

DAQ-Middleware 1.4.0 開発マニュアル

千代浩司

高エネルギー加速器研究機構

素粒子原子核研究所

\$Date: 2015/02/26 02:05:14 \$

概要

これは DAQ-Middleware 開発用マニュアルです。次の事項を解説します。

1. DAQ-Middleware 1.4.0 開発環境の準備方法
2. DAQ-Middleware 1.4.0 開発環境の使い方
3. サンプルコンポーネントの作成と起動方法

DAQ-Middleware 1.4.0 で実装されている事項については「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

このテキストで作製するサンプルコンポーネントはデータをリードアウトモジュールから読み取る SampleReader コンポーネント、および SampleReader コンポーネントからデータを受け取ってヒストグラムを画面に表示する SampleMonitor コンポーネントです。SampleReader は、とくにハードウェアを必要としないようにするためソフトウェアエミュレータからデータを読むことにしました。

このマニュアルで前提とするプログラミング技能は以下のとおりです。

1. C および C++ 言語でプログラムが書けること。
2. gcc, make など開発ツールが使えること。
3. ネットワーク機器からデータを読みとるソケットプログラミングができること。

目次

1	このマニュアルについて	5
2	開発環境の準備	6
2.1	VMware Player を使う場合	6
2.2	Scientific Linux 5.x、6.x、7.x に RPM バイナリをインストールする方法	7
2.3	ソースからインストールする方法	9
2.4	インストールの確認	9
2.5	インストール後のディレクトリ構造	10
3	DAQ-Middleware の概要	13
3.1	コンポーネントの構成	14
3.2	コンポーネントの状態、および遷移	15
3.3	コンポーネント間を流れるデータフォーマット	16
3.4	InPort、OutPort のデータの読み書き	17
3.5	エラーが起きたときの処理	18
4	コンポーネント開発環境	19
4.1	newcomp コマンド	19
4.2	Makefile の書き方	22
5	開発ディレクトリの準備	25
6	Skeleton コンポーネントによる状態遷移の確認	25
7	単純なコンポーネントの作成例	29
8	この文書で開発するデータ収集システムの概要	33
9	ソフトウェアエミュレータ	33
9.1	セットアップ	33
9.2	起動	34
9.3	ソフトウェアエミュレータのデータフォーマット	34
9.4	エミュレータからのデータの確認	34
10	SampleReader コンポーネントの開発	35
10.1	SampleReader.h の変更	37

目次

10.2	SampleReader.cpp の変更	39
10.3	Makefile の変更	46
11	SampleMonitor コンポーネントの開発	46
11.1	SampleData.h の作成	47
11.2	SampleMonitor.h の変更	48
11.3	SampleMonitor.cpp の変更	49
11.4	SampleMonitorComp.cpp の変更	54
11.5	Makefile の変更	55
12	起動および動作確認	55
13	パラメータの Condition データベース化	60
13.1	Condition データベースを使ったヒストグラムのテスト	64
14	WebUI の使い方	67
14.1	ソフトウェアパッケージの確認	67
14.2	操作方法	68
付録 A	DAQ-Middleware で提供しているライブラリについて	71
付録 B	この解説書の変更履歴	73
付録 C	rpm および yum コマンドを使用してセットアップしたときのログ	74
付録 D	ソースからインストールする場合のヒント	77
付録 E	リモートブートセットアップ	79
E.1	ネットワーク疎通の確認	79
E.2	xinetd のインストール	80
E.3	xinetd の設定	80
E.4	コンフィギュレーションファイルの作成	81
E.5	システムの起動	81
E.6	リモートブートのしくみ	83
付録 F	Scientific Linux インストール方法	85
F.1	新規にインストールする場合 (SL 5.x)	85
F.2	開発環境をあとから追加する場合 (SL 5.x)	86
F.3	新規にインストールする場合 (SL 6.x)	87
F.4	開発環境をあとから追加する場合 (SL 6.x)	87

F.5	新規にインストールする場合 (SL 7.x)	88
F.6	開発環境をあとから追加する場合 (SL 7.x)	89
F.7	SL 5.x, 6.x での SELinux, iptables の設定	90
F.8	SL 7.x での SELinux, パケットフィルターの設定	90
付録 G	コマンドラインから DAQ オペレータに指示を出す	92
付録 H	InPort リングバッファのパラメータの設定	93

1 このマニュアルについて

このマニュアルは以下のような構成になっています。

- DAQ-Middleware 1.4.0 開発環境の準備方法 (第 2 節)
- DAQ-Middleware のごく簡単な概要 (第 3 節)
- DAQ-Middleware 1.4.0 開発環境の使い方 (第 4 節)
- 開発ディレクトリの準備 (第 5 節)
- Skeleton コンポーネントによる状態遷移の確認 (第 6 節)
- 簡単なコンポーネントによるコンポーネント間データ転送 (第 7 節)
- サンプルコンポーネントの作成と起動方法 (第 8 節以降)
- Condition データベースを使ったパラメータの変更 (第 13 節)

このマニュアルではコンポーネントを開発する方法について解説します。データ収集システムへ配備する方法についての解説はありません。

DAQ-Middleware 1.4.0 の設計と実装、使用できるクラス、メソッドについては「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

このマニュアルで解説する DAQ コンポーネントのソースコードについて

このマニュアルで解説するソースコードは DAQ-Middleware 1.4.0 をインストールすると `/usr/share/daqmw/examples` ディレクトリ以下に入ります。手で入力するのが大変である場合とか、このマニュアルのソースコード説明で抜けているのではないと思われる部分はこのディレクトリ以下のソースコードを参照してください。

このマニュアルのバージョン

このマニュアルの版は、表紙の `Date :` をみて区別してください。

Scientific Linux 5.x、6.x、7.x への DAQ-Middleware のセットアップ

Scientific Linux 5、6、7 上へのセットアップは 2.2 節で解説する `yum` コマンドで行うのが簡単です。

DAQ-Middleware のソースについて

使用する OS のバイナリが提供されていない、バージョンにあっていない、DAQ-Middleware に付属するライブラリを変更したい等の理由により DAQ-Middleware のソースを取得したい場合は第 2.3 節の URL からダウンロードしてください。

2 開発環境の準備

現在のところ DAQ-Middleware 1.4.0 の開発環境を準備する方法には

1. DAQ-Middleware 開発グループが用意した VMware Player イメージを利用する
2. Scientific Linux 5.x、6.x、7.x へ RPM バイナリをインストールする (SL 5.x、6.x 用は i686 (32 ビット)、x86_64 (64 ビット) 両方のバイナリを用意しています。SL 7.x は SL 7.x 自体が x86_64 用 (64 ビット) のものしかリリースされていないので x86_64(64 ビット) のみのバイナリを用意しています)。
3. 自力で依存物をセットアップしソースからインストールする

の三つの方法があります。以下この順番で説明します。

2.1 VMware Player を使う場合

この解説書で必要なソフトウェアのインストール、設定を済ませた VMware Player で使える Scientific Linux 5.8 ディスクイメージを用意しました。このイメージを使えばすぐに DAQ-Middleware コンポーネントの作成作業にとりかかることができます。

VMware Player は VMware のサイト <http://www.vmware.com/jp/products/player/> からダウンロードしてください。最新版の VMware Player 3.x は使用する CPU によってはインストールすることができません。VMware Player 3.x を使用するためには CPU が CMOV、PAE、TSC、FXSAVE 命令をサポートしている必要があります。最近の CPU のほとんどはこの命令をサポートしていると思われます。CPU として Pentium M を使用した計算機には PAE がなくて 3.x はインストールできませんでした。3.x をインストールできない場合は <http://www.vmware.com/download/player/download.html> から VMware Player 2.5.4 をダウンロードしてください。

VMware Player で使える Scientific Linux 5.8 ディスクイメージは <http://daqmw.kek.jp/vmplayer/sl-55-daqmw.zip> からダウンロードしてください。一般ユーザーとしてユーザー名 daq、パスワード abcd1234 が登録されています。また root のパスワードは abcd1234 です。

このイメージではメモリーとして 1024MB を指定してあります。この解説書で作るコンポーネントシステムでは支障がないと思いますが、もっとメモリーを必要とする場合は VMware Player アイコンをクリックしたあと出てくる画面で、当該イメージを選択し、右側「仮想マシン設定の編集」を選び、ハードウェア メモリを選択し調節してください。

この解説書で開発するモニターコンポーネントのヒストグラムの作成では ROOT (<http://root.cern.ch/>) を使用していますがこれは /usr/local/root 以下に入っています。daq ユーザーでログインすると環境変数 ROOTSYS の値として /usr/local/root となるように設定しています。

2 開発環境の準備

この VMware Player イメージは Scientific Linux 5.8 のイメージを作成し、そこに次節で述べる RPM を使ってバイナリをインストールして作成したものです。

2.2 Scientific Linux 5.x、6.x、7.x に RPM バイナリをインストールする方法

Scientific Linux は RedHat Enterprise Linux を元に作られた Linux ディストリビューションです。詳細は <http://www.scientificlinux.org/> を見てください。Scientific Linux 5.x, 6.x, 7.x のセットアップ方法については付録 F をご覧ください。

2.2.1 DAQ-Middleware 2008.10 ~ 2009.10 版を使用していた人向けの注意

DAQ-Middleware 1.0.0 から rpm の配布 URL が変わりました。アップデートの際には、次節のコマンドを使ってセットアップを行う前に古い環境を削除してください。削除するには <http://daqmw.kek.jp/src/daqmw-rpm> にあるファイルをダウンロードして root ユーザで

```
# chmod +x daqmw-rpm
# ./daqmw-rpm distclean
```

としてください。chmod しないで `sh daqmw-rpm distclean` でもかまいません。

(註) このファイルはシェルスクリプトで、`daqmw-rpm distclean` で以下のコマンドを順に実行しています。

```
rpm -e kek-daqmiddleware-repo
rpm -e OpenRTM-aist
rm -fr /var/cache/yum/kek-daqmiddleware
```

2.2.2 セットアップ方法

Scientific Linux 5.x、6.x、7.x 環境下に DAQ-Middleware 開発グループが用意した RPM バイナリをインストールするには <http://daqmw.kek.jp/src/daqmw-rpm> にあるファイルをダウンロードして root ユーザーで以下のコマンドを実行してください。

```
root# chmod +x daqmw-rpm
root# ./daqmw-rpm install
```

chmod するかわりに `sh daqmw-rpm install` でもかまいません。

(註) このファイルはシェルスクリプトで `daqmw-rpm install` で以下のコマンドを順に実行しています。

```
root# rpm -ihv http://daqmw.kek.jp/rpm/el5/noarch/kek-daqmiddleware-repo-2-0.noarch.rpm
root# yum --disablerepo='*' --enablerepo=kek-daqmiddleware install DAQ-Middleware
```

このコマンドを実行した際のログを付録 C にのせておきます。

この yum コマンドでインストールされる rpm パッケージは以下のとおりです。32 ビット用 rpm パッケージのダウンロードファイルサイズは 22MB、64 ビット用 rpm パッケージのダウンロードファイルサイズは 23MB、インストール後のファイルサイズの合計は 32 ビット用で 78MB、64 ビット用で 95MB です。

- DAQ-Middleware-1.4.0
- OpenRTM-aist-1.0.0 (+ まだリリースされていないパッチ)
- OmniORB サーバー、ライブラリ、開発環境
 - omniORB-doc-4.1.6
 - omniORB-servers-4.1.6
 - omniORB-utils-4.1.6
 - omniORB-devel-4.1.6
 - omniORB-4.1.6
- SL 5.x では xerces-c-2.7.0 および xerces-c-devel-2.7.0、SL 6.x では xerces-c-3.0.1 および xerces-c-devel-3.0.1
- xalan-c-1.10.0 および xalan-c-devel-1.10.0

モニターコンポーネントの作成でヒストグラムに必要なソフトウェア (ROOT など) がある場合は別途インストールする必要があります。また 13 節で解説する Condition データベースを使用するには

- boost

パッケージが必要です。boost は Scientific Linux 5.x、6.x、7.x の配布物に含まれていますので yum コマンド等でインストールしてください。

2.2.3 yum コマンドを使ったアップデート方法

バグフィックス、例題追加などの理由で新しいバージョンの DAQ-Middleware が出た場合には次のコマンドでアップデートすることができます (新規インストールする場合と同一のコマンドです)。

```
wget http://daqmw.kek.jp/src/daqmw-rpm
chmod +x daqmw-rpm
./daqmw-rpm install
```

2.2.4 アンインストールする方法

アンインストールするには以下のコマンドを実行します。

```
# ./daqmw-rpm distclean
```


2 開発環境の準備

ひとつずつ消していくには、たとえば

```
# rpm -qa --last | less
```

とするとインストールした rpm パッケージ名が、時間順で表示されます (最近のものが最初に表示されます) ので、これを見て rpm -e コマンドで依存物のほうから順番に消していくこともできます。

2.3 ソースからインストールする方法

DAQ-Middleware のソースは <http://daqmw.kek.jp/src/> にあります。ファイル名は DAQ-Middleware-M.m.p.tar.gz 形式になっていて M、m、p には数が入ります。

ソースからインストールするにはこれをダウンロードして展開し、make; make install する必要があります。コンパイルには OpenRTM-aist が必要で、また動作させるには OmniORB が必要です。依存物を用意するのは大変ですから RPM あるいは yum でセットアップできる場合はこれらを使ってバイナリファイルをダウンロードしインストールするのが簡単でおすすめです。

Scientific Linux、CentOS、RedHat Enterprise Linux (5.x、6.x、7.x) 以外で使いたい場合のコンパイルのヒントを付録 D にまとめてあります。

2.4 インストールの確認

自力でインストール、セットアップした場合は以下の方法で開発環境が正常にセットアップできたかどうか確認することができます。

Skeleton コンポーネントを使って、開発環境が整っているか確認します。Skeleton コンポーネントは DAQ-Middleware 1.4.0 のセットアップが済んでいれば /usr/share/daqmw/examples/Skeleton/ ディレクトリ以下にソースがインストールされています。

以下の要領で開発環境が整っているか確認します。

```
% cp -r /usr/share/daqmw/examples/Skeleton . (最後にドット(".")があります)
% cd Skeleton
% ls
Makefile Skeleton.cpp Skeleton.h SkeletonComp.cpp
% make
```

正常ですと SkeletonComp という実行形式ファイルができます。make コマンドがエラーとなって異常終了した場合は、原因を追求し解決しておく必要があります。解決方法はエラーの内容によります*1。

*1 たとえば Scientific Linux 5.x 上で libuuid 関連でコンパイルが失敗する場合は Scientific Linux 5.x の配布パッケージから e2fsprogs-devel パッケージをインストールする必要があります。

2.5 インストール後のディレクトリ構造

DAQ-Middleware をインストールしたあとにどのようなファイルがインストールされたか調べるには RPM からインストールした場合、および VMware Player を使っている場合は

```
rpm -ql DAQ-Middleware
```

で調べることができます。ソースからインストールした場合にはこのような手軽な方法はありません。

VMware Player を使う場合も、native Linux 環境で rpm を使ってセットアップを行った場合も DAQ-Middleware のディレクトリ構造は以下のようになっています。

/usr/bin

ユーザが手でコマンドを起動するコマンド類がここに入っています。以下のコマンドが入っています。各コマンドの具体的な利用方法についてはのちほど必要になったところで説明します。

run.py, run.pyc, run.pyo

システムコンフィギュレーションファイルを読んでそれに書かれているコンポーネントを起動し、最後に DaqOperator を起動するコマンドです。Python で書かれています。run.pyc, run.pyo はそれぞれ run.py のバイトコンパイルファイル、およびオプティマイズされたバイトコンパイルファイルです。

daqmw-rpm

RPM で DAQ-Middleware をセットアップするときのユーティリティです。使用方法については第 2.2.1 節、第 2.2.2 節をご覧ください。なお rpm -e DAQ-Middleware で DAQ-Middleware のアンインストールを実行するとこのコマンドも消えます。再セットアップ等でこのコマンドが必要な場合は <http://daqmw.kek.jp/src/daqmw-rpm> からダウンロードすることができます。

condition_xml2json および xml2json-with-attribute.xslt

DAQ-Middleware ではラン毎に変わるコンポーネントで使用するパラメータは Condition データベースファイルに記述します。記述は XML で行います。各コンポーネントはこのデータベースを読んでパラメータを取得します。実際には、XML そのものを解析するには時間がかかるので XML で書かれたものを JSON フォーマットに変換したものを作り、各コンポーネントはこの JSON フォーマットのものを読みます。XML から JSON フォーマットの変換にはこの condition_xml2json を使います。xml2json-with-attribute.xslt には変換に必要な XSL スタイルシートです。

newcomp

新規コンポーネントを作成するときに必要なファイルのテンプレートを作成するコマンドです。第 4.1 節で詳しく解説しています。

/usr/include/daqmw

このディレクトリに DAQ-Middleware の基本となるクラスファイル等が入っています。また下記ライブラリの API インクルードファイルもあります。

/usr/include/daqmw/idl

IDL ファイル置場。

/usr/lib/daqmw (32 ビット) あるいは /usr/lib64/daqmw (64 ビット)

DAQ コンポーネントを作成するうえで必ず使う (であろう) ファイルが入っています。現在はソケットライブラリおよびコンディション関連のライブラリ (json_spirit) が入っています。コンポーネントをコンパイルする Makefile 中で下記 /usr/share/daqmw/mk/ にある comp.mk を使うとこのディレクトリは DAQMW_LIB_DIR として指定できます。この機能を使うと 32 ビットおよび 64 ビットでコンポーネントソースを共有することができます。たとえば

```
LDLIBS += -L$(DAQMW_LIB_DIR) -lSock
```

などとして使います。comp.mk については 4.2 節を見てください。

このディレクトリにインストールされるライブラリ、および使用上の注意について付録 A にまとめてありますのでご覧になってください。

/usr/libexec/daqmw/DaqOperator

DAQ オペレータコンポーネントの実行形式ファイルです。ソースは /usr/share/daqmw/DaqOperator/ 以下にあります。

/usr/share/daqmw/conf

DAQ-Middleware の設定ファイル (コンフィギュレーションファイル、およびコンディションファイル) の雛型がこのディレクトリにあります。この文書で SampleReader および SampleMonitor を動かすときに使用するコンフィギュレーションファイルはこのディレクトリの下に sample.xml という名前が入っています。実行ファイルの配置場所によっては sample.xml はそのままでは使えません。どう変更するかは第 12 節を見てください。

/usr/share/daqmw/DaqOperator

DAQ オペレータコンポーネントのソースコード一式が入っています。DAQ システムに必要なコンポーネントのソースをひとつのディレクトリにまとめたい場合には DAQ オペレータコンポーネントについてはこのディレクトリからソースをコピーしてください。

/usr/share/daqmw/docs

DAQ-Middleware のドキュメントがこのディレクトリにあります。

`/usr/share/daqmw/etc`

複数の計算機を使って DAQ システムを構築するさいにはリモートブート機能が必要になります。xinetd を使ってリモートブートを実現する場合の雛型ファイルがこのディレクトリに入っています。リモートブートのセットアップについては付録 E を御覧ください。

`/usr/share/daqmw/examples`

例題コンポーネントがこの下のディレクトリに、まとめられています。このディレクトリの下にはこの文書で開発するコンポーネントのソース (SampleReader および SampleMonitor) がこの名前が入っています。

`/usr/share/daqmw/mk`

Makefile の記述を簡略化するための定型コマンドがこの下の `comp.mk` にまとめられています。Makefile の書き方については 4.2 節を見てください。

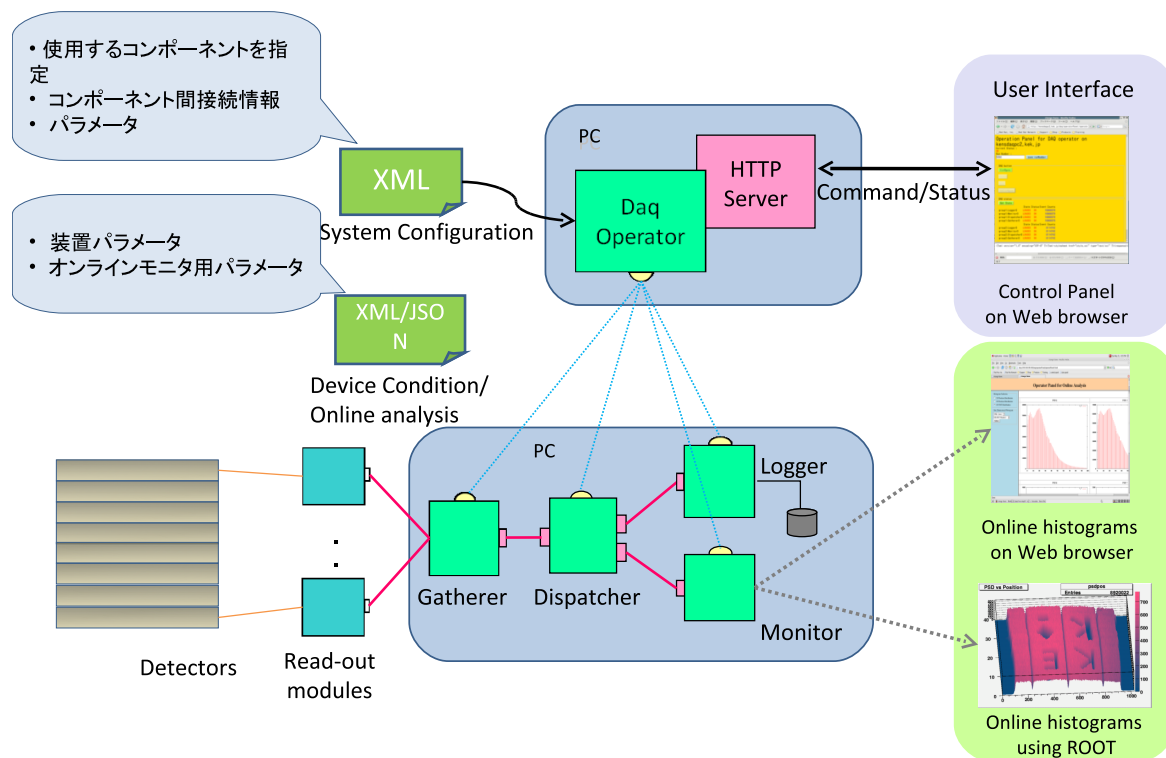


図 1 DAQ-Middleware overview

3 DAQ-Middleware の概要

DAQ-Middleware のアーキテクチャ、コンポーネントの仕様、コンポーネント間を流れるデータのフォーマット等については「DAQ-Middleware 1.1.0 技術解説書」[2] に書かれています。コンポーネント開発を始めるまえに読んでおいてください。この節ではコンポーネントのコードを書く上で知っておかなければならないことがらをまとめておきます。

図 1 に DAQ-Middleware 構成概要図を示します。DAQ-Middleware では複数の DAQ コンポーネント (以下では単にコンポーネントといいます) でデータ収集を行います。コンポーネントを統括するのが DaqOperator です。DaqOperator が各コンポーネントに接続、データ収集開始、終了等のコマンドを送ってランコントロールを行います。DaqOperator は XML で書かれたシステムコンフィギュレーションファイルを読んで、どのようなコンポーネントがあるのか、どのコンポーネントとどのコンポーネントを接続するのか等のシステム情報を把握します。DaqOperator に対する指示は他の上位システム (フレームワークなど) が行います。DaqOperator にはふたつのモードがあります。ひとつは DaqOperator が HTTP で指示を受け取るウェブモード、もうひとつはキーボード入力から指示を受けるコンソールモードです。この開発マニュアルではコンソールモードで DaqOperator に指示をだします。ラン毎に変わるパラメータは XML でコンディションファイルとして記述します。XML のパースは重い処理になるので XML で書かれたファイルを JSON

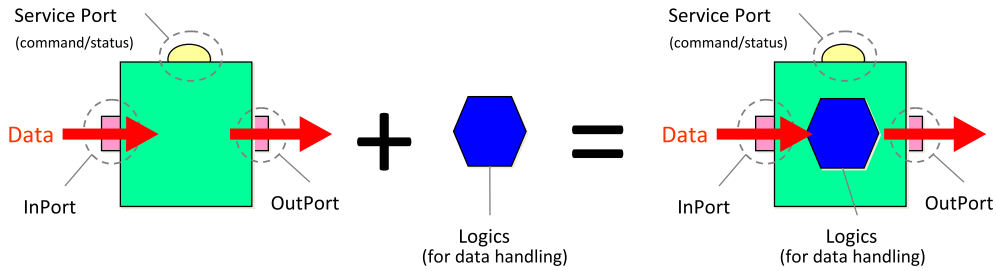


図 2 ロジック実装

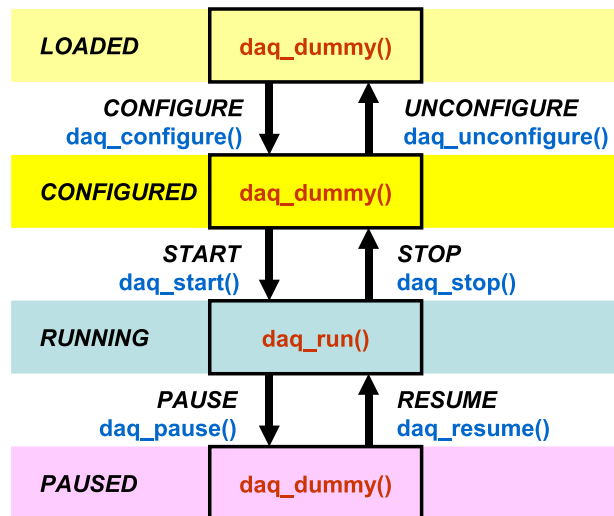


図 3 ステートチャート

フォーマットに変換しておき、各コンポーネントが JSON フォーマットで書かれたファイルを読みパラメータを取得します。

3.1 コンポーネントの構成

図 2 はコンポーネントの構成を表したものです。DAQ-Middleware ではデータ収集は複数のコンポーネントが通信して行います。図 2 左辺第 1 項は DAQ-Middleware で既に提供されているパーツを表しています。DAQ-Middleware ではデータの受け取りに InPort、送り出しに OutPort を使用します。InPort には上流コンポーネントから送られてきたデータが入ります。OutPort には下流コンポーネントに送りたいデータを書きます。異なるコンポーネント間の OutPort — InPort の通信機能は DAQ-Middleware が提供します。またコンポーネントには ServicePort があり、これは DaqOperator からのランコントロール命令の受信、ステータス情報の送信等に使われます。

コンポーネント開発者は自身が実現したいロジックをプログラミングします。たとえばリードアウトモジュールからデータを読み取って後段コンポーネントにデータを送るコンポーネントのロ

3 DAQ-MIDDLEWARE の概要

関数名	状態遷移/状態	プログラミング例
daq_configure()	LOADED → CONFIGURED	DaqOperator からのパラメータリスト (リードアウトモジュールの IP アドレス、ポート番号など) を受け取り値をセットする。
daq_start()	CONFIGURED → RUNNING	ソケットを作成し、リードアウトモジュールに connect する。
daq_run()	RUNNING	リードアウトモジュールからデータを読む。後段コンポーネントにデータを送る。
daq_stop()	RUNNING → CONFIGURED	リードアウトモジュールから disconnect する。

表 1 Gatherer のプログラミング例

関数名	状態遷移/状態	プログラミング例
daq_configure()	LOADED → CONFIGURED	DaqOperator からのパラメータリストを受け取り値をセットする。
daq_start()	CONFIGURED → RUNNING	ヒストグラムデータ作成。
daq_run()	RUNNING	上流コンポーネントからデータを読みデコードしてヒストグラムデータにフィルする。定期的にヒストグラム図を書く。
daq_stop()	RUNNING → CONFIGURED	最終ヒストグラムデータを使ってヒストグラム図を書く。

表 2 Monitor のプログラミング例

ジックはリードアウトモジュールからソケットを使ってデータを読み、アウトポートヘデータを書くというものです。アウトポートヘデータを書くとデータは自動的に後段コンポーネントに送られ、コンポーネント開発者はこの部分を自分で実装する必要はありません。この他にロジックの例として、InPort にきたデータを読んでヒストグラムを書くなどがあります。

3.2 コンポーネントの状態、および遷移

各コンポーネントは起動している間、図 3 にあるステートチャートの状態のうちどれかの状態にあります。状態としては LOADED、CONFIGURED、RUNNING、PAUSED が定義されています。たとえば実行形式ファイルが計算機にロードされ、コンポーネントがプロセスとして走り出した直後の状態は LOADED です。DaqOperator から CONFIGURE の指示がくるまで LOADED の状態を保ちます。CONFIGURE の指示がくると CONFIGURED の状態に遷移します。遷移

するさいに、その遷移で定義されている関数が一度だけ実行されます。たとえば LOADED から CONFIGURED に遷移するときには `daq_configure()` が実行されます。

ひとつの状態にある間、その状態に対応する関数が何度も呼ばれます。たとえば RUNNING の状態にあるときには `daq_run()` が呼ばれます。1 回の `daq_run()` 終了後、STOP コマンドがきていなければもう一度 `daq_run()` が呼ばれます。以下 STOP コマンドがくるまで `daq_run()` は何度も呼ばれます。各状態にある間、呼ばれる関数は、その関数のなかで永遠にブロックすることがないようにプログラミングする必要があります。永遠にブロックすると次の状態に遷移するコマンドが発行されてもそれを受け取ることができなくなるからです。たとえば `daq_run()` でソケットを使ってデータを読む場合にはソケットにタイムアウトを付けて永遠にブロックすることはないようにする必要があります*²。

コンポーネントはこれらの関数を実装することにより実現します。例としてリードアウトモジュールからのデータ読みだしを行うコンポーネント (`gatherer`) で実装することがらを表 1 に示します。また、ヒストグラムを作成し画面に表示するコンポーネント (`monitor`) の例を表 2 に示します。これはあくまで例であり必ずしもこれらを実装しなければならないということではありません。

3.3 コンポーネント間を流れるデータフォーマット

図 4 に各コンポーネント間を流れるデータフォーマットを示します。この図にあるコンポーネントヘッダ、コンポーネントフッタはリードアウトモジュールが送ってくるデータ中 (にあるかもしれない) ヘッダ、フッタとは無関係です。コンポーネントヘッダ、コンポーネントフッタのフォーマットを図 5 に示します。

2 バイトある header magic には、両バイトとも `0xe7` が入ります。2 バイトある footer magic には、両バイトとも `0xcc` が入ります。

コンポーネントヘッダ、コンポーネントフッタにある合計 4 バイトの Reserved のバイトはユーザーの方々が自由に使ってよい領域です。今後 DAQ-Middleware フレームワークとしてこの領域を使うことはありません。たとえば複数のリードアウトモジュールを使うがイベントデータにモジュール番号をいれる場所がないがどのモジュールのデータか判別できるようにしたいときなどに利用します。

上流コンポーネントが下流コンポーネントにデータを送るときにはヘッダー中の `EventByteSize` に何バイトのイベントデータを送ろうとしたかを書いておきます。データを受け取った下流コンポーネントは実際に受け取ったイベントデータサイズとヘッダー中の `EventByteSize` を比べ読み落としがなかったかどうかを検証します。また上流コンポーネントは下流コンポーネントにデータを送るのが何回目であるかをフッター中の `sequence number` にセットします。データを受け取った下流コンポーネントは上流コンポーネントからデータを受け取った回数を記憶しておき、デー

*² ソケットの `read()` はデフォルトでは読むデータがない場合はそこで永遠にブロックします。

3 DAQ-MIDDLEWARE の概要

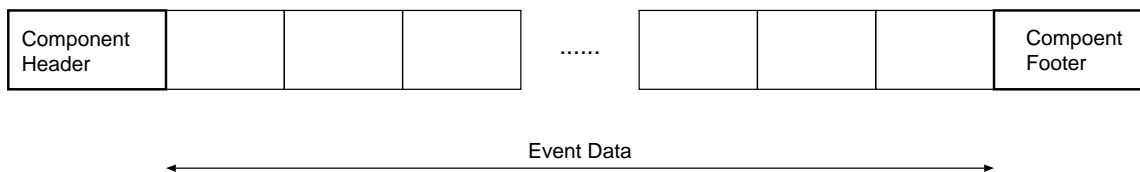


図 4 コンポーネント間を流れるデータのフォーマット。コンポーネントヘッダとフッタのフォーマットについては図 5 を参照。

Component Header

Header Magic (0xe7)	Header Magic (0xe7)	Reserved	Reserved	Data ByteSize (24:31)	Data ByteSize (16:23)	Data ByteSize (8:15)	Data ByteSize (0:7)
0	7 8	15 16	23 24	31 32	39 40	47 48	55 56 63

Component Footer

Footer Magic (0xcc)	Footer Magic (0xcc)	Reserved	Reserved	sequence number (24:31)	sequence number (16:23)	sequence number (8:15)	sequence number (0:7)
0	7 8	15 16	23 24	31 32	39 40	47 48	55 56 63

図 5 コンポーネントヘッダ、フッタフォーマット。

タを受け取ったらフッター中の sequence number と比較します。これによりデータを受け取れなかったことがなかったかどうかを検証することができます。

EventByteSize および sequence number のセット、ゲットに関するメソッドは set_event_byte_size(), inc_sequence_num(), get_sequence_num() などがあります。全てのメソッドは「DAQ-Middleware 1.1.0 技術解説書」[2] で解説されています。

イベントデータフォーマットはユーザーが策定します。

3.4 InPort、OutPort のデータの読み書き

InPort、OutPort をひとつずつ持つ Sample コンポーネントを例に InPort、OutPort のデータ読み書きを説明します。

Sample.h では以下のように InPort、OutPort およびそれぞれの Port で使用するバッファを定義します。

```
private:
    TimedOctetSeq      m_in_data; // InPort
    InPort<TimedOctetSeq> m_InPort;

    TimedOctetSeq      m_out_data; // OutPort
    OutPort<TimedOctetSeq> m_OutPort;
```

m_in_data および m_out_data は Sample.cpp のコンストラクタで InPort、OutPort のデータバッファになります。

```

Sample::Sample(RTC::Manager* manager)
: DAQMW::DaqComponentBase(manager),
  m_InPort("sample_in", m_in_data),
  m_OutPort("sample_out", m_out_data),

```

上流コンポーネントから InPort に到着したデータは `m_InPort.read()` で読みます。戻り値は `true` あるいは `false` で、`false` の場合は `check_inPort_status(m_InPort)` で InPort の状態を確認します。`check_inPort_status(m_InPort)` が `BUF_TIMEOUT` の場合は通常はリトライするようにコードを書きます。`BUF_FATAL` の場合は次節にある `fatal_error_report()` を使って `DaqOperator` に致命的エラーが起こったことを報告します。正常にデータが読めた場合は、データは `m_in_data.data` 配列に入ります。読んだ長さを取得するには `m_in_data.data.length()` メソッドを使います。

下流コンポーネントにデータを送るには OutPort にデータを書きます。まず `m_out_data.data.length(データ長)` で送るデータ長を指定します。データ長の単位はバイトです。次に、送るデータを `m_out_data.data` 配列に書きます。次に `m_OutPort.write()` を実行します。`m_OutPort.write()` の戻り値は `true` あるいは `false` で `true` の場合は正常にデータが送られたことを示します。`false` の場合は `check_outPort_status(m_OutPort)` を使って OutPort の状態を確認します。`check_outPort_status()` が `BUF_TIMEOUT` であった場合は通常リトライするようにコードを書きます。`BUF_FATAL` であった場合は次節の `fatal_error_report()` を使って致命的エラーが起きたことを `DaqOperator` に知らせます。

詳細については「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

3.5 エラーが起きたときの処理

コンポーネントで致命的エラーが起きた場合は `fatal_error_report()` を使って `DaqOperator` にエラーが起きたことを通知するようにします。`fatal_error_report()` の詳細は「DAQ-Middleware 1.1.0 技術解説書」[2] をごらんください。なにが致命的エラーかはコンポーネント開発者が決めます。致命的エラーが発生した場合の対処は上位システム、あるいは人間が行います。

4 コンポーネント開発環境

4.1 newcomp コマンド

コンポーネント開発作業を始めるにあたって 2.4 節のように Skeleton コンポーネントをコピーしてから、ファイルを書き換えるのもよいですが、実際にコンポーネントに適切な名前を付けることになると、インクルードガード名、コンポーネント名のように機械的に変更しなければならないところがあります。そこでこれらの作業を自動化するために newcomp というコマンドを用意しました。/usr/bin/newcomp にあります。このコマンドに開発するコンポーネント名を引数として指定すると、指定したコンポーネント名のディレクトリをカレントディレクトリに作成し、その下に以下のファイルを作ります (例として newcomp MyMonitor とした例を示します):

- Makefile
- MyMonitor.h
- MyMonitor.cpp
- MyMonitorComp.cpp

```
% newcomp MyMonitor
% ls
MyMonitor
% cd MyMonitor
% ls
Makefile  MyMonitorComp.cpp  MyMonitor.cpp  MyMonitor.h
```

ファイル名中 MyMonitor の部分は newcomp の引数で指定したコンポーネント名に置き換わります。

Makefile 中のインクルードガード名が MYMONITOR になっていたりコンポーネント名を定義するところが mymonitor になっている以外のロジックの中身は Skeleton コンポーネントと同一です。この状態で make できるようになっていますので一度 make コマンドを使って開発環境が正常にセットされているかどうか確認することができます。^{*3}

これらのファイルのうち MyMonitorComp.cpp はプログラムのみにて main() 関数になにか入れたい場合以外は変更する必要はありません。この文書で作成する SampleMonitor コンポーネントは、ヒストグラムを書くのに ROOT を使用しますが、このコンポーネントでは main() 関数で TApplication オブジェクトを生成するために SampleMonitorComp.cpp を変更します。

MyMonitor.cpp で daq_start()、daq_run() などのメソッドを実装してコンポーネントを作成します。

^{*3} make コマンドを実行すると autogen ディレクトリが作成されそこには自動生成されたファイルが入ります。コンポーネント開発では autogen ディレクトリのファイルは変更する必要はありません。

コンポーネント間のデータ流に着目し、他のコンポーネントへデータは送るが、他のコンポーネントからデータを受け取ることがないコンポーネントを Source 型コンポーネント (あるいは Source タイプコンポーネント) といいます。それとは逆に他のコンポーネントからデータは受け取るが、他のコンポーネントへデータを送ることがないコンポーネントを Sink 型コンポーネント (あるいは Sink タイプコンポーネント) といいます。newcomp コマンドには、作るコンポーネントの型にあわせて InPort、OutPort を削除、追加するオプションがあります。利用できるコンポーネントタイプは newcomp -h で newcomp コマンドのヘルプメッセージを表示させるとできます。

```
$ newcomp -h
Usage: newcomp [-f] [-t component_type] NewCompName
(中略)
You may specify component type as -t option. Valid component types are:

null
sink
source
(後略)
```

上の newcomp -h ででてくる null 型はほとんどのメソッドが空の雛型ファイルを作るものです。-t でタイプを指定しなかった場合は null を指定したのと同じものことができます。Source 型でもないし Sink 型でもないコンポーネントを作成する場合はこの null 型 (あるいは -t で型を指定しない) を作成して作業を始めてください (中身はなにもありませんが、実装すべきメソッド (の空のもの) は全て入っています)。

Source タイプのコンポーネントを作成するには

```
% newcomp -t source MySampleReader
```

とします。MySampleReader のところは自分が使いたいコンポーネント名に置き換えてください。また Sink タイプのコンポーネントを作成するには

```
% newcomp -t sink MySampleMonitor
```

とします。MySampleMonitor のところは自分が使いたいコンポーネント名に置き換えてください。

4.1.1 Source 型のロジック

newcomp -t source MyReader とすると雛型として MyReader.h 内に

```
1 private:
2     TimedOctetSeq      m_out_data;
3     OutPort<TimedOctetSeq> m_OutPort;
4
5 private:
6     int daq_dummy();
7     int daq_configure();
8     int daq_unconfigure();
```

4 コンポーネント開発環境

```
9     int daq_start();
10    int daq_run();
11    int daq_stop();
12    int daq_pause();
13    int daq_resume();
14
15    int parse_params(::NVList* list);
16    int read_data_from_detectors();
17    int set_data(unsigned int data_byte_size);
18    int write_OutPort();
19
20    static const int SEND_BUFFER_SIZE = 4096;
21    unsigned char m_data[SEND_BUFFER_SIZE];
22    unsigned int m_recv_byte_size;
```

が定義されます。下から 3 行目と 2 行目がリードアウトモジュールからのデータ読みだしに使用するバッファ (の雛型) です。読み取りロジックは MyReader.cpp の

```
1 int MyReader::read_data_from_detectors()
2 {
3     int received_data_size = 0;
4     /// write your logic here
5     return received_data_size;
6 }
```

の部分に書くように雛型ができます。ここで想定している read_data_from_detectors() の仕様は

- 戻り値は読んだバイト数
- 読んだデータは m_data に入れる

です。これはあくまでも雛型ですので、必ずしもこういうふうにコンポーネントを実装しなければならないということではありません。

4.1.2 Sink 型のロジック

newcomp -t sink MyMonitor すると雛型として MyMonitor.cpp 内に

```
1 check_header_footer(m_in_data, recv_byte_size); // check header and footer
2 unsigned int event_byte_size = get_event_size(recv_byte_size);
3
4 //////////////// Write component main logic here. ////////////////
5 // online_analyze();
6 ///////////////////////////////////////////////////////////////////
7
8 inc_sequence_num(); // increase sequence num.
9 inc_total_data_size(event_byte_size); // increase total data byte size
```

が定義されます。この雛型は 5 行目の online_analyze() 関数にヒストグラムを書くなどの処理を入れると想定してこうしてあります。event_byte_size には、上流コンポーネントから送

られてきたデータのうち、コンポーネントヘッダ、フッタを除いたユーザーデータの長さがバイトサイズで入ります。ユーザーデータの中身は `m_in_data.data[HEADER_BYTE_SIZE]` から `m_in_data.data[HEADER_BYTE_SIZE + event_byte_size - 1]` になります (図 4 を参照してください)。このデータからヒストグラムを書くなどのロジックを実装することになります。これはあくまでも雛型ですので、必ずしもこういうふうにコンポーネントを実装しなければならないということではありません。

4.2 Makefile の書き方

`newcomp` コマンドでできる Makefile を以下に示します (`newcomp MyMonitor` とした場合の例です):

```
COMP_NAME = MyMonitor

all: $(COMP_NAME)Comp

SRCS += $(COMP_NAME).cpp
SRCS += $(COMP_NAME)Comp.cpp

# sample install target
#
# MODE = 0755
# BINDIR = /tmp/mybinary
#
# install: $(COMP_NAME)Comp
#     mkdir -p $(BINDIR)
#     install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)

include /usr/share/daqmw/mk/comp.mk
```

`MyMonitor.cpp` および `MyMonitorComp.cpp` の処理は `/usr/share/daqmw/mk/comp.mk` に書かれていますので Makefile 中にこれらふたつのファイルを手で追加する必要はありません (追加するとエラーになります)。

`newcomp` コマンドでできたファイルのみを使ってコンポーネントを実装する場合は Makefile は変更する必要はありません。ソースファイル (*.cpp) を追加した場合には Makefile に以下のように `SRCS` 変数に追加します (`ModuleUtils.cpp` と `FileUtils.cpp` を追加した場合の例を示します):

```
1 COMP_NAME = MyMonitor
2
3 all: $(COMP_NAME)Comp
4
5 SRCS += $(COMP_NAME).cpp
6 SRCS += $(COMP_NAME)Comp.cpp
7 #
8 # ModuleUtils.cpp と FileUtils.cpp を追加した例
9 #
10 SRCS += ModuleUtils.cpp
11 SRCS += FileUtils.cpp
```

4 コンポーネント開発環境

```
12 # sample install target
13 #
14 #
15 # MODE = 0755
16 # BINDIR = /tmp/mybinary
17 #
18 # install: $(COMP_NAME)Comp
19 #     mkdir -p $(BINDIR)
20 #     install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)
21
22 include /usr/share/daqmw/mk/comp.mk
```

10 行目と 11 行目が追加したファイルの行です。あるいは OBJS 変数にオブジェクトファイル名で指定することもできます:

```
OBJS += ModuleUtils.o
OBJS += FileUtils.o
```

以下のように SRCS 変数と OBJS 変数に同一ファイルを指定することはできません (コンパイルがシンボルの多重定義で失敗します):

```
(これはだめな例)
SRCS += FileUtils.cpp
OBJS += FileUtils.o
```

また OBJS 変数に誤ってソースファイル名 (*.cpp) を書くと make clean でそのソースファイルが消えてしまいますのでご注意ください。

FileUtils.o の作成に FileUtils.h と FileUtils.cpp が必要な場合、以下のように依存関係を書いておくと、FileUtils.h あるいは FileUtils.cpp を変更した場合に、ソースファイル全体をコンパイルしなおすのではなく、変更があったソースファイルだけコンパイルしなおすようになります。

```
FileUtils.o: FileUtils.h FileUtils.cpp
```

newcomp コマンドで生成された cpp ファイルの依存関係 (たとえば newcomp MyMonitor とした場合の MyMonitor.o および MyMonitorComp.o の依存関係) は comp.mk 内に書かれているので書く必要はありません。

comp.mk では CPPFLAGS として -I.、-I/usr/include/daqmw および -I/usr/include/daqmw/idl を追加しています。make コマンドを実行するディレクトリ中の *.h ファイル読み込みのために -I. を追加する必要はありません。これら以外、および /usr/include 以外の場所にあるインクルードファイルを読み込ませたい場合には CPPFLAGS += -I/path/to/myheader_dir のように CPPFLAGS += を使います。

また外部のライブラリを使いたい場合には LDLIBS 変数に追加します。たとえば mylibrary というライブラリを使う場合でこのライブラリのインクルードファイルが /usr/local/include 以

下にあり、ライブラリファイルが`/usr/local/lib/libmylibrary.so`であった場合は以下のよう
に Makefile に追加します。13 行目と 14 行目が追加した行です。

```

1  COMP_NAME = MyMonitor
2
3  all: $(COMP_NAME)Comp
4
5  SRCS += $(COMP_NAME).cpp
6  SRCS += $(COMP_NAME)Comp.cpp
7
8  #
9  # インクルードファイルが/usr/local/include にあり
10 # ライブラリファイルが/usr/local/lib/libmylibrary.so にある
11 # ライブラリを使用する場合は次のように Makefile に追加します。
12 #
13 CPPFLAGS += -I/usr/local/include
14 LDLIBS   += -L/usr/local/lib -lmylibrary
15
16 # sample install target
17 #
18 # MODE = 0755
19 # BINDIR = /tmp/mybinary
20 #
21 # install: $(COMP_NAME)Comp
22 #     mkdir -p $(BINDIR)
23 #     install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)
24
25 include /usr/share/daqmw/mk/comp.mk

```

DAQ-Middleware が提供するライブラリ (ソケットライブラリ、json ライブラリ) のディ
レクトリ (32 ビット SL では `/usr/lib/daqmw`、64 ビット SL では `/usr/lib64/daqmw`)
は `DAQMW_LIB_DIR` として参照可能です。インクルードファイルがあるディレクトリ
(`/usr/include/daqmw`) は前述のようにすでに `CPPFLAGS` に追加されているので追加する必要は
ありません。

`make` コマンドを実行すると `autogen` ディレクトリが作成されそこには自動生成されたファイ
ルが入ります。コンポーネント開発では `autogen` ディレクトリのファイルは変更する必要はあり
ません。

コンポーネントを開発する場所はどこでもかまいませんが DAQ-Middleware に含まれている
makefile サブルーチンユーティリティ (`comp.mk`) は 1 ディレクトリ 1 コンポーネントを前提とし
て作られています。

DAQ-Middleware に添付されている `comp.mk` サブルーチンを使う場合はソースファイルの拡張
子は `.cpp` を、インクルードファイルの拡張子は `.h` を使ってください。これ以外のもの (たとえば
`.cc` とか `.hh`) を使うとコンパイルが正常に行われません。

5 開発ディレクトリの準備

この解説では開発システムに daq ユーザーとしてログインして作業するものとして解説を行います。複数のコンポーネントを作成することになるので、それらをまとめるため開発ディレクトリを /home/daq/MyDaq とすることにしてこのディレクトリを作成します。

```
% cd
% mkdir MyDaq
% cd MyDaq
% pwd
/home/daq/MyDaq
```

6 Skeleton コンポーネントによる状態遷移の確認

Skeleton コンポーネントで状態遷移を確認してみます。Skeleton コンポーネントは、コンポーネント動作に必要なすべてのメソッドが、中身が空の状態の実装してあるコンポーネントです。前節で作った開発用ディレクトリに移動して newcomp コマンドで Skeleton コンポーネントのソースを作ります (できるソースは /usr/share/daqmw/examples/Skeleton 以下にあるものと同一です)。新たにできた Skeleton ディレクトリに移動し、make を実行します。

```
% cd
% cd MyDaq
% newcomp Skeleton
% cd Skeleton
% ls
Makefile  SkeletonComp.cpp  Skeleton.cpp  Skeleton.h
% make
(途中省略)
% ls -l SkeletonComp
-rwxrwxr-x 1 daq daq 281923 Apr  1 09:00 SkeletonComp
```

続けてこのコンポーネントを動かすためのコンフィギュレーションファイルをコピーします。

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

skel.xml をエディタで開いて execPath を調べます。execPath は上で作った SkeletonComp 実行ファイルを指定している必要があります。この例のとおりやった場合は変更する点はありませんが、違うディレクトリにある場合は SkeletonComp ファイルがあるパスをフルパスで指定するように編集してください。/usr/share/daqmw/conf/skel.xml のコード部分を以下に示します。

```
1 <configInfo>
2   <daqOperator>
3     <hostAddr>127.0.0.1</hostAddr>
```

```

4   </daqOperator>
5   <daqGroups>
6     <daqGroup gid="group0">
7       <components>
8         <component cid="SkeletonComp0">
9           <hostAddr>127.0.0.1</hostAddr>
10          <hostPort>50000</hostPort>
11          <instName>Skeleton0.rtc</instName>
12          <execPath>/home/daq/MyDaq/Skeleton/SkeletonComp</execPath>
13          <confFile>/tmp/daqmw/rtc.conf</confFile>
14          <startOrd>1</startOrd>
15          <inPorts>
16          </inPorts>
17          <outPorts>
18          </outPorts>
19          <params>
20          </params>
21        </component>
22      </components>
23    </daqGroup>
24  </daqGroups>
25 </configInfo>

```

タグの詳細については「DAQ-Middleware 1.1.0 技術解説書」[2]を参照してください。execPathでコンポーネント実行ファイルのフルパスを指定しています。このコンポーネントは他のコンポーネントと接続することはないのでinPorts、OutPortsは空になっています。

「DAQ-Middleware 1.1.0 技術解説書」[2]に書かれているとおりDAQ-MiddlewareではDAQシステムの統括はDaqOperatorが行います。コンポーネントの接続、データ収集の開始、終了の指示はDaqOperatorが行いますが、指示されるほうのコンポーネントは別の方法で既に起動されている必要があります(DaqOperatorが各コンポーネントを起動するわけではありません)。各コンポーネントをブートする方法にはshellのコマンドラインから起動する、xinetdを使ってネットワークブートを行うなどの方法があります。ここではDAQ-Middlewareに含まれている/usr/bin/run.pyコマンドのローカルブート機能を使ってブートを行います。

run.pyでオプション-l(数字の“いち”ではなくてエル)を指定すると、run.pyは最後の引数で指定したコンフィギュレーションファイルを読み、起動すべきコンポーネントのパス名を取得します。そのパスにあるコンポーネントをローカル計算機で起動したあと、DaqOperatorをローカル計算機で起動します。またrun.pyで-cオプションを指定すると、run.pyはDaqOperatorをコンソールモードで起動します。コンソールモードで起動したDaqOperatorはキーボードからユーザからの指示を読みます。また定期的に各コンポーネントが取り扱ったデータバイト数を端末に表示します(各コンポーネントは定期的にDaqOperatorに自身が処理したデータバイト数を報告しています)。Skeletonコンポーネントではデータは流れないのでデータバイト数は0のままになっています。

```

% cd
% cd MyDaq
% run.py -c -l skel.xml

```

6 SKELETON コンポーネントによる状態遷移の確認

(あるいは `run.py -cl skel.xml` のようにオプションをまとめて指定することもできます)

`run.py` を起動してしばらく待つと (計算機の CPU 性能で差はありますがおおよそ 4 秒くらい)、

```
Command:    0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume

RUN NO: 0
start at:      stop at:

GROUP:COMP_NAME      EVENT  SIZE      STATE      COMP STATUS
group0:SkeletonComp0:      0      LOADED      WORKING
```

のようになります。先に述べたようにこの文字を出力しているのは `DaqOperator` で、`DaqOperator` はコマンドキー入力を待っています。今のシステムではコンポーネントは `Skeleton` コンポーネントのみで現在の状態は `STATE` 欄から `LOADED` であることがわかります。利用できるコマンドは 1 行目の `Command:` と書いてある行に表示されています。コマンド入力に対応する数字キーを押すことで行います。状態遷移はひとつづつ行う必要があります。たとえばこの `LOADED` の状態で、`start` を押すと不適切な入力と判断されます。コマンドを入力すると `DaqOperator` は各コンポーネントに遷移命令を送ります。

上の画面で `STATE` 欄が `LOADED` になっていることを確認してください。0 を押して `configure` すると `STATE` 欄が `CONFIGURED` に変わります。続けて 1 を押して `start` するとランナンバーの入力を求められるので適当にランナンバー (1 とか) を入力するしてください。すると `STATE` 欄が `RUNNING` に変わります。続けて 2 を押して `stop` すると `STATE` 欄が `CONFIGURED` に変わります。

2 を押してコンポーネントを `stop` させたあとに `Ctrl-C` を押すと `DaqOperator` に `SIGINT` が送られて `DaqOperator` が終了します。`DaqOperator` と同時に `run.py` が起動したコンポーネントにも `SIGINT` が送られます (`run.py` から起動した `DaqOperator`、および各コンポーネントが同一プロセスグループに属しているため)。通常コンポーネントのほうが先に `exit` して `DaqOperator` は数回コンポーネントと接続しようとししますので、`Ctrl-C` を押したあと画面に

```
### ERROR:      : cannot connect
```

という行がコンポーネントの数だけ表示されます。少し待って `DaqOperator` が終了します。

`run.py` から起動されたコンポーネントの標準出力、標準エラー出力は `/tmp/daqmw/log`。コンポーネントプログラム名に保存されます。いまの場合は `/tmp/daqmw/log.SkeletonComp` に保存されます。

Skeleton コンポーネントの状態遷移の確認

Skeleton コンポーネントを少し改造してコンポーネントの状態遷移の確認をしてみます。`/usr/share/daqmw/examples/` 以下にある全てのコンポーネントソースファイルでは `m_debug` 変数が定義されていて、これを使ってデバッグメッセージの出力を行ったりとめたりできるようになって

います。プログラムのには単純に

```
if (m_debug) {
    std::cerr << "debug message here" << std::endl;
}
```

と書きます。

まず Skeleton コンポーネントのコンストラクタ (Skeleton.cpp 内にあります) の初期化のところを以下のように書き換えます。

```
Skeleton::Skeleton(RTC::Manager* manager)
: DAQMW::DaqComponentBase(manager),
  m_InPort("skeleton_in", m_in_data),
  m_OutPort("skeleton_out", m_out_data),

  m_in_status(BUF_SUCCESS),
  m_out_status(BUF_SUCCESS),

  m_debug(true) // false から true に書き換え
```

また定期的に呼び出される onExecute のデバッグメッセージは今回は関係ないのでコメントアウトしておきます。

```
RTC::ReturnCode_t Skeleton::onExecute(RTC::UniqueId ec_id)
{
    // std::cerr << "*** onExecute\n"; // 出力抑制のためコメントアウト
    daq_do();

    return RTC::RTC_OK;
}
```

さらに daq_dummy() では元の状態ではデバッグメッセージが入っていないので追加しておきます。

```
int Skeleton::daq_dummy()
{
    std::cerr << "Skeleton::dummy" << std::endl; // 追加

    return 0;
}
```

これで make して前節と同様に run.py -cl skel.xml で run.py から Skeleton コンポーネントを起動、DaqOperator をコンソールモードで起動します。コンポーネントのログは/tmp/daqmw/以下にできます。Skeleton コンポーネントのログは/tmp/daqmw/log.SkeletonComp にできますのでこれを tail -f で見ながら DaqOperator に CONFIGURE 等の指示を出し、遷移する際のメソッド、ある状態にあるあいだじゅう繰り返されるメソッドが実際に呼ばれていることを確認してください。

7 単純なコンポーネントの作成例

コンポーネント間のデータ転送ができるようになるために本節では、単純なコンポーネントを作成してみます。

作るコンポーネントは `newcomp` でできる Source 型コンポーネント (ここでは `TinySource` コンポーネントと名付けます) と Sink 型コンポーネント (ここでは `TinySink` コンポーネントと名付けます) です。TinySource は自分でデータを作ってそれを送るコンポーネントで、TinySink は受け取ったデータを 16 進で標準エラー出力に出すコンポーネントとします。DAQ-Middleware 1.4.0 をインストールすると `TinySource`、`TinySink` のソースはそれぞれ `/usr/share/daqmw/examples/TinySource`、`/usr/share/daqmw/examples/TinySink` に入ります。使用するコンフィギュレーションファイルは `/usr/share/daqmw/conf/tiny.xml` です。

```
% cd
% cd MyDaq
% newcomp -t source TinySource
% newcomp -t sink TinySink
% cp /usr/share/daqmw/conf/tiny.xml .
```

として `TinySource`、`TinySink` を以下のように変更します。

TinySource.cpp の変更

```
1 int TinySource::read_data_from_detectors()
2 {
3     int received_data_size = 0;
4     /// write your logic here
5     usleep(500000); // 追加
6     for (int i = 0; i < SEND_BUFFER_SIZE; i++) { // 追加
7         m_data[i] = (i % 256); // 追加
8     } // 追加
9     received_data_size = SEND_BUFFER_SIZE; // 追加
10    /// end of my tiny logic
11
12    return received_data_size;
13 }
```

`TinySource.h` で確保されたバッファに単純に数値をうめこんでいます。あまり頻繁にぐるぐるまわしても大変ですから 5 行目で 0.5 秒 sleep させています。

TinySink.h の変更

```
1 private:
2     int daq_dummy();
3     int daq_configure();
4     int daq_unconfigure();
5     int daq_start();
6     int daq_run();
7     int daq_stop();
```

```

8   int daq_pause();
9   int daq_resume();
10
11  int parse_params(::NVList* list);
12  int reset_InPort();
13
14  unsigned int read_InPort();
15  //int online_analyze();
16  static const unsigned int RECV_BUFFER_SIZE = 4096; // 追加
17  unsigned char m_data[RECV_BUFFER_SIZE];           // 追加
18  BufferStatus m_in_status;
19  bool m_debug;

```

16行目と17行目でInPortにあるデータをコピーするバッファを追加しました。

TinySink.cpp の変更

```

1  check_header_footer(m_in_data, recv_byte_size); // check header and footer
2  unsigned int event_byte_size = get_event_size(recv_byte_size);
3
4  //////////// Write component main logic here. ////////////
5  // online_analyze();
6  if (event_byte_size > RECV_BUFFER_SIZE) {           // 追加
7      fatal_error_report(USER_DEFINED_ERROR1, "Length Too Large"); // 追加
8  }                                                   // 追加
9  memcpy(m_data, &m_in_data.data[HEADER_BYTE_SIZE], event_byte_size); // 追加
10 for (unsigned int i = 0; i < event_byte_size; i++) { // 追加
11     fprintf(stderr, "%02X ", m_data[i]);           // 追加
12     if ((i + 1) % 16 == 0) {                       // 追加
13         fprintf(stderr, "\n");                   // 追加
14     }                                             // 追加
15 }                                               // 追加
16 ////////////////////////////////////////////////////
17
18 inc_sequence_num();                               // increase sequence num.
19 inc_total_data_size(event_byte_size);           // increase total data byte size

```

memcpy() で m_in_data.data[HEADER_BYTE_SIZE] から event_byte_size ぶん memcpy() でデータをコピーしています。バッファオーバーランしないように memcpy() の前にイベントデータバイト数を確認してバッファサイズ (RECV_BUFFER_SIZE) より大きかったら致命的エラーが発生したと判断することにして fatal_error_report() で DaqOperator に報告しています。10行目からの for ループで取り出したデータを標準エラー出力に出力しています。

変更したらコンパイルします。

```

% cd
% cd MyDaq
% cd TinySource
% make
% cd ..
% cd TinySink
% make
% cd ..

```

7 単純なコンポーネントの作成例

動作テスト /usr/share/daqmw/conf/tiny.xml が Tiny コンポーネント用コンフィギュレーションファイルです。コピーして使用します。execPath が上で作ったコンポーネントの実行ファイルのフルパスになっているかどうか確認してください。違っていたらエディタで編集します。/usr/share/daqmw/conf/tiny.xml のコード部分を以下に示します。

```
1 <configInfo>
2   <daqOperator>
3     <hostAddr>127.0.0.1</hostAddr>
4   </daqOperator>
5   <daqGroups>
6     <daqGroup gid="group0">
7       <components>
8         <component cid="TinySource0">
9           <hostAddr>127.0.0.1</hostAddr>
10          <hostPort>50000</hostPort>
11          <instName>TinySource0.rtc</instName>
12          <execPath>/home/daq/MyDaq/TinySource/TinySourceComp</execPath>
13          <confFile>/tmp/daqmw/rtc.conf</confFile>
14          <startOrd>2</startOrd>
15          <inPorts>
16          </inPorts>
17          <outPorts>
18            <outPort>tinysource_out</outPort>
19          </outPorts>
20          <params>
21          </params>
22        </component>
23        <component cid="TinySink0">
24          <hostAddr>127.0.0.1</hostAddr>
25          <hostPort>50000</hostPort>
26          <instName>TinySink0.rtc</instName>
27          <execPath>/home/daq/MyDaq/TinySink/TinySinkComp</execPath>
28          <confFile>/tmp/daqmw/rtc.conf</confFile>
29          <startOrd>1</startOrd>
30          <inPorts>
31            <inPort from="TinySource0:tinysource_out">tinysink_in</inPort>
32          </inPorts>
33          <outPorts>
34          </outPorts>
35          <params>
36          </params>
37        </component>
38      </components>
39    </daqGroup>
40  </daqGroups>
41 </configInfo>
```

TinySource コンポーネントは OutPort をひとつ持つので 17 行目の OutPorts で OutPort をひとつ指定しています。また TinySink コンポーネントは InPort をひとつ持つので 30 行目の InPorts で InPort をひとつ指定しています。その他タグの詳細は「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

では起動してみます。

```
% cd
% cd MyDaq
% ls tiny.xml (tiny.xml があることを確認。まだコピーしていなかったら下のコマンドを実行)
% cp /usr/share/daqmw/conf/tiny.xml .
% run.py -c -l tiny.xml
(どんどんログに出力されているので 5 秒程度で 2 を押して止めてください)。
(CONFIGURED になったら Ctrl-C で run.py を抜ける)。
```

run.py は、コンポーネントが出すエラー出力を/tmp/daqmw/log. コンポーネントプログラム名に出力するようにコンポーネントを起動します。/tmp/daqmw/log.TinySinkComp に fprintf() で出力したデータが記録されていることを確認してください。

以上でコンポーネント間での通信ができるようになりました。実際の DAQ システムでは Source 型コンポーネントはリードアウトモジュールからデータを読むようにソケットプログラミングをして、読んだデータを後段のコンポーネントに送るということになります。また Sink 型コンポーネントは上流コンポーネントから送られてきたデータを単に標準エラー出力に出すのではなくヒストグラム化ツールを使ってデコード、およびヒストグラムを書くということになります。

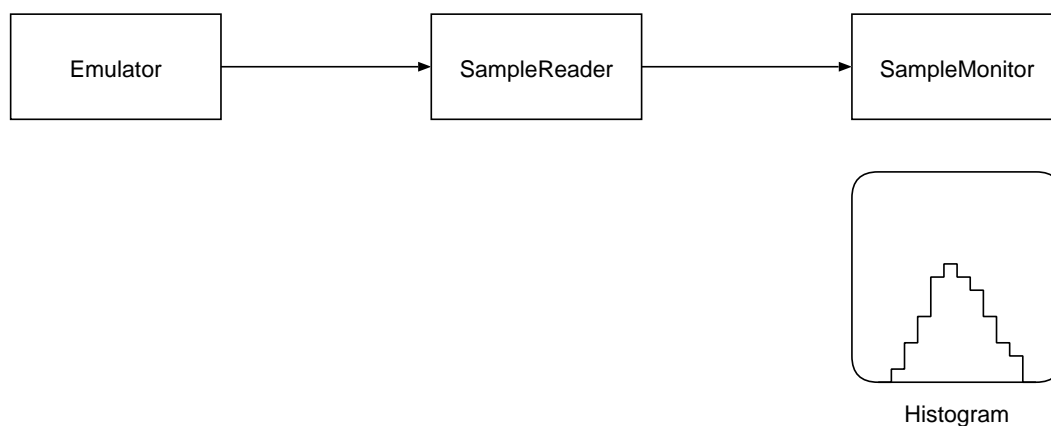


図 6 この文書で開発するデータ収集システムの概要図。SampleReader が Emulator からデータを読み、後段の SampleMonitor に送る。SampleMonitor は受け取ったデータをデコードしてヒストグラムを書いて画面上に表示する。

8 この文書で開発するデータ収集システムの概要

この文書で開発するデータ収集システムの概要を図 6 に示します。SampleReader が Emulator からデータを読み、後段の SampleMonitor に送ります。SampleMonitor は受け取ったデータをデコードしてヒストグラムを書いて画面上に表示するというシンプルなデータ収集システムです。Emulator としてここではソフトウェアエミュレータを使用します。

9 ソフトウェアエミュレータ

9.1 セットアップ

DAQ-Middleware 1.2.0 からは `/usr/bin/daqmw-emulator` としてインストールされていますのでなにもする必要はありません^{*4}。DAQ-Middleware 1.2.0 以前のものを使う場合は以下に示す URL からダウンロードできます。

```
http://daqmw.kek.jp/src/daqmw-emulator.tar.gz
```

ダウンロードしたら `/home/daq/MyDaq/` ディレクトリ以下に展開します。

```
% cd
% cd MyDaq
% lftpget http://daqmw.kek.jp/src/daqmw-emulator.tar.gz
% tar xf daqmw-emulator.tar.gz
% ls
daqmw-emulator
```

^{*4} ソースは `/usr/share/daqmw/daqmw-emulator/` 以下にはいています。

```
% cd daqmw-emulator
% make
% cp emulator ~/bin/daqmw-emulator
```

9.2 起動

コマンドラインから

```
% daqmw-emulator
```

とするとポート 2222 で接続を待ちます。接続があるとすぐにデータを送りはじめます。オプションをなにもつけない場合は約 8kB/s でデータを送ります。転送レートを変更するには `-t` オプションで指定して

```
% daqmw-emulator -t 128k
```

のようにします。これで 128kB/s でデータを送るようになります。 `-t 1M` を指定すると 1MB/s でデータを送るようになります。あまり大きな値を指定すると、特に VMware Player で使用している場合は計算機負荷がかかるのでさげましょう。

停止させるには通常よくやるように Ctrl-C を押します。

9.3 ソフトウェアエミュレータのデータフォーマット

この解説書で使用するソフトウェアエミュレータのデータフォーマットを図 7 に示します。1 イベントデータを送るのに 8 バイト使用します。最初にシグネチャ (マジック) `0x5a` がきます。デコードするにはこのバイトがこの値になっているかどうかで違うところを読んでいないかどうか確認することができます。続けてデータフォーマットバージョン (`0x01`) がきます。その次にモジュール番号がきます。このソフトウェアエミュレータでは `0x0~0x7` の値が入っています。次の 1 バイトは将来の拡張用として予約になっています。最後の 4 バイトにイベントデータ (整数値) がはいつています。このイベントデータには `0.000~1000.000` までの数値を 1000 倍して整数に丸めた値が入っています。数値の意味がある複数バイトをネットワークで送る場合、どのようなバイト順で送って来るのか決める必要があります。このソフトウェアエミュレータではネットワークバイトオーダーで送って来るようになっています。読み取り側でホストバイトオーダーに変換するには `ntohl()` 関数を使用します。

9.4 エミュレータからのデータの確認

エミュレータからどういうデータがやってくるのか確認しておきましょう。 `nc` コマンドを使うのが簡単です。以下のようにコマンドを実行します。このコマンドは複数行に分けるのではなく 1

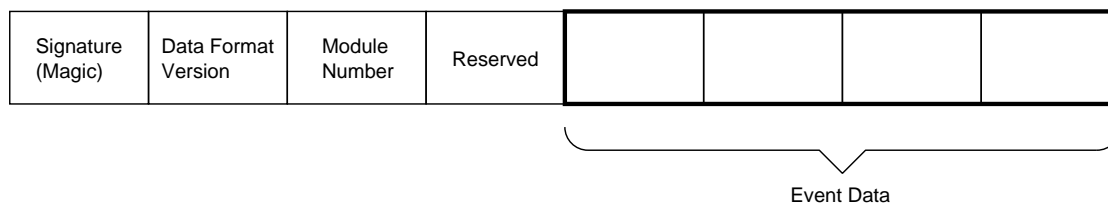


図7 ソフトウェアエミュレータからやってくるデータのデータフォーマット。1 イベントデータを送るのに8バイト使用する。イベントデータは物理量的には0.000~1000.000の値をとるもので、エミュレータはこれを1000倍して4バイト整数値に丸めて送ってくる。バイトオーダーはネットワークバイトオーダーになっている。0バイトから3バイトまではメタデータ。マジックは0x5aに固定。データフォーマットバージョンは0x01固定。モジュール番号は0x01から0x07を送ってくるが、この解説書ではモジュール番号は使用しない。

行で投入してください*5。

```
(sleep 10; pkill -f /usr/bin/nc) & /usr/bin/nc 127.0.0.1 2222 > data.out
```

これでncコマンドが127.0.0.1のポート2222に接続します。読んだデータはdata.outファイルに保存されます。読み込み時間はsleepで指定した秒数でここでは10秒です。データフォーマットについては前節をごらんください。適当にデコードして(たとえば8バイト読んで、4バイト目から8バイト目をintとしてとりだしntohl()でホストバイトオーダーに変換し1000.0で割るプログラムを書くなどする)ヒストグラムを書くと図8のように100、200、300、…、800にピークがある図になります。この図を画面に表示し、定期的にアップデートされるようなシステムを作ることがこの解説の目的です。

エミュレータデータの詳細になりますが、図8の100付近のピークのデータは全てモジュール番号が0になっています。200付近のピークのデータは全てモジュール番号が1になっています。800付近のピークのデータは全てモジュール番号が7になっています。図8はモジュール番号は無視して全てのモジュールからのデータを重ね合わせたものになっています。この文書ではエミュレータから送られてくるモジュール番号は利用しません。

10 SampleReader コンポーネントの開発

以下で解説するSampleReaderおよびSampleMonitorのコードは/usr/share/daqmw/examples/以下のSampleReaderディレクトリ、およびSampleMonitorディレクトリにあります。newcompで作った雛型ファイルとの変更点はたとえばdiffコマンドを以下のように使って調べることができます。

*5 単にpkill ncとするとncプロセス以外の“nc”という文字列を含んだ他のプロセスへもシグナルが送られてその結果それらの関係ないプロセスもexitしてしまいますのでここではncをフルパスで指定しています。

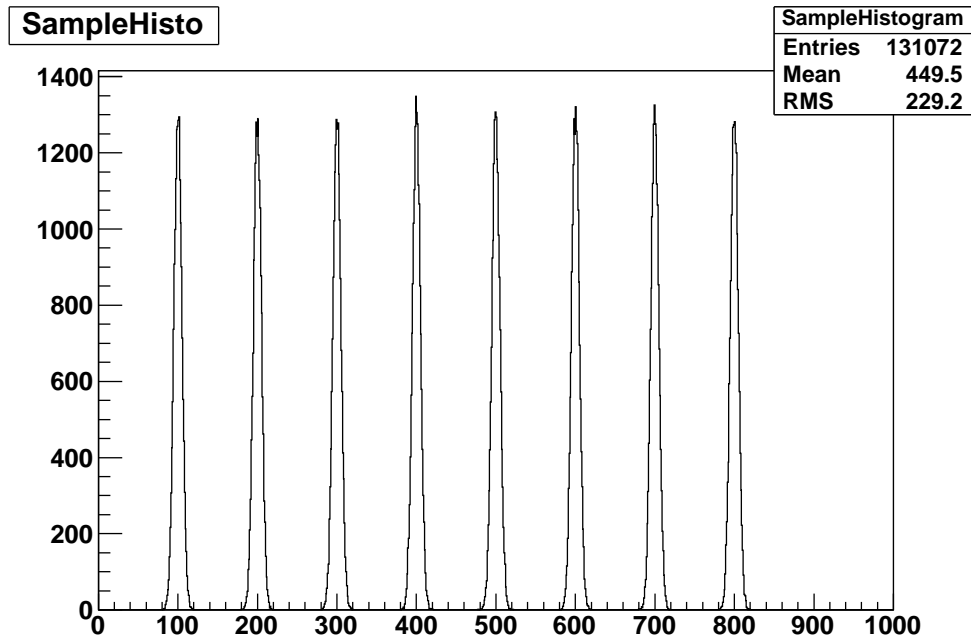


図 8 この解説書で使うソフトウェアエミュレータからのデータのヒストグラム

```
% mkdir diff-test
% cd diff-test
% newcomp -t source SampleReader
% ls
SampleReader
% newcomp -t sink SampleMonitor
% ls
SampleMonitor SampleReader
% mv SampleReader SK-SampleReader
% mv SampleMonitor SK-SampleMonitor
% cp -r /usr/share/daqmw/examples/SampleReader .
% cp -r /usr/share/daqmw/examples/SampleMonitor .
% diff -urNp SK-SampleReader SampleReader | less
% diff -urNp SK-SampleMonitor SampleMonitor | less
```

diff コマンドの `-p` オプションを使うと、以下のように変更があった行を示す `@@` の行と一緒にその変更がなんという関数名のところであるのが表示されるようになるので、変更箇所の判別に役立ちます。

```
@@ -85,6 +87,9 @@ int SampleReader::daq_configure()
(以下変更点がでてくる)
```

では実際にコンポーネントを開発してみましょう。ここでは第 8 節で書いたような DAQ システムを構成するコンポーネントを作成することにします。この節ではエミュレータからデータを読み取って後段のコンポーネントに送る `SampleReader` コンポーネントを作成します。まず

SampleReader コンポーネントのデータ読み取り部分の仕様を考えます。ここでは以下のようにしました。

- ソケット部分については DAQ-Middleware 付属の Sock ライブラリを使用する。
- 接続に失敗したら致命的エラーが起きたと考えることにする。
- ソケットからの読み取りバッファとして 1024 バイト用意する。
- 一度に 1024 バイト必ず読むことにする。
- 2 秒以内に 1024 バイト読めなかった場合は致命的エラーが起きたと考えることにする。
- エミュレータの IP アドレス、およびポート番号はコンフィギュレーションファイルで指定する。
- `daq_run()` 中、後段のコンポーネントにデータを送ることができなかった場合は次の `daq_run()` ではエミュレータから新たにデータを読むことはせず、送れなかったデータを再送する。

DAQ-Middleware 付属の Sock ライブラリのインクルードファイルは `/usr/include/daqmw/Sock.h` で、ライブラリファイルは `/usr/lib/daqmw/libSock.so` です。

仕様が決まったら実装作業に移ります。まず `newcomp -t source SampleReader` とコマンドを実行して Source タイプコンポーネントを指定して雛型ファイルを作ります。またできたディレクトリ (いまの場合は `SampleReader`) に移動して `make` し、開発環境が正常かどうか確認しておきます。

```

1 % newcomp -t source SampleReader (雛型ファイルを作成する)
2 % cd SampleReader (SampleReader ディレクトリができているので移動)
3 % ls (作られたファイルを見る)
4 Makefile SampleReader.cpp SampleReader.h SampleReaderComp.cpp
5 % make (開発環境の確認)
6 rm -fr autogen (正常なら SampleReaderComp という実行形式
7 mkdir autogen ファイルができる)
8 (中略)
9 % ls (できたファイルの確認)
10 Makefile SampleReader.h SampleReaderComp* SampleReaderComp.o
11 SampleReader.cpp SampleReader.o SampleReaderComp.cpp autogen/
12 % make clean (オブジェクトファイル、実行形式ファイルおよび
13 自動生成されたファイル (autogen ディレクトリ
14 以下) の消去)
15 % ls
16 Makefile SampleReader.cpp SampleReader.h SampleReaderComp.cpp

```

10.1 SampleReader.h の変更

SampleReader.h を以下のように変更します。

10.1.1 Sock ライブラリの利用

まず Sock ライブラリを使えるようにします。

```

1 #include "DaqComponentBase.h"
2
3 #include <daqmw/Sock.h> // 追加
4
5 using namespace RTC;

```

この変更点は 4 行目の#include の追加でこれは DAQ-Middleware 付属の Sock ライブラリを利用できるようにするものです。

10.1.2 メンバー変数等の追加

メンバー変数、定数を変更します。

```

1     int set_data(unsigned int data_byte_size);
2     int write_OutPort();
3
4     DAQMW::Sock* m_sock;           /// 追加 socket for data server
5
6     static const int EVENT_BYTE_SIZE = 8;    // 追加 event byte size
7     static const int SEND_BUFFER_SIZE = 1024; // 変更
8     unsigned char m_data[SEND_BUFFER_SIZE];
9     unsigned int m_recv_byte_size;          // 追加
10
11    BufferStatus m_out_status;
12
13    int m_srcPort;                       // 追加 Port No. of data server
14    std::string m_srcAddr;                // 追加 IP addr. of data server

```

変更内容はコメントで書いたとおりです。

- (4 行目) Sock オブジェクト追加
- (6 行目) エミュレータからやってくるデータは 1 イベントデータが 8 バイトなのでそれを定義した。
- (7 行目) 上で述べたように 1 回のリードで 1024 バイト読むことにしたのでそれを定義。
- (13 行目) エミュレータの IP ポート番号を指定する変数。ポート番号はコンフィギュレーションファイルから取得する。
- (14 行目) エミュレータの IP アドレス変数。IP アドレスはコンフィギュレーションファイルから取得する。

この他 9 行目で m_recv_byte_size という変数をメンバー変数に追加しています。これは上記の仕様で「daq_run() 中、後段のコンポーネントにデータを送ることができなかった場合は次の daq_run() ではエミュレータから新たにデータを読むことはせず、送れなかったデータを再送する」と決めたのでそれを実装するための変数です。再送のためには前回の daq_run() でエミュレータから何バイトデータを読んだか記憶しておく必要があります。この変数はそのために使います。今回は必ず 1024 バイト読むと決めましたが今後の拡張を考えてこのようにしました。

10.2 SampleReader.cpp の変更

次に SampleReader.cpp の変更に移ります。ここでは状態チャート (図 3) にある関数毎に解説します。

コンストラクタ

```

1 SampleReader::SampleReader(RTC::Manager* manager)
2   : DAQMW::DaqComponentBase(manager),
3     m_OutPort("samplerreader_out", m_out_data),
4     m_sock(0), // 追加
5     m_recv_byte_size(0),
6     m_out_status(BUF_SUCCESS),
7
8     m_debug(false)
9 {
10  // Registration: InPort/OutPort/Service
11
12  // Set OutPort buffers
13  registerOutPort("samplerreader_out", m_OutPort);
14
15  init_command_port();
16  init_state_table();
17  set_comp_name("SAMPLEREADER");
18 }

```

データ読み取りソケットオブジェクト (m_sock) を 0 に初期化します。この値が 0 かどうかでソケットオブジェクトを delete するべきかどうか判定できます。後述 daq_stop() を参照してください。

fatal_type

```

1 using DAQMW::FatalType::DATAPATH_DISCONNECTED;
2 using DAQMW::FatalType::OUTPORT_ERROR;
3 using DAQMW::FatalType::USER_DEFINED_ERROR1;
4 using DAQMW::FatalType::USER_DEFINED_ERROR2;

```

using 宣言を使って、fatal_error_report() の引数で名前空間名を省略できるようにしています。fatal_error_report() については「DAQ-Middleware 1.1.0 技術解説書」[2] をご覧ください。

daq_configure()

```

1 int SampleReader::daq_configure()
2 {
3     std::cerr << "*** SampleReader::configure" << std::endl;
4
5     ::NVList* paramList;
6     paramList = m_daq_service0.getCompParams();
7     parse_params(paramList);

```

```

8
9     return 0;
10 }

```

上は、`newcomp -t source SampleReader` が作成した `daq_configure()` です。今回の実装ではここで変更する事項はありません。6行目の `getCompParams()` でコンフィギュレーションファイルで指定されたパラメータを取得しています。取得したパラメータの解析、および変数への設定は `parse_params()` で行うようになっていきますので続けて `parse_params()` の変更に移ります。

`parse_params()` を変更して `m_srcAddr` および `m_srcPort` 変数にコンフィギュレーションファイルから取得した値をセットできるようにします。

```

1 int SampleReader::parse_params(::NVList* list)
2 {
3     bool srcAddrSpecified = false; // 追加
4     bool srcPortSpecified = false; // 追加
5
6     std::cerr << "param list length:" << (*list).length() << std::endl;
7
8     int len = (*list).length();
9     for (int i = 0; i < len; i+=2) {
10        std::string sname = (std::string)(*list)[i].value;
11        std::string svalue = (std::string)(*list)[i+1].value;
12
13        std::cerr << "sname: " << sname << " ";
14        std::cerr << "value: " << svalue << std::endl;
15
16        // 追加 (m_srcAddr および m_srcPort の設定)
17        if ( sname == "srcAddr" ) {
18            srcAddrSpecified = true;
19            if (m_debug) {
20                std::cerr << "source addr: " << svalue << std::endl;
21            }
22            m_srcAddr = svalue;
23        }
24        if ( sname == "srcPort" ) {
25            srcPortSpecified = true;
26            if (m_debug) {
27                std::cerr << "source port: " << svalue << std::endl;
28            }
29            char* offset;
30            m_srcPort = (int)strtol(svalue.c_str(), &offset, 10);
31        }
32    }
33
34    // 追加 (srcAddr および srcPort が取得できていなければ致命的エラー
35    // とする)
36    if (!srcAddrSpecified) {
37        std::cerr << "### ERROR:data source address not specified\n";
38        fatal_error_report(USER_DEFINED_ERROR1, "NO SRC ADDRESS");
39    }
40    if (!srcPortSpecified) {
41        std::cerr << "### ERROR:data source port not specified\n";
42        fatal_error_report(USER_DEFINED_ERROR2, "NO SRC PORT");
43    }
44 }

```


10 SAMPLEREADER コンポーネントの開発

```
45     return 0;
46 }
```

bool 変数を追加 (3 行目、4 行目) して、これから追加する m_srcAddr および m_srcPort 変数の取得に成功したか失敗したか記録できるようにしておきます。

list 変数にはコンフィギュレーションファイルで指定されたパラメータが、パラメータ名、値、パラメータ名、値、パラメータ名、値、… の順に入っていますので必要なパラメータ名があるかどうか線形サーチします。見つければ変数に値をセットします。これを行っているのが 17 行以下のコードです。値は文字列になっていますので strtol() で数値に変更しています。また 36 行目以下で m_srcAddr および m_srcPort が取得できないときには致命的エラーが起きたと判断することにして fatal_error_report() でエラーが起きたことを DaqOperator に通知していません。以上で daq_configure() の変更は終了です。

daq_start()

```
1 int SampleReader::daq_start()
2 {
3     std::cerr << "*** SampleReader::start" << std::endl;
4
5     m_out_status = BUF_SUCCESS;
6
7     // 以下を追加
8     try {
9         // Create socket and connect to data server.
10        m_sock = new DAQMW::Sock();
11        m_sock->connect(m_srcAddr, m_srcPort);
12    } catch (DAQMW::SockException& e) {
13        std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
14        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
15    } catch (...) {
16        std::cerr << "Sock Fatal Error : Unknown" << std::endl;
17        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
18    }
19
20    // Check data port connections
21    bool outport_conn = check_dataPort_connections( m_OutPort );
22    if (!outport_conn) {
23        std::cerr << "### NO Connection" << std::endl;
24        fatal_error_report(DATAPATH_DISCONNECTED);
25    }
26
27    return 0;
28 }
```

ここでは 9 行目以下を追加しました。追加した内容は、Sock オブジェクトを生成し (10 行目)、m_srcAddr および m_srcPort で指定されたサーバー (今回の場合はエミュレータ) に接続します。接続に失敗したら致命的失敗が起きたと考えることに仕様を決めましたのでそのコードを追加しています (12 行目以下)。check_dataPort_connection() は後段コンポーネントと接続ができていかどうかを確認する関数です。接続できていなければ致命的エラーが起きたとしています (21 行

目以下)。

daq_run()

```

1 int SampleReader::daq_run()
2 {
3     if (m_debug) {
4         std::cerr << "*** SampleReader::run" << std::endl;
5     }
6
7     if (check_trans_lock()) { // check if stop command has come
8         set_trans_unlock(); // transit to CONFIGURED state
9         return 0;
10    }
11
12    if (m_out_status == BUF_SUCCESS) { // previous OutPort.write() successfully done
13        int ret = read_data_from_detectors();
14        if (ret > 0) {
15            m_rcv_byte_size = ret;
16            unsigned long sequence_num = get_sequence_num();
17            set_data(m_rcv_byte_size); // set data to OutPort Buffer
18        }
19    }
20
21    if (write_OutPort() < 0) {
22        ; // Timeout. do nothing.
23    }
24    else { // OutPort write successfully done
25        inc_sequence_num(); // increase sequence num.
26        inc_total_data_size(m_rcv_byte_size); // increase total data byte size
27    }
28
29    return 0;
30 }

```

daq_run() の部分は newcomp コマンドでできたものから変更したところはありません。以下、コードの解説をします。

(7~10 行目) stop コマンドがきているかどうかをチェックしているところです。stop コマンドがきていたら check_trans_lock() は true を返します。この場合は set_trans_lock() を呼んで CONFIGURED ステートに移行して daq_run() を終了します。stop コマンドがきていなかった場合はデータの読み取りを行います (ただし前回の daq_run() で後段のコンポーネントにデータを送ることができなかった場合は新たにエミュレータからデータを読まずに、前回読んだデータを再送すると仕様で決めたのでそのようにコーディングします)。check_trans_lock()、set_trans_lock() の意味と詳細は「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

(12~19 行目) if で前回の daq_run() でデータを正常に後段コンポーネントに送れたかどうかを調べています (m_out_status が BUF_SUCCESS の場合にはデータが正常に送られています)。前回の daq_run() で後段のコンポーネントにデータが送れている場合はエミュレータからデータ読みだしを行います。読み出したデータは SampleReader.h で定義された

m_data 配列に入るように read_data_from_detector() を書きます。この関数はこのあと取り上げます。データが読めれば read_data_from_detector() は読んだバイト数を返す (ように read_data_from_detector() を書く) ので 13 行目で戻り値を調べています。16 行目の set_data() は、コンポーネント間ヘッダ、フッタ、データを OutPort バッファにセットする関数で、その実装はこのあと解説します。

(20 行目) write_OutPort() は後段コンポーネントにデータを送る関数でこの実装もこのあと取り上げます。後段コンポーネントにデータを送れたら inc_sequence_num() を使ってこのコンポーネントが保持しているシーケンス番号を 1 増やします。また inc_total_data_size() を使って、このコンポーネントが保持している、いままで取り扱ったイベントデータバイト数を増やします。inc_sequence_num()、inc_total_data_size() およびコンポーネントが保持しているシーケンス番号等のデータについては「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

read_data_from_dataectors()

```

1 int SampleReader::read_data_from_detectors()
2 {
3     int received_data_size = 0;
4
5     /// write your logic here
6     /// read 1024 byte data from data server
7     int status = m_sock->readAll(m_data, SEND_BUFFER_SIZE);
8     if (status == DAQMW::Sock::ERROR_FATAL) {
9         std::cerr << "### ERROR: m_sock->readAll" << std::endl;
10        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
11    }
12    else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
13        std::cerr << "### Timeout: m_sock->readAll" << std::endl;
14        fatal_error_report(USER_DEFINED_ERROR2, "SOCKET TIMEOUT");
15    }
16    else {
17        received_data_size = SEND_BUFFER_SIZE;
18    }
19
20    return received_data_size;
21 }

```

上で述べたようにエミュレータからのデータ読みだしは read_data_from_detectors() で行っています。その仕様は

- 戻り値は読み取ったバイト数。
- 読み取ったデータは m_data メンバー変数が示す配列に入れる。
- データの読みだしに失敗した場合は致命的エラーとする。
- 読みだしタイムアウト (Sock ライブラリのデフォルトの 2 秒を採用) が起きた場合は致命的エラーとする。

です。

ソケット関連の読みだしは必ずしも指定したバイト数分読み出せると決まっているわけではありません (たとえば 100 バイト指定して読もうとしたとき 50 バイトしかデータが到着していなかった場合など)。指定したバイト数だけかならず読めるようにしたい場合は自分で関数を書く必要があります。このような関数は一般的によく使われるので DAQ-Middleware では Sock ライブラリで `readAll()` 関数を提供しています。`readAll()` の引数はふたつで、最初の引数にデータが読めた場合にデータを格納する配列を指定します。2 番目の引数に読むバイト数を指定します。`readAll()` の戻り値は

- 読み取りエラーが起きた場合は `DAQMW::Sock::ERROR_FATAL` を返す
- 読み取りタイムアウト (デフォルトで 2 秒) が起きた場合は `DAQMW::Sock::ERROR_TIMEOUT` を返す
- 正常に読みだしができた場合は読み出したバイト数を返す

となっています。7 行目で `readAll()` を使い、8 行目から 18 行目では戻り値によって、致命的エラーが起きていないかどうか調べています。致命的エラーが起きていなくて正常に読めた場合は上記 `read_data_from_detectors()` の仕様にしたがい読み出したバイト数を返しています (20 行目)。

`set_data()`

```

1 int SampleReader::set_data(unsigned int data_byte_size)
2 {
3     unsigned char header[8];
4     unsigned char footer[8];
5
6     set_header(&header[0], data_byte_size);
7     set_footer(&footer[0]);
8
9     ///set OutPort buffer length
10    m_out_data.data.length(data_byte_size + HEADER_BYTE_SIZE + FOOTER_BYTE_SIZE);
11    memcpy(&(m_out_data.data[0]), &header[0], HEADER_BYTE_SIZE);
12    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE]), &m_data[0], data_byte_size);
13    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE + data_byte_size]), &footer[0],
14           FOOTER_BYTE_SIZE);
15
16    return 0;
17 }

```

`set_data()` の部分は `newcomp` コマンドが作成した雛型と変更ありません。`set_data()` では後段コンポーネントに送るデータにコンポーネントヘッダ、およびフッタを追加しています*6。

まず `header` 配列、および `footer` 配列を確保します。つぎに `set_header()` を使って、ヘッ

*6 コンポーネント間のデータフォーマットは既に図 4 に示したとおりです。またその図のヘッダフォーマット、フッタフォーマットは図 5 にあるとおりです。ヘッダ、フッタフォーマット中の `DataByteSize` および `sequence number` は後段のコンポーネントが `check_header_footer()` を使って上流コンポーネントからのデータの受け取りで読み落としがなかったかどうかの検証を行うのに使用します。

ダの 32 ビット目から 63 ビット目にある `DataByteSize` をセットします。また `set_footer()` を使って `sequence number` をセットします。 `sequence number` は、プライベート変数 `m_loop` の値を使ってあらわに指定する必要はないので `set_footer()` の引数は `footer` バッファの先頭へのポインタのみです。10 行目の `m_out_data.data.length()` で引数に `OutPort` に書くバイト数を指定し `m_out_data.data` バッファを確保しています。続いてこのバッファに、さきほど作ったヘッダ、フッタ、およびデータを `memcpy()` を使ってコピーします (11 行目から 14 行目)。

`write_OutPort()`

```

1 int SampleReader::write_OutPort()
2 {
3     //////////////////////////////////////////////////// send data from OutPort //////////////////////////////////////
4     bool ret = m_OutPort.write();
5
6     //////////////////////////////////////////////////// check write status //////////////////////////////////////
7     if (ret == false) { // TIMEOUT or FATAL
8         m_out_status = check_outPort_status(m_OutPort);
9         if (m_out_status == BUF_FATAL) { // Fatal error
10            fatal_error_report(OUTPORT_ERROR);
11        }
12        if (m_out_status == BUF_TIMEOUT) { // Timeout
13            return -1;
14        }
15    }
16    m_out_status = BUF_SUCCESS;
17    return 0; // successfully done
18 }

```

`m_OutPort` の `write()` メソッドでアウトポートにデータを書きます。書くデータは `set_data()` で `m_out_data.data` 配列に格納されたデータです。 `false` が返ってきたら、 `check_outPort_status(m_OutPort)` でアウトポートの状態を調べます。致命的エラーであれば `fatal_error_report()` を使って `DaqOperator` に致命的エラーが起きたことを通知します。タイムアウトであった場合には -1 を返して呼び出し側にタイムアウトしたことを知らせます。 `write()` が成功した場合には `m_out_status` に `BUF_SUCCESS` をセットして終了します。

`daq_stop()`

```

1 int SampleReader::daq_stop()
2 {
3     std::cerr << "*** SampleReader::stop" << std::endl;
4
5     if (m_sock) { // 追加
6         m_sock->disconnect(); // 追加
7         delete m_sock; // 追加
8         m_sock = 0; // 追加
9     } // 追加
10
11    return 0;
12 }

```

ここでは Sock ライブラリの `disconnect()` メソッドを使ってエミュレータから切断しています。システムによっては、データ読み取りソケットを作るまえに `daq_stop()` に移行するようにプログラムする場合がありますのでその対応のため `m_sock` の値を見てソケットを作ったかどうかを判定しています*7 (SampleReader ではなくても動作はしますが今後、これをもとに変更を加えることを考えてこのようにしてあります)。

`daq_pause()` および `daq_resume()` `daq_pause()` および `daq_resume()` には変更がありません。

10.3 Makefile の変更

SampleReader では Sock ライブラリを使用することにしましたので Makefile でライブラリファイルの位置を指定する必要があります。以下のように Makefile を変更します。

```
SRCS += $(COMP_NAME).cpp
SRCS += $(COMP_NAME)Comp.cpp

# 以下の行を追加
LDLIBS += -L/usr/lib/daqmw -lSock
```

Sock ライブラリのインクルードファイルは `/usr/include/daqmw/Sock.h` にあります。本来ですとこの場所は標準の場所ではありませんので `CPPFLAGS` に `-I/usr/include/daqmw` を追加するなどの対処が必要になりますが、この追加は Makefile 中、最後に `include` されている `/usr/share/daqmw/comp.mk` で追加されているので行う必要はありません。なお、たとえば `ROOT` のように `/usr/local/root/include` を見る必要があるなど `/usr/include/daqmw` 以外の非標準な場所にあるインクルードファイルを読ませる必要がある場合はそのように Makefile に書く必要がありますのでご注意ください (たとえば次節の SampleMonitor はそうなっています)。

これで `make` してエラーになる場合はエラーメッセージを見て対処します。

11 SampleMonitor コンポーネントの開発

続いて SampleReader からデータを受け取ってヒストグラムし画面上に表示する SampleMonitor コンポーネントの開発に移ります。ここではヒストグラムを書くツールとして `ROOT`*8 を使用します。

SampleReader 同様 `newcomp` コマンドで雛型ファイルを作成します。今度は Sink 型コンポーネントの作成ですので `-t sink` を指定します。

```
% cd (ホームディレクトリに移動する)
% mkdir MyDaq (作ってなければ作る。この名前でもよい)
```

*7 たとえば、データ読み取りを開始するまえに SiTCP slow control パケットを投げる必要がある場合を考えます。そのコードを `daq_start()` 内のデータ読み取りソケット作成前に書き、slow control に失敗したらソケットを作るまえに `fatal_error_report()` を実行するようにプログラムする場合があります。

*8 <http://root.cern.ch/>

11 SAMPLEMONITOR コンポーネントの開発

```
% cd MyDaq
% newcomp -t sink SampleMonitor (-t sink を指定して newcomp を実行)
% cd SampleMonitor
% ls                               (できたファイルを確認する)
Makefile SampleMonitor.cpp SampleMonitor.h SampleMonitorComp.cpp
% make                             (make してみて開発環境が OK かどうか確認する)
rm -fr autogen
mkdir autogen
(中略)
% ls
Makefile          SampleMonitor.h SampleMonitorComp*   SampleMonitorComp.o
SampleMonitor.cpp SampleMonitor.o  SampleMonitorComp.cpp autogen
% make clean
```

ヒストグラムの仕様はここではヒストグラムの最小値は 0、最大値は 1000、ビンの数は 100 にします。

このコンポーネントはだいたい以下のような動作をします。

1. 上流コンポーネントからデータを読む。
2. コンポーネントヘッダ、フッタを使って読み落としがないかどうか確認する。
3. 複数あるイベントデータをデコードする。
4. イベントデータを ROOT ヒストグラムデータにインクリメントする。
5. ヒストグラム図をアップデートする条件がととのっていればアップデートする。
6. 以下くりかえし。

この SampleMonitor コンポーネントではヒストグラム図をアップデートできる条件として、何回 `daq_run()` がまわったかを `get_sequence_num()` で取得し、一定回数ごとにヒストグラム図をアップデートすることにしました。

11.1 SampleData.h の作成

今回のエミュレータからのデータフォーマットは比較的、単純であるため、データ構造を構造体として定義しなくても取り扱うことが可能かもしれませんが、将来の拡張のため^{*9}イベントデータフォーマットの構造体を定義しておきます。ここでは `SampleData.h` という名前のファイルを新しく作り、以下のように定義しました。エミュレータからのイベントデータフォーマットについては第 9.3 節の図 7 を参照してください。

```
#ifndef SAMPLEDATA_H
#define SAMPLEDATA_H

const int ONE_EVENT_SIZE = 8;

struct sampleData {
    unsigned char magic;
    unsigned char format_ver;
};
```

*9 および他人 (2 か月後の自分を含む) のため

```

    unsigned char module_num;
    unsigned char reserved;
    unsigned int data;
};
#endif

```

11.2 SampleMonitor.h の変更

SampleMonitor.h を以下のように変更します。

インクルードファイル

```

1 #include <arpa/inet.h> // 追加 for ntohl()
2
3 /////////////// ROOT Include files ///////////////
4 #include "TH1.h" // 追加
5 #include "TCanvas.h" // 追加
6 #include "TStyle.h" // 追加
7 #include "TApplication.h" // 追加
8
9 #include "SampleData.h" // 追加

```

1 行目の<arpa/inet.h>は `ntohl()` 関数用です。3 行目から 7 行目はヒストグラムを作るのに ROOT を使用するのでそれ様のインクルードファイルです。9 行目の `SampleData.h` に前節で見たようにエミュレータからのでデータフォーマットが構造体として定義されています。

変数およびメソッド

```

1 int decode_data(const unsigned char* mydata); // 追加
2 int fill_data(const unsigned char* mydata, const int size); // 追加
3
4 BufferStatus m_in_status;
5
6 /////////////// ROOT Histogram ///////////////
7 TCanvas *m_canvas; // 追加
8 TH1F *m_hist; // 追加
9 int m_bin; // 追加
10 double m_min; // 追加
11 double m_max; // 追加
12 int m_monitor_update_rate; // 追加
13 unsigned char m_recv_data[4096]; // 追加
14 unsigned int m_event_byte_size; // 追加
15 struct sampleData m_sampleData; // 追加

```

1 行目の `decode_data()` はデータをデコードするメソッドです。2 行目の `fill_data()` は ROOT ヒストグラムデータにデータをフィルするメソッドです。7 行目から 11 行目はヒストグラム用の変数で、ヒストグラム図は 7 行目で定義した `m_canvas` 上に描画されます。このモニターではヒストグラム図を更新するタイミングとして `daq_run()` を実行した回数をもとにすることに決

めました。具体的には 12 行目の `m_monitor_update_rate` 変数で指定した回数ぶん `daq_run()` が走ったときにヒストグラム図をアップデートしています。13 行目の `m_recv_data` は上流コンポーネントから送られてきたデータのうちコンポーネントヘッダ、フッタをとりのぞいたデータを格納するバッファです。14 行目の `m_event_byte_size` は上流コンポーネントからのデータについて 1 回の読みだしで何バイト読めたかを保持する変数です*¹⁰。15 行目の `m_sampleData` はエミュレータからのデータのフォーマットを定義した構造体です。デコードしたデータをここに入れてヒストグラムデータヘインクリメントする際はこの変数の値を使って行います。

11.3 SampleMonitor.cpp の変更

変数の初期化

```

1 SampleMonitor::SampleMonitor(RTC::Manager* manager)
2   : DAQMW::DaqComponentBase(manager),
3     m_InPort("samplemonitor_in", m_in_data),
4     m_in_status(BUF_SUCCESS),
5     m_canvas(0),           // 追加
6     m_hist(0),            // 追加
7     m_bin(0),             // 追加
8     m_min(0),             // 追加
9     m_max(0),             // 追加
10    m_monitor_update_rate(30), // 追加
11    m_event_byte_size(0),   // 追加
12    m_debug(false)

```

使用する変数の初期化を行っています。

daq_dummy()

```

1 int SampleMonitor::daq_dummy()
2 {
3     if (m_canvas) {           // 追加
4         m_canvas->Update(); // 追加
5         // daq_dummy() will be invoked again after 10 msec.
6         // This sleep reduces X servers' load.
7         sleep(1);           // 追加
8     }                       // 追加
9
10    return 0;
11 }

```

CONFIGURED ステートに移行した場合 (stop の指令を出したあとなど) に ROOT 以外の window を ROOT の canvas 上に移動させて、またもとに戻した場合そのままヒストグラムが消えたままになってしまうので、CONFIGURED ステートでも定期的にヒストグラムを描画するようにしています。ここでやっているのは、すでに作られたヒストグラム図の描画のみで、ヒスト

*¹⁰ SampleReader は 1024 バイト送ってくるのでここで読めるバイト数はいつも 1024 バイトなのですが、今後の拡張のために変数を準備しました。

グラム図のアップデートは行っていません。

daq_unconfigure()

```

1 int SampleMonitor::daq_unconfigure()
2 {
3     std::cerr << "*** SampleMonitor::unconfigure" << std::endl;
4     if (m_canvas) {          // 追加
5         delete m_canvas;    // 追加
6         m_canvas = 0;       // 追加
7     }                       // 追加
8
9     if (m_hist) {           // 追加
10        delete m_hist;      // 追加
11        m_hist = 0;         // 追加
12    }                       // 追加
13    return 0;
14 }

```

ここではヒストグラム図を書く canvas とヒストグラムデータを delete しています。

daq_start()

```

1 /////////////////////////////////////////////////// CANVAS FOR HISTOS ///////////////////////////////////////////////////
2 if (m_canvas) {          // 追加
3     delete m_canvas;    // 追加
4     m_canvas = 0;       // 追加
5 }                       // 追加
6 m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400); // 追加
7
8 /////////////////////////////////////////////////// HISTOS ///////////////////////////////////////////////////
9 if (m_hist) {           // 追加
10    delete m_hist;      // 追加
11    m_hist = 0;         // 追加
12 }                       // 追加
13
14 int m_hist_bin = 100;   // 追加
15 double m_hist_min = 0.0; // 追加
16 double m_hist_max = 1000.0; // 追加
17
18 gStyle->SetStatW(0.4);  // 追加
19 gStyle->SetStatH(0.2);  // 追加
20 gStyle->SetOptStat("em"); // 追加
21
22 m_hist = new TH1F("hist", "hist", m_hist_bin, m_hist_min, m_hist_max); // 追加
23 m_hist->GetXaxis()->SetNdivisions(5); // 追加
24 m_hist->GetYaxis()->SetNdivisions(4); // 追加
25 m_hist->GetXaxis()->SetLabelSize(0.07); // 追加
26 m_hist->GetYaxis()->SetLabelSize(0.06); // 追加

```

ここではヒストグラム変数 m_canvas および m_hist の設定を行っています。

m_canvas も m_hist も stop のあともう一度 start する場合に備えて値が設定されているかどうか

か調べて、設定されていなければいったん delete して 0 で初期化した後、new しています*¹¹。14~16 行目でヒストグラムのパラメータを指定しています。18~26 行目で ROOT のコマンドを使ってヒストグラム図のパラメータを指定しています。この詳細は ROOT のマニュアルを見てください。

daq_run()

```

1 int SampleMonitor::daq_run()
2 {
3     if (m_debug) {
4         std::cerr << "*** SampleMonitor::run" << std::endl;
5     }
6
7     unsigned int recv_byte_size = read_InPort();
8     if (recv_byte_size == 0) { // Timeout
9         return 0;
10    }
11
12    check_header_footer(m_in_data, recv_byte_size); // check header and footer
13    m_event_byte_size = get_event_size(recv_byte_size); // 変更
14
15    //////////// Write component main logic here. ////////////
16    memcpy(&m_recv_data[0], &m_in_data.data[HEADER_BYTE_SIZE], m_event_byte_size); // 追加
17
18    fill_data(&m_recv_data[0], m_event_byte_size); // 追加
19
20    if (m_monitor_update_rate == 0) { // 追加
21        m_monitor_update_rate = 1000; // 追加
22    } // 追加
23
24    unsigned long sequence_num = get_sequence_num(); // 追加
25    if ((sequence_num % m_monitor_update_rate) == 0) { // 追加
26        m_hist->Draw(); // 追加
27        m_canvas->Update(); // 追加
28    } // 追加
29    //////////////////////////////////////
30    inc_sequence_num(); // increase sequence num.
31    inc_total_data_size(m_event_byte_size); // 変数名変更 increase total data byte size
32
33    return 0;
34 }

```

(7~10 行目) 後述する read_InPort() を使って Inport にあるデータの読みだしを試みます (読めた場合は、読んだデータは m_in_data.data 配列に入ります)。read_InPort() の戻り値は

- timeout が起きた場合は 0 を返す
- データが読めた場合は読めたバイト数を返す

ように実装しています。データが読めた場合のバイト数にはコンポーネントヘッダとフッタの分も含まれています。なおこの解説書が取り扱う範囲では read_InPort() は newcomp -t sink で

*¹¹ daq_stop() でも上に見るように、念のため delete のあと 0 にしています

きる雛型ファイルから変更する点はありません。

(12 行目) データが読めた場合は、まず `check_header_footer()` でシーケンス番号に矛盾がないかどうか確認します。`check_header_footer()` で異常が見付かった場合は `fatal_error_report()` で `DaqOperator` にエラーを通知して、コンポーネントはアイドル状態に遷移するようになっています。

(13 行目) `get_event_size()` 関数を使って、イベントデータのバイト数を取得しています。

(16 行目) `read_InPort()` でデータが読めた場合、データは `m_in_data.data` 配列に入っていますので、デコードのためにイベントデータ列のみを `memcpy()` でコピーしています。

(18 行目) ヒストグラムデータに fill する関数 `fill_data()` を使ってインクリメントしています。`fill_data()` の実装は後述します。

(20~28 行目) このモニターではヒストグラム図は、データを受け取れた `daq_run()` の回数をもとにヒストグラム図をアップデートするタイミングを決めています。24 行目で `get_sequence_num()` で自身がもっているシーケンス番号をとりだして、25 行目で `m_monitor_update_rate` で指定された回数ごとにヒストグラム図をアップデートし、画面に描画しています。割算をするので念のため 20 行めの `if` ブロックで 0 でないかどうか確認しています。

(29~30 行目) シーケンス番号をインクリメントし、また取り扱ったイベントバイト数をインクリメントしています。シーケンス番号、`get_sequence_num()` および `inc_total_data_size()` の意味は「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

`read_InPort`

```

1 unsigned int SampleMonitor::read_InPort()
2 {
3     //////////////// read data from InPort Buffer ////////////////
4     unsigned int recv_byte_size = 0;
5     bool ret = m_InPort.read();
6
7     //////////////// check read status ////////////////
8     if (ret == false) { // false: TIMEOUT or FATAL
9         m_in_status = check_inPort_status(m_InPort);
10        if (m_in_status == BUF_TIMEOUT) { // Buffer empty.
11            if (check_trans_lock()) { // Check if stop command has come.
12                set_trans_unlock(); // Transit to CONFIGURE state.
13            }
14        }
15        else if (m_in_status == BUF_FATAL) { // Fatal error
16            fatal_error_report(INPORT_ERROR);
17        }
18    }
19    else {
20        recv_byte_size = m_in_data.data.length();
21    }
22
23    if (m_debug) {
24        std::cerr << "m_in_data.data.length():" << recv_byte_size
25                << std::endl;
26    }
27

```

```

28     return recv_byte_size;
29 }

```

この関数では InPort からデータ読みだしを試みています。この関数は `newcomp -t sink` ができる関数そのままになっていて変更する点はありません。以下コードの解説をします。

(5 行目) `read()` メソッドを使って InPort からデータ読みだしを試みます。読んだデータは `m_in_data.data` 配列に入ります。続いて `read()` で読んだ結果を調べます。

(8~14 行目) `read()` が `false` を返した場合、`check_inPort_status()` で InPort の状態を調べます。`BUF_TIMEOUT` が返ってきた場合は読むべきデータがなかったという意味です。この場合は `check_trans_lock()` で STOP コマンドがきていたかどうか調べて、STOP コマンドがやってきた場合には CONFIGURE ステートに戻るようになっています。

(15~17 行目) `check_inPort_status()` が `BUF_FATAL` を返してきたら致命的エラーが起きたと判断することにして、`fatal_error_report()` で DaqOperator にエラーをなげています。`fatal_error_report()` を読んだ結果コンポーネント自身はアイドル状態に移行します。

(19 行目~21 行目) `read()` メソッドが `true` を返した場合には正常にデータが読めたという意味ですので、`m_in_data.data.length()` メソッドを使って読めたデータ長 (単位はバイト) を取得しています。

(28 行目) 読めたデータの長さ、あるいは正常動作しているがデータがなかった場合は 0 を返してこの関数は終了しています。

`fill_data()`

```

1 // この関数全体を追加
2 int SampleMonitor::fill_data(const unsigned char* mydata, const int size)
3 {
4     for (int i = 0; i < size/(int)ONE_EVENT_SIZE; i++) {
5         decode_data(mydata);
6         float fdata = m_sampleData.data/1000.0; // 1000 times value is received
7         m_hist->Fill(fdata);
8
9         mydata+=ONE_EVENT_SIZE;
10    }
11    return 0;
12 }

```

デコードしたデータをヒストグラムに fill するルーチンです。引数はデータバイト列へのポインタと、データバイト列の長さです。データバッファの先頭から 1 イベントサイズ (エミュレータの場合は `ONE_EVENT_SIZE = 8` バイト) ごとにスキャンして行ってイベントデータを取り出しています^{*12}。とりだしたデータは順次 ROOT の `Fill()` を使ってヒストグラムデータにフィルしています。

^{*12} ここでは使っていませんが、データ構造体を定義して、バイト列を構造体でフィットしてスキャンする方法をとる場合は、構造体にはアライメントの問題があるのを認識しておく必要があります。

decode_data()

```

1 int SampleMonitor::decode_data(const unsigned char* mydata) // この関数全体を追加
2 {
3     m_sampleData.magic      = mydata[0];
4     m_sampleData.format_ver = mydata[1];
5     m_sampleData.module_num = mydata[2];
6     m_sampleData.reserved   = mydata[3];
7     unsigned int netdata    = *(unsigned int*)&mydata[4];
8     m_sampleData.data       = ntohl(netdata);
9
10    if (m_debug) {
11        std::cerr << "magic: "      << std::hex << (int)m_sampleData.magic      << std::endl;
12        std::cerr << "format_ver: " << std::hex << (int)m_sampleData.format_ver << std::endl;
13        std::cerr << "module_num: " << std::hex << (int)m_sampleData.module_num << std::endl;
14        std::cerr << "reserved: "   << std::hex << (int)m_sampleData.reserved   << std::endl;
15        std::cerr << "data: "       << std::dec << (int)m_sampleData.data       << std::endl;
16    }
17
18    return 0;
19 }

```

データのデコードをする関数を decode_data としてまとめてみました。ここではデコードしたデータはメンバー変数 m_sampleData に入れています。

daq_stop()

```

1 int SampleMonitor::daq_stop()
2 {
3     std::cerr << "*** SampleMonitor::stop" << std::endl;
4
5     m_hist->Draw();      // 追加
6     m_canvas->Update(); // 追加
7
8     reset_InPort();
9
10    return 0;
11 }

```

5 行目と 6 行目を追加しました。この目的は daq_stop 時にそれまでインクリメントされたデータを元にヒストグラムを書きなおすことです。これで stop したときにヒストグラム図中の Entries の数と DaqOpertor が端末上に表示するイベントバイト数 ÷ 8(1 イベントバイト数) があうことになります。

11.4 SampleMonitorComp.cpp の変更

SampleReader コンポーネントでは main() 関数がある SampleReaderComp.cpp の変更は必要ありませんでしたが、SampleMonitor ではヒストグラムを作るのに ROOT を使用したので main() 関数で TApplication オブジェクトを作成する必要があります。以下のように SampleMonitorComp.cpp を変更します。

12 起動および動作確認

```
1 int main (int argc, char** argv)
2 {
3     RTC::Manager* manager;
4     manager = RTC::Manager::init(argc, argv);
5
6     // for root application
7     TApplication theApp("App", &argc, argv); // 追加
8
9     // Initialize manager
10    manager->init(argc, argv);
```

11.5 Makefile の変更

このコンポーネントではヒストグラム化に ROOT を使いますのでそのインクルードファイル、およびライブラリの位置をコンパイラに教える必要があります。Makefile を以下のように書き換えます。まず Makefile の先頭に

```
ifndef ROOTSYS
$(error This program requires ROOTSYS environment variable\
but does not defined. Please define ROOTSYS as follows at\
shell prompt: "export ROOTSYS=/usr/local/root". If you don't install\
ROOT in /usr/local/root, please substitute your ROOT root directory)
endif
```

を追加します。これは下の CPPFLAGS および LDLIBS 変数を設定するところで ROOT のユーティリティプログラム root-config の呼び出し中で ROOTSYS 環境変数を使っているためです。

CPPFLAGS、LDLIBS については以下の行を追加します。

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

root-config --glibs の方は LDLIBS で使えるように先頭に-L が付いた値が返りますので右辺の先頭に-L を付ける必要はありません。一方 root-config --incdir では-I を付ける必要があります。

これで make を実行し、SampleMonitorComp 実行形式ファイルができるかどうか確認します。

12 起動および動作確認

これでコンポーネント開発は終了しましたので、エミュレータからデータを読ませてヒストグラムを画面上に表示してみましょう。

第5節で述べたように

```
/home/daq/MyDaq
/home/daq/MyDaq/SampleReader
```

```
/home/daq/MyDaq/SampleMonitor
```

というディレクトリ構造をもとに起動方法について説明します。

「DAQ-Middleware 1.1.0 技術解説書」[2] に書かれているとおり DAQ-Middleware では DAQ システムの統括は DaqOperator が行います。既に起動されているコンポーネントの接続、データ収集の開始、終了の指示は DaqOperator が行います。各コンポーネントをブートする方法には手で起動する、xinetd を使ってネットワークブートを行うなどの方法があります。ここでは DAQ-Middleware に含まれている /usr/bin/run.py コマンドのローカルブート機能を使ってブートを行います。

DAQ-Middleware では XML 文書による DAQ システムのコンフィギュレーションが可能です。詳細は「DAQ-Middleware 1.1.0 技術解説書」[2] を御覧ください。ここでは /usr/share/daqmw/conf/sample.xml をコピーして使います。

```
% cd
% pwd
/home/daq/MyDaq
% cp /usr/share/daqmw/conf/sample.xml .
```

この文書で仮定しているディレクトリ構造になっている場合は変更する点はありませんが、このようになっていない場合は以下の点を変更することが必要です。

- 2 箇所ある execPath をコンポーネントのファイルのパス名に置き換える

/usr/share/daqmw/conf/sample.xml のコード部分を以下に示します。

```
1 <configInfo>
2   <daqOperator>
3     <hostAddr>127.0.0.1</hostAddr>
4   </daqOperator>
5   <daqGroups>
6     <daqGroup gid="group0">
7       <components>
8         <component cid="SampleReader0">
9           <hostAddr>127.0.0.1</hostAddr>
10          <hostPort>50000</hostPort>
11          <instName>SampleReader0.rtc</instName>
12          <execPath>/home/daq/MyDaq/SampleReader/SampleReaderComp</execPath>
13          <confFile>/tmp/daqmw/rtc.conf</confFile>
14          <startOrd>2</startOrd>
15          <inPorts>
16          </inPorts>
17          <outPorts>
18            <outPort>samplerreader_out</outPort>
19          </outPorts>
20          <params>
21            <param pid="srcAddr">127.0.0.1</param>
22            <param pid="srcPort">2222</param>
23          </params>
24        </component>
25      </components>
26    </daqGroup>
27  </daqGroups>
28 </configInfo>
```



```

26         <hostAddr>127.0.0.1</hostAddr>
27         <hostPort>50000</hostPort>
28         <instName>SampleMonitor0.rtc</instName>
29         <execPath>/home/daq/MyDaq/SampleMonitor/SampleMonitorComp</execPath>
30         <confFile>/tmp/daqmw/rtc.conf</confFile>
31         <startOrd>1</startOrd>
32         <inPorts>
33             <inPort from="SampleReader0:samplerreader_out">samplemonitor_in</inPort>
34         </inPorts>
35         <outPorts>
36         </outPorts>
37         <params>
38         </params>
39     </component>
40 </components>
41 </daqGroup>
42 </daqGroups>
43 </configInfo>

```

SampleReader コンポーネントは OutPort をひとつ持ちますので 17 行目の OutPorts でそれを指定しています。また SampleReader コンポーネントはパラメータとしてエミュレータの IP アドレスとポートを指定することにしましたので 20 行目の params でそれらを指定しています。SampleReader コンポーネントのソース (SampleReader.cpp) では parse_params() でここで指定された値を取得しています。SampleMonitor コンポーネントは InPort をひとつ持ちますので 32 行目の InPorts で指定しています。その他のタグについての詳細は「DAQ-Middleware 1.1.0 技術解説書」[2] を参照してください。

では起動してみます。

エミュレータ起動用端末を開いてエミュレータを起動しておきます。

```
% daqmw-emulator
```

以下のように run.py を起動します。

```

% cd /home/daq/MyDaq
% ls
SampleReader SampleMonitor emulator sample.xml
% run.py -c -l sample.xml

```

run.py のオプション-c は DaqOperator がコンソールモードで起動するオプションで、これを指定すると DaqOperator は定期的に各コンポーネントが取り扱ったデータバイト数を端末に表示します (各コンポーネントは定期的に DaqOperator に自身が処理したデータバイト数を報告しています)。run.py のオプション-l は sample.xml から起動するコンポーネントのパスを探し、そのパスにあるコンポーネントをローカル計算機で起動します。

run.py を起動してしばらく待つと (計算機の CPU 性能で差はありますがおおよそ 4 秒くらい)、のようになります。この文字を出力しているのは DaqOperator で、DaqOperator はこの状態でコマンドキー入力を待っています。利用できるコマンドは 1 行目の Command: と書いてある行に表

示されています。コマンド入力に対応する数字キーを押すことで行います。状態遷移はひとつづつ行う必要があります。たとえばこの状態で、start を押すと不適切な入力と判断されます。コマンドを入力すると DaqOperator は各コンポーネントに遷移命令を送ります。run.py 起動直後のこの状態でコンポーネントは状態遷移図 図 3 の UNCONFIGURED になっています。

```
Command:    0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume

RUN NO: 0
start at:      stop at:

GROUP:COMP_NAME      EVENT  SIZE      STATE      COMP STATUS
group0:SampleReader0:      0      0      LOADED      WORKING
group0:SampleMonitor0:      0      0      LOADED      WORKING
```

この状態で 0 を押ししばらく待つと CONFIGURED 状態に移行します。

```
Command:    0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume

RUN NO: 0
start at:      stop at:

GROUP:COMP_NAME      EVENT  SIZE      STATE      COMP STATUS
group0:SampleReader0:      0      0      CONFIGURED  WORKING
group0:SampleMonitor0:      0      0      CONFIGURED  WORKING
```

次に 1 を押すとランナンバーを聞いてきますので適当に 1 等の数字を入力します。これで DaqComponent が SampleReader と SampleMonitor に start の指示を出します。DaqOperator は各コンポーネントから報告された処理したデータバイト数を画面に表示し数秒に一度、更新します。またこのコードであたえた m_monitor_update_rate の値 (30) ではヒストグラム図は 4 秒に一度程度アップデートされます。終了するには 2 を押して各コンポーネントを STOP 状態に遷移させます。このときの画面の状態を図 9 に示します。

2 を押してコンポーネントを stop させたあとに Ctrl-C を押すと DaqOperator に SIGINT が送られて DaqOperator が終了します。DaqOperator と同時に run.py が起動したコンポーネントにも SIGINT が送られます (run.py から起動した DaqOperator、および各コンポーネントが同一プロセスグループに属しているため)。通常コンポーネントのほうが先に exit して DaqOperator は数回コンポーネントと接続しようとするので、Ctrl-C を押したあと画面に

```
### ERROR:    : cannot connect
### ERROR:    : cannot connect
```

という行がコンポーネントの数だけ表示されます。少し待って DaqOperator が終了します。

ヒストグラムの更新頻度は m_monitor_update_rate の値で指定されています。SampleMonitor.cpp 中の初期化の部分

```
1 SampleMonitor::SampleMonitor(RTC::Manager* manager)
2   : DAQMW::DaqComponentBase(manager),
3   m_InPort("samplemonitor_in", m_in_data),
```

12 起動および動作確認

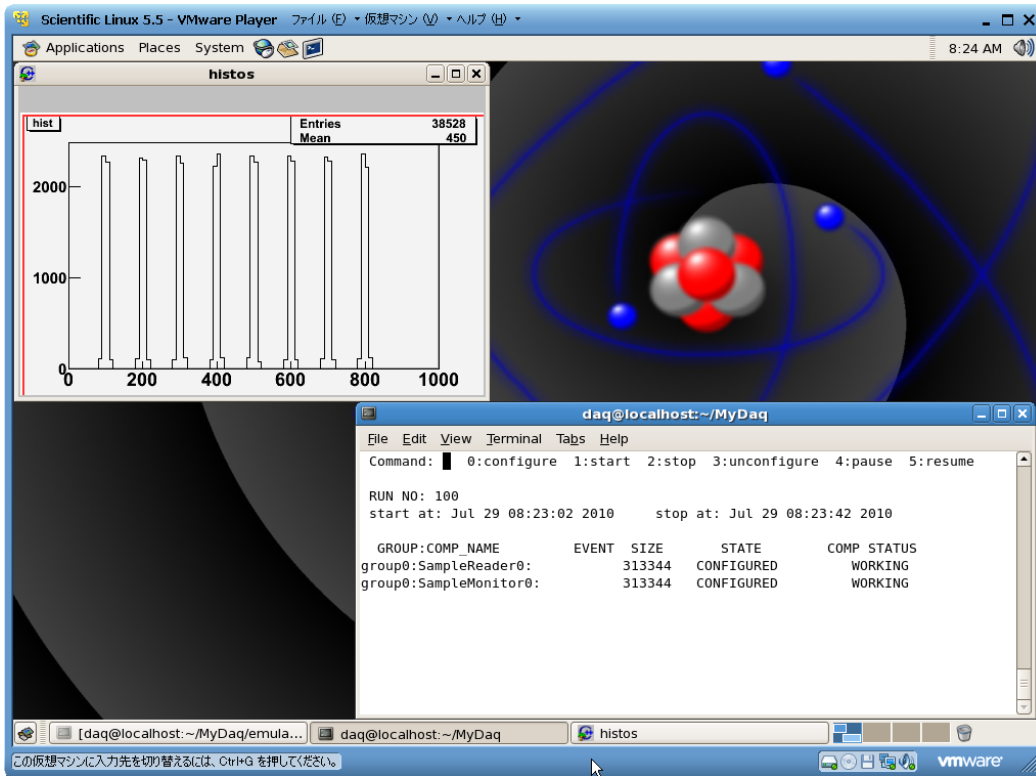


図9 データ収集終了した画面。エミュレータを起動した端末は最小化して下のパネルにおさめてある。

```
4 |     m_in_status(BUF_SUCCESS),
5 |     m_canvas(0),           // 追加
6 |     m_hist(0),            // 追加
7 |     m_bin(0),             // 追加
8 |     m_min(0),             // 追加
9 |     m_max(0),             // 追加
10 |    m_monitor_update_rate(30), // 追加
11 |    m_event_byte_size(0),   // 追加
12 |    m_debug(false)
```

の `m_monitor_update_date()` の括弧内の数値を小さくすると更新頻度があがります。このコードのままでは更新頻度を変更するにはソースコード自身を変更し、再コンパイルする必要があります (数値を変更しやってみてください)。再コンパイル無しで変更するには DAQ-Middleware では、次節の Condition データベースを使います。

13 パラメータの Condition データベース化

DAQ-Middleware ではラン毎に変わるようなパラメータをシステムにあてるために Condition データベースという枠組があります。前節の SampleMonitor コンポーネントでは

- ヒストグラムのビン数
- ヒストグラムの最小値
- ヒストグラムの最大値
- ヒストグラムを更新する頻度

が決め打ちになっていて、これらを変更するにはソースコードを変更する必要がありました。Condition データベースを使用するとソースコードの改変をしなくてもパラメータを変更することができるようになります。

Condition データベースについては別に解説文書「Condition データベースの開発マニュアル」[3] がありますのでこれを参照してください。ここではこのマニュアル中の「class を用いた実装」に沿って、上記ヒストグラムのパラメータをセットすることにします。パラメータをセットするタイミングは `daq_start()` 時に行うことにします。

`/usr/share/daqmw/examples/ConditionSampleMonitor` ディレクトリ以下に、これから述べる変更を行ったソースがあります。`/usr/share/daqmw/examples/SampleMonitor` を元に変更してありたとえ

```
% cd /usr/share/daqmw/examples
% diff -uprN SampleMonitor ConditionSampleMonitor
```

とすると変更点がわかるようになっていきます。また `condition.xml` のサンプルは `/usr/share/daqmw/conf/condition.xml` にあります。以下では前述の SampleMonitor のソースをコピーして Condition データベースを使えるように改造することにします。

```
% cd
% cd MyDaq
% cp -r SampleMonitor ConditionSampleMonitor
% cd ConditionSampleMonitor
```

新規に `ConditionSampleMonitor.h` と `ConditionSampleMonitor.cpp` ファイルを作成しこの中でパラメータを保持する変数およびパラメータを取得するクラス `ConditionSampleMonitor` クラスを作ります。また、DAQ コンポーネントが Condition データベースを読めるようにするためには `JsonSpirit` ライブラリ、および `boost_regex` ライブラリをリンクする必要があります。そこで増えたソースファイルの分もあわせて `Makefile` を変更します。変更点は

- `JsonSpirit` ライブラリ、および `boost_regex` ライブラリをリンクするようにする
- ソースファイル `ConditionSampleMonitor.cpp` が増えたのでこれを `SRCS` に追加する

13 パラメータの CONDITION データベース化

の 2 点です。

```
SRCS += ConditionSampleMonitor.cpp
ConditionSampleMonitor.o: ConditionSampleMonitor.h ConditionSampleMonitor.cpp
LDLIBS += -L/usr/lib/daqmw -lJsonSpirit -lboost_regex
```

パラメータのデータ構造は、ConditionSampleMonitor.h 内で構造体 monitorParam で定義することにします。

```
1 #ifndef _CONDITION_SAMPLEMONITOR_H
2 #define _CONDITION_SAMPLEMONITOR_H 1
3
4 #include <string>
5 #include "Condition.h"
6
7 struct monitorParam {
8     unsigned int hist_bin;
9     unsigned int hist_min;
10    unsigned int hist_max;
11    unsigned int monitor_update_rate;
12 };
13
14 typedef struct monitorParam monitorParam;
15
16 class ConditionSampleMonitor : public Condition {
17 public:
18     ConditionSampleMonitor();
19     virtual ~ConditionSampleMonitor();
20     bool initialize(std::string filename);
21     bool getParam(std::string prefix, monitorParam* monitorParam);
22 private:
23     Json2ConList m_json2ConList;
24     conList      m_conListSampleMonitor;
25 };
26
27 #endif
```

次に condition.json ファイルを読み monitorParam 構造体変数にセットするメソッドを ConditionSampleMonitor.cpp に追加します。

```
1 #include "ConditionSampleMonitor.h"
2
3 ConditionSampleMonitor::ConditionSampleMonitor() {}
4 ConditionSampleMonitor::~~ConditionSampleMonitor() {}
5
6 bool
7 ConditionSampleMonitor::getParam(std::string prefix, monitorParam* monitorParam)
8 {
9     setPrefix(prefix);
10    unsigned int hist_bin;
11    unsigned int hist_min;
12    unsigned int hist_max;
13    unsigned int monitor_update_rate;
14
15    if (find("hist_bin", &hist_bin)) {
16        monitorParam->hist_bin = hist_bin;
```

```

17     }
18     else {
19         std::cerr << prefix + " hist_bin not found" << std::endl;
20         return false;
21     }
22
23     if (find("hist_min", &hist_min)) {
24         monitorParam->hist_min = hist_min;
25     }
26     else {
27         std::cerr << prefix + " hist_min not found" << std::endl;
28         return false;
29     }
30
31     if (find("hist_max", &hist_max)) {
32         monitorParam->hist_max = hist_max;
33     }
34     else {
35         std::cerr << prefix + " hist_max not found" << std::endl;
36         return false;
37     }
38
39     if (find("monitor_update_rate", &monitor_update_rate)) {
40         monitorParam->monitor_update_rate = monitor_update_rate;
41     }
42     else {
43         std::cerr << prefix + " monitor_update_rate not found" << std::endl;
44         return false;
45     }
46
47     return true;
48 }
49
50 bool ConditionSampleMonitor::initialize(std::string filename)
51 {
52     if (m_json2ConList.makeConList(filename, &m_conListSampleMonitor) == false) {
53         std::cerr << "### ERROR: Fail to read the Condition file "
54             << filename << std::endl;
55     }
56     init(&m_conListSampleMonitor);
57     return true;
58 }

```

以上で ConditionSampleMonitor クラスの準備ができました。

続けて SampleMonitor 側で Condition データベースを使ってヒストグラムのパラメータを取得するようにします。まず、Condition データベースのファイル名 CONDITION_FILE、およびパラメータを保持する構造体 m_monitorParam を SampleMonitor.h に追加:

```

////////// ROOT Histogram //////////
TCanvas *m_canvas;
TH1F     *m_hist;
unsigned char m_recv_data[4096];
unsigned int m_event_byte_size;
struct sampleData m_sampleData;
////////// Condition database //////////
static const std::string CONDITION_FILE; // 追加

```

13 パラメータの CONDITION データベース化

```
    monitorParam m_monitorParam;          // 追加

    bool m_debug;
};
```

さらに SampleMonitor.cpp で CONDITION_FILE に値を代入します。

```
static const char* samplemonitor_spec[] =
{
    "implementation_id", "SampleMonitor",
    "type_name",        "SampleMonitor",
    "description",      "SampleMonitor component",
    "version",          "1.0",
    "vendor",           "Kazuo Nakayoshi, KEK",
    "category",         "example",
    "activity_type",    "DataFlowComponent",
    "max_instance",    "1",
    "language",         "C++",
    "lang_type",        "compile",
    ""
};

const std::string SampleMonitor::CONDITION_FILE = "./condition.json"; // 追加
```

つぎにこのクラスを使用するように DAQ コンポーネントのソースを変更します。

まず conditionSampleMonitor.h をインクルードするように SampleMonitor.cpp を変更します。

```
#include "SampleMonitor.h"
#include "ConditionSampleMonitor.h" // 追加
```

次に SampleMonitor.cpp で set_condition() 関数を追加し、ConditionSampleMonitor クラスを使ってパラメータを取得します。また daq_start() で set_condition() を呼ぶようにします。

```
// この関数全体を追加
int set_condition(std::string condition_file, monitorParam *monitorParam)
{
    ConditionSampleMonitor conditionSampleMonitor;
    conditionSampleMonitor.initialize(condition_file);
    if (conditionSampleMonitor.getParam("common_SampleMonitor_", monitorParam)) {
        std::cerr << "condition OK" << std::endl;
    }
    else {
        throw "SampleMonitor condition error";
    }

    return 0;
}

int SampleMonitor::daq_start()
{
    std::cerr << "*** SampleMonitor::start" << std::endl;
}
```

```

m_in_status = BUF_SUCCESS;

try {
    set_condition(CONDITION_FILE, &m_monitorParam); // 追加
} // 追加
catch (std::string error_message) { // 追加
    std::cerr << error_message << std::endl; // 追加
    fatal_error_report(USER_DEFINED_ERROR1, "Condition error"); // 追加
} // 追加
catch (...) { // 追加
    std::cerr << "unknown error" << std::endl; // 追加
    fatal_error_report(USER_DEFINED_ERROR1, "Unknown error"); // 追加
} // 追加

```

さらにヒストグラムのピンの数、最小値、最大値としてこの取得した値を使うように、TH1F()の引数を変更します。またヒストグラムを更新するタイミングを決めているところも変更します。

```

m_hist = new TH1F("hist", "hist",
                 m_monitorParam.hist_bin, // 引数変更
                 m_monitorParam.hist_min, // 引数変更
                 m_monitorParam.hist_max); // 引数変更

```

```

unsigned long sequence_num = get_sequence_num();
if ((sequence_num % m_monitorParam.monitor_update_rate) == 0) { // 変更
    m_hist->Draw();
    m_canvas->Update();
}

```

また SampleMonitor のコンストラクタから m_hist_bin 等の初期化をしている部分を削除します。

```

SampleMonitor::SampleMonitor(RTC::Manager* manager)
: DAQMW::DaqComponentBase(manager),
  m_InPort("samplemonitor_in", m_in_data),
  m_in_status(BUF_SUCCESS),
  m_canvas(0),
  m_hist(0),
  // m_hist_bin, m_hist_min, m_hist_max, m_monitor_update_rate の初期化を削除
  m_event_byte_size(0),
  m_debug(false)

```

13.1 Condition データベースを使ったヒストグラムのテスト

パラメーターの値を condition.xml で与えます。サンプルは/usr/share/daqmw/conf/condition.xml にありますのでこれを/home/daq/MyDaq ディレクトリにコピーします。

```

% cd /home/daq/MyDaq
% cp /usr/share/daqmw/conf/condition.xml .

```


13 パラメータの CONDITION データベース化

コンポーネントは直接、この xml ファイルを読むのではなく、JSON 形式に変換したファイル `condition.json` を読みます。JSON 形式への変換は `condition_xml2json` コマンドを使って行います。

```
% condition_xml2json condition.xml
```

このコマンドで `condition.json` ファイルができます^{*13}。起動は、Condition データベース化前と同様に `run.py -c -l sample.xml` コマンドで行います。コンポーネント実行ファイルのパスが変わりました (ディレクトリが `SampleMonitor` から `ConditionSampleMonitor` に変わったので `sample.xml` ファイルの `execPath` を起動するコンポーネントのフルパスに変更します。以下に `sample.xml` の変更点を示します。

```
<component cid="SampleMonitor0">
  <hostAddr>127.0.0.1</hostAddr>
  <hostPort>50000</hostPort>
  <instName>SampleMonitor0.rtc</instName>
  <!-- execPath changed -->
  <execPath>/home/daq/MyDaq/ConditionSampleMonitor/SampleMonitorComp</execPath>
  <!-- execPath changed ~~~~~ -->
  <confFile>/tmp/daqmw/rtc.conf</confFile>
  <startOrd>1</startOrd>
  <inPorts>
    <inPort from="SampleReader0:samplerreader_out">samplemonitor_in</inPort>
  </inPorts>
  <outPorts>
  </outPorts>
  <params>
  </params>
</component>
```

起動は以前と同様 `run.py -c -l sample.xml` で行います^{*14}。ヒストグラムビン数、最小値、最大値等が上の `condition.xml` の値になっていることを確認してください。また `condition.xml` のパラメータを変更後、再び `condition_xml2json condition.xml` で `condition.json` をアップデートし、コンポーネントを起動させて、ヒストグラムが Condition ファイルで指定した値になっていることを確認してください。

ヒストグラムビン数 100、最小値 0、最大値 150 と Condition データベースでセットした場合の例を図 10 に示します。DaqOperator が端末に示したバイト数 423936 バイトから、全部で 52992 個のイベントデータを収集したことがわかります。エミュレータからのデータは 100、200、…、800 を中心としたデータを均等に送ってきています。全データ数 52992 個のうち 8 分の 1 の 6624 個のデータがヒストグラムにインクリメントされたことが図中左側ヒストグラム図の Entries の欄で確認することができます。

*13 このコマンドはシェルスクリプトでその内部で `Xalan` コマンドを使っていますので `xalan` パッケージが必要です。

*14 あるいは `sample.xml` は以前のままにして `sample.xml` を `conditionsample.xml` にコピーして、上記の `execPath` の変更を行い `run.py -c -l conditionsample.xml` として起動する手もあります。

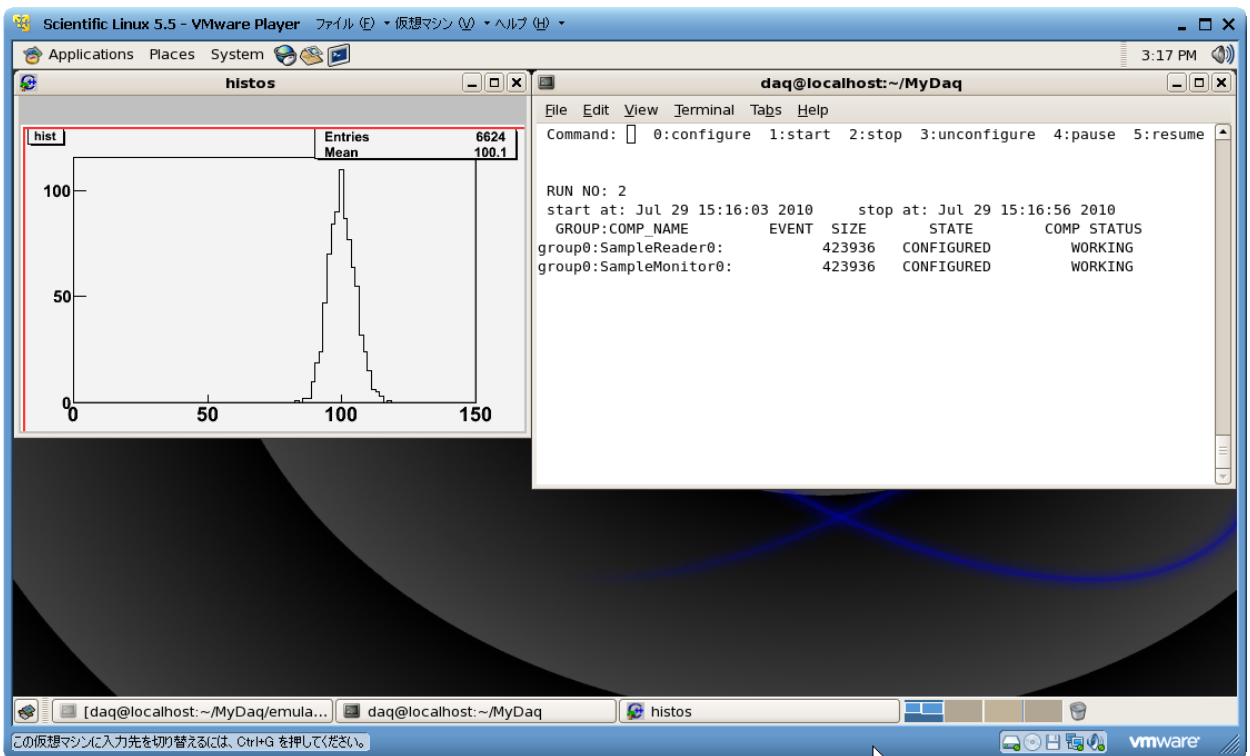


図 10 Condition データベースを使ってヒストグラムの最小値を 0、最大値を 150、ビン数を 100 としてデータを読み取った例。ヒストグラム図中の Entries の数値については本文を参照。

14 WebUI の使い方

いままでは DaqOperator をコンソールモードで起動して、DaqOperator に対してキーボードから指示を出していました。DAQ-Middleware 1.1.0 からは、WebUI が追加されました。この節ではその使い方について説明します。

DAQ-Middleware 1.1.1 までは使用できる Web ブラウザは firefox のみでした。DAQ-Middleware 1.1.2 から IE、firefox、chrome、safari が使えます。ただし IE の場合は図 11 の一番下の “XML Response from DaqOperator” の欄に「XML 文書が表示できません」と表示され、DaqOperator から返ってきた XML が表示されません (ボタンを押して操作を行うことはできます)。

14.1 ソフトウェアパッケージの確認

WebUI を使うには Scientific Linux 5.x では mod_python パッケージが、その他の Linux distribution (Scientific Linux 6.x を含む) では mod_wsgi パッケージが必要です。既にインストールされているかどうかを確認するには

```
% rpm -q mod_python あるいは rpm -q mod_wsgi
```

で確認します。“package mod_python is not installed” といわれたらインストールされていないので次のコマンドでインストールします。

```
% su (ルートのパスワードを答える)
# yum install mod_python あるいは yum install mod_wsgi
```

mod_python あるいは mod_wsgi をインストール直後は httpd がそれらを使えるようになっていないので httpd を再起動します。

```
# service httpd restart
```

httpd がすでに起動していなかった場合は “Stopping httpd: [FAILED]” と出ますが、”Starting httpd: [OK]” と出れば OK です。

リブート時に自動的に httpd が起動するようにするには

```
# chkconfig httpd on
```

とします。

また SELinux が有効になっていると WebUI が動かないので SELinux を無効にしておく必要があります。有効になっているかどうかの検査方法、無効化する方法については付録 F.7 節をご覧ください。

14.2 操作方法

コンポーネントは `-c` なしで `run.py -l` として起動します。

```

1 [daq@localhost MyDaq]$ run.py -l sample.xml
2 Use config file sample.xml
3 Use /usr/share/daqmw/conf/config.xsd for XML schema
4 Use /usr/libexec/daqmw/DaqOperatorComp for DAQ-Operator
5 Conf file validated: sample.xml
6 start new naming service... done
7 Local Comps booting... done
8 Now booting the DAQ-Operator... done
9 [daq@localhost MyDaq]$

```

必要なコンポーネントが全て起動したかどうかは `pgrep` コマンドで確認できます。

```

1 [daq@localhost MyDaq]$ pgrep -fl Comp
2 6189 /home/daq/MyDaq/SampleReader/SampleReaderComp -f /tmp/daqmw/rtc.conf
3 6208 /home/daq/MyDaq/SampleMonitor/SampleMonitorComp -f /tmp/daqmw/rtc.conf
4 6217 /usr/libexec/daqmw/DaqOperatorComp -f ./rtc.conf -h 127.0.0.1 -p 9876 -x sample.xml
5 [daq@localhost MyDaq]$

```

先頭に表示されている数字はプロセス ID です。この例題システムで必要なコンポーネント (SampleReader、SampleMonitor、DaqOperator) が全て起動していることがわかります。

ここで Web ブラウザを起動し (上側パネルの “System” の右側のアイコンをクリックすると起動します)、<http://localhost/daqmw/operatorPanel/operatorPanel0.html> にアクセスします。このときの画面の状態を図 11 にしめします。

Web 画面上、上から Current Status、Run Number 入力欄、DAQ ボタン、DAQ status、DaqOperator からの XML レスポンスが表示されています。

Run Number は一回のランが終了すると自動でインクリメントされます。また手動で番号を指定することもできます。番号欄に数字を入れて “save runNumber” ボタンを押すとセーブされます。

DaqOperator への指示は DAQ ボタンを押すことで行います。そのときの state によって押せるボタンが決まっています。緑色になっているボタンが現在、押すことが可能なボタンです。

DAQ status の欄は、コンポーネント名、現在の State、Status、およびイベント数が表示されます。

この状態で Configure ボタンを押し、続けて Begin を押すとデータ収集が開始され、ヒストグラム図が表示されます。このときの画面を図 12 に示します。終了するには End ボタンを押します。続けて Begin ボタンを押すと次のランのデータ収集が始まります。End ボタンに続いて Unconfigure ボタンを押すと LOADED ステートに移行し、DAQ Status の State の欄が LOADED になります。

14 WEBUI の使い方

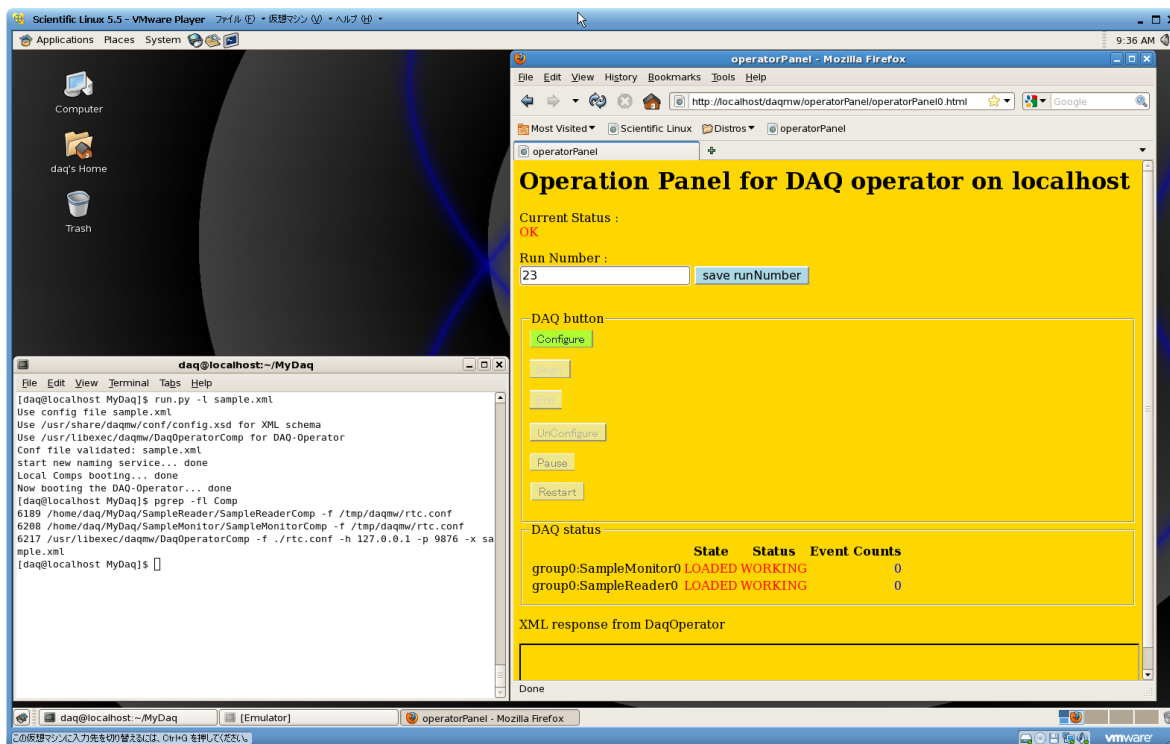


図 11 WebUI。エミュレータを動かし(下のパネルに納められている Emulator と書かれた端末で起動している)、`run.py -l sample.xml` として各コンポーネントを起動し、Web ブラウザで `http://localhost/daqmw/operatorPanel/operatorPanel0.html` にアクセスしたところ。

コンポーネントプロセスを停止させるにはコマンドラインから `kill` コマンドを使います。kill コマンドはデフォルトでは 15 文字までしかプロセス名を見ないので単に “`kill Comp`” とするとコンポーネントに SIGTERM が送られないことがあります。各プロセス名の全てを見るようにするには `-f` を付けて `kill` コマンドを実行します。

```
% kill -f Comp
```

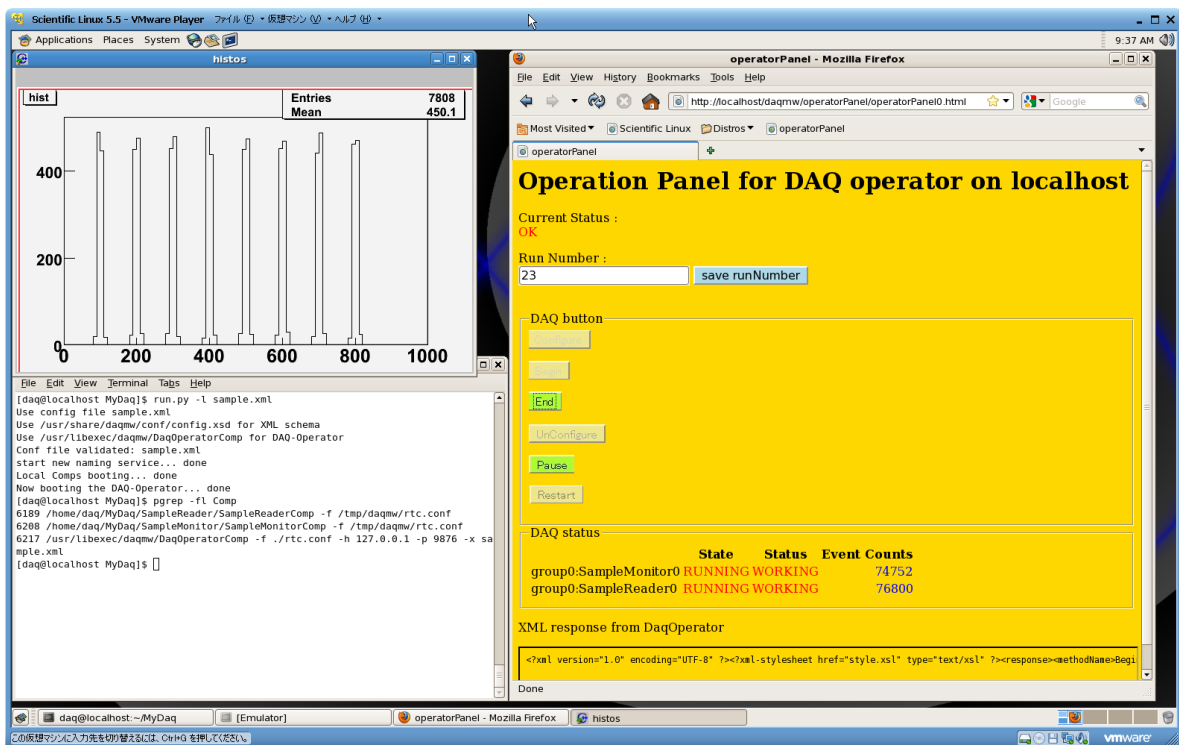


図 12 RUNNING のときの画面の状態。左側ヒストグラム図は定期的に更新されている。

付録 A DAQ-Middleware で提供しているライブラリについて

DAQ-Middleware ではどの DAQ システムでも使用されると思われるライブラリを提供していません。現時点では

- C で書かれたソケットライブラリ
- C で書かれた SiTCP Bus Control Protocol (BCP) 用ライブラリ
- C++ で書かれたソケットライブラリ
- Condition データベースで使用する JsonSpirit ライブラリ

が提供されています。インクルードファイルは `/usr/include/daqmw` に、シェアードライブラリファイル、およびスタティックライブラリファイルは 32bit Scientific Linux (SL) では `/usr/lib/daqmw` に、64bit SL では `/usr/lib64/daqmw` にインストールされます。少々、数が多い点と、OS、その他のアプリケーションパッケージが提供するライブラリファイルから分離するのを目的に独自ディレクトリにインストールすることにしました。

`/usr/lib/daqmw` は `/usr/lib` のような `ld.so` が見る標準ディレクトリではないのでこれらのライブラリを使用しているコンポーネントを動かす場合には以下の方法を使って `ld.so` に `/usr/lib/daqmw` も検索するように指示を出す必要があります。

(1) 環境変数 `LD_LIBRARY_PATH` を使う方法

環境変数 `LD_LIBRARY_PATH` に 32bit SL では `/usr/lib/daqmw` を、64bit SL では `/usr/lib64/daqmw` を追加します。このマニュアルでは使用していませんが、`xinetd` を使ってネットワークブートする場合、リモート計算機で起動されるプロセスの環境変数には `LD_LIBRARY_PATH` は設定されないので `LD_LIBRARY_PATH` を使う場合には `xinetd` の設定ファイル (`/etc/xinetd.d/` ディレクトリ以下にあるファイル) で `env` アトリビュートを使って設定する方法を使う必要があります。

(2) `ld.so` 設定ファイルを使用する方法

Scientific Linux では `/etc/ld.so.conf.d/` ディレクトリ以下にファイルを作成しそのファイルにシェアードライブラリファイルがあるディレクトリを指定しておくと、`ld.so` が指定されたディレクトリも検索するようになります。この機能を利用する場合には `/etc/ld.so.conf.d/daqmw.conf` ファイルを作ってその中に 32bit SL の場合は `/usr/lib/daqmw` と、64bit SL の場合は `/usr/lib64/daqmw` と書きます。ファイルを作成した直後にこの機能を有効にするには `root` ユーザーで一度

```
root# ldconfig
```

を実行する必要があります。リブート後は自動で有効になるのでリブートするたびに `ldconfig` コマンドを実行する必要はありません。

設定を書くファイルの置き場所は OS により異なります。Linux の場合は `/etc/ld.so.conf` あるいはそこから読み出される `/etc/ld.so.conf.d/` ディレクトリ以下のファイルに置くという場合がほとんどです。

DAQ-Middleware ではこちらの `/etc/ld.so.conf.d/` ディレクトリに設定ファイルを置く方法を採用しました。この方法の利点は、各ユーザーが、実行時に環境変数 `LD_LIBRARY_PATH` の値をいちいち気にする必要がなくなること、また、複数の計算機で分散させてデータ収集を行う場合にコンポーネントの起動時に `LD_LIBRARY_PATH` を渡す工夫 (上の `xinetd` の場合の `env` アトリビュートを使うなど) をする必要がなくなることです。

ソースからインストールする場合は、`make install` 時に `/etc/ld.so.conf.d/` ディレクトリがあるかどうか調べ、あれば `daqmw.conf` ファイルを作るようにしています。ファイルに書く内容は `/usr/lib64/` ディレクトリがあれば `/usr/lib64/daqmw` と書き、ない場合は `/usr/lib/daqmw/` と書くようになっています。

付録 B この解説書の変更履歴

この解説書の変更履歴を以下にまとめておきます。

2010-08

- Initial revision。
- DAQ-Middleware 1.0.0 用。
- 2010 年 8 月に行われた DAQ-Middleware 講習会で使用。

2011-01

- DAQ-Middleware 1.0.1(/usr/share/daqmw/docs/DAQ-Middleware-DevManual.pdf) にバンドル。
- DAQ-Middleware の概要について追加。

2011-02

- タイトルを DAQ-Middleware 1.0.2 向けに変更。
- 中身は 1.0.1 用マニュアルから変更ありません。

2011-06

- 64bit をサポートした DAQ-Middleware 1.1.0 向けに変更。
- DAQ-Middleware 1.1.0 から WebUI を追加したのでその説明を追加。
- ソースからコンパイルする際のヒントを追加。
- その他字句の修正など。

2011-10

- リモートブートセットアップ方法について追加。
- Scientific Linux 5.x のインストール方法について追加。

2012-04

- DAQ-Middleware 1.2.0 向けの更新。

付録 C rpm および yum コマンドを使用してセットアップしたときのログ

```
[root@localhost ~]# rpm -ihv http://daqmw.kek.jp/rpm/el5/noarch/
( ) kek-daqmiddleware-repo-2-0.noarch.rpm (長いので( )で折り返しています)
Retrieving http://daqmw.kek.jp/rpm/el5/noarch/kek-daqmiddleware-repo-2-0.noarch.rpm
Preparing... ##### [100%]
 1:kek-daqmiddleware-repo ##### [100%]
[root@localhost ~]# yum --enablerepo=kek-daqmiddleware install DAQ-Middleware
Loaded plugins: kernel-module
kek-daqmiddleware | 951 B 00:00
kek-daqmiddleware/primary | 5.1 kB 00:00
kek-daqmiddleware 15/15
Setting up Install Process
Resolving Dependencies
--> Running transaction check
----> Package DAQ-Middleware.i386 0:1.0.0-0.el5 set to be updated
--> Processing Dependency: OpenRTM-aist >= 1.0.0 for package: DAQ-Middleware
--> Processing Dependency: xerces-c-devel for package: DAQ-Middleware
--> Processing Dependency: libomniDynamic4.so.0 for package: DAQ-Middleware
--> Processing Dependency: libcoil.so.0 for package: DAQ-Middleware
--> Processing Dependency: libRTC-1.0.0.so.0 for package: DAQ-Middleware
--> Processing Dependency: xalan-c-devel for package: DAQ-Middleware
--> Processing Dependency: libxerces-c.so.27 for package: DAQ-Middleware
--> Processing Dependency: libomniORB4.so.0 for package: DAQ-Middleware
--> Processing Dependency: libomnithread.so.3 for package: DAQ-Middleware
--> Running transaction check
----> Package OpenRTM-aist.i386 0:1.0.0-2.el5 set to be updated
--> Processing Dependency: omniORB-doc for package: OpenRTM-aist
--> Processing Dependency: omniORB-bootscripts for package: OpenRTM-aist
--> Processing Dependency: omniORB-utils for package: OpenRTM-aist
--> Processing Dependency: omniORB-servers for package: OpenRTM-aist
--> Processing Dependency: omniORB-devel for package: OpenRTM-aist
----> Package omniORB.i386 0:4.0.7-4.el5 set to be updated
----> Package xalan-c-devel.i386 0:1.10.0-2.el5 set to be updated
--> Processing Dependency: xalan-c = 1.10.0-2.el5 for package: xalan-c-devel
--> Processing Dependency: libxalanMsg.so.110 for package: xalan-c-devel
--> Processing Dependency: libxalan-c.so.110 for package: xalan-c-devel
----> Package xerces-c.i386 0:2.7.0-1.el5.rf set to be updated
----> Package xerces-c-devel.i386 0:2.7.0-1.el5.rf set to be updated
--> Running transaction check
----> Package omniORB-bootscripts.i386 0:4.0.7-4.el5 set to be updated
----> Package omniORB-devel.i386 0:4.0.7-4.el5 set to be updated
----> Package omniORB-doc.i386 0:4.0.7-4.el5 set to be updated
----> Package omniORB-servers.i386 0:4.0.7-4.el5 set to be updated
----> Package omniORB-utils.i386 0:4.0.7-4.el5 set to be updated
----> Package xalan-c.i386 0:1.10.0-2.el5 set to be updated
--> Finished Dependency Resolution
Beginning Kernel Module Plugin
Finished Kernel Module Plugin

Dependencies Resolved
```

```
=====
Package Arch Version Repository Size
=====
```

付録 C RPM および YUM コマンドを使用してセットアップしたときのログ

```
Installing:
DAQ-Middleware          i386      1.0.0-0.el5          kek-daqmiddleware    1.0 M
Installing for dependencies:
OpenRTM-aist            i386      1.0.0-2.el5          kek-daqmiddleware    6.6 M
omniORB                 i386      4.0.7-4.el5          kek-daqmiddleware    6.4 M
omniORB-bootscripts     i386      4.0.7-4.el5          kek-daqmiddleware    6.1 k
omniORB-devel           i386      4.0.7-4.el5          kek-daqmiddleware    2.9 M
omniORB-doc             i386      4.0.7-4.el5          kek-daqmiddleware    986 k
omniORB-servers         i386      4.0.7-4.el5          kek-daqmiddleware    59 k
omniORB-utils           i386      4.0.7-4.el5          kek-daqmiddleware    37 k
xalan-c                 i386      1.10.0-2.el5         kek-daqmiddleware    1.2 M
xalan-c-devel           i386      1.10.0-2.el5         kek-daqmiddleware    443 k
xerces-c                i386      2.7.0-1.el5.rf       kek-daqmiddleware    1.6 M
xerces-c-devel          i386      2.7.0-1.el5.rf       kek-daqmiddleware    649 k
```

Transaction Summary

```
=====
Install      12 Package(s)
Upgrade      0 Package(s)
```

Total download size: 22 M
Is this ok [y/N]: y (y を入力する)

Downloading Packages:

```
(1/12): omniORB-bootscripts-4.0.7-4.el5.i386.rpm | 6.1 kB    00:00
(2/12): omniORB-utils-4.0.7-4.el5.i386.rpm     | 37 kB     00:00
(3/12): omniORB-servers-4.0.7-4.el5.i386.rpm   | 59 kB     00:00
(4/12): xalan-c-devel-1.10.0-2.el5.i386.rpm    | 443 kB    00:00
(5/12): xerces-c-devel-2.7.0-1.el5.rf.i386.rpm | 649 kB    00:00
(6/12): omniORB-doc-4.0.7-4.el5.i386.rpm       | 986 kB    00:00
(7/12): DAQ-Middleware-1.0.0-0.el5.i386.rpm    | 1.0 MB    00:00
(8/12): xalan-c-1.10.0-2.el5.i386.rpm          | 1.2 MB    00:00
(9/12): xerces-c-2.7.0-1.el5.rf.i386.rpm       | 1.6 MB    00:00
(10/12): omniORB-devel-4.0.7-4.el5.i386.rpm    | 2.9 MB    00:00
(11/12): omniORB-4.0.7-4.el5.i386.rpm          | 6.4 MB    00:00
(12/12): OpenRTM-aist-1.0.0-2.el5.i386.rpm     | 6.6 MB    00:00
```

```
-----
Total                               10 MB/s | 22 MB    00:02
```

```
Running rpm_check_debug
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
```

Running Transaction

```
Installing      : omniORB                1/12
Installing      : xerces-c                2/12
Installing      : xerces-c-devel          3/12
Installing      : omniORB-doc             4/12
Installing      : omniORB-utils           5/12
Installing      : omniORB-servers         6/12
Installing      : xalan-c                 7/12
Installing      : omniORB-devel           8/12
Installing      : xalan-c-devel           9/12
Installing      : omniORB-bootscripts     10/12
Installing      : OpenRTM-aist            11/12
Installing      : DAQ-Middleware          12/12
```

Installed:

```
DAQ-Middleware.i386 0:1.0.0-0.el5
```

Dependency Installed:

```
OpenRTM-aist.i386 0:1.0.0-2.el5          omniORB.i386 0:4.0.7-4.el5
```

付録 C RPM および YUM コマンドを使用してセットアップしたときのログ

```
omniORB-bootscripts.i386 0:4.0.7-4.e15  omniORB-devel.i386 0:4.0.7-4.e15
omniORB-doc.i386 0:4.0.7-4.e15         omniORB-servers.i386 0:4.0.7-4.e15
omniORB-utils.i386 0:4.0.7-4.e15      xalan-c.i386 0:1.10.0-2.e15
xalan-c-devel.i386 0:1.10.0-2.e15     xerces-c.i386 0:2.7.0-1.e15.rf
xerces-c-devel.i386 0:2.7.0-1.e15.rf
```

```
Complete!
[root@localhost ~]#
```

付録 D ソースからインストールする場合のヒント

Scientific Linux、CentOS、RedHat Enterprise Linux (5.x、6.x) にセットアップする場合は RPM バイナリを利用するのが簡単です。その他の OS のバイナリは現在のところ、作成されていませんので、これら以外の OS で DAQ-Middleware を使用したい場合はソースからコンパイルする必要があります。この節では DAQ-Middleware をソースからインストールする場合のヒントをまとめておきます。

DAQ-Middleware が動作するためには以下のソフトウェアが必要です。OS ディストリビューションにこれらのソフトウェアが含まれている場合はそれをインストールするのが簡単かと思えます。

- omniORB 一式 (4.1.x)
- OpenRTM-aist 1.0.0 + パッチ
- xerces-c 2.x あるいは 3.x
- xalan-c 1.10
- boost

DAQ-Middleware は OpenRTM-aist をベースとして使用しています。OpenRTM-aist は omniORB を使っています。また DAQ-Middleware では XML 関連ライブラリとして xerces-c を使っています。コンディションデータベースについて XML から json ファイルへの変換に xalan-c と boost を使っています。サンプルコンポーネントには boost を使っているものがあります。

omniORB は開発環境、および omniidl、omniNames が動作することが必要です。OS ディストリビューションが配布するものをいれた場合リブート時に omniNames が自動起動する場合があります。開発時に omniNames に関するトラブルを防止するために run.py は起動すると omniNames をいったん停止し、再び起動する動作をします。OS 起動時に自動起動するようにしておくと開発者のユーザー権限では止めることができないことが多いので、omniNames は自動起動しないように設定しておきます。RHEL 系では root ユーザーで “chkconfig omniNames off” を実行しておきます。

xerces-c は 2.x と 3.x の両方に対応しています。xalan-c は xerces-c を使っています。Linux ディストリビューションが提供するバイナリを使用する場合は xerces-c のバージョンにあった xalan-c をインストールする必要があります。

OpenRTM-aist のパッチは <http://daqmw.kek.jp/rpm/SRPMS/> にある OpenRTM-aist-1.0.0-X.r1971.e15.src.rpm に含まれています。この SRPM ファイルには OpenRTM-aist-1.0.0 相当のファイルも含まれていますので rpm2cpio コマンド等で取り出して以下のように作業します。作業には Autotools が必要です。

```
tar xf OpenRTM-aist-r1971.tar.gz
cd OpenRTM-aist-r1971
```

```
SRPMS にふくまれていたパッチを全部あてる（具体的コマンド省略）
sh build/autogen
./configure --prefix=/usr (--prefix の値は必要に応じて変える)
make
make install
```

これらの準備ができれば DAQ-Middleware のソースを <http://daqmw.kek.jp/src/> から取得して展開し

```
make
make install
```

します。

アンインストールするには、Makefile の `uninstall` ターゲットが使えます。

```
cd DAQ-Middleware-1.x.y
make uninstall
```

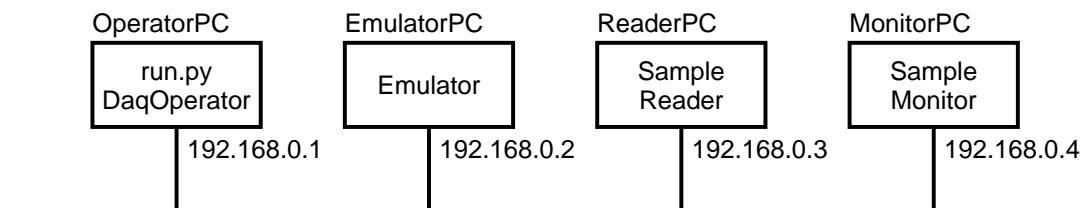


図 13 リモートブート解説のための計算機構成図。OperatorPC, EmulatorPC 等はホスト名である。IP アドレスは下にも書いてあります。各計算機で走らせる DAQ 構成物は四角の中に書いてあります。

付録 E リモートブートセットアップ

DAQ コンポーネントを複数の計算機に分散させてシステムを構成する方法について解説します。この節で例として取り上げる計算機構成を図 13 にのせておきました。この例は ReaderPC で SampleReader を動かし、ネットワーク経由で MonitorPC 上で動いている SampleReader に読み取ったデータを送る構成になっています。

複数計算機に DAQ コンポーネントを分散させるので、ネットワーク越しに DAQ コンポーネントを起動するしくみが必要になります。ここでは xinetd を使います。

xinet プログラムファイルの他に関連するファイルは /etc/services、/etc/xinetd.d/bootComps、/usr/share/daqmw/etc/remote-boot/bootComps.py です。

E.1 ネットワーク疎通の確認

ネットワーク経由でデータをやりとりするので各計算機間で通信ができる必要があります。ping に応答があるからといって他の通信ができるとは限りませんので注意する必要があります。iptables 等でパケットフィルタリングを実施している場合にはそれを停止させておくトラブル防止になります^{*15}。iptables でのパケットフィルタリングを停止するには

```
root# service iptables stop
```

とします。起動時に停止状態になっているようにするには

```
root# chkconfig iptables off
```

とします。

^{*15} パケットフィルタリング等の安全確保手段を停止してよいかどうかは各自で接続しているネットワークセキュリティポリシー等を確認してください。

E.2 xinetd のインストール

rpm -q xinetd で xinetd がインストールされているかどうかわかります。インストールされていない場合は OS 配布リポジトリから

```
root# yum install xinetd
```

でインストールします。DAQ コンポーネントを起動する全ての計算機で xinetd を使えるようにする必要があります。図 13 の例では ReaderPC、MonitorPC に xinetd が必要になります。

E.3 xinetd の設定

/etc/services—に以下のコマンドで bootComps を追加します (>ではなくて>が 2 個の>>であることに注意してください)。

```
root# cat /usr/share/daqmw/etc/remote-boot/services.sample >> /etc/services
```

xinetd から起動されるプログラムを定義する /etc/xinetd.d/bootComps を以下のようにして作ります (1 行が長いので () で折り返していますが実行時には 1 行で入力します)。

```
root# cp /usr/share/daqmw/etc/remote-boot/bootComps-xinetd.sample ( )
/etc/xinetd.d/bootComps
```

xinetd を起動するには以下のようにします。

```
root# service xinetd restart
```

これで

```
root# service xinetd restart
Stopping xinetd: [FAILED] (まだ起動されていない場合は FAILED とですが
Starting xinetd: [ OK ] Starting xinetd が OK になればよいです)
root#
```

となったあと /var/log/messages を見て正常に起動されたかどうか確認します。エラーが出た場合は出力されたメッセージを読んで修正します。また実際に動作するかどうかは、他の計算機から

```
% echo hello | nc remote-host 50000
```

とします (remote-host の部分は xinetd を設定したホスト名あるいはその IP アドレスに置き換えてください)。これで


```
-1 need more info.
```

と出力されれば OK です。このように出力されない場合は xinetd を起動するようにした計算機の /var/log/messages を見て確認する、あるいはパケットフィルターリングが行われていてパケットが届いていないなどを確認します。bootComps-xinetd.sample では daq というユーザー権限で DAQ コンポーネントを起動するようになっていますので DAQ コンポーネントを起動する計算機では daq というユーザーが登録されている必要があります (変更するには /etc/xinetd.d/bootComps の user = の右辺を変更します)。ブート時に xinetd を自動起動するためには

```
root# chkconfig xinetd on
```

とします。

run.py を動かす計算機で DAQ コンポーネントを起動する場合でも DAQ コンポーネントは xinetd 経由で起動する必要があります。

DAQ コンポーネントのログは /tmp/daqmw/ ディレクトリ内に作られます。ログは DAQ コンポーネントのプロセスの権限で作られますので既に他のユーザーが所有する /tmp/daqmw/ ディレクトリがある場合はそのディレクトリごと消去しておいてください。

E.4 コンフィギュレーションファイルの作成

図 13 の構成の場合のコンフィギュレーションファイルを図 14 に示します。

execPath には DAQ コンポーネントプログラムファイルがある場所を指定します。confFile には DAQ コンポーネントが起動するときネームサーバー稼働している計算機等の情報が書かれたファイルで、run.py が自動生成してリモート計算機に送ります。置く場所はいつものところ run.py 内で /tmp/daqmw/rtc.conf に固定されていますのでコンフィギュレーションファイルで指定してもその指定は無視されます。

E.5 システムの起動

システムの起動は Emulator を起動後、コンフィギュレーションファイルを remote.xml だとして

```
daq% run.py -c remote.xml
```

として行います。

SampleMonitor のヒストグラムの描画は、デフォルトでは X の許可がないのでどこにも出てきませんが、DaqOpeator が表示するイベント数を見ると正常に読み取りができているかどうかわかります。

```

<configInfo>
  <daqOperator>
    <hostAddr>192.168.0.1</hostAddr>
  </daqOperator>
  <daqGroups>
    <daqGroup gid="group0">
      <components>
        <component cid="SampleReader0">
          <hostAddr>192.168.0.3</hostAddr>
          <hostPort>50000</hostPort>
          <instName>SampleReader0.rtc</instName>
          <execPath>/home/daq/MyDaq/SampleReader/SampleReaderComp</execPath>
          <confFile>/tmp/daqmw/rtc.conf</confFile>
          <startOrd>2</startOrd>
          <inPorts>
          </inPorts>
          <outPorts>
            <outPort>samplerreader_out</outPort>
          </outPorts>
          <params>
            <param pid="srcAddr">192.168.0.2</param>
            <param pid="srcPort">2222</param>
          </params>
        </component>
        <component cid="SampleMonitor0">
          <hostAddr>192.168.0.4</hostAddr>
          <hostPort>50000</hostPort>
          <instName>SampleMonitor0.rtc</instName>
          <execPath>/home/daq/MyDaq/SampleMonitor/SampleMonitorComp</execPath>
          <confFile>/tmp/daqmw/rtc.conf</confFile>
          <startOrd>1</startOrd>
          <inPorts>
            <inPort from="SampleReader0:samplerreader_out">samplemonitor_in</inPort>
          </inPorts>
          <outPorts>
          </outPorts>
          <params>
          </params>
        </component>
      </components>
    </daqGroup>
  </daqGroups>
</configInfo>

```

図 14 図 13 の構成の場合のコンフィギュレーションファイル例

ヒストグラムを OperatorPC 上に表示するには OperatorPC 上で

```
daq@OpeartorPC% xhost +MonitorPC
```

として

```
daq@OperatorPC% run.py -d OperatorPC:0 -c remote.xml
```

として行います。表示されない場合は X の通信が許可されているかどうかをたとえば

```
daq@OperatorPC% ssh -x MonitorPC (小文字-x で SSH X forwarding を無効にする)
daq@MonitorPC% DISPLAY=OperatorPC:0 xterm
```

として MonitorPC の X クライアントが OperatorPC の X 上に表示できるかどうか確認したり、-d の引数を IP アドレス:0 にして試す、などします。

リモート計算機で作ったヒストグラムを X 上に表示させようとして表示できない場合には画面描画をしている X が tcp listen しているかどうか確認してください。SL など Gnome を使っているシステムでは /etc/gdm/custom.conf の [security] セクションに

```
[security]
DisallowTCP=false
```

と書いて X サーバーの再起動 (いったんログアウトするなど) する必要があります。さらに表示する PC 側で、X サーバーに他 PC からの接続を許可するように設定する必要があります。

```
xhost +
```

とすると全ての計算機から X サーバーに接続が許可されます。一部の計算機からのみ接続できるようにするためには

```
xhost +hostname.example.jp
```

のように接続を許可したい計算機のホスト名、あるいは IP アドレスを指定します。

E.6 リモートブートのしくみ

リモートブートのしくみを図 15 に示します。この図は、前節でとりあげた DaqOperator、SampleReader、SampleMonitor がそれぞれ別の計算機で稼働するシステムのブートを示しています。

1. run.py がコマンドラインから起動されます。起動された run.py は引数で指定されたコンフィギュレーションファイルをパースしてどの計算機で DAQ コンポーネントを起動するか把握します。
2. run.py は DAQ コンポーネントを起動する計算機の Port 50000 にアクセスし rtc.conf 作成の指示を出します。指示を受け取るのは xinetd から起動された bootComps.py です。
3. bootComps.py は rtc.conf を作成し、作成できたかどうかを run.py に通知します。
4. run.py は rtc.conf が各計算機で作成できたら続けて xinetd を経由して bootComps.py に対して DAQ コンポーネントを起動する指示をだします。起動するコンポーネントのパスは config.xml に書かれているものを使用します。
5. 起動した DAQ コンポーネントは各計算機で作られた rtc.conf をもとに Naming Service が

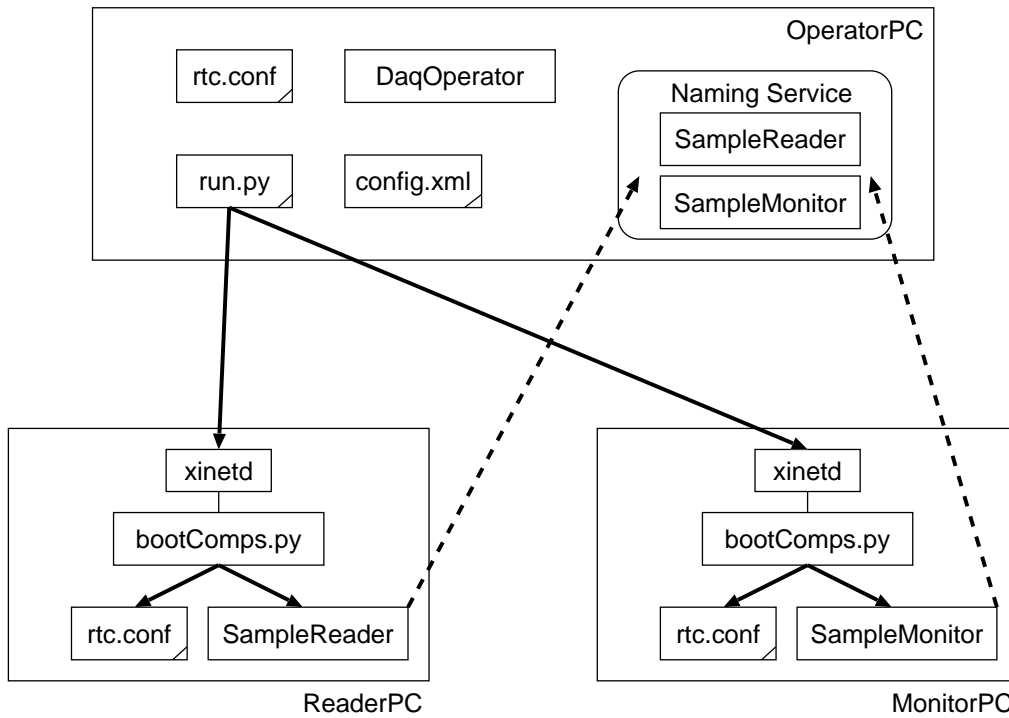


図 15 リモートブートのしくみ。

動いている計算機を把握し、Naming Service にアクセスし自身の情報を登録します。

6. run.py は続いて DaqOperator を起動します。DaqOperator は起動後、config.xml を見てシステムを把握し、各 DAQ コンポーネントの情報を Naming Service へ問い合わせます。その後 DAQ コンポーネント間を接続します。

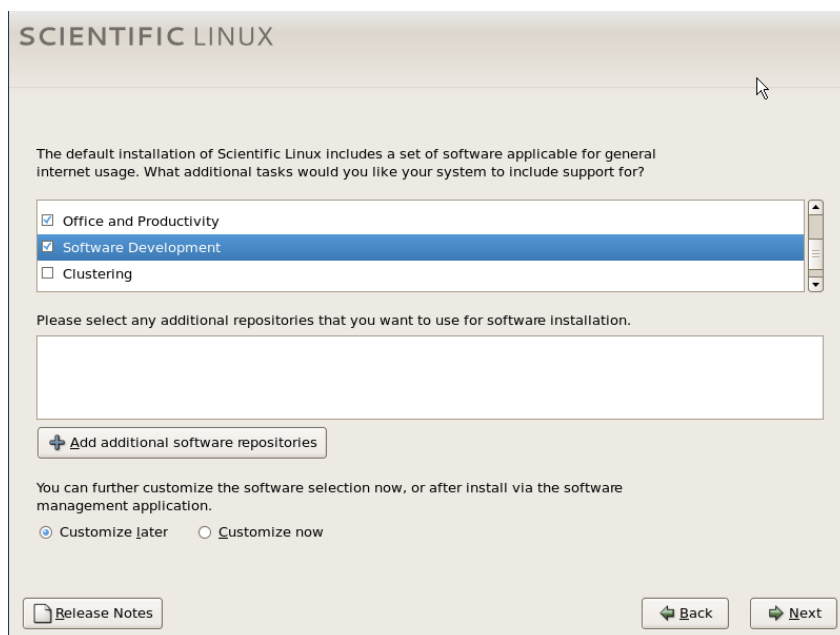


図 16 計算機タスクの選択

付録 F Scientific Linux インストール方法

Scientific Linux のウェブサイトは <http://www.scientificlinux.org/> です。インストール CD、DVD イメージはこのサイト、あるいは下記の日本にあるミラーから取得することができます。

- <http://ftp.riken.jp/Linux/scientific/> (理研)
- <http://ftp.ne.jp/Linux/distributions/scientificlinux/> (KDDI 研究所)
- <http://ftp.jaist.ac.jp/pub/Linux/scientific/> (JAIST)
- <http://reflx1.kek.jp/scientific/> (KEK (KEK 内部からのみ接続可能))

インストールは GUI にしたがってやっていけばあまりまよるところはないと思いますが、パッケージについて、DAQ コンポーネントを開発する場合にどのパッケージをインストールしておけばよいか、以下で解説します。

F.1 新規にインストールする場合 (SL 5.x)

インストールの途中で、そのマシンで主として行う作業についての画面がでてきますが、ここで Software Development にチェックを追加しておく (図 16) と必要なパッケージがほぼインストールされます。WebUI を使う場合は `mod_python` が必要ですのでインストール後に

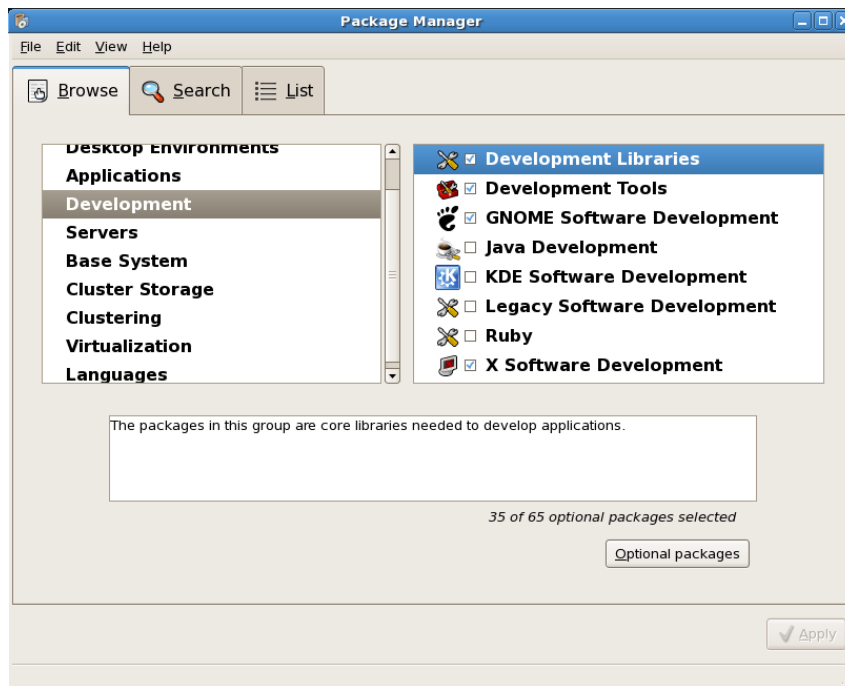


図 17 SL 5.x での開発パッケージの追加

```
root# yum install mod_python
```

としてインストールしておきます。また Scientific Linux 5.x では上のパッケージセレクションではエディタとして Emacs がインストールされないので必要ならインストール後に

```
root# yum install emacs
```

などしてインストールしておきます。

SELinux、iptables の設定について F.7 節をご覧ください。

F.2 開発環境をあとから追加する場合 (SL 5.x)

すでに Gnome Desktop としては使えるが、開発環境 (gcc, make 等) をいれていない場合には、メニューバー左端のアイコンをクリックしてでてくるメニューから “Add/Remove Software” を起動して “Development” から以下のものをチェックして apply します (図 17)。mod_python と Emacs の追加については前節をごらんください。

- Development Libraries
- Development Tools
- GNOME Software Development

- X Software Development

SELinux、iptables の設定について F.7 節をご覧ください。

F.3 新規にインストールする場合 (SL 6.x)

インストールの途中で、そのマシンで主として行う作業についての画面がでてきますが、ここで Software Development にチェックを追加しておく (図 18) と必要なパッケージがほぼインストールされます。DAQ コンポーネントのコンパイルに必要な libuuid-devel パッケージはインストールされていないので、

```
# yum install libuuid-devel
```

としてインストールしておきます。WebUI を使う場合は mod_wsgi が必要ですのでインストール後に

```
root# yum install mod_wsgi
```

としてインストールしておきます。また Scientific Linux 6.x では上のパッケージセレクションではエディタとして Emacs がインストールされない所以需要ならインストール後に

```
root# yum install emacs
```

などしてインストールしておきます。

SELinux、iptables の設定について F.7 節をご覧ください。

F.4 開発環境をあとから追加する場合 (SL 6.x)

すでに Gnome Desktop としては使えるが、開発環境 (gcc, make 等) をいれていない場合には、yum コマンドを使って Development tools、Additional development グループをインストールします^{*16}。さらに libuuid-devel パッケージが必要です。

```
root# yum groupinstall 'Development tools'  
root# yum groupinstall 'Additional development'  
root# yum install libuuid-devel
```

libuuid-devel、mod_wsgi および emacs の追加については前節をご覧ください。

SELinux、iptables の設定について F.7 節をご覧ください。

^{*16} GUI でこれらのグループを選択することはできないようです。

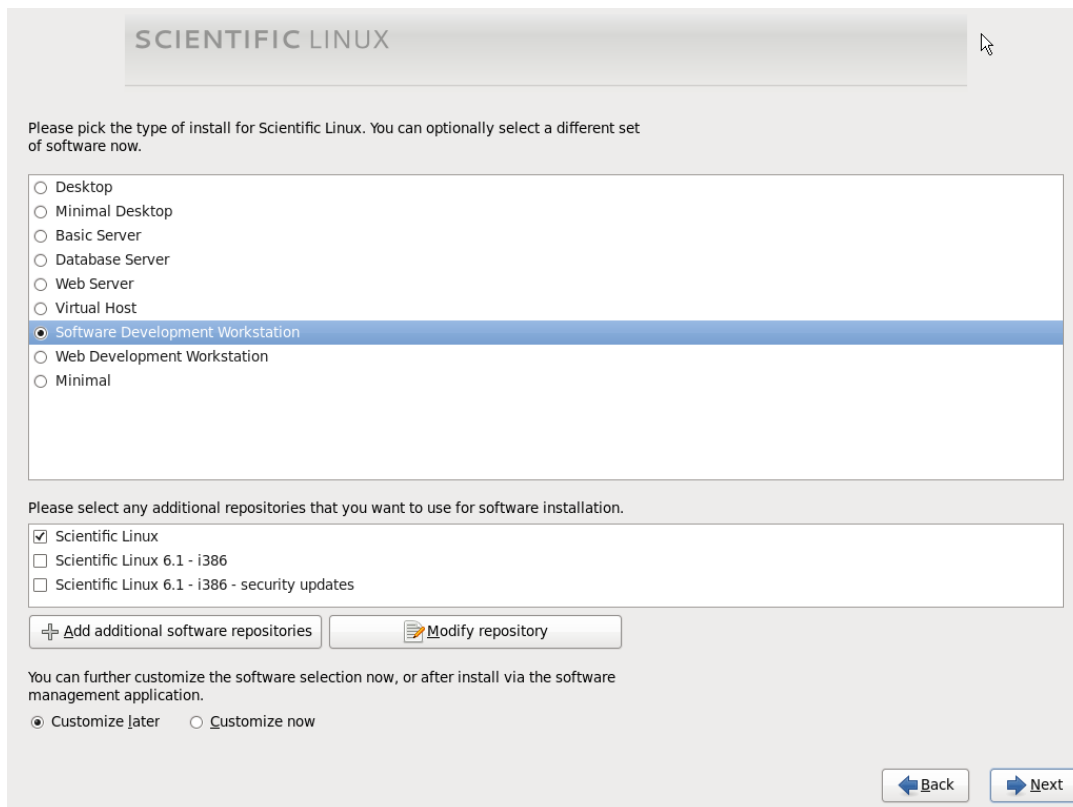


図 18 SL 6.x 計算機タスクの選択

F.5 新規にインストールする場合 (SL 7.x)

インストールの途中で、そのマシンで主として行う作業についての画面がでてきますが、ここで Development and Creative Workstation にチェックを追加しておく (図 19) と必要なパッケージがほぼインストールされます。DAQ コンポーネントのコンパイルに必要な libuuid-devel パッケージ、および boost-devel パッケージはインストールされていないので、

```
# yum install libuuid-devel
# yum install boost-devel
```

としてインストールしておきます。WebUI を使う場合は mod_wsgi が必要ですのでインストール後に

```
root# yum install mod_wsgi
```

としてインストールしておきます。また Scientific Linux 7.x では上のパッケージセレクションではエディタとして Emacs がインストールされない所以需要ならインストール後に

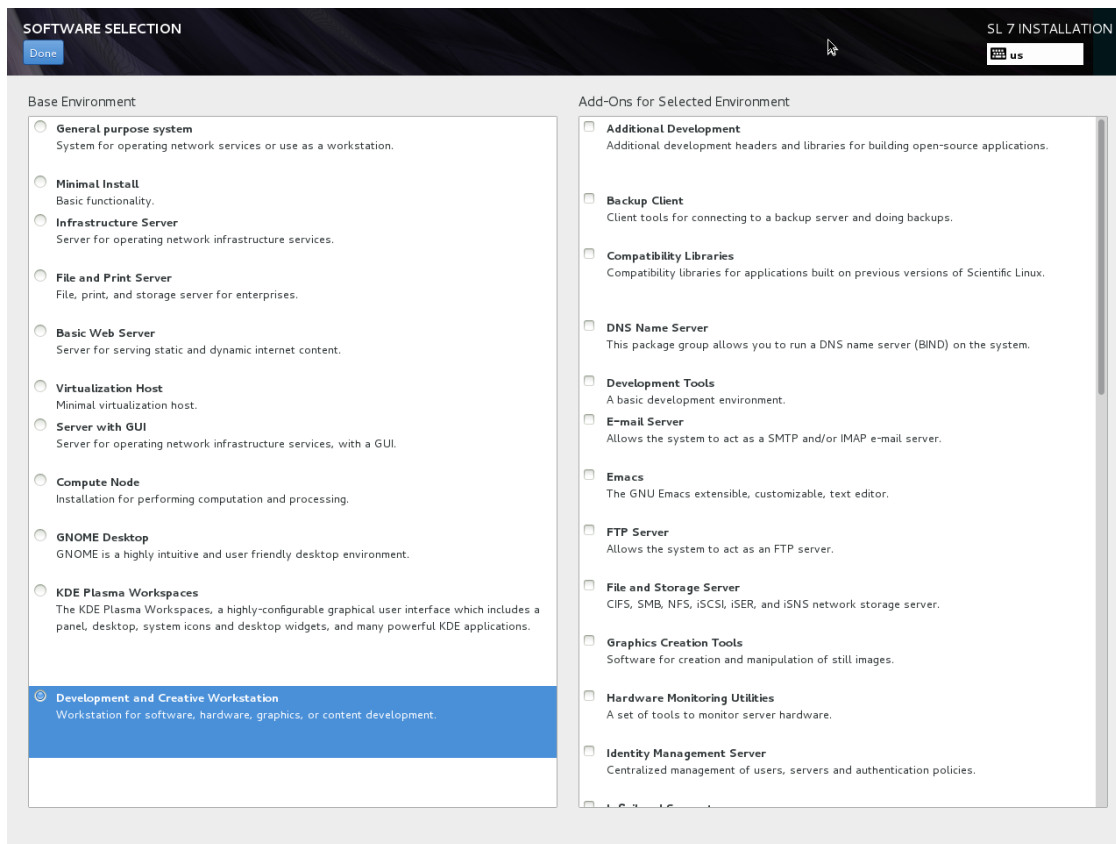


図 19 SL 7.x 計算機タスクの選択

```
root# yum install emacs
```

などしてインストールしておきます。

SELinux、iptables の設定について F.7 節をご覧ください。

F.6 開発環境をあとから追加する場合 (SL 7.x)

SL 7.x では SOFTWARE SELECTION でのデフォルトが “General purpose system” になっています。これでセットアップすると Gnome Desktop としては使えますが、開発環境 (gcc, make 等) が入っていません。この状態で開発環境をセットアップするには次のコマンドを実行します。

```
root# yum groupinstall 'Development and Creative Workstation'
root# yum install boost-devel
root# yum install libuuid-devel
root# yum install mod_wsgi
```

SELinux、パケットフィルターの設定について F.8 節をご覧ください。

F.7 SL 5.x, 6.x での SELinux, iptables の設定

SELinux が有効になっていると WebUI が動作しないので SELinux を無効にする必要があります。

SELinux が有効になっているかどうかは `getenforce` コマンドでわかります。

```
% getenforce
```

有効の場合は `Enforcing` あるいは `Permissive` と表示されます。無効の場合は `Disabled` と表示されます。

無効にするには `/etc/sysconfig/selinux` で `SELINUX=` で始まる行を

```
SELINUX=disabled
```

と書き換えてリブートします。

`iptables` でパケットフィルターをしていると `DAQ-Middleware` で必要とするパケットの送信、受信ができない場合があります。トラブルを避けるには `iptables` のパケットフィルターを停止させるのが単純でよいと思います。

現在 `iptables` でパケットフィルターリングがされているかどうかの確認は

```
root# service iptables status
```

で行います。`iptables` が稼働している場合はパケットフィルタールールが表示されます。稼働していない場合は

```
iptables: Firewall is not running.
```

と表示されます。

有効になっていた場合に、無効にするには

```
root# service iptables stop
```

とします。リブート後も自動で無効にするには

```
root# chkconfig iptables off
```

とします。

F.8 SL 7.x での SELinux, パケットフィルターの設定

SELinux が有効になっていると WebUI が動作しないので SELinux を無効にする必要があります。

SELinux が有効になっているかどうかは `getenforce` コマンドでわかります。

```
% getenforce
```

有効の場合は `Enforcing` あるいは `Permissive` と表示されます。無効の場合は `Disabled` と表示されます。

無効にするには `/etc/sysconfig/selinux` で `SELINUX=` で始まる行を

```
SELINUX=disabled
```

と書き換えてリブートします。

パケットフィルタが有効な状態だと DAQ-Middleware で必要とするパケットの送信、受信ができない場合があります。トラブルを避けるにはパケットフィルタを停止させるのが単純でよいと思います。Scientific Linux 7.x からは `firewalld` でパケットフィルタの設定をするようになりました。現在パケットフィルタが有効になっているかどうかは以下のコマンドを実行して確認します。

```
root# systemctl status firewalld
```

このコマンドの出力に以下のように `Active: active (running) since ...` の行が含まれていればパケットフィルタが有効になっています:

```
firewalld.service - firewalld - dynamic firewall daemon
  Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled)
  Active: active (running) since Thu 2015-02-26 08:04:36 JST; 23s ago
  Main PID: 584 (firewalld)
  CGroup: /system.slice/firewalld.service
          584 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid

Feb 26 08:04:28 hostname.kek.jp systemd[1]: Starting firewalld - dynamic fir....
Feb 26 08:04:36 hostname.kek.jp systemd[1]: Started firewalld - dynamic fire....
Hint: Some lines were ellipsized, use -l to show in full.
```

すぐに停止するには以下のコマンドを実行します。

```
root# systemctl stop firewalld
```

再起動後も無効にするには

```
root# systemctl disable firewalld
```

と入力します。

付録 G コマンドラインから DAQ オペレータに指示を出す

http を使って DAQ オペレータに指示を出すツールとしてウェブブラウザ以外に Python を使ったコマンドラインツールを用意しました。/usr/bin/daqcom が実行ファイルです。単に daqcom と実行するとヘルプメッセージがでます。SL 5.x では

```
daq% daqcom http://localhost/daqmw/operatorPanel/ -g state
```

とすると状態が取得できます。SL 6.x では URL が変更になって

```
daq% daqcom http://localhost/daqmw/scripts/ -g state
```

になりました。くわしくは参考文献 [4] を御覧ください。

付録 H InPort リングバッファのパラメータの設定

DAQ-Middleware 1.3.0 から InPort のリングバッファのパラメータ値を設定できるようになりました。設定は設定ファイル (config.xml) を使って行います。設定例を下記に示します。設定ファイルで指定しない場合はデフォルト値が適用されます。設定可能なパラメータ、値、デフォルト値は以下のとおりです。

```
<inPorts>
<inPort from="SampleReader0:samplerereader_out"
  buffer_length="4096"
  buffer_read_timeout="0.010"
  buffer_write_timeout=0.010">samplemonitor_in</inPort>
</inPorts>
```

buffer_length (単位: 個、デフォルト値 256)

InPort リングバッファ長を指定します。単位はバッファ数 (個) です。デフォルトは 256 個です (1.2.2 およびそれ以前は 128 個の固定値でした)。たとえばハードディスクのように大抵は、すばやく書けるがときどき書きだしに時間がかかる (たとえば 0.1 秒以上かかる) 場合に、書きだしに時間がかかっている間に到着したデータをバッファリングする目的で buffer_length の値を大きくしてデータ収集速度を確保するためにこのパラメータを使います。

(註)InPort リングバッファ中のバッファは、上流から送られてきたデータ長にあわせて動的に確保されます。

buffer_read_timeout (単位: 秒、デフォルト値 0.005)

buffer_read_empty_policy が block の場合 (block がデフォルトです) のリードの最大待ち時間を指定します。単位は秒です。デフォルト値は 0.005 (5 ミリ秒) です (1.2.2 およびそれ以前は 0.1 秒 (100 ミリ秒) の固定値でした)。buffer_read_empty_policy については下記をごらんください。

buffer_write_timeout (単位: 秒、デフォルト値 0.005)

buffer_write_full_policy が block の場合 (block がデフォルトです) のライトの最大待ち時間を指定します。単位は秒です。デフォルト値は 0.005 (5 ミリ秒) です (1.2.2 およびそれ以前は 0.1 秒 (100 ミリ秒) の固定値でした)。buffer_write_full_policy については下記をごらんください。

buffer_read_empty_policy (デフォルト値: block)

InPort のリングバッファが空のときの InPort.read() の動作を設定します。指定できる値は block、および do_nothing です。デフォルト値は block です (1.2.2 およびそれ以前は block で動作しており、指定はできませんでした)。通常は block を指定しておくことを推奨します。

`block` を指定するとリングバッファが空のとき、最大 `buffer_read_timeout` で指定した秒数までデータがくるのを待ちます。待っている間にデータがきた場合は `InPort.read()` はきたデータをすぐに読みます最大時間待ってもデータがこなかった場合、`InPort.read()` は `false` を返します。この場合、エラーかどうかの判定は `check_inPort_status()` メソッドで行います。エラーの場合は `check_inPort_status()` は `BUF_FATAL` を返し、単にデータがこなくて読めなかった場合は `BUF_TIMEOUT` を返します。

`do_nothing` を指定するとノンブロッキングリードの動作になります。これを指定した場合は `InPort` のリングバッファが空の場合 `InPort.read()` は、すぐに `false` で返ります。エラーかどうかの判定には `check_inPort_status()` を使います。リングバッファにデータがなく読めなかった場合は `check_inPort_status()` は `BUF_NODATA` を返します。エラーの場合は `BUF_FATAL` を返します。`do_nothing` を指定すると、`InPort` リングバッファが空の状態が続くときに連続してリードするようにプログラムを作ると CPU を著しく消費し、結果としてデータ収集速度が劣化することがあるので注意が必要です。

`buffer_write_full_policy` (デフォルト値: `block`)

`InPort` のリングバッファがフルのときの `OutPort.write()` の動作を設定します。指定できる値は `block`、および `do_nothing` です。デフォルト値は `block` です (1.2.2 およびそれ以前は `block` で動作しており、指定はできませんでした)。通常は `block` を指定しておくことを推奨します。

`block` を指定した場合、下流側 `InPort` リングバッファがフルのとき、最大 `buffer_write_timeout` で指定した秒数まで空きができるのを待ちます。待っている間に空きができた場合は `OutPort.write()` はすぐに書きます。最大時間待ってもデータを書けなかった場合、`OutPort.write()` は `false` を返します。この場合、エラーかどうかの判定は `check_outPort_status()` メソッドで行います。エラーの場合は `check_outPort_status()` は `BUF_FATAL` を返し、バッファがフルで書けなかった場合は `BUF_TIMEOUT` を返します。

`do_nothing` を指定するとノンブロッキングライトの動作になります。これを指定した場合は、下流側 `InPort` のリングバッファがフルの場合 `OutPort.write()` は、すぐに `false` で返ります。エラーかどうかの判定は `check_outPort_status()` を使います。バッファフルで書けなかった場合は `check_outPort_status` は `BUF_NOBUF` を返します。エラーの場合は `BUF_FATAL` を返します。`do_nothing` を指定すると、`InPort` リングバッファがフルの状態が続くときに連続してライトするようにプログラムを作ると CPU を著しく消費し、結果としてデータ収集速度が劣化することがあるので注意が必要です。

参考文献

- [1] DAQ-Middleware Home page <http://daqmw.kek.jp/>
- [2] DAQ-Middleware 1.1.0 技術解説書、2011 年 6 月、
<http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-Tech.pdf>
- [3] 安芳次、千代浩司、Condition データベースの開発マニュアル、2010 年 8 月 3 日、
<http://daqmw.kek.jp/docs/ConditionDevManual-1.0.0.pdf>
- [4] DAQOperator 操作用 Python コマンドマニュアル、2012 年 4 月、
<http://daqmw.kek.jp/docs/PythonCommandForDAQMWOperation.pdf>