bytes] [-p port num]
- デフォルトは –t 8k –b 1k (8kB/sec, 1 回1kB) –p 2222
- 数値はm, kのサフィックスが使える
- 指定された転送レートをできるだけ守るようにデータを送る
- 送ってくるデータフォーマット:

Magic	Format Version	Module Number	Reserved	Event Data	Event Data	Event Data	Event Data

Magic: 0x5a
Format Version: 0x01
Module Number: 0x00 – 0x07
Event Data: 適当にガウシャン風。100, 200, 300, ... 800にピークがある。
1000倍した整数値で送ってくる。ネットワークバイトオーダー。

このようなデータフォーマットになっているか、hexdumpコマンドで確認してください。

ex05 課題　ヒストグラムの大きさを変える

変更点

SampleMonitor　ヘッダファイル

- Tcanvasの縦のサイズをあらわすグローバル変数を追加
- Tcanvasの横のサイズをあらわすグローバル変数を追加

SampleMonitor　cppファイル

- parse_params関数で、「縦のサイズ」と「横のサイズ」を取得し、グローバル変数に格納

int型にする必要があります。SampleReaderのポート番号を取得する処理を参考にしてください

- daq_start関数で

(修正前）m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400);

（修正後） m_canvas = new TCanvas(“c1”, “histos”, 0, 0,「横のサイズ」,「縦のサイズ」);

コンフィグレーションファイル

- 「縦のサイズ」と「横のサイズ」のパラメータを追加

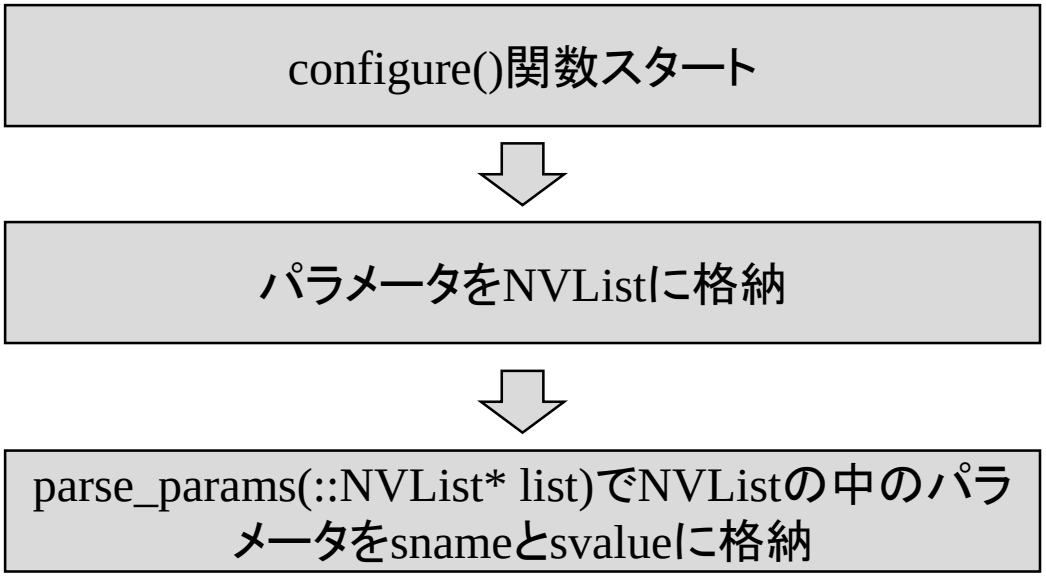
ex05 パラメータ取得

目的

パラメータの設定方法について学習する

コンポーネントプログラムにおけるパラメータ取得処理

→configure関数で読み込みを行っている。



SampleReader (SampleReader.cpp)
daq_configure() パラメータの取得

```
int SampleReader::daq_configure()
{
    std::cerr << "*** SampleReader::configure" << std::endl;

    ::NVList* paramList;
    paramList = m_daq_service0.getCompParams();
    parse_params(paramList);

    return 0;
}
```

SampleReader - daq_configure()

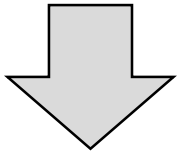
```
<!-- config.xml -->
<params>
    <param pid="srcAddr">127.0.0.1</param>
    <param pid="srcPort">2222</param>
</params>
```

```
int SampleReader::parse_params(::NVList* list)
{
    int len = (*list).length();
    for (int i = 0; i < len; i+=2) {
        std::string sname  = (std::string)(*list)[i].value;
        std::string svalue = (std::string)(*list)[i+1].value;
        if ( sname == "srcAddr" ) {
            m_srcAddr = svalue;
        }
        if ( sname == "srcPort" ) {
            char* offset;
            m_srcPort = (int)strtol(svalue.c_str(), &offset, 10);
        }
    }
```

ex06 コンポーネント間のデータについて

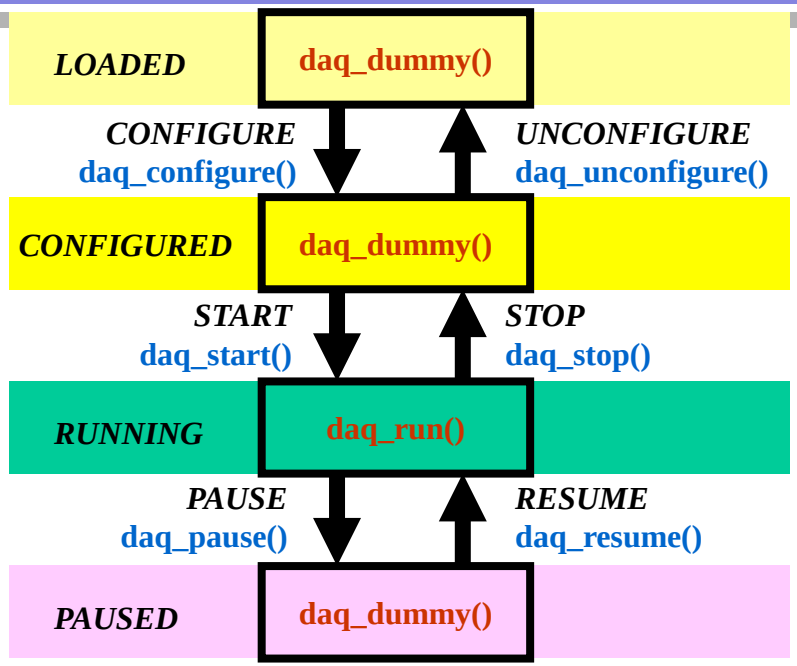
- SampleReaderおよびSampleMonitorの処理について説明

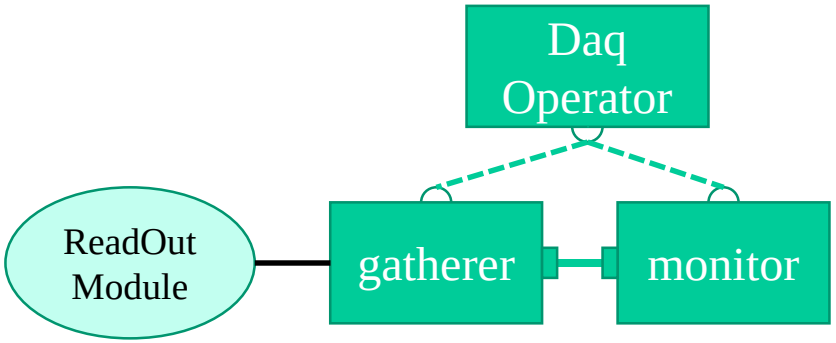
- コンポーネント間のデータについて説明
  （DAQ-Middlwareオリジナルのデータ構造があることに注意）



- SampleReader-SampleMonitor間のデータを確認できる 処理を追加する。

SampleReader, Monitorの仕様





Gatherer (SampleReader)

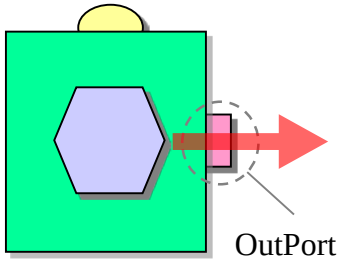
daq_configure(): リードアウトモジュールのIPアドレス、ポートを取得（DAQ-Operatorからふってくる）
daq_start(): リードアウトモジュールに接続
daq_run(): リードアウトモジュールからデータを読んで後段コンポーネントにデータを送る
daq_stop(): リードアウトモジュールから切断。

Monitor (SampleMonitor)

daq_start(): ヒストグラムデータの作成
daq_run(): 上流コンポーネントからデータをうけとり、デコードしてヒストグラムデータをアップデートする。定期的にヒストグラム図を書く
daq_stop(): 最終データを使ってヒストグラム図を書く

SampleReader （SampleReader.h、cpp)

```
// SampleReader.h
class SampleReader
    : public DAQMW::DaqComponentBase
{
private:
    TimedOctetSeq          m_out_data;
    OutPort<TimedOctetSeq> m_OutPort;
```




```
// SampleReader.cpp
SampleReader::SampleReader(RTC::Manager* manager)
    : DAQMW::DaqComponentBase(manager),
      m_OutPort("samplereader_out", m_out_data),
      m_sock(0),
      m_recv_byte_size(0),
      m_out_status(BUF_SUCCESS),
```

SampleReader - daq_start()

```
int SampleReader::daq_start()
{
    m_out_status = BUF_SUCCESS;

    // リードアウトモジュールに接続
    try {
        // Create socket and connect to data server.
        m_sock = new DAQMW::Sock();
        m_sock->connect(m_srcAddr, m_srcPort);
    } catch (DAQMW::SockException& e) {
        std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    } catch (...) {
        std::cerr << "Sock Fatal Error : Unknown" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
```

SampleReader - daq_run()

```
int SampleReader::daq_run()
{
    if (check_trans_lock()) {  // check if stop command has come
        set_trans_unlock();    // transit to CONFIGURED state
        return 0;
    }
    if (m_out_status == BUF_SUCCESS) {
        int ret = read_data_from_detectors();
        if (ret > 0) {
            m_recv_byte_size = ret;
            set_data(m_recv_byte_size); // set data to OutPort Buffer
        }
    }
    if (write_OutPort() < 0) {
        ;      // Timeout. do nothing.
    }
    else {     // OutPort write successfully done
        inc_sequence_num();                          // increase sequence num.
        inc_total_data_size(m_recv_byte_size);  // increase total data byte size
    }
```

readを行う関数
readしたデータはm_dataに格納される
戻り値はreadしたデータのBYTE数

後段のコンポーネントに送るデータの作成
(m_out_dataの作成)

後段のコンポーネントにデータを送信

SampleReader – set_data()

```
int SampleReader::set_data(unsigned int data_byte_size)
{
    unsigned char header[8];
    unsigned char footer[8];

    set_header(&header[0], data_byte_size);
    set_footer(&footer[0]);

    ///set OutPort buffer length
    m_out_data.data.length(data_byte_size + HEADER_BYTE_SIZE + FOOTER_BYTE_SIZE);
    memcpy(&(m_out_data.data[0]), &header[0], HEADER_BYTE_SIZE);
    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE]), &m_data[0], data_byte_size);
    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE + data_byte_size]), &footer[0],
           FOOTER_BYTE_SIZE);

    return 0;
}
```

headerとfotterを作成

m_out_dataを作成
(header + Data + footer)

SampleReader - daq_run()

```
int SampleReader::read_data_from_detectors()
{
    int received_data_size = 0;

    /// read 1024 byte data from data server
    int status = m_sock->readAll(m_data, SEND_BUFFER_SIZE);
    // 書き方はいろいろあるがここでは先にエラーチェックを書いた
    if (status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "### ERROR: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
    // ここではデータがタイムアウトで読めなかったらエラーとなるように決めた
    else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "### Timeout: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR2, "SOCKET TIMEOUT");
    }
    else {
        received_data_size = SEND_BUFFER_SIZE;
    }

    return received_data_size;
}
```

SEND_BUFFER_SIZE(=1024BYTE)だけデータをreadする
readしたデータはm_dataに格納

エラー処理

通常の処理

SampleMonitor - SampleData.h

```
#ifndef SAMPLEDATA_H
#define SAMPLEDATA_H

const int ONE_EVENT_SIZE = 8;

struct sampleData {
  unsigned char magic;
  unsigned char format_ver;
  unsigned char module_num;
  unsigned char reserved;
  unsigned int  data;
};

#endif
```

データフォーマット構造体を定義。
デコードしたらすぐにこの構造体に代入して、変数名で処理できるようにする。

Magic	Format Version	Module Number	Reserved	Event Data	Event Data	Event Data	Event Data

SampleMonitor.h

```
////////// ROOT Histogram //////////
    TCanvas *m_canvas;
    TH1F    *m_hist;
    int      m_bin;
    double   m_min;
    double   m_max;
    int      m_monitor_update_rate;
    unsigned char m_recv_data[4096];
    unsigned int  m_event_byte_size;
    struct sampleData m_sampleData;

    bool m_debug;
};
```

SampleMonitor.cpp - daq_dummy()

```
int SampleMonitor::daq_dummy()
{
    if (m_canvas) {
        m_canvas->Update();
        // daq_dummy() will be invoked again after 10 msec.
        // This sleep reduces X servers' load.
        sleep(1);
    }

    return 0;
}
```

SampleMonitor - daq_configure()

```
int SampleMonitor::daq_configure()
{
    ::NVList* paramList;
    paramList = m_daq_service0.getCompParams();
    parse_params(paramList);

    return 0;
}
int SampleMonitor::parse_params(::NVList* list)
{
    int len = (*list).length();
    for (int i = 0; i < len; i+=2) {
        std::string sname  = (std::string)(*list)[i].value;
        std::string svalue = (std::string)(*list)[i+1].value;

        if (sname == "monitorUpdateRate") {
            if (m_debug) {
                std::cerr << "monitor update rate: " << svalue << std::endl;
            }
            char *offset;
            m_monitor_update_rate = (int)strtol(svalue.c_str(), &offset, 10);
        }
```

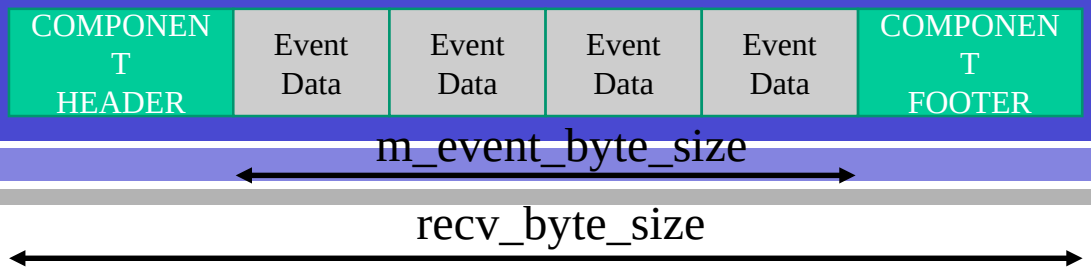
ヒストグラムの更新レートをコンフィグレーションファイルから取得

SampleMonitor - daq_start()

```
int SampleMonitor::daq_start()
{
    m_in_status  = BUF_SUCCESS;
    ////////////////// CANVAS FOR HISTOS /////
    if (m_canvas) {
        delete m_canvas;
        m_canvas = 0;
    }
    m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400);

    ////////////////       HISTOS      ///////////////////
    if (m_hist) {
        delete m_hist;
        m_hist = 0;
    }

    int m_hist_bin = 100;
    double m_hist_min = 0.0;
    double m_hist_max = 1000.0;

    m_hist = new TH1F("hist", "hist", m_hist_bin, m_hist_min, m_hist_max);
```

ヒストグラム、キャンバス
を定義

SampleReader - daq_run()



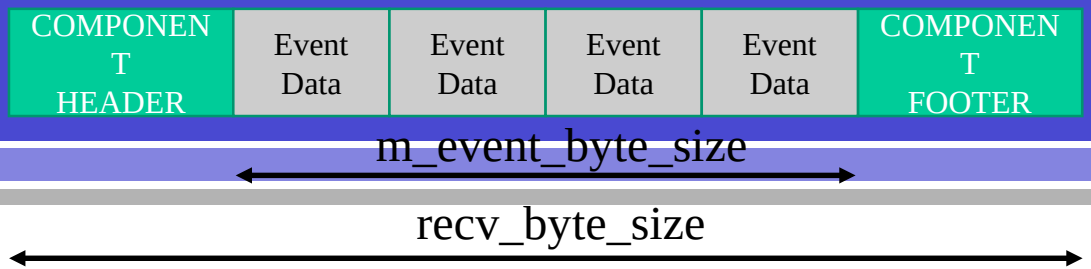

```
int SampleMonitor::daq_run()
{
    unsigned int recv_byte_size = read_InPort();
    if (recv_byte_size == 0) {
        return 0;
    }


    check_header_footer(m_in_data, recv_byte_size); // check header and footer
    m_event_byte_size = get_event_size(recv_byte_size);


    //////////////  Write component main logic here. //////////////
    memcpy(&m_recv_data[0], &m_in_data.data[HEADER_BYTE_SIZE], m_event_byte_size);
```

データを前段のコンポーネントから受信
データはm_in_dataに格納

ヘッダフッタの確認
イベントデータのデータ長を取得
（DAQ-MWオリジナルの関数を使用している。詳細は技術解説書を読むこと）

イベントデータをm_recv_dataに格納

SampleReader - daq_run() 続き




```
fill_data(&m_recv_data[0], m_event_byte_size);
if (m_monitor_update_rate == 0) {
    m_monitor_update_rate = 1000;
}
unsigned long sequence_num = get_sequence_num();
if ((sequence_num % m_monitor_update_rate) == 0) {
    m_hist->Draw();
    m_canvas->Update();
}
```

ヒストグラムにデータをfill処理する関数

シーケンス番号がm_monitor_update_rateで割り切れるときに、ヒストグラムを更新

SampleMonitor - fill_data()

```
int SampleMonitor::fill_data(const unsigned char* mydata, const int size)
{
    for (int i = 0; i < size/(int)ONE_EVENT_SIZE; i++) {
        decode_data(mydata);
        float fdata = m_sampleData.data/1000.0; // 1000 times value is received
        m_hist->Fill(fdata);

        mydata+=ONE_EVENT_SIZE;
    }
    return 0;
}
```

1イベント分のデータを出コードし、m_sampleData.dataに格納

ONE_EVENT_SIZEは8BYTE

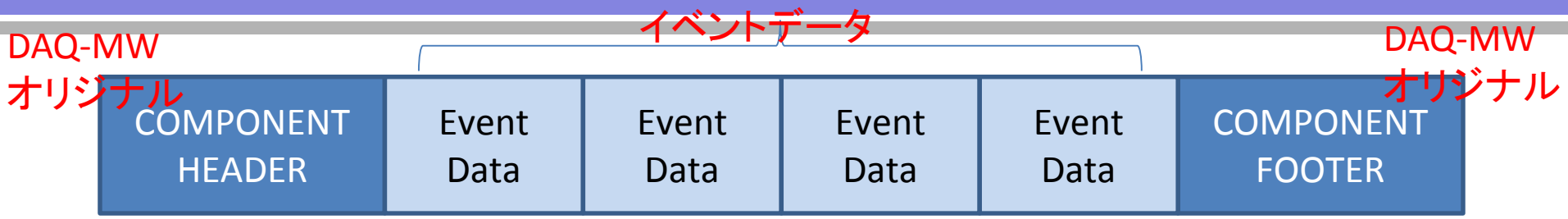
SampleMonitor - decode_data()

```
int SampleMonitor::decode_data(const unsigned char* mydata)
{
    m_sampleData.magic      = mydata[0];
    m_sampleData.format_ver = mydata[1];
    m_sampleData.module_num = mydata[2];
    m_sampleData.reserved   = mydata[3];
    unsigned int netdata    = *(unsigned int*)&mydata[4];
    m_sampleData.data       = ntohl(netdata);
```

Magic	Format Version	Module Number	Reserved	Event Data	Event Data	Event Data	Event Data

ntohl(): ネットワークバイトオーダーからホストバイトオーダーへ変換

コンポーネント間のデータフォーマット



Component Header

Header Magic (0xe7)	Header Magic (0xe7)	Reserved	Reserved	Data Byte Size	Data Byte Size	Data Byte Size	Data Byte Size

Data Byte Sizeには下流コンポーネントに何バイトのイベントデータを送ろうとしたかを入れる
下流側ではDataByteSizeを読んでデータが全部読めたかどうか判断する

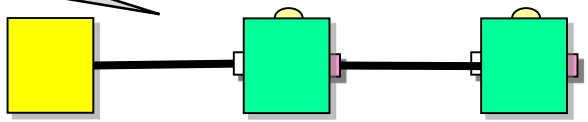
Component Footer

Footer Magic (0xcc)	Footer Magic (0xcc)	Reserved	Reserved	Seq. Num	Seq. Num	Seq. Num	Seq. Num

Sequence Numberにデータを送るのは何回目かを入れる
下流側では受け取った回数を自分で数えておいて、Sequence Numberとあうかどうか確認する

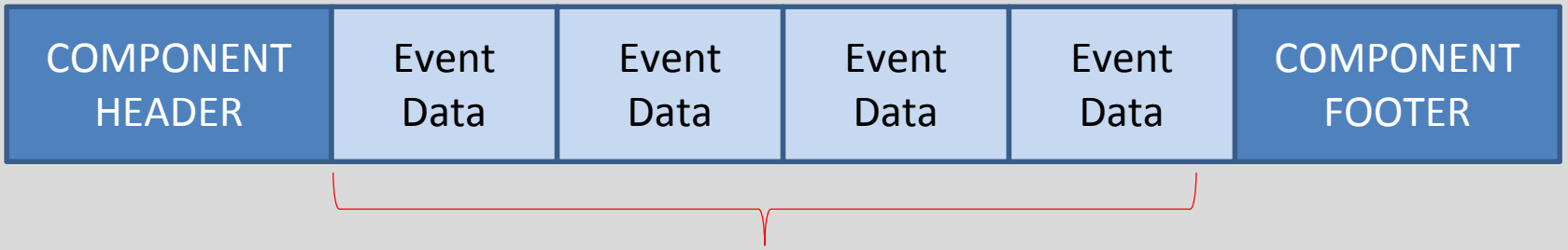
SampleReader – SampleMonitorの場合

1回のdaq_run関数で
1024BYTE
読み込む



エミュレータ　Reader　Monitor

ReaderからMonitorへのデータ



ex06 コンポーネント間のデータについて

実習内容

シーケンス番号が20で割り切れる時、SampleReaderが送るデータの初めの20Byteを確認する。

ログ出力の例

COMPONENT HEADER

```
sequence_num = 580
Data0 = 0xe7
Data1 = 0xe7
Data2 = 0x0
Data3 = 0x0
Data4 = 0x0
Data5 = 0x0
Data6 = 0x4
Data7 = 0x0
Data8 = 0x5a
Data9 = 0x1
Dataa = 0x0
Datab = 0x0
Datac = 0x0
Datad = 0x1
Datae = 0x79
Dataf = 0x80
Data10 = 0x5a
Data11 = 0x1
Data12 = 0x1
Data13 = 0x0
```

ex07 ボードを読むシステムを動かしてみる (Reader - Logger)

手順

1.RawDataLoggerコンポーネントの作成
2.RawDataReader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

名前を変えるだけ

プログラムの変更が必要

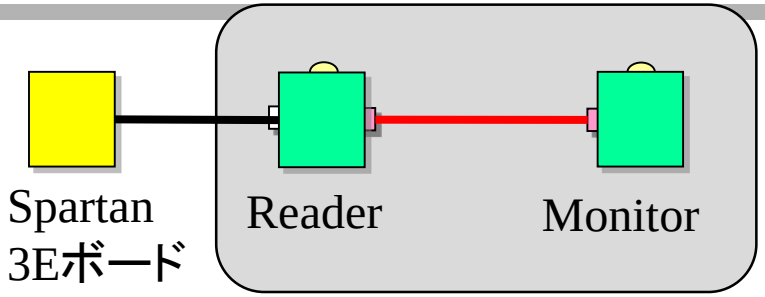
- SampleReader

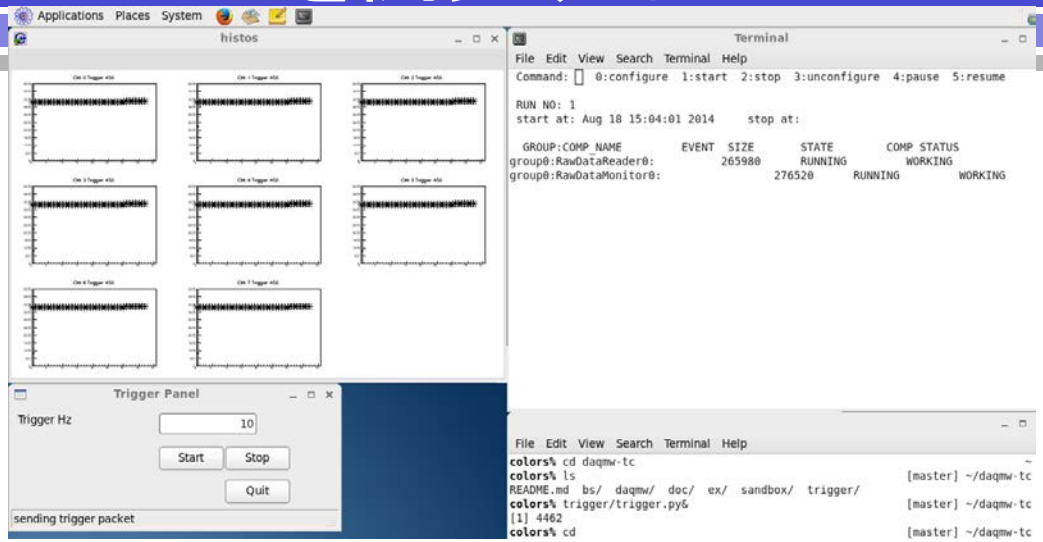
1024BYTEごとreadしていた

- RawDataReader

1. はじめにヘッダをread
2. ヘッダからデータ長を取得
3. 取得したデータ長の分だけread

ex08 DAQ-Middlewareで モニターコンポーネントを開発する

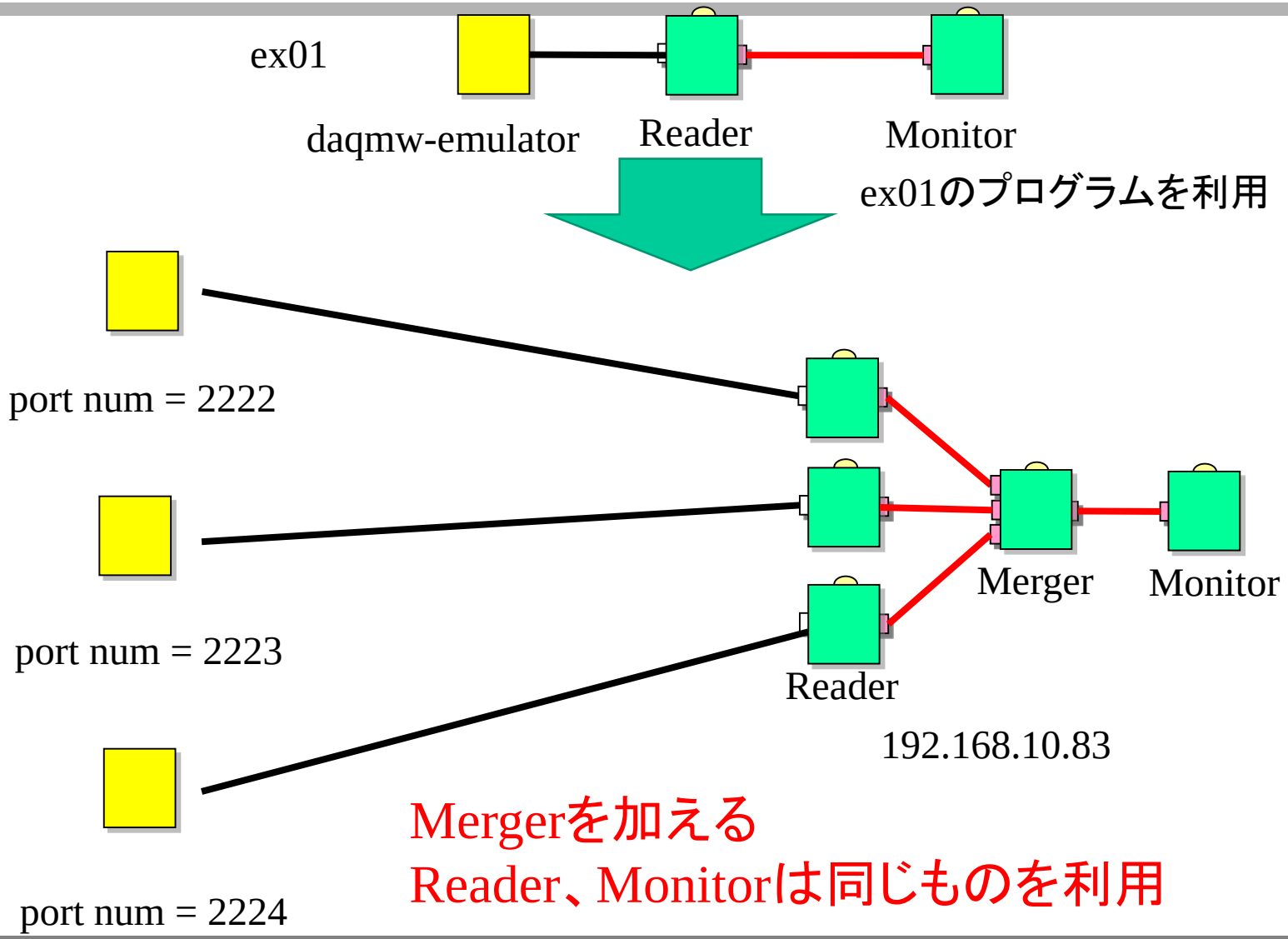




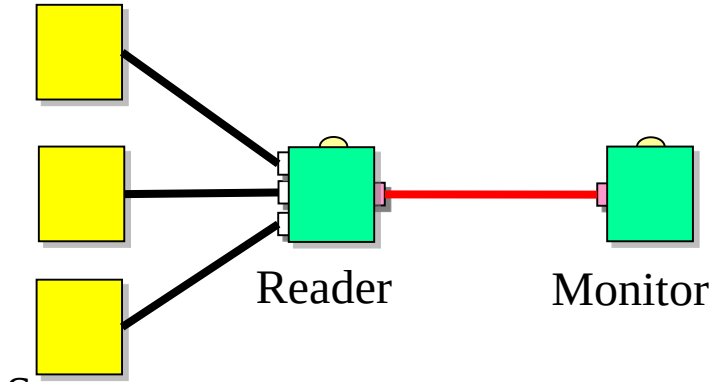
- ex07で使ったReaderを利用。
  Readerは1イベントごと、データをMonitorに送っている。
  →read_data_from_detectors関数に処理内容が書かれている。

- MonitorはSampleMonitorを利用して自分で作る。
  SampleMonitorからの変更点のヒントはweb上に記載してある。
  DAQ-Middleware特有の関数があるので、理解が難しい箇所があります。
  →濱田に質問していただくか、マニュアルを参照してください。

ex09 Mergerを利用して複数台のPCからデータを収集する



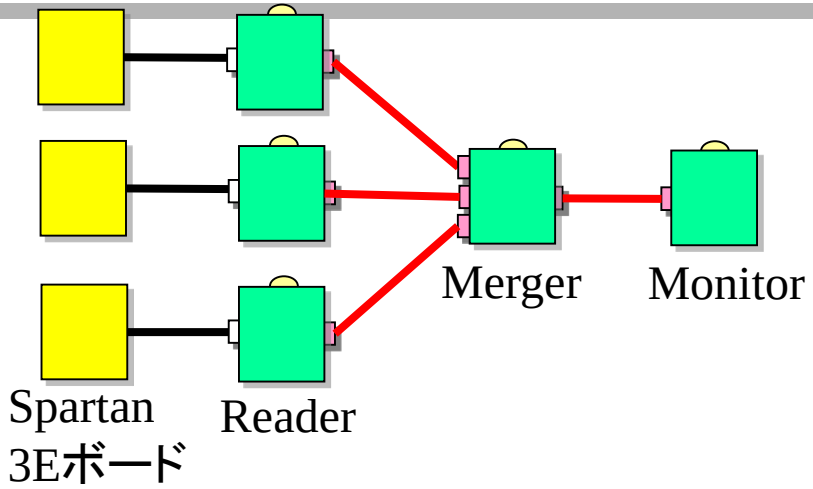
DAQ-Middleware　多重読みだしの例



例1　Readerでepoll等を利用して多重読み込みを行う

（メリット）
- コンポーネントが少ないので使用するリソースが少なくても済む

（デメリット）
- Readerの作成が難しい
- プロセスを分けないと、1CPUにReaderの分の負荷が大きくなってしまう



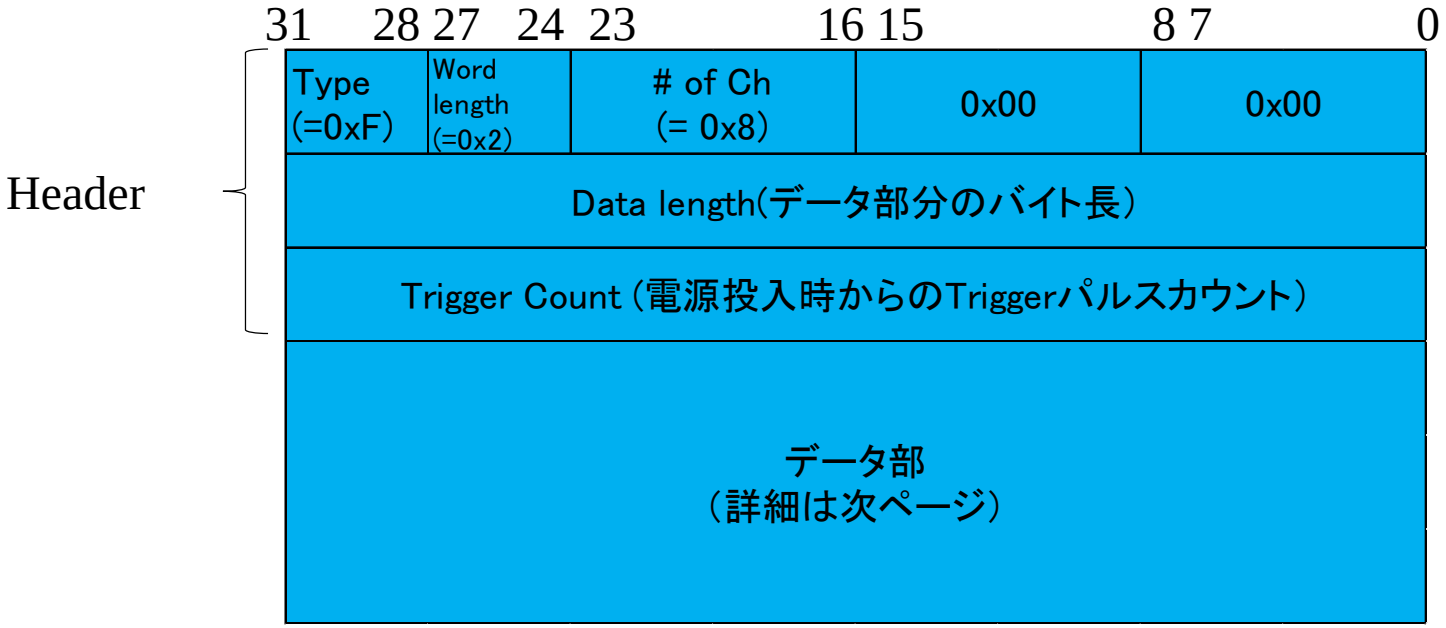
例2　複数のReaderとMergerを利用する

（メリット）
- Readerは全て1台の読み出しなので簡単に作れる。
- Readerの負荷を分散できる

（デメリット）
コンポーネントが多いので使用するリソースが多くなる

BACKUP

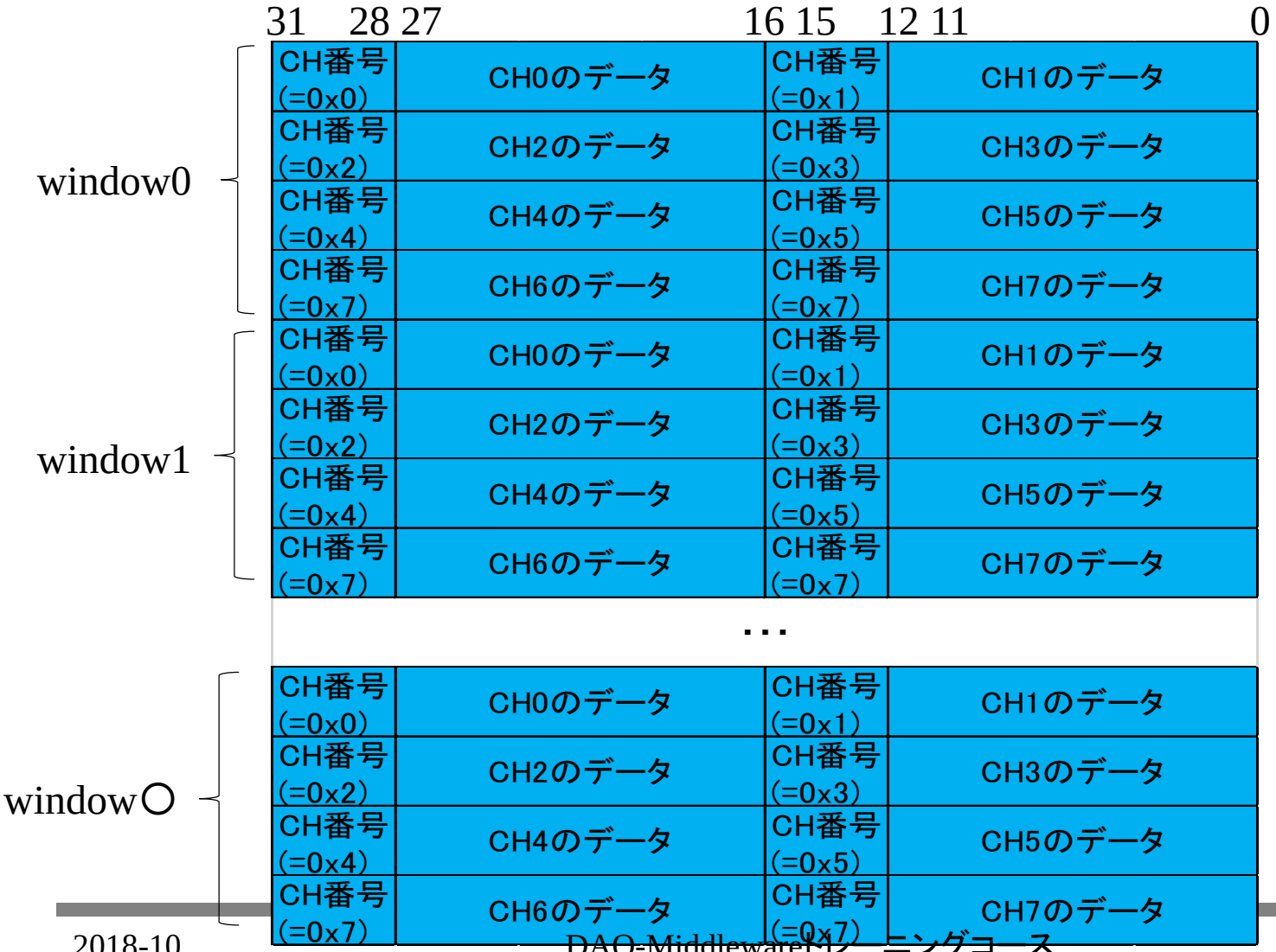
データ転送パケットフォーマット（全体）



※複数バイトの場合、ビックエンディアン

データ転送パケットフォーマット（データ部）

window数の分だけ、データが送られてくる

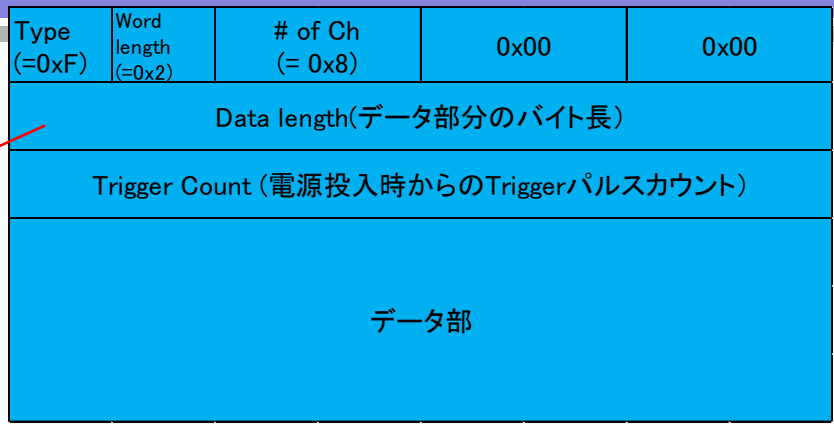


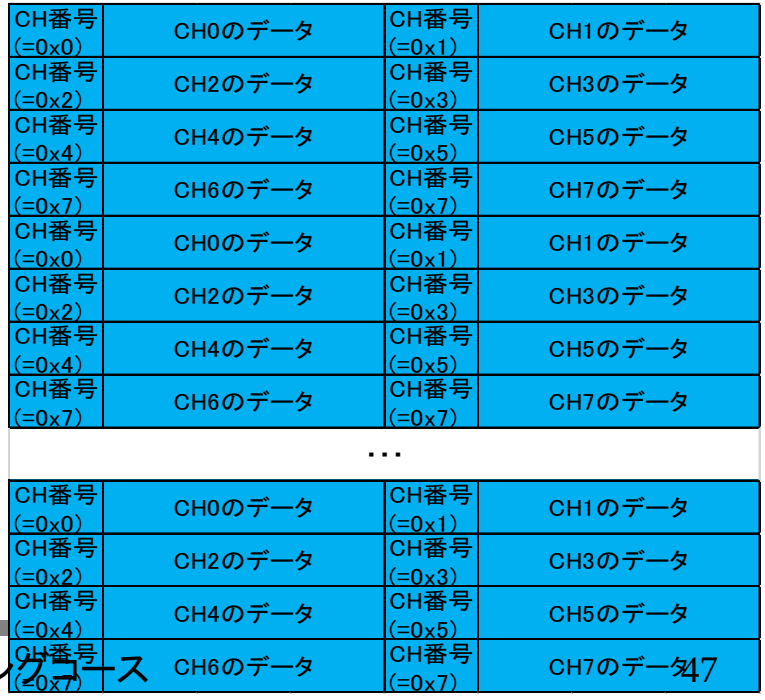
※複数バイトの場合、ビックエンディアン

データ量について

- 1windowあたりのデータ量
  = 2Byte (=1ch分のデータ)
  × ch数

- Data length（データ部分のバイト長）
  = 1windowあたりのデータ量
  × window数
  = 2Byte (=1ch分のデータ)
  × ch数
  × window数



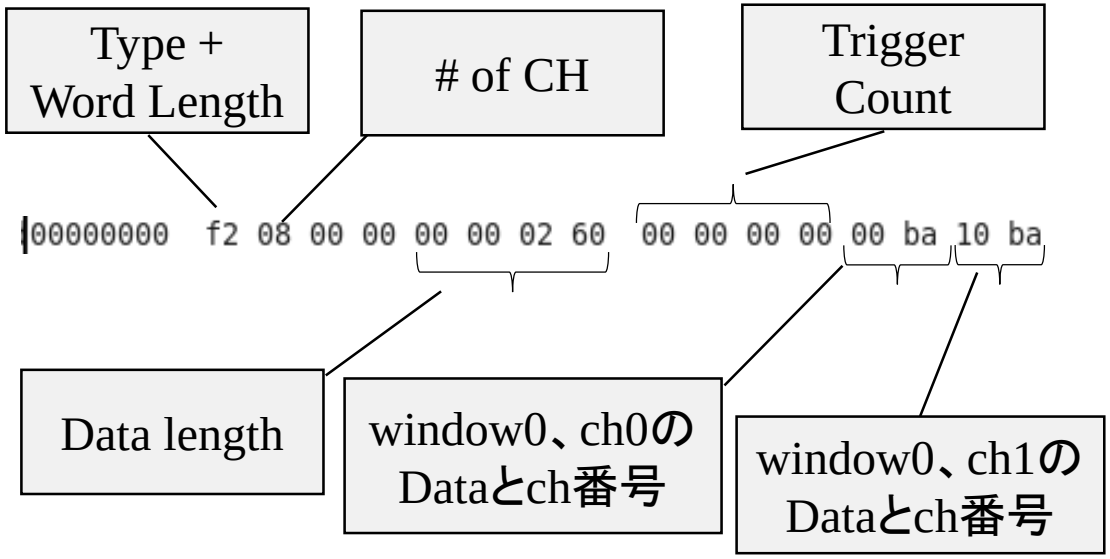


sample.datの確認

サンプルデータ(sample.dat)の確認

```
% hexdump -Cv ~/daqmw-tc-network/bs/sample.dat | less
```

サンプルデータの初めの数Byte

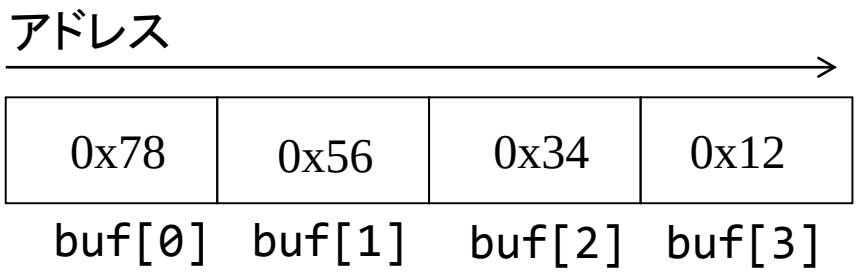


Type (=0xF)	Word length (=0x2)	# of Ch (= 0x8)	0x00	0x00
Data length(データ部分のバイト長)				
Trigger Count（電源投入時からのTriggerパルスカウント）				
データ部				

CH番号 (=0x0)	CH0のデータ	CH番号 (=0x1)	CH1のデータ
CH番号 (=0x2)	CH2のデータ	CH番号 (=0x3)	CH3のデータ
CH番号 (=0x4)	CH4のデータ	CH番号 (=0x5)	CH5のデータ
CH番号 (=0x7)	CH6のデータ	CH番号 (=0x7)	CH7のデータ
CH番号 (=0x0)	CH0のデータ	CH番号 (=0x1)	CH1のデータ
CH番号 (=0x2)	CH2のデータ	CH番号 (=0x3)	CH3のデータ
CH番号 (=0x4)	CH4のデータ	CH番号 (=0x5)	CH5のデータ
CH番号 (=0x7)	CH6のデータ	CH番号 (=0x7)	CH7のデータ
…			
CH番号 (=0x0)	CH0のデータ	CH番号 (=0x1)	CH1のデータ
CH番号 (=0x2)	CH2のデータ	CH番号 (=0x3)	CH3のデータ
CH番号 (=0x4)	CH4のデータ	CH番号 (=0x5)	CH5のデータ
CH番号 (=0x7)	CH6のデータ	CH番号 (=0x7)	CH7のデータ

ネットワークバイトオーダー

0x 78 56 34 12 の順に送られてきたデータを



intとしての解釈

little endian　0x 12345678 = 305419896　（順序が逆）

bit endian　0x 78563412 = 2018915346　（そのままの順）

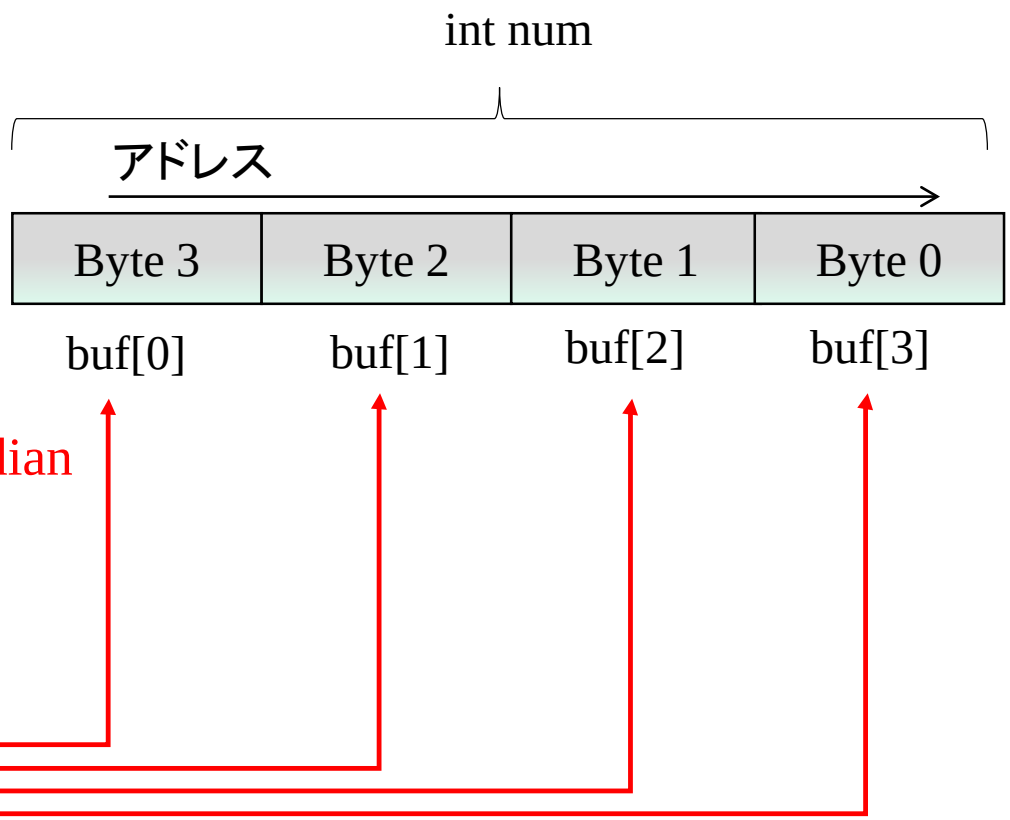
ネットワークバイトオーダーはbig endian

ネットワークバイトオーダー

union(共用体)は様々な型のデータを共通のメモリー領域で管理

byte_order.cpp (一部)

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



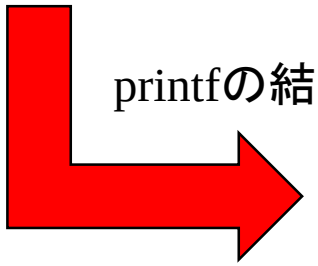
byte_order.cpp (一部)

```
x.num = 0x12345678;
```

byte_order.cppではbuf[] のアドレスと格納されている値を表示する

ネットワークバイトオーダー

```
my_num x, y;
x.num = 0x12345678;

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



printfの結果例

```
% ./byte_order

x: 0x7fff78597440 0 0x78
x: 0x7fff78597441 1 0x56
x: 0x7fff78597442 2 0x34
x: 0x7fff78597443 3 0x12
```

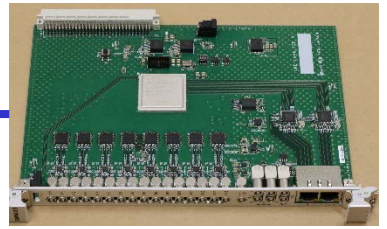
※アドレス値は環境によって異なるが、必ず+1されていく

htonl()関数を使うとどうなりますか？

（ex02と同様、プログラムをexからsandboxにコピーして、プログラムを起動してみて下さい）

ネットワークバイトオーダー

インテルCPU搭載



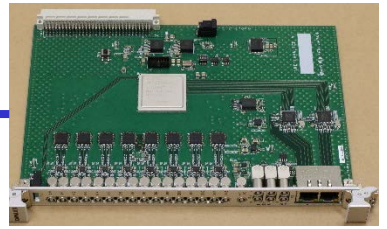
ホストオーダー:
リトルエンディアン

ビックエンディアンで送受信

- データ送信時にhtonl関数、htons関数を使って、
  リトルエンディアンからビックエンディアンに変換
- データ送信時にntohl関数、ntohs関数を使って、
  ビックエンディアンからリトルエンディアンに変換

ネットワークバイトオーダー

モトローラCPU搭載



ホストオーダー：
ビックエンディアン

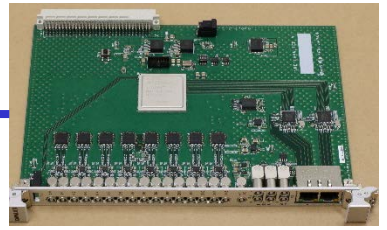
ビックエンディアンで送受信

- データ送信時にhtonl関数、htons関数を使って、
  ビックエンディアンからビックエンディアンに変換（つまり、変わらない）
- データ送信時にntohl関数、ntohs関数を使って、
  ビックエンディアンからビックエンディアンに変換（つまり、変わらない）

関数を使えば、ホストオーダーがどちらでも対応できる

ネットワークバイトオーダー

インテルCPU搭載



ホストオーダー:
リトルエンディアン

リトルエンデンィアンで送受信

注意！！
リトルエンディアンで送信することもある。
この時は、htonl関数などを使わない等の対応が必要。

仕様書や作成者に聞いて、
エンディアンを確認することが重要