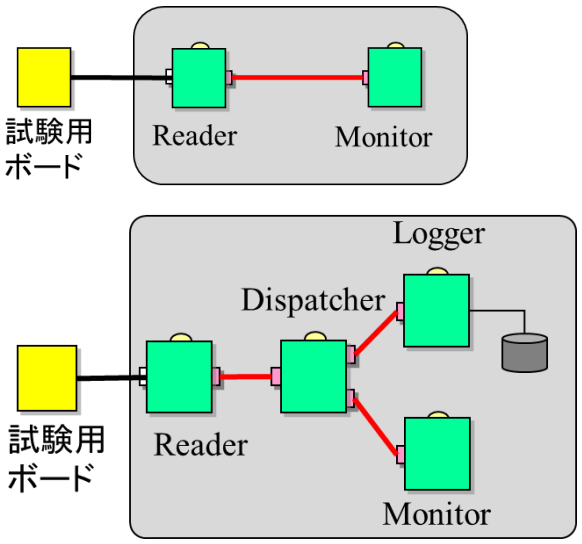
DAQ-Middleware トレーニングコース実習

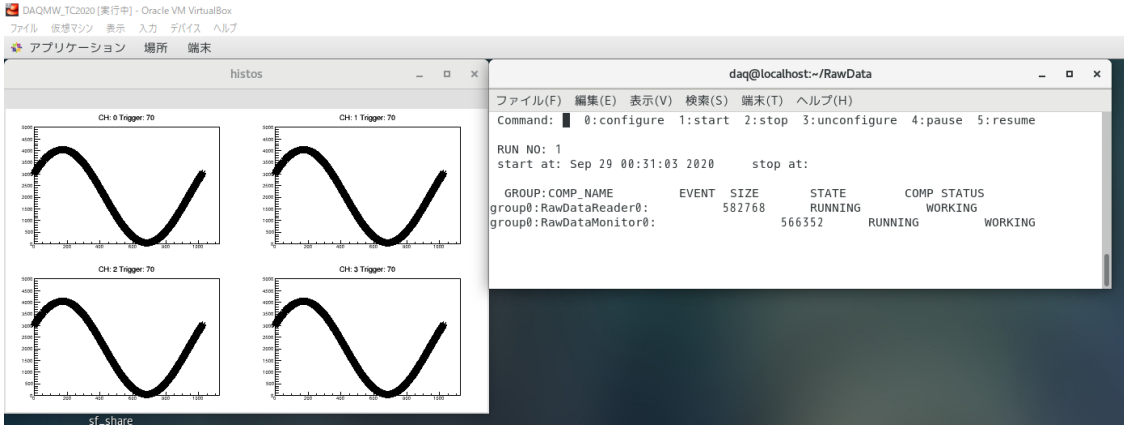
濱田英太郎
高エネルギー加速器研究機構
素粒子原子核研究所

目標と目的

「目標」

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





「目的」

DAQ-MWを利用したソフトウェアの基本を学ぶ

試験用ボード

スイッチで設定を可能にする。

・データ種類

00 : サイン波

01 : 三角波

10 : 矩形波

11 : パルス波

・周波数（1イベ

データを送る周

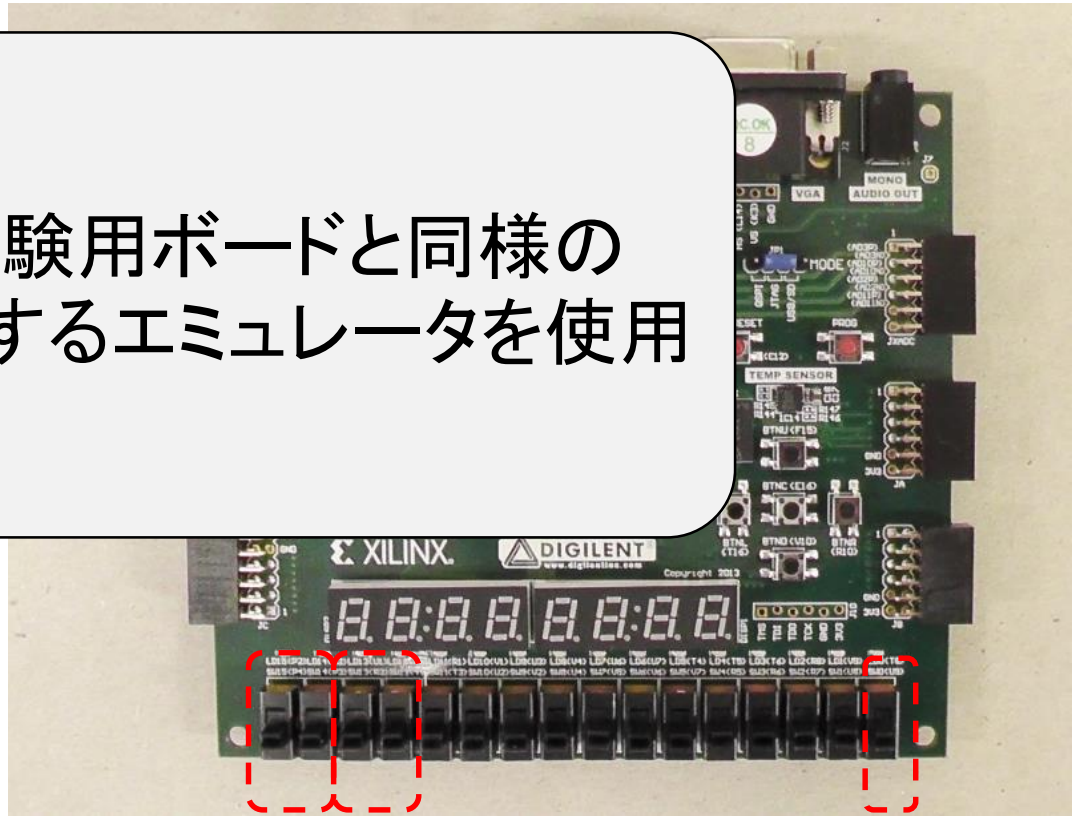
00 : 1Hz

01 : 5Hz

10 : 10Hz

11 : 100Hz

今年度は試験用ボードと同様の
データを出力するエミュレータを使用



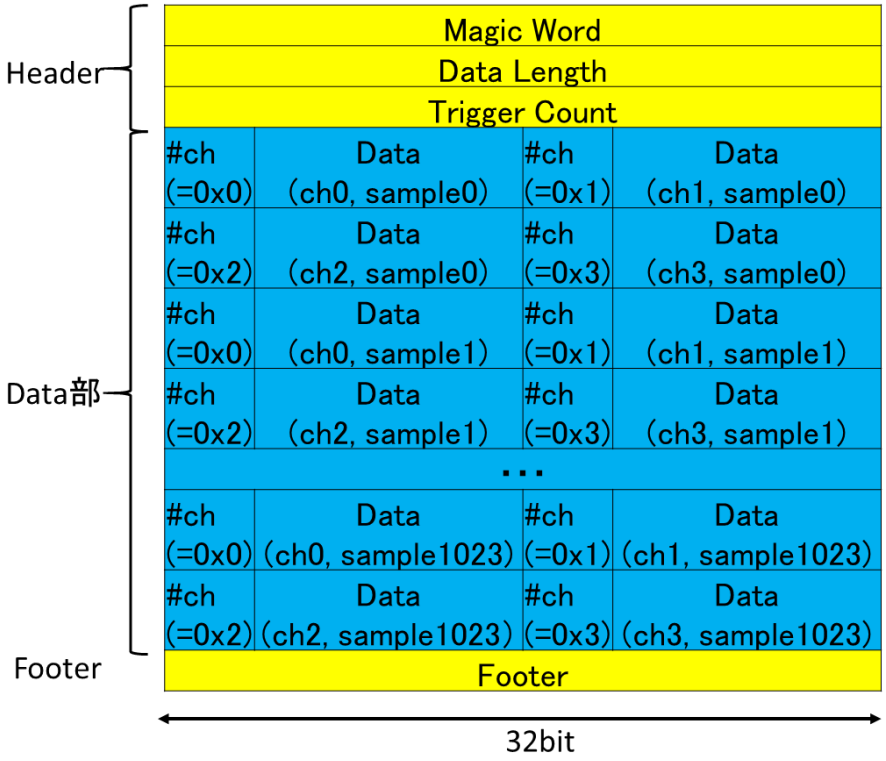
データ種類　周波数　データ送信ON

・データ送信ON

0 : データ送信OFF

1 : データ送信ON

エミュレータ　データフォーマット



「Header」
・Magic Word
常に　0x01234567

・Datalength
Data部のバイト長

・Trigger Count
1イベントのデータを送るごとに+1されていく。

「Data部」
・各データは16bit
（上位4bitはch番号、下位12bitにデータ値）
・1イベントはsample0から順々にsample1023まで1024sampleを送る
・各sampleはch0からch3までの4ch分を送る

「Footer」
・Footer
常に　0x89ABCDEF

※全てビッグエンディアン

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（余裕がある人向け）
  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
```

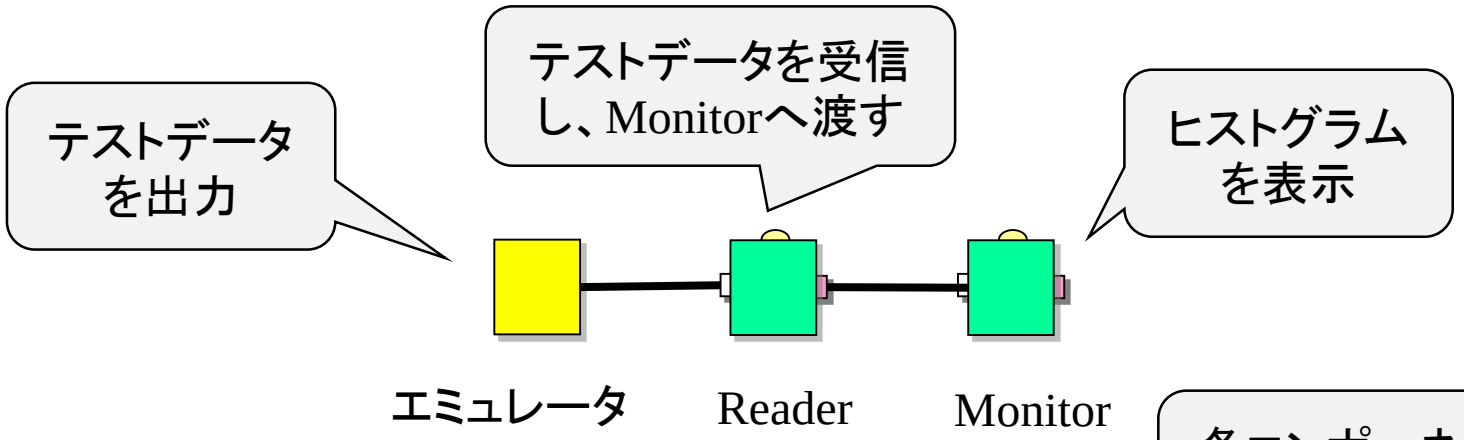
ホームディレクトリに「daqmw-tc2」というディレクトリが追加されます。

実習ファイル　中身の説明

- ex
  実習で行う項目の解説
- sandbox
  特に使わない
- doc
  ボードの出力データフォーマットの説明資料がある
- trigger
  特に使わない
- 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）

~/MyDaqディレクトリを作り、その中でプログラムを作っていく
(daqmw-tc2の中ではない)

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



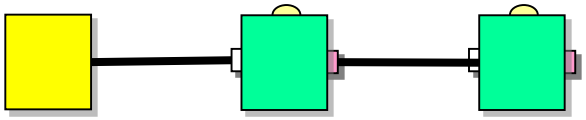
手順

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. データ収集開始
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

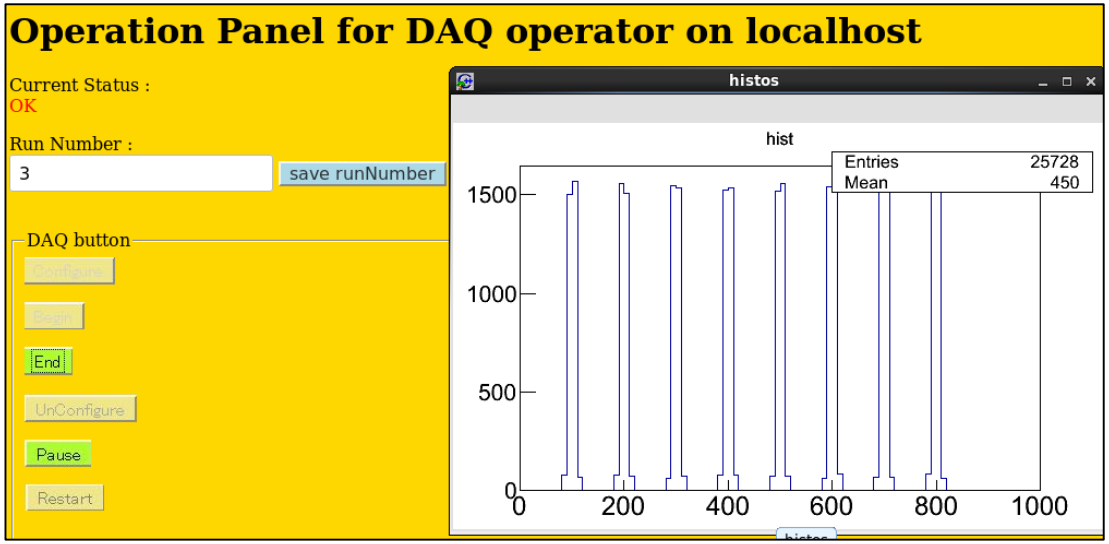
設定ファイル（コンポーネントとその接続情報等）

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-Middleware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）

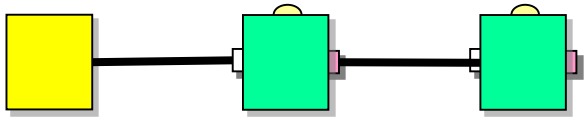
目的

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。

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



エミュレータ　Reader　Monitor

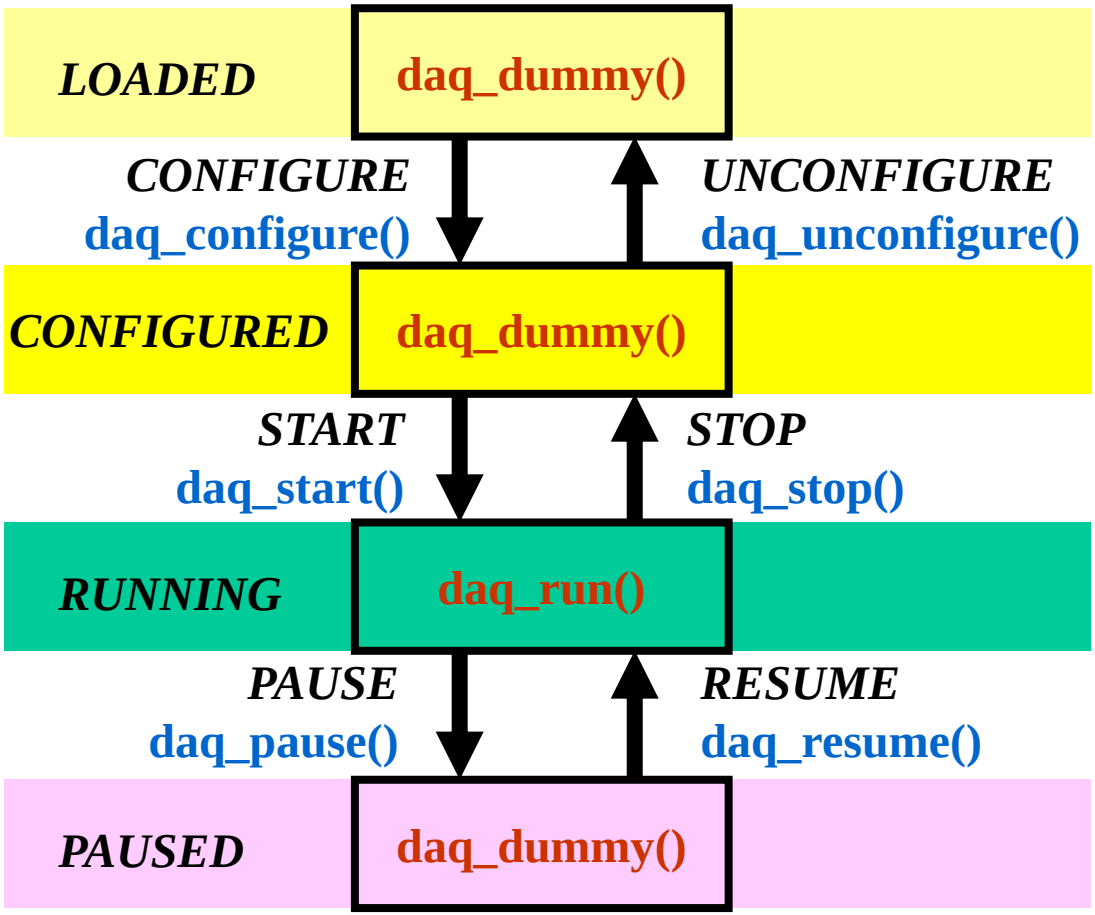
手順

1. SampleReaderのconfigureするときに、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

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。

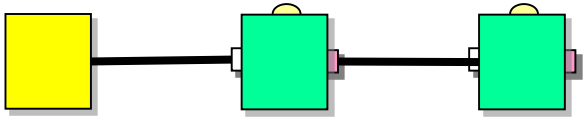
ex04　コンフィグレーションファイルの編集（コンポーネント構成）

目的

コンフィグレーションファイルの編集方法を学ぶ。

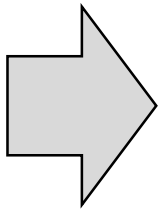
行うこと

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。

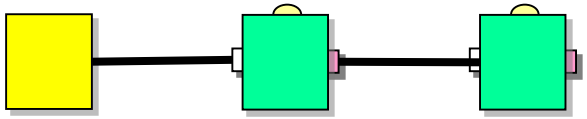


エミュレータ　Reader　Monitor

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



エミュレータ　Reader　Logger

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3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

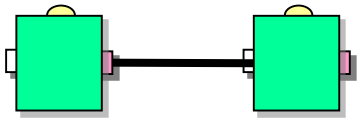
コンポーネントの実行形式ファイルのパス

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)

outportの設定

初期パラメータ

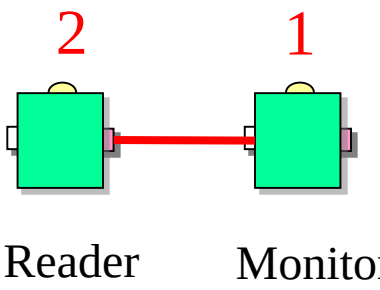


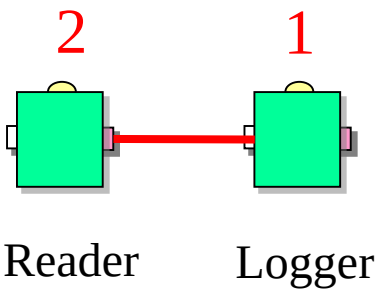
Reader　Monitor

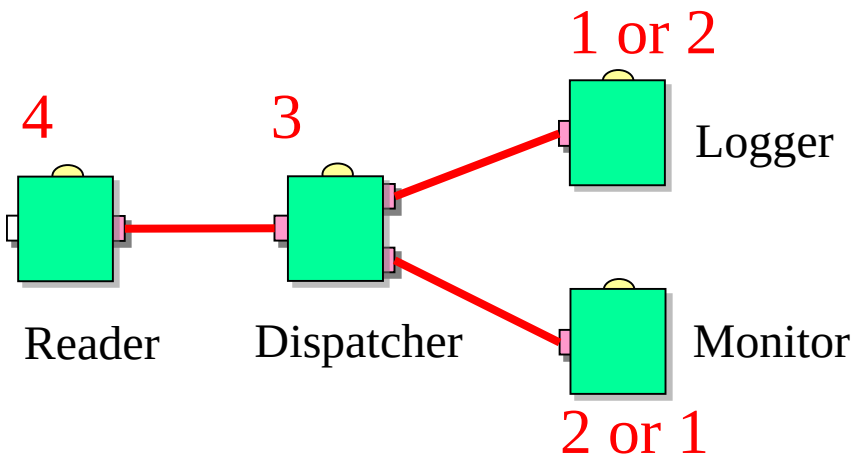
コンフィグレーションファイル（スタートコマンド投入の際の順番）

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

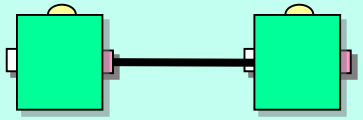
コンポーネントの実行形式ファイルのパス

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

outport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

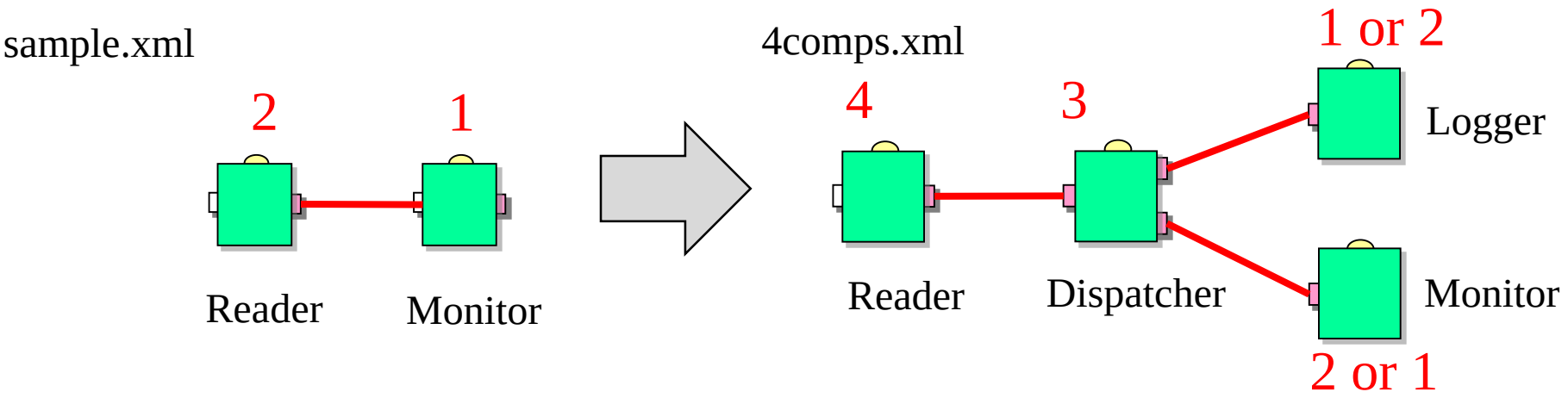
Magic: 0x5a
Format Version: 0x01
Module Number: 0x00 – 0x07
Event Data: 複数のガウス関数。100, 200, 300, ... 800にピークがある。
1000倍した整数値で送ってくる。ネットワークバイトオーダー。

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。

```
% hexdump -Cv（ファイル名）| less
```

EX04 課題

DispatcherとLoggerを追加し、4つのコンポーネントで動作するシステムを作成する



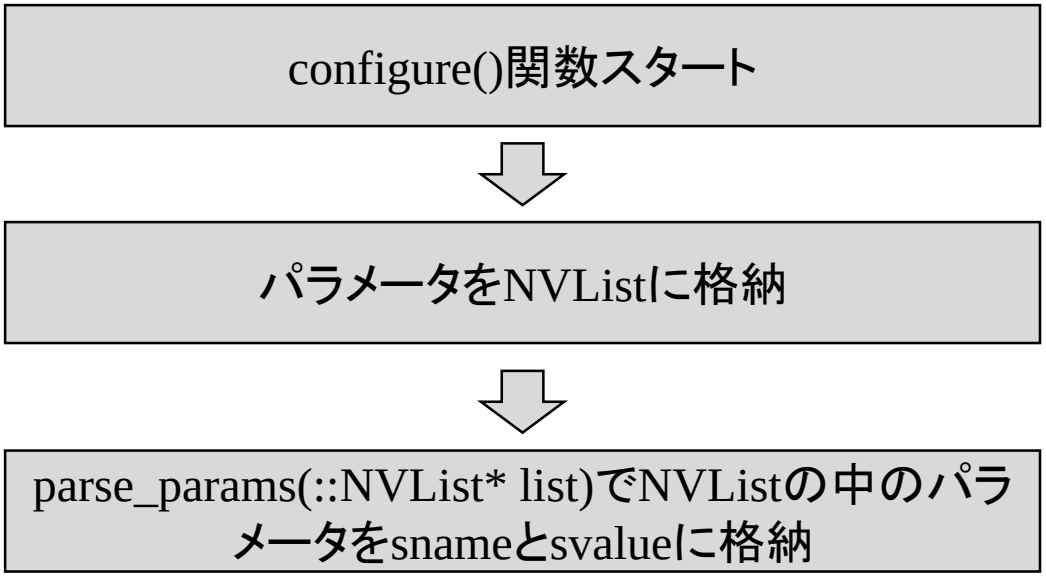
ex05 パラメータ取得

目的

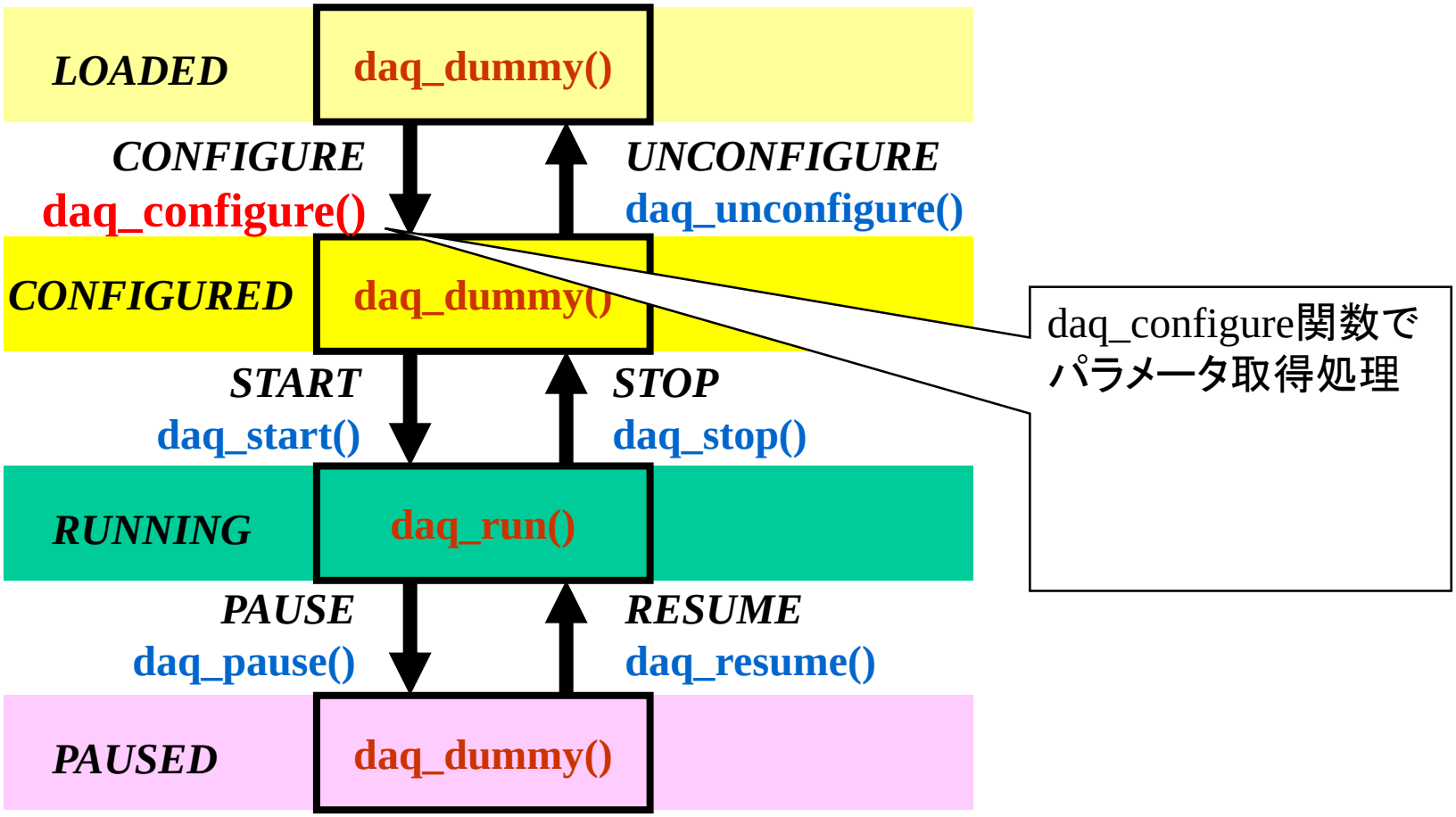
パラメータの設定方法について学習する

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。



コンポーネント状態遷移



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;
}
```

コンフィグレーションファイルからパラメータを取り出し、paramListに格納する

(*list)[0].value　(*list)[1].value

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

(*list)[2].value　(*list)[3].value

コンフィグレーションファイル（sample.xml）の一部

0	"srcAddr"
1	"127.0.0.1"
2	"srcPort"
3	"2222"

paramList（NVList)に格納される値

SampleReader - daq_configure()

```
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);
        }
    }
```

m_srcAddr = ”127.0.0.1”

m_srcPort = 2222
(strtol関数等で文字列をint型に変更)

0	"srcAddr"
1	"127.0.0.1"
2	"srcPort"
3	"2222"

for文 1回目のloopのsname

for文 1回目のloopのsvalue

for文 2回目のloopのsname

for文 2回目のloopのsvalue

paramList（NVList)に格納された値

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変数を追加
- Tcanvasの横のサイズをあらわす変数を追加

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

ex06 コンポーネント間のデータについて

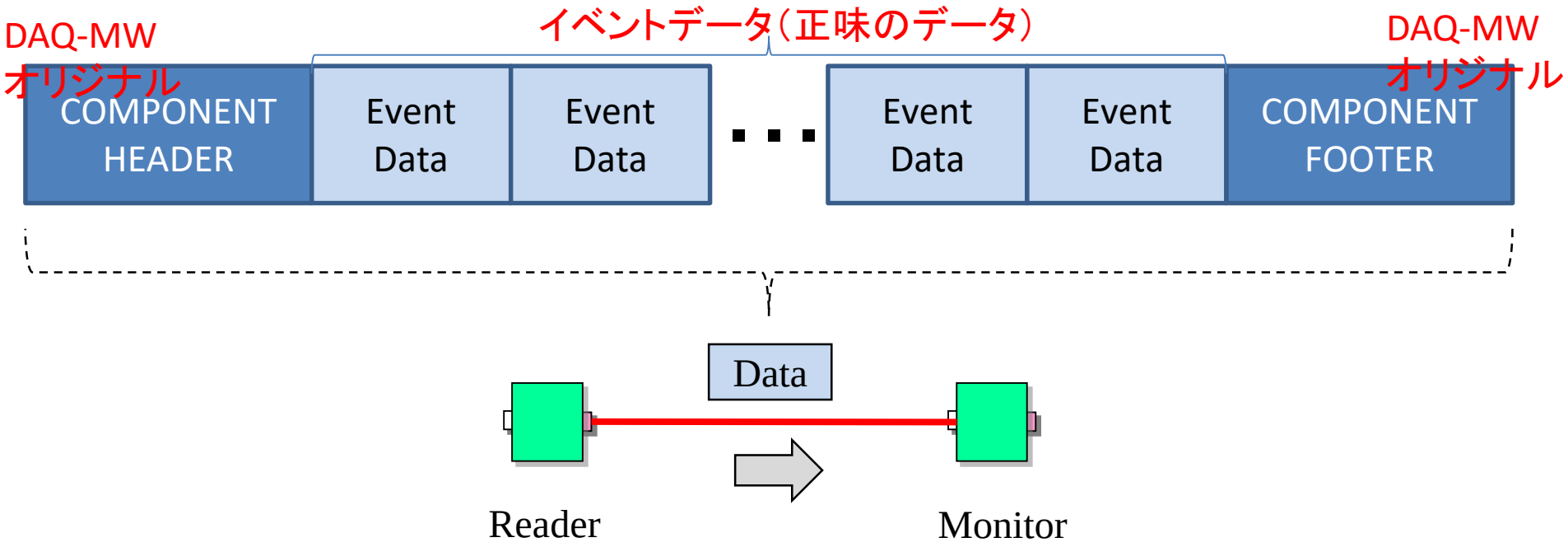
課題:

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

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

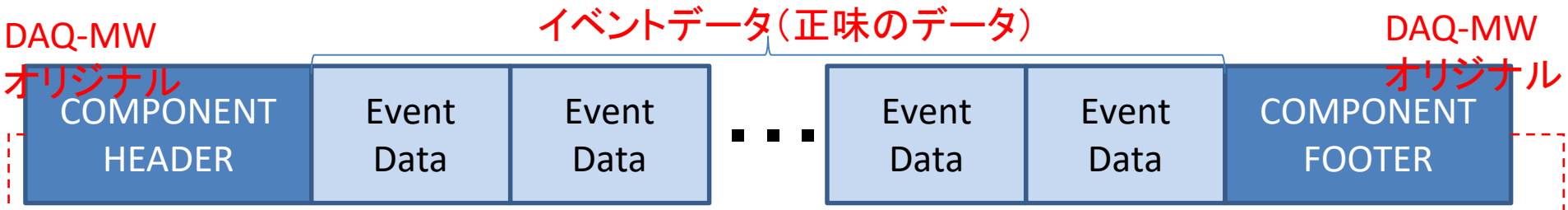
- SampleReaderおよびSampleMonitorのプログラムについて説明

コンポーネント間のデータフォーマット



DAQ-Middlewareオリジナルのヘッダとフッタがある

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

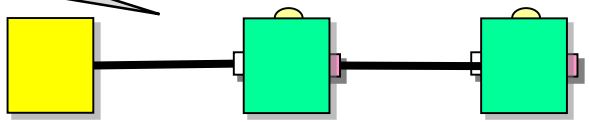
Component Footer

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

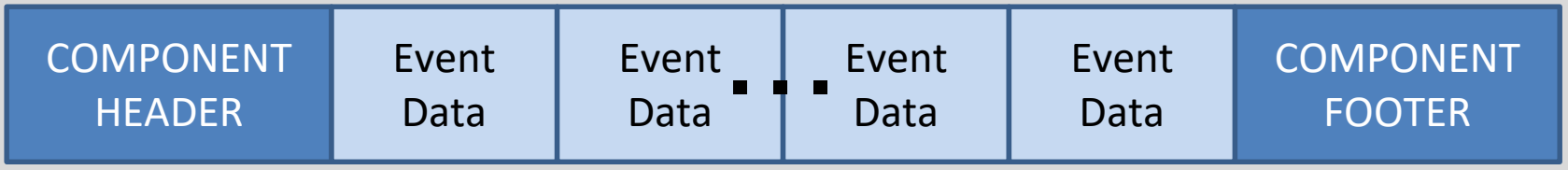
SampleReader – SampleMonitorの場合

1回のdaq_run関数で
1024BYTE
読み込む



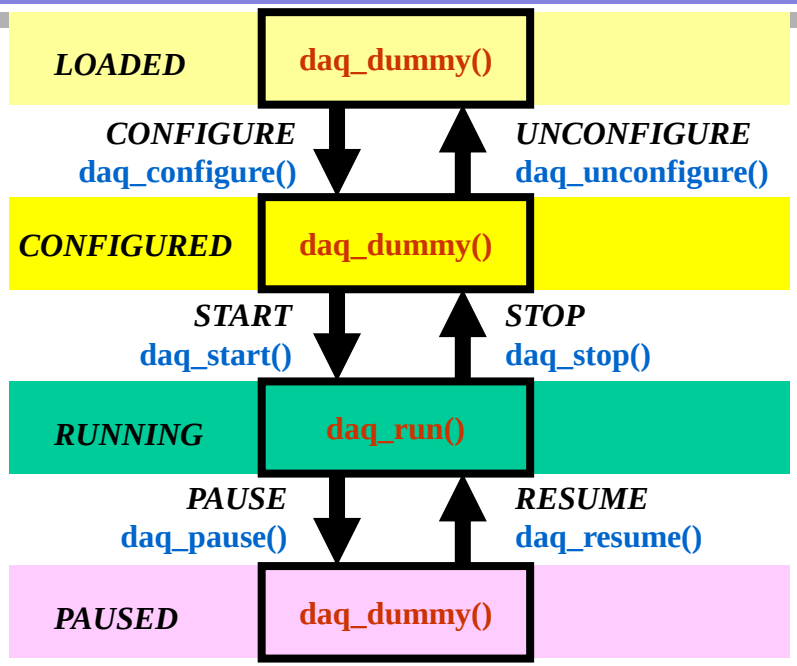
エミュレータ　Reader　Monitor

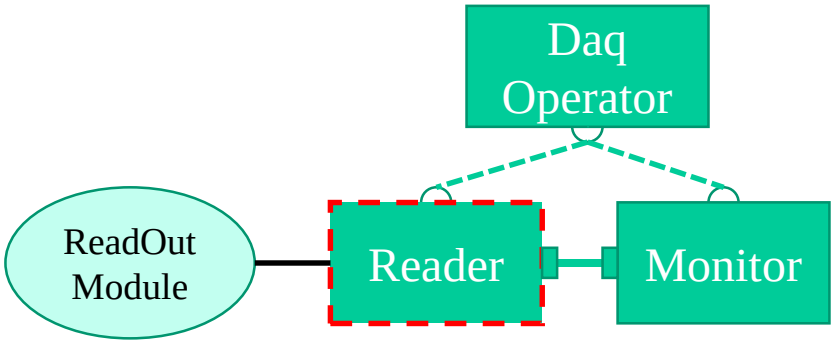
ReaderからMonitorへのデータ



1024BYTE

SampleReaderの概要





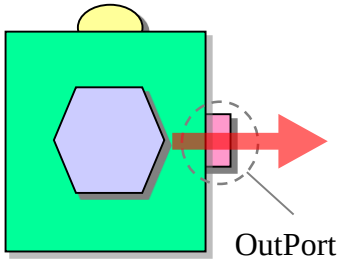
Gatherer (SampleReader)

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_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_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。

SampleReader (SampleReader.h)

重要な変数についての紹介




```
// SampleReader.h
class SampleReader
    : public DAQMW::DaqComponentBase
{
private:
    TimedOctetSeq          m_out_data;

（省略…）

    unsigned char m_data[SEND_BUFFER_SIZE]

（省略…）
```

outportのデータバッファ

読み込んだデータ用のバッファ
(SEND_BUFFER_SIZE = 1024))

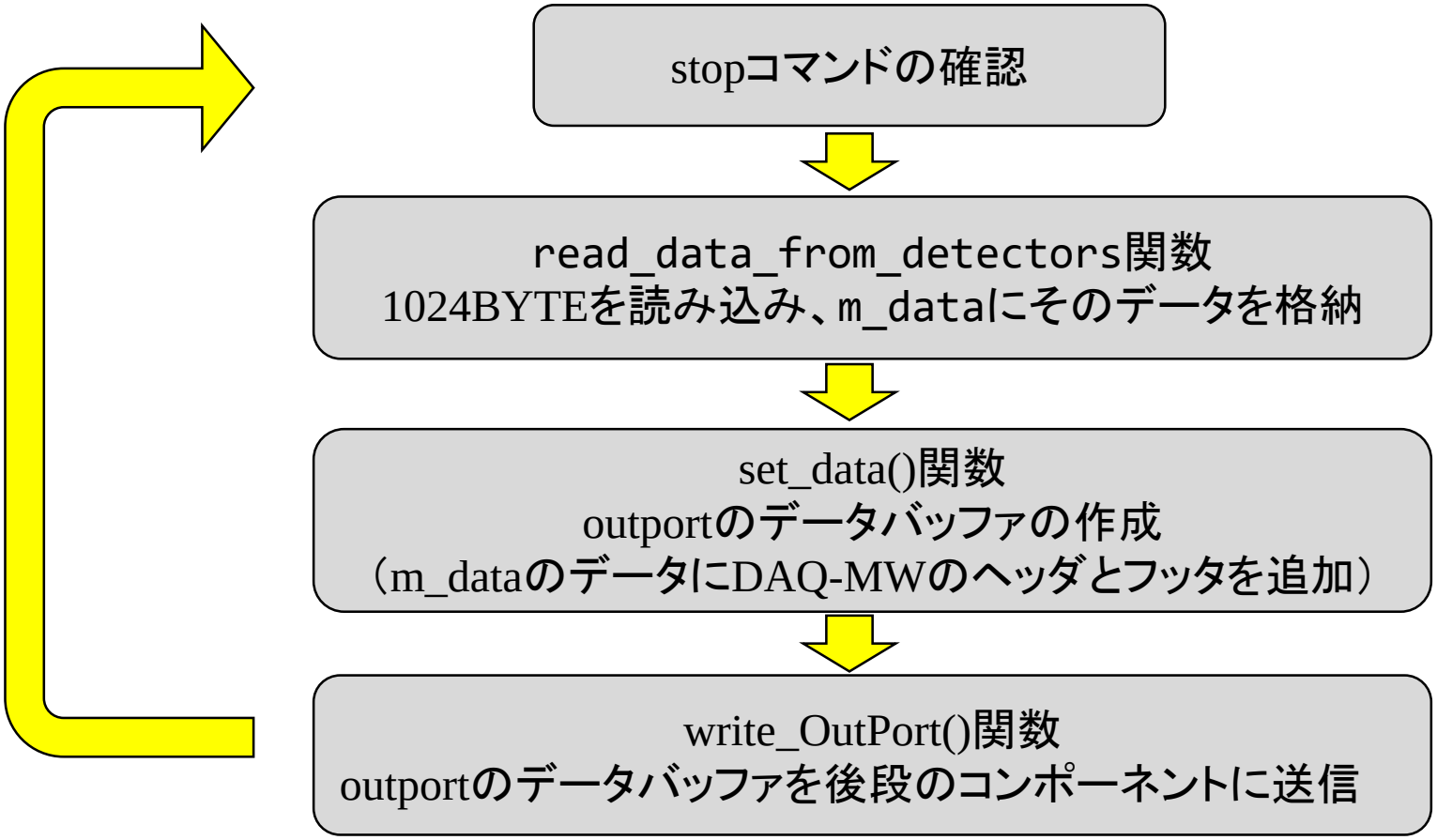
SampleReader - daq_start()

リードアウトモジュールに接続する

```
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()

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



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);
        }
    }
    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
    }
```

stopコマンドの確認

①1024BYTEを読み込み、
m_dataにそのデータを格納

②outportのデータバッファの作成
（m_out_dataの作成）

③outportのデータバッファを
後段のコンポーネントに送信

SampleReader - read_data_from_detectors ()
1024BYTEを読み込み、m_dataにそのデータを格納

```
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);

    if (status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "### ERROR: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
    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
(SEND_BUFFER_SIZE = 1024)

SampleReader - set_data()
outportのデータバッファの作成

```
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;
}
```

DAQ-MWオリジナルの関数
headerとfooterを作成

m_out_dataを作成
- m_out_data.lengthで後段コンポーネントに送るデータの長さを指定 (header + Data + footer)
- m_out_data.dataに送るデータの中身を指定

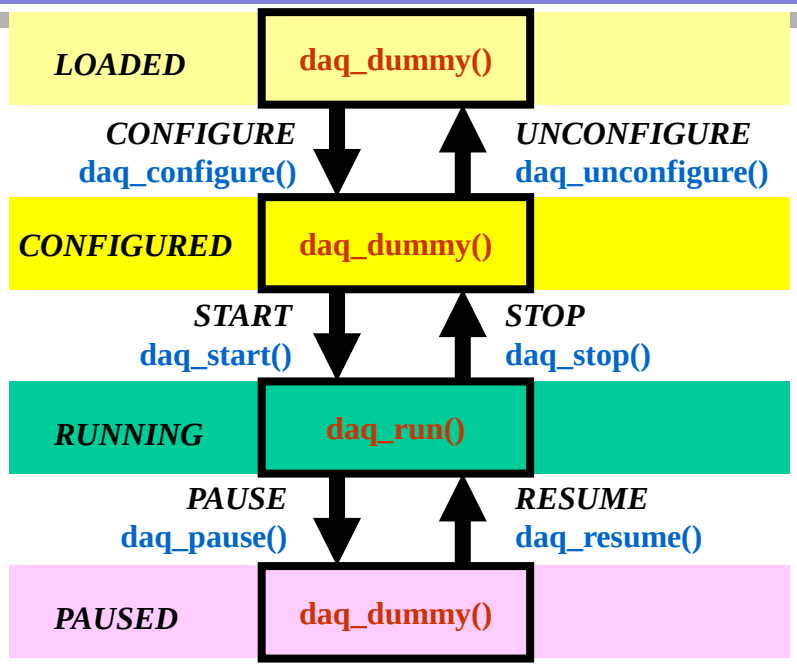
SampleReader::write_OutPort()
outportのデータバッファを後段のコンポーネントに送信

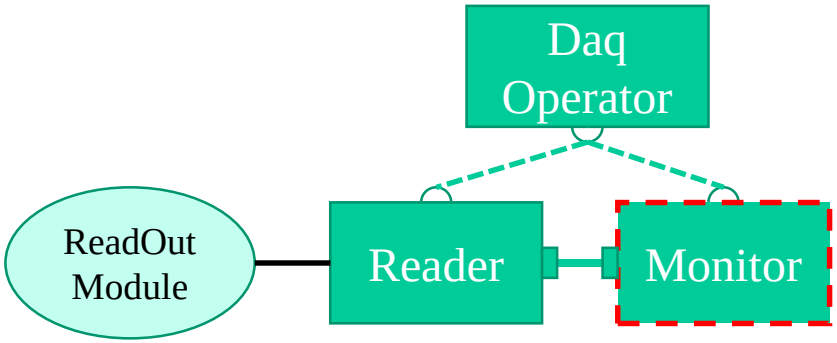
```
int SampleReader::write_OutPort()
{
  ////////////////// send data from OutPort  //////////////////
  bool ret = m_OutPort.write();

  ////////////////////// check write status ///////////////////
  if (ret == false) {  // TIMEOUT or FATAL
    m_out_status  = check_outPort_status(m_OutPort);
    if (m_out_status == BUF_FATAL) {   // Fatal error
      fatal_error_report(OUTPORT_ERROR);
    }
    if (m_out_status == BUF_TIMEOUT) { // Timeout
      return -1;
    }
  }
  else {
    m_out_status = BUF_SUCCESS; // successfully done
  }
```

後段のコンポーネントにデータ送信

エラー処理

SampleMonitorの概要





Monitor (SampleMonitor)

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_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

SampleMonitor - SampleMonitor.h

```
TimedOcteSeq             m_in_data
InPort<TimedOcteSeq>     m_InPort

（省略）

    int      m_monitor_update_rate;
    unsigned char m_recv_data[4096];
    unsigned int  m_event_byte_size;
    struct sampleData m_sampleData;

    bool m_debug;
};
```

前段のコンポーネントから受信したデータはm_in_dataに格納

m_in_data.dataからDAQ-MWのヘッダとフッタを取り除いた正味のデータを格納

エミュレータのデータフォーマットに対応させた構造体（sampleDataはsampleData.hで定義されている）

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。
（エミュレータのデータ構造より）

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

エミュレータが出力するデータのフォーマット

SampleMonitor - daq_start()
ヒストグラムデータの作成

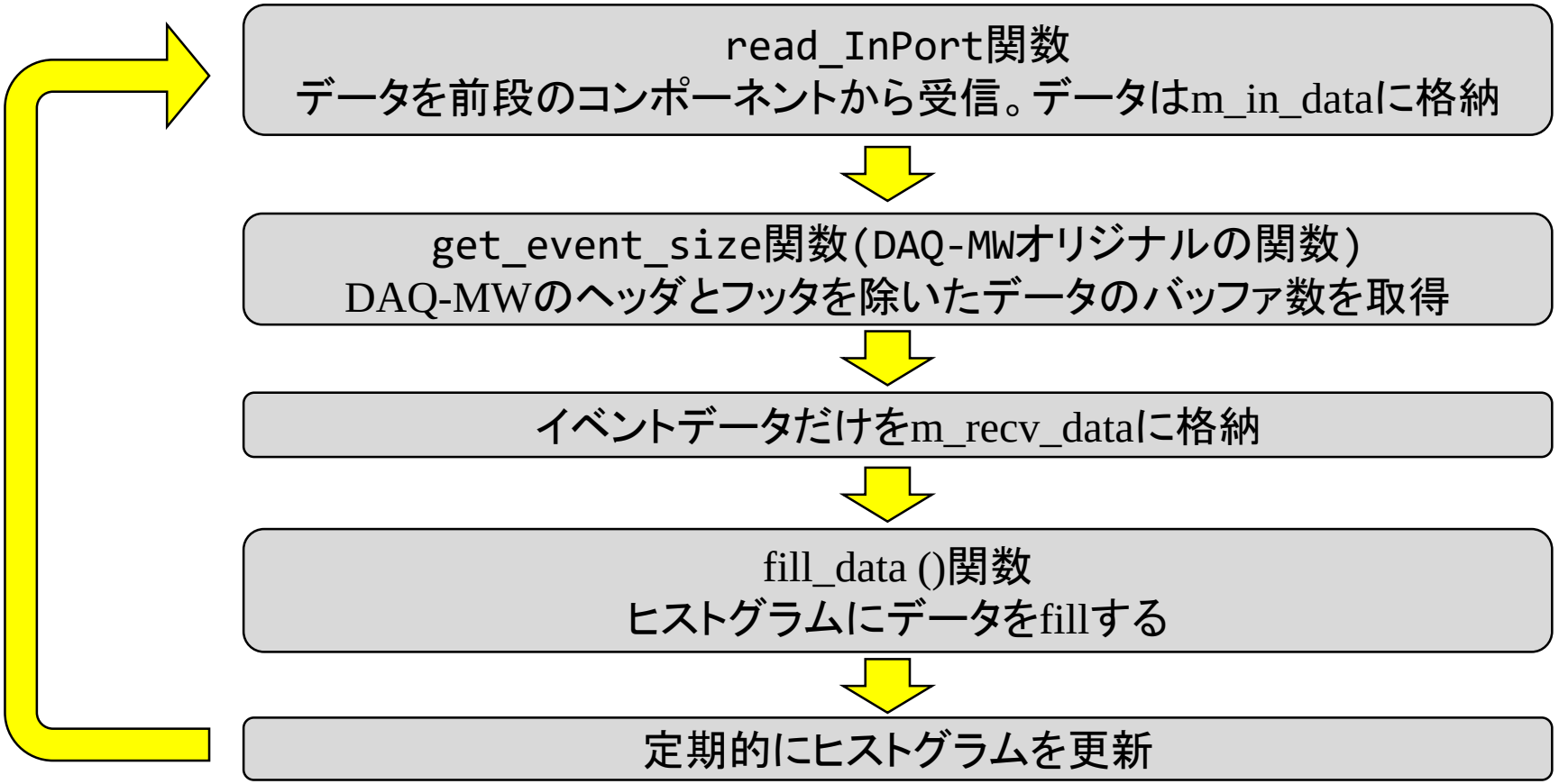
```
int SampleMonitor::daq_start()
{
    m_in_status  = BUF_SUCCESS;
    ///////////////// CANVAS FOR HISTOS ///////////////////
    if (m_canvas) {
        delete m_canvas;
        m_canvas = 0;
    }
    m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400);

    ////////////////       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);
```

キャンバスを定義
（ROOT特有の書き方）

ヒストグラムを定義
（ROOT特有の書き方）

SampleMonitor - daq_run()

上流コンポーネントからデータを受け取り、デコードしてヒストグラムにデータをFillする。定期的にヒストグラムを更新する。



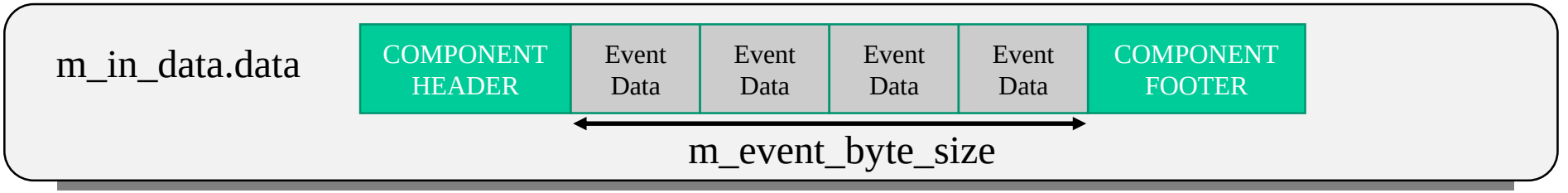
SampleMonitor - daq_run()

```
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);

    （続く）
```

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

DAQ-MWのヘッダとフッタを除いたデータのバッファ数を取得
（get_event_sizeはDAQ-MWオリジナルの関数）

m_in_data.dataはDAQ-MWのヘッダやフッタも含まれる。
そこからイベントデータだけをm_recv_dataに格納



SampleMonitor - 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

シーケンス番号を+1する

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

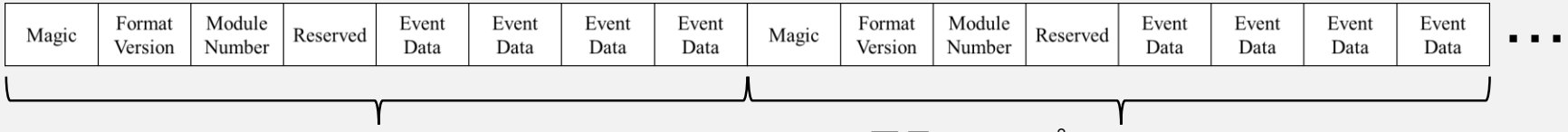
SampleMonitor - fill_data()
ヒストグラムにデータをfill処理する関数

ONE_EVENT_SIZEは8BYTE

```
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デコードし、
m_sampleData.dataに格納
（次のページでこの関数について説明）

mydata（エミュレータからの1024Byteのイベントデータ）



最初のループ
始めの1イベントデータを取り出し、1000で
割った値をヒストグラムにFillする

2回目のループ
次の1イベントデータを取り出し、1000で
割った値をヒストグラムにFillする

SampleMonitor - decode_data()
エミュレータのデータをデコードする

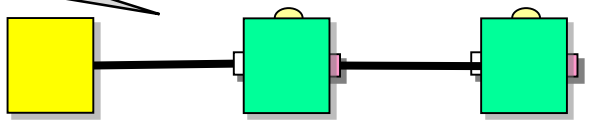
```
int SampleMonitor::decode_data(const unsigned char* mydata)
{
    m_sampleData.magic      = mydata[0];
    m_sampleData.format_ver = mydata[1];
    m_sampleData.module_num = mydata[2];
    m_sampleData.reserved   = mydata[3];
    unsigned int netdata    = *(unsigned int*)&mydata[4];
    m_sampleData.data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

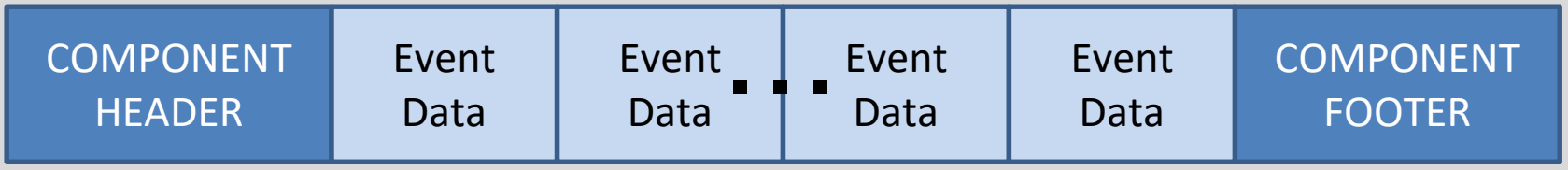
SampleReader – SampleMonitorの場合

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



1024BYTE

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

エミュレータデータ
1イベント目

```
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)

ここまで作ったプログラムを利用し、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作成する

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.データがセーブされていることを確認

SampleLoggerの
名前を変えるだけ

この章の課題

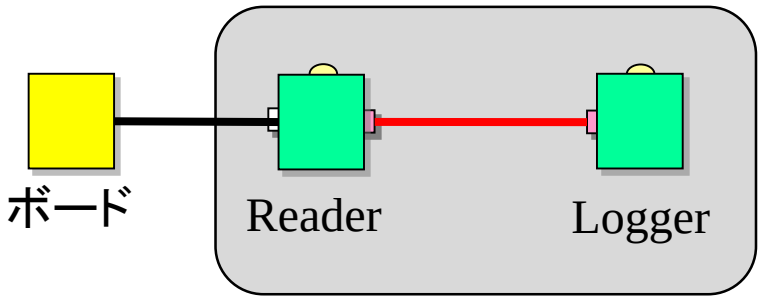
SampleReaderのプログラムを修正

- SampleReader

1024BYTEごとreadする

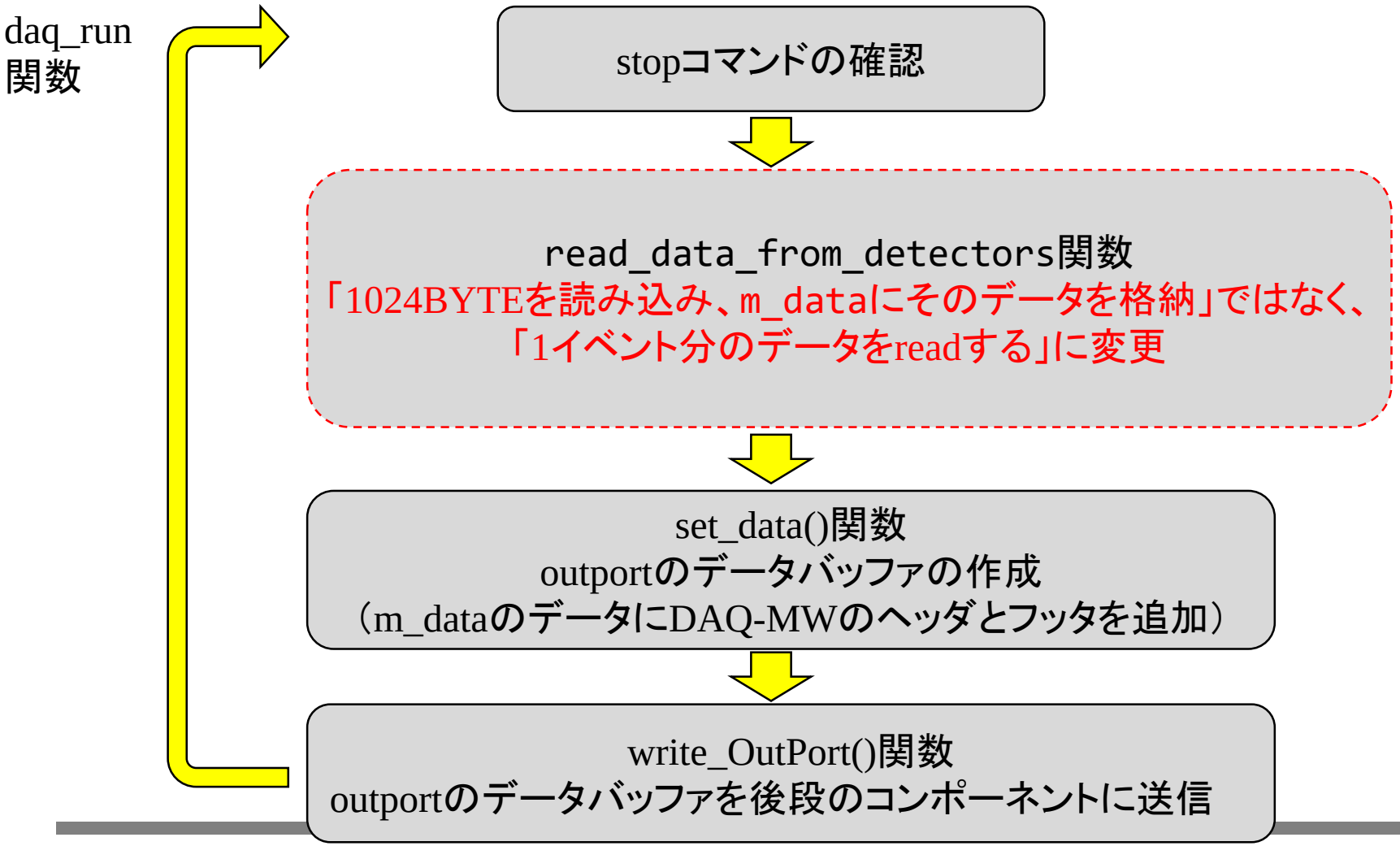
- RawDataReader

1イベント分のデータをreadする

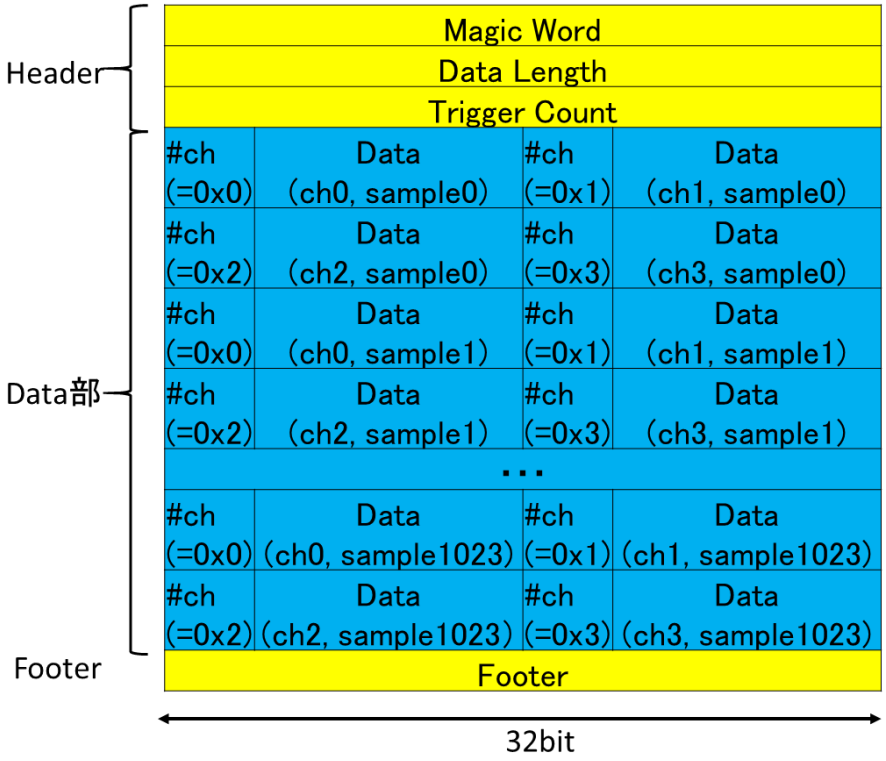


SampleReaderからRawDataReaderを作るためには

SampleReader.cppのread_data_from_detectors関数を編集する



エミュレータ　データフォーマット



「Header」
・Magic Word
常に　0x01234567

・Datalength
Data部のバイト長

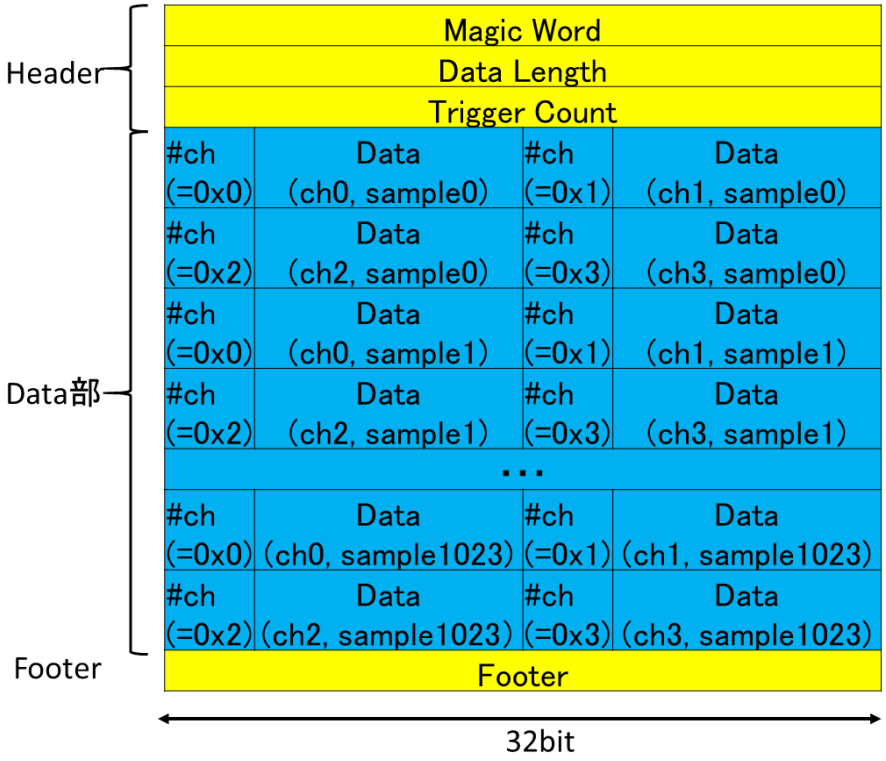
・Trigger Count
1イベントのデータを送るごとに+1されていく。

「Data部」
・各データは16bit
（上位4bitはch番号、下位12bitにデータ値）
・1イベントはsample0から順々にsample1023まで1024sampleを送る
・各sampleはch0からch3までの4ch分を送る

「Footer」
・Footer
常に　0x89ABCDEF

※全てビッグエンディアン

SampleReaderからRawDataReaderを作るためには

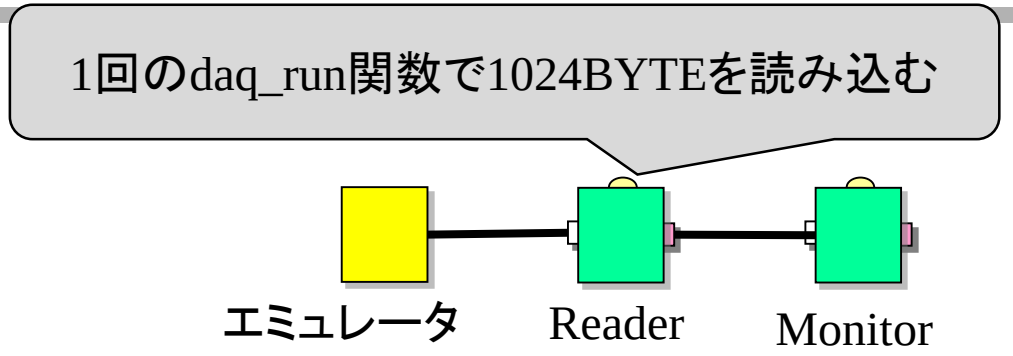


read_data_from_detectors関数で
1イベント分のデータをreadする処理の流れ

1. はじめにヘッダ（12BYTE）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
4. フッタ（4BYTE）をread
5. 戻り値を1イベント分の長さにするに変更

SampleReaderからRawDataReaderを作るためには

SampleReaderでは…



read_data_from_detectors関数が1024BYTEを読み込んでいる

```
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);
（省略）
    else {
        received_data_size = SEND_BUFFER_SIZE;
    }
    return received_data_size;
}
```

m_dataに1イベント分のデータが入るように編集すること

1イベント分のデータ長が戻り値にすること

SampleReaderからRawDataReaderを作るためには

RawDataReader.hも少し編集が必要

```
// SampleReader.h
class SampleReader
    : public DAQMW::DaqComponentBase
{
private:
    TimedOctetSeq          m_out_data;

（省略…）

    unsigned char m_data[SEND_BUFFER_SIZE]

（省略…）
```

m_dataのサイズはSEND_BUFFER_SIZE（=1024）であったが、これでは1イベント分のデータは入らない。
1024×1024にすれば十分な量を確保できる

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

```
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
今回は、この関数の中身を修正する

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

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

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

    /// write your logic here
    /// read 1024 byte data from data server
    int status = m_sock->readAll(&m_data[0], HEADER_SIZE);
    if (status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "### ERROR: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
    else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "### Timeout: Header Read Timeout" << std::endl;
        return DAQMW::Sock::ERROR_TIMEOUT;

    }

    int data_length = 0;

    // get data part length
    data_length = get_data_length(&m_data[0], HEADER_SIZE);
```

データをHEADER_SIZE(= ヘッダの12Byte)だけread

m_dataの先頭に、そのデータを格納

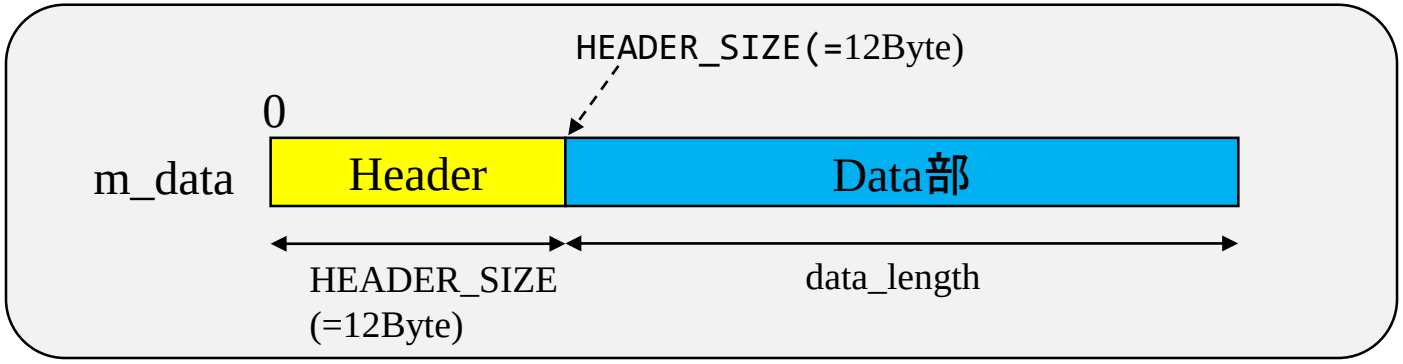
ヘッダからデータ長を取得する関数
(3ページ後で解説)

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

データ部を読むためにをデータ長の分だけread
m_dataの中のヘッダの後にそのデータを格納

（続き）

```
// Then read data part (data_length bytes)
    status = m_sock->readAll(&m_data[HEADER_SIZE], data_length);
    if (status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "### ERROR: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
    else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "### Timeout: Data Read Timeout" << std::endl;
        return DAQMW::Sock::ERROR_TIMEOUT;
    }
```



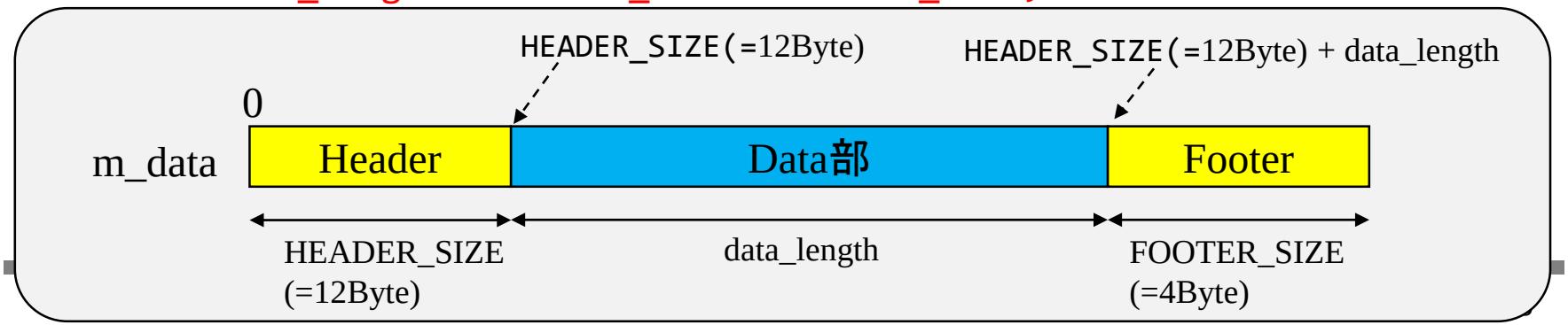
ex07 ボードを読むシステムを動かしてみる (Reader - Logger)　解答例

データをFOOTER_SIZE(= 4Byte)だけread
m_dataの中のデータ部の後に格納

（続き）

```
// Finally, Read footer
   status = m_sock->readAll(&m_data[HEADER_SIZE + data_length], FOOTER_SIZE);
   if (status == DAQMW::Sock::ERROR_FATAL) {
       std::cerr << "### ERROR: m_sock->readAll" << std::endl;
       fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
   }
   else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
       // Header read timeout is not an error
       std::cerr << "### Timeout: Footer Read Timeout" << std::endl;
       return DAQMW::Sock::ERROR_TIMEOUT;
   }
   return data_length + HEADER_SIZE + FOOTER_SIZE;
```

1イベント当たりのサイズ
（＝データ長 + ヘッダの長さ　+ フッタの長さ）



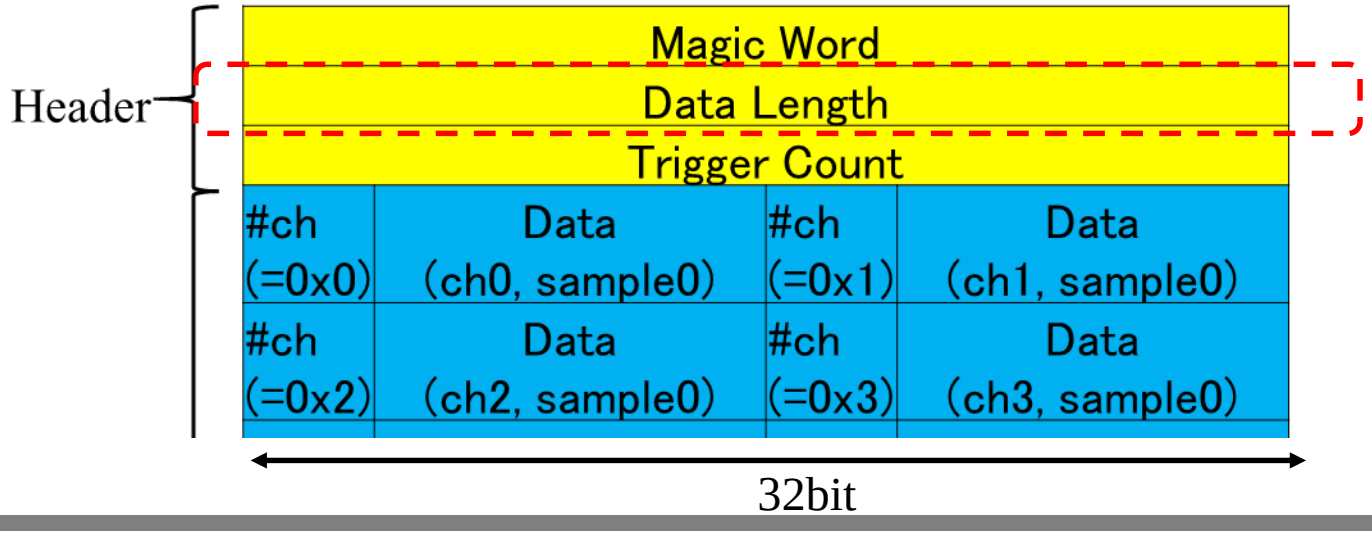
ex07 ボードを読むシステムを動かしてみる (Reader - Logger)　解答例

（続き）

```
int RawDataReader::get_data_length(unsigned char *buf, int buflen)
{
    unsigned int *length;
    unsigned int rv;
    length = (unsigned int *)&buf[4];
    rv  = ntohl(*length);

    return rv;
}
```

ヘッダからデータ長を取得している

バイトオーダーの変換



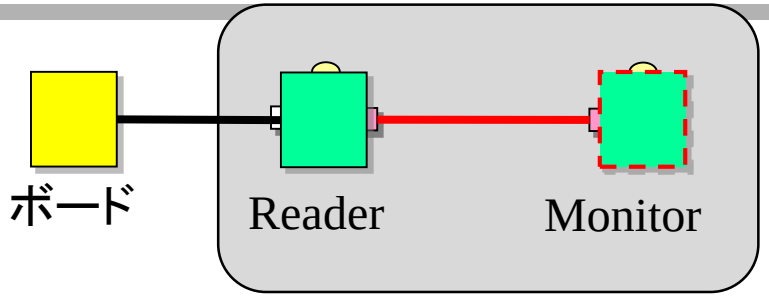
ex07 ボードを読むシステムを動かしてみる (Reader - Logger)　解答例

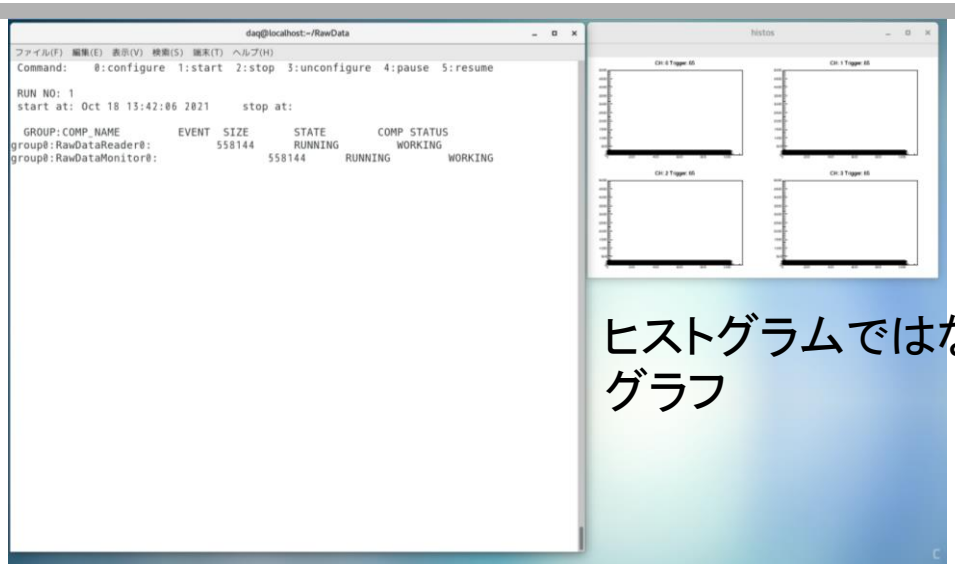
解答のプログラムは

daqmw-tc2/daqmw-tc2/daqmw/RawDataReader

に置いてある。

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。

- MonitorはSampleMonitorを利用して自分で作る。
  受け取った1イベント分のデータをデコードし、グラフにセットする。
  （DAQ-Middleware特有の関数やROOT特有の書き方があるので、理解が難しい箇所があることに注意）。

SampleMonitorからRawDataMonitorを作るためには

- RawDataMonitor.cppのdaq_run関数の編集をする。
  （主にRawDataMonitor.cppのfill_data関数）

- 他の場所の変更は説明いたします。
  具体的は内容は用意したプログラムを利用してください。

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.hの編集」

ヘッダファイルは用意したファイルをコピーして利用すること

RawDataMonitor.h （SampleMonitor.hとの違い）

```
#include "TGraph.h"
// #include "TH1.h"

（省略）

//      int decode_data(const unsigned char* mydata);

（省略）

 ////////// ROOT Graph /////////
    TCanvas *m_canvas;
    RawDataPacket rdp;
    const static int N_GRAPH = 4;
    const static int N_ROW_IN_CANVAS = 2;
    TGraph   *m_graph[N_GRAPH];

（省略）
```

TH1（ヒストグラム用のクラス）ではなく、Tgraph（グラフ用のクラス）

サンプル用のエミュレータのデータのデコード関数は必要ない

ボードのデコードを行うクラスのインスタンス

Tgraphに必要

SampleMonitorからRawDataMonitorを作るためには「新しいファイルを追加する場合」

初日に行ったデコードプログラム（RawDataPacket）を利用するためには、

以下のようにして、RawDataPacket.hおよびRawDataPacket.cppをコピーし、

```
% cp ~/daqmw-tc2/daqmw-tc2/daqmw/RawDataMonitor/RawDataPacket.h   .
% cp ~/daqmw-tc2/daqmw-tc2/daqmw/RawDataMonitor/RawDataPacket.c   .
```

さらに Makefileに

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

を追加

（RawDataPacket.hおよびRawDataPacket.cppを使わなくてもプログラムを作成することができる）

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp　コンストラクタ」

コンストラクタはweb上に書いてある。コピーして利用すること。

※コンストラクタ･･･インスタンスを作成したタイミングで実行される関数

```
RawDataMonitor::RawDataMonitor(RTC::Manager* manager)
    : DAQMW::DaqComponentBase(manager),
      m_InPort("rawdatamonitor_in",   m_in_data),
      m_in_status(BUF_SUCCESS),
      m_monitor_update_rate(30),
      m_event_byte_size(0),
      m_canvas(0),
      m_debug(false)
{
    （省略）
    for (int i = 0; i < N_GRAPH; i++) {
        m_graph[i] = 0;
    }
}
```

メンバ変数の初期化を行う
（必要のないものは削除）

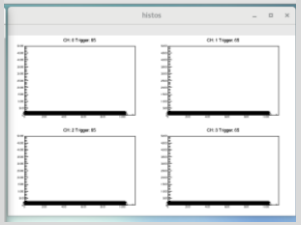
配列はコンストラクタ初期化できないので、関数のなかで初期化を行う

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp　daq_start」

daq_startはweb上に書いてある。
コピーして利用すること。

```
int RawDataMonitor::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);
    int col, row;
    row = N_ROW_IN_CANVAS;
    col = N_GRAPH / row;
    if (N_GRAPH % row != 0) {
        col ++;
    }
    m_canvas->Divide(col, row);
    for (int i = 0; i < N_GRAPH; i++) {
        if (m_graph[i]) {
            delete m_graph[i];
            m_graph[i] = 0;
        }
        m_graph[i] = new TGraph();
    }
```

TCanvasの作成
（ROOT特有の
書き方）

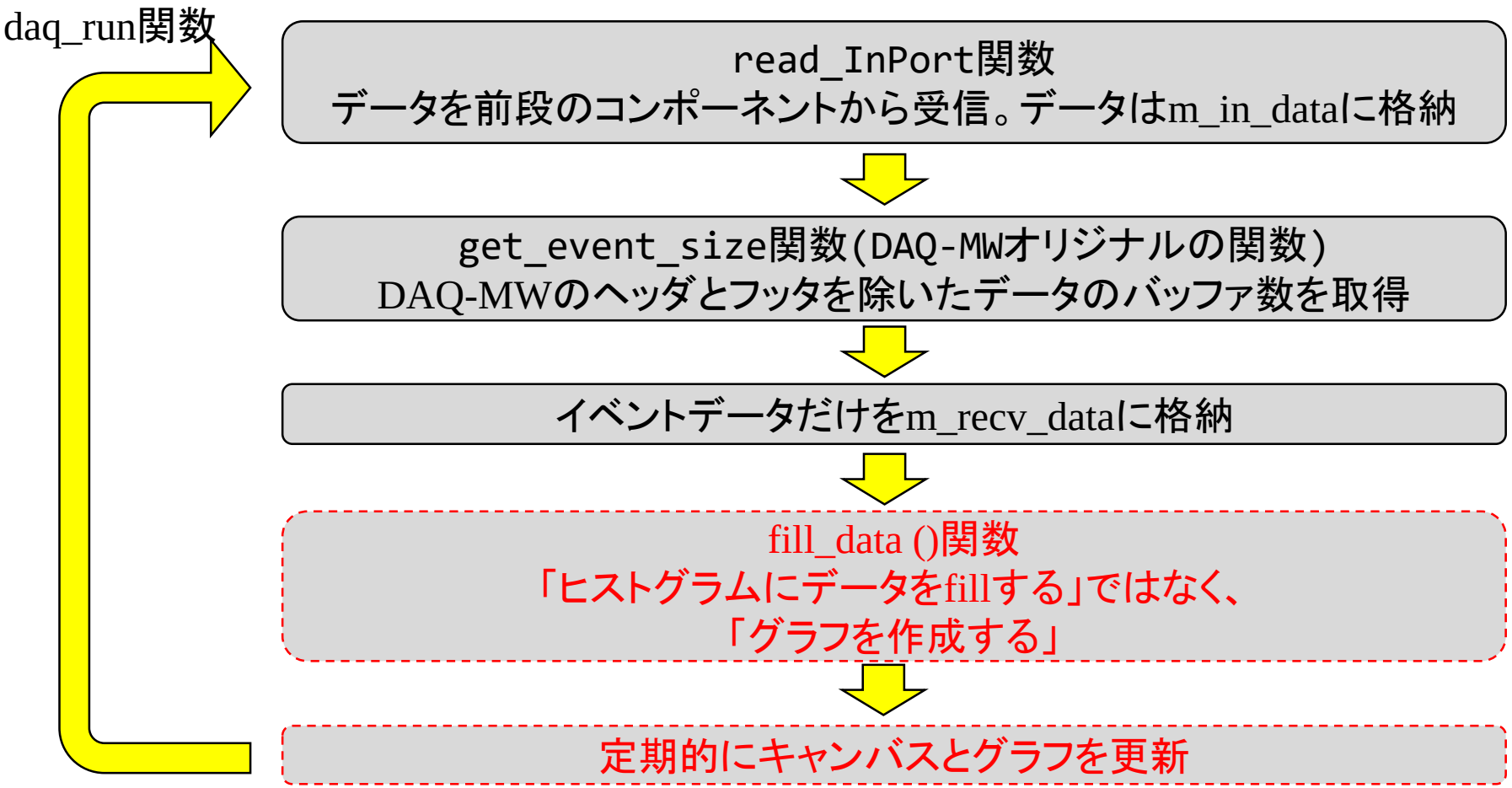


4つに分割

Tgraphの作成
（ROOT特有の書き方）

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp daq_run」

この章の課題では、主にfill_data関数を編集する



SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp daq_run」

```
int RawDataMonitor::daq_run()
{
(省略)
    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) {
         for (int i = 0; i < N_GRAPH; i++) {
             m_canvas->cd(i + 1);
             m_graph[i]->Draw("AC*");
         }
         m_canvas->Update();
     }
```

m_recv_data(1イベント分のデータを格納)をデコードし、グラフを作成する関数

「SampleMonitorとは違う箇所」
キャンバスとグラフの更新

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp fill_data」

データをデコードし、グラフを作成する

```
int RawDataMonitor::fill_data(const unsigned char* mydata, const int size)
{
```

この部分を自分で作る
mydataをデコードし、データをunsigned short型のdata[n_ch][window_size]に格納する。
window数をint型のwindow_sizeに代入する。
必要なら昨日の講義で作成したRawDataPacketクラスを利用すること。

```
for (int i = 0; i < N_GRAPH; i++) {
        for (int w = 0; w < window_size; w++) {
            m_graph[i]->SetPoint(w, w, data[i][w]);
        }
        m_graph[i]->SetMinimum(0.0);
        m_graph[i]->SetMaximum(5000.0);
        m_graph[i]->SetTitle(Form("CH: %d Trigger: %d", i, trigger_count));
    }
```

N_GRAPHはch数
（ヘッダファイルで定義）

得たデータをグラフにSetPoint
（グラフの点を作る）

必要ならこの部分も修正が必要。

```
   return 0;
}
```

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp daq_stop」

daq_stopはweb上に書いてある。コピーして利用すること。

```
int RawDataMonitor::daq_stop()
{
    std::cerr << "*** RawDataMonitor::stop" << std::endl;

    for (int i = 0; i < N_GRAPH; i++) {
        m_graph[i]->Draw();
    }
    m_canvas->Update();

    reset_InPort();

    return 0;
}
```

キャンバスとグラフの更新

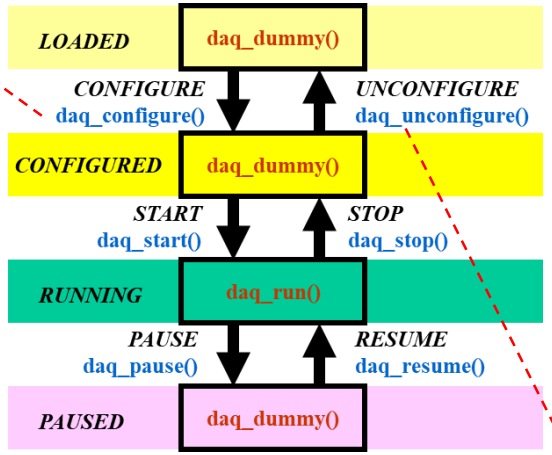
SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp daq_unconfigure」

daq_unconfigureはweb上に書いてある。コピーして利用すること。

```
int RawDataMonitor::daq_unconfigure()
{
    std::cerr << "*** RawDataMonitor::unconfigure" << std::endl;
    if (m_canvas) {
        delete m_canvas;
        m_canvas = 0;
    }
    for (int i = 0; i < N_GRAPH; i++) {
        if (m_graph[i]) {
            delete m_graph[i];
            m_graph[i] = 0;
        }
    }
    return 0;
}
```

キャンバスとグラフの削除

daq_startでキャンバスとグラフをnewで作っていた。



daq_startでキャンバスとグラフをdeleteしないといけない。

SampleMonitorからRawDataMonitorを作るためには「RawDataMonitor.cpp daq_unconfigure」

使わなくなったdecode_data関数は削除すること
（残したままだとMakeできない）

```
int RawDataMonitor::decode_data(const unsigned char* mydata)
{
    m_sampleData.magic       = mydata[0];
 （省略）


}
```

ex08 DAQ-Middlewareで モニターコンポーネントを開発する　解答例

RawDataMonitor.cpp

RawDataPackerクラスを利用し、デコードを行っている

```
int RawDataMonitor::fill_data(const unsigned char* mydata, const int size)
{
    rdp.set_buf(mydata, size);
    int window_size   = rdp.get_window_size();
    int n_ch          = rdp.get_num_of_ch();
    int trigger_count = rdp.get_trigger_count();

    unsigned short data[n_ch][window_size];

    for (int w = 0; w < window_size; w++) {
        for (int ch = 0; ch < n_ch; ch ++) {
            data[ch][w] = rdp.get_data_at(ch, w);
        }
    }
```

rdpのm_bufにmydata（=1イベント分のデータ）をセット

window数の取得

ch数の取得

各window、chのデータを取得

（続く）

ex08 DAQ-Middlewareで モニターコンポーネントを開発する　解答例

（続き）

```
    for (int i = 0; i < N_GRAPH; i++) {
        for (int w = 0; w < window_size; w++) {
            m_graph[i]->SetPoint(w, w, data[i][w]);
        }
        m_graph[i]->SetMinimum(0.0);
        m_graph[i]->SetMaximum(5000.0);
        m_graph[i]->SetTitle(Form("CH: %d Trigger: %d", i, trigger_count));
    }

    rdp.reset_buf();

    return 0;
}
```

得たデータをグラフに SetPoint

次のイベントに備え、リセットする

ex08 DAQ-Middlewareで モニターコンポーネントを開発する　解答例

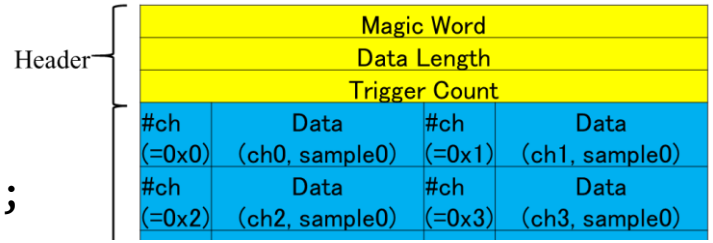
RawDataPacket.cpp

```
int RawDataPacket::set_buf(const unsigned char *buf, const int buf_len)
{
    m_buf = buf;
    m_buf_len = buf_len;

    return 0;
}

int RawDataPacket::get_trigger_count()
{
    unsigned int *trig;
    int rv;
    trig = (unsigned int *)&m_buf[TRIGGER_POS];
    rv = ntohl(*trig);
    return rv;
}
```

rdpのm_bufにmydata（=1イベント分のデータ）をセット

トリガーカウントを取得

TRIGGER_POS = 8



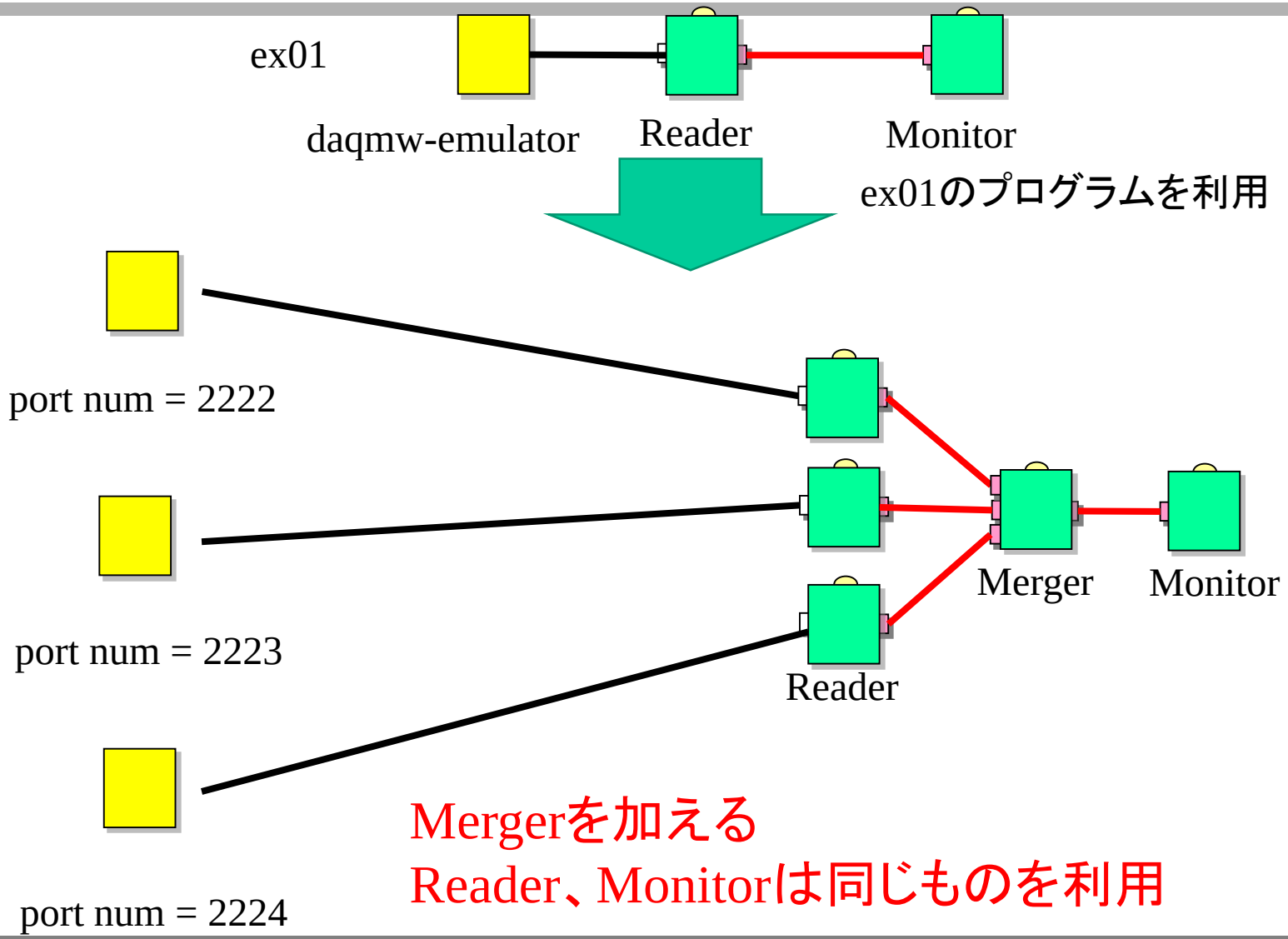
ex08 DAQ-Middlewareで モニターコンポーネントを開発する　解答例

解答のプログラムは

daqmw-tc2/daqmw-tc2/daqmw/RawDataMonitor

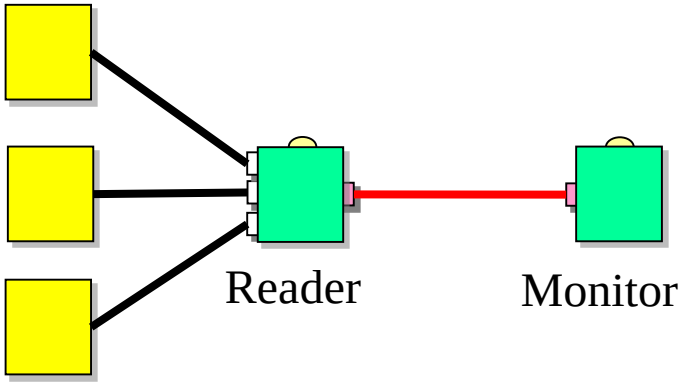
に置いてある。

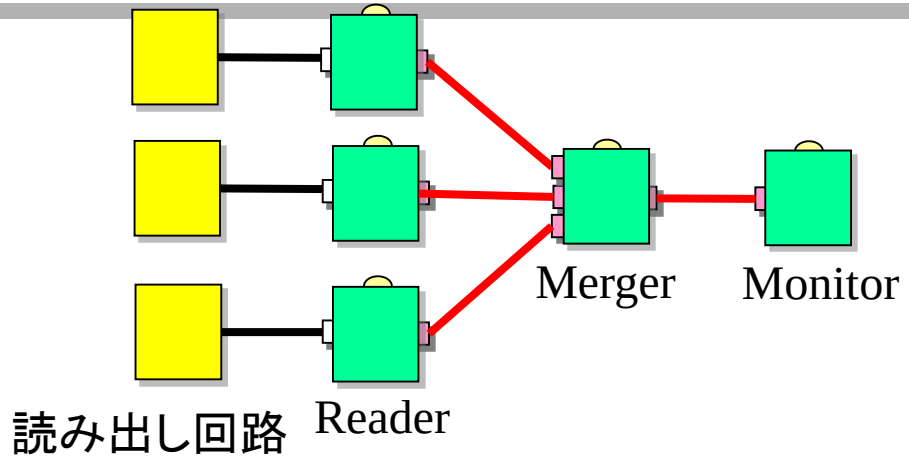
ex09 Mergerを利用して複数台のネットワークノードからデータを収集する



Mergerを加える
Reader、Monitorは同じものを利用

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

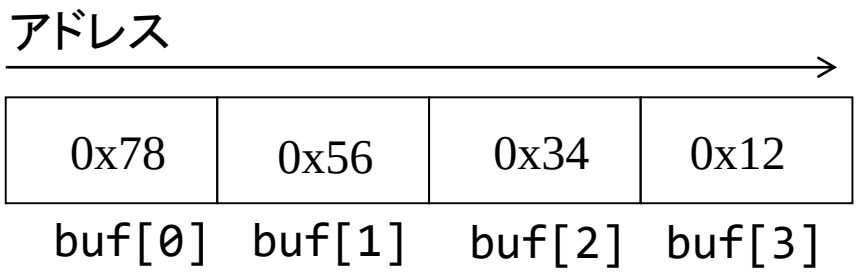
（メリット）
- Readerは全て1台の読み出しなので簡単に作れる。
- Readerの負荷を分散できる

（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

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）

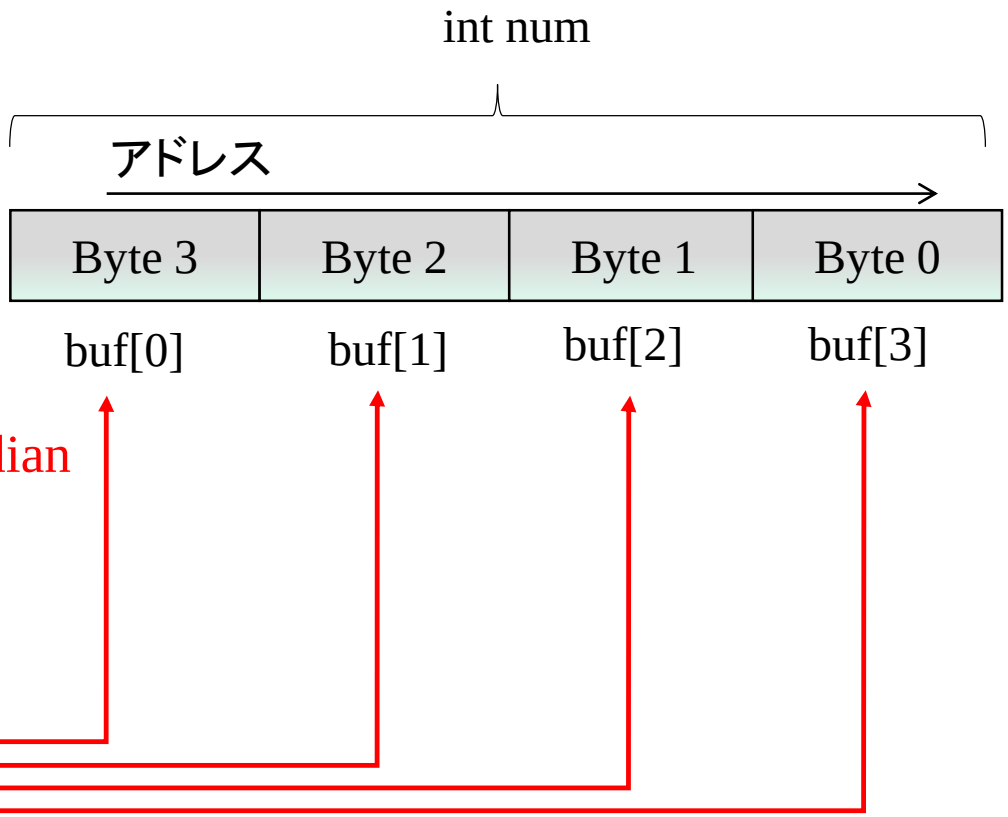
ネットワークバイトオーダーはbig endian

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];
};
```



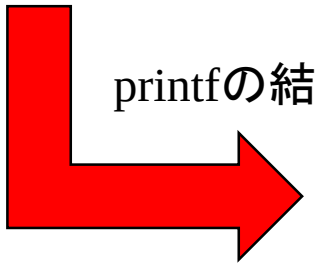
byte_order.cpp (一部)

```
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
```

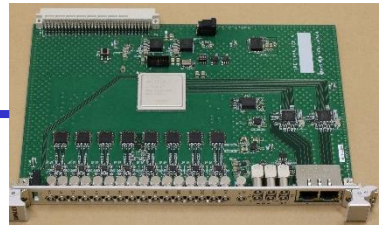
※アドレス値は環境によって異なるが、必ず+1されていく

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



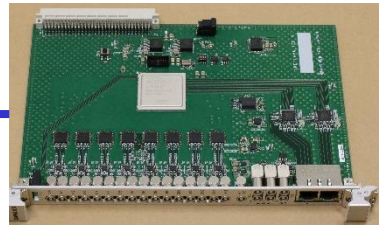
ホストオーダー:
リトルエンディアン

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