データ収集システムの開発

データ収集システム入門

エレクトロニクスDAQセミナー2010

2010年7月29日

KEK素核研エレクトロニクスシステムグループ

安　芳次

課題

- 達成目標: データ収集システムを利用するだけでなくシステム開発の道筋が理解できるようになる。
- 内容: データ収集システムを開発するという立場で概観し、必要となる知識とその方法について説明する。

話の筋

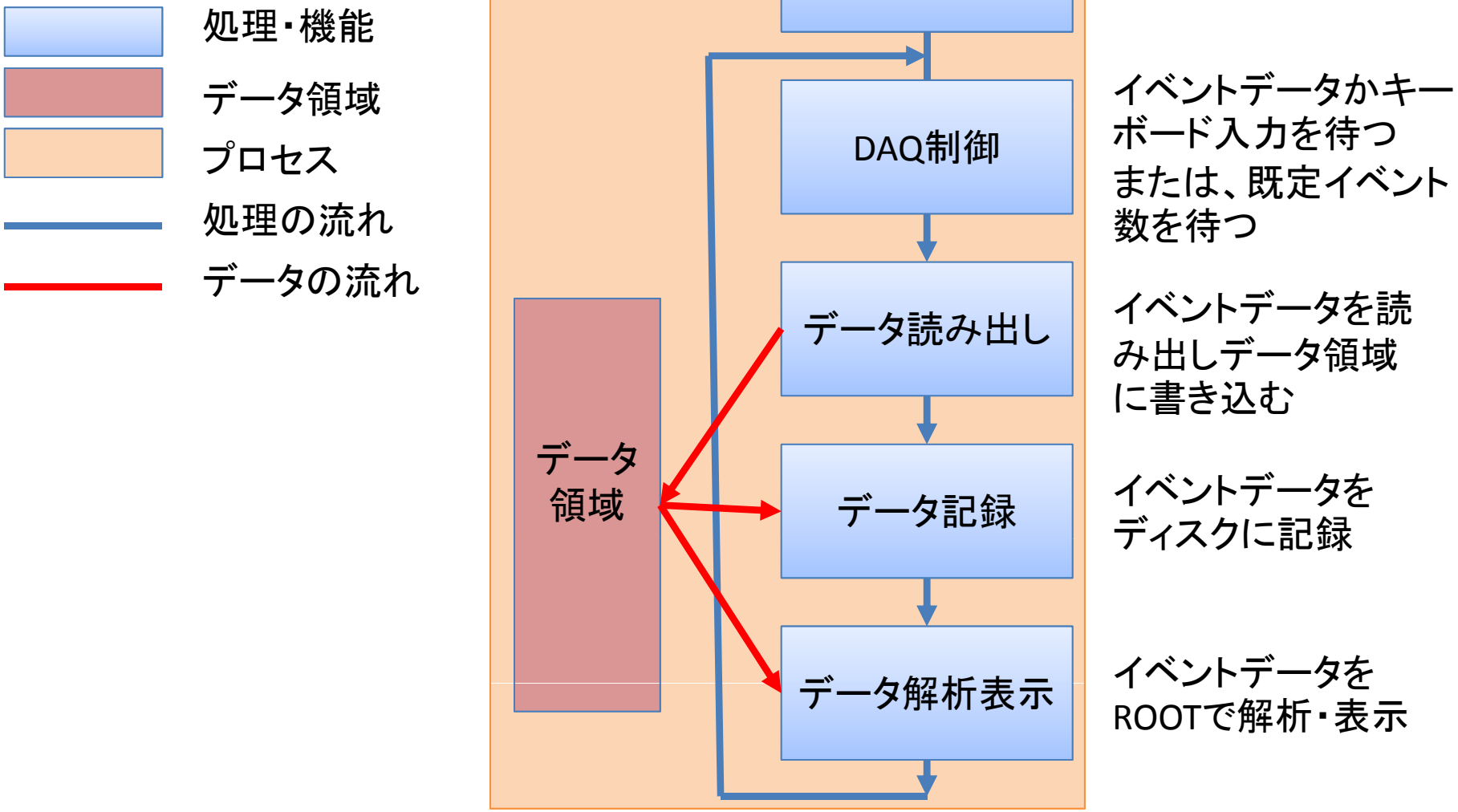
- 機能要素であるパラメータ（データベース）入力・データ読み出し・記録・解析表示(モニタ)・DAQ制御をどのように組み立てるか？簡単さ、性能、スケーラビリティーの視点から見てみよう。（トリガーは重要な機能要素だが、今回は含めない）
- まずは、「最も簡単なデータ収集法」
  - 簡単ではあるが、すべての要素が含まれる
  - 一通りのことはできるが、DAQ操作性・性能・スケーラビリティーの面で限界
- 次に、「GUIを用いた簡単なデータ収集法」
  - GUIはDAQ操作を簡単にするが、「最も簡単なデータ収集法」の延長ではプログラミングできない
  - DAQ操作性はよくなるが、性能・スケーラビリティーの面で限界
- 最後に、「DAQ-Middlewareを用いたデータ収集法」
  - スケーラビリティーの実現、性能向上にはフレームワーク（ミドルウエア）が必要
- まとめ

技術的にキーとなる概念

- パラメータ・データの共有・分散
- マルチ・シングルのプロセス・スレッド
- 随時処理・平行処理

- （ネットワークを超えた、1つのシステム内の）プロセス間通信
- 同期処理（平行する処理手順の保障）・排他処理（資源の競合回避・クリティカルセクション）

最も簡単なデータ収集法



最も簡単なデータ収集法

- selectシステムコールによるDAQ制御の改善
  - 複数のイベントを待つ方法として強力な技術
- 広い意味でのデータベース
- デバッグ

selectシステムコールによるDAQ制御の改善

- selectシステムコールは複数のイベントを同時に待つ
  - TCP/IPのreadでデータが到着
  - TCP/IPのreadでタイムアウト
  - Keyboardからの入力
- tcpselect関数は
  - データが到着したら、1
  - タイムアウトが発生したら、2
  - Keyboardからの入力があったら、0
  - エラーが生じたら、-1

```
fd_set fds, readfds;
int tcpsock;
struct timeval timeout;

tcpsockにsocketを生成する
FD_ZERO(&readfds);
FD_SET(tcpsock, &readfds);
FD_SET(0, &readfds);

int tcpselect() {
  memcpy(&fds, &readfds, sizeof(fd_set));
  timeout.tv_sec = 0;
  timeout.tv_usec = 100000; // 100 msec
  int n = select(tcpsock+1, &fds, NULL, NULL,
&timeout);
  if(n==0)return 2; // timeout
  if(FD_ISSET(tcpsock), &fds) return 1:
  else if(FD_ISSET(0, &fds) return 0;
  else return -1;
}
```

データベース

- システム構成や実験条件のパラメータなどDAQにおいて必要となるデータベースをどのように作成するか？
- データの形式はいろいろある
  - テキストやCSV
  - SQL（リレーショナルデータベース）
  - XMLやJSON（Web技術で発達したデータベース）
- データベースは汎用化・自動化のキーとなる技術
  - プログラムを変更することなく、パラメータを変更してシステム構成・実験条件を変える
  - データベースに従ってデータ収集の手順を決めてゆく

データベース(テキストやCSV)

- Comma-Separated Values (CSV)は、いくつかのフィールド（項目）をコンマ「,」で区切ったテキストデータおよびテキストファイル。拡張子は.csv、MIMEタイプはtext/csv。(Wikipediaから)
- テキストやCSVの形式は簡単に実装できるという点で優れているが、データのスキーマ（属性や関係の関連の定義）がないので、長期の保存には適していない。

テキスト

```
100
INIT
10.0
```

CSV

```
Value, 100
Setup, INIT
Temp, 20.0
```

データベース（SQL）

- リレーショナルデータベース管理システム (RDBMS) において、データの操作や定義を行うためのデータベース言語（ Wikipediaから）
- ORACLE, MySQL, PostgreSQLなどよく整備されている
- いろいろなプログラミング言語でのサポートも充実している
- しかし、スキーマを一度決めると、その後の変更が容易ではない

スキーマ

```
create table tgcparameter (
id char(30) default '' not null,
Name char(20) default '' not null,
Value char(20) default '' not null,
Readable int(1) default 0 not null,
Writable int(1) default 0 not null,
Configure int(1) default 0 not null,
ISS int(1) default 0 not null,
state char(5) default '' not null,
primary key(id) );
```

データの入力

```
insert into tgcparameter values
("PS_00_JRC_SEU", "Jrc/Seu", "nonQ", 1, 0, 0, 0, "c"),
("PS_00_PPS0_BCID_DELAYA", "Pps0/BcidDelayA",
"1111111111", 1, 1, 1, 1, "c");
```

データの検索

```
select * from tcpparameter;
```

データベース(XML やJSON ）

- Extensible Markup Language(XML) は、個別の目的に応じたマークアップ言語作成のため、汎用的に使うことができる仕様、および仕様により策定される言語(Wikipediaから）
- JSON（ジェイソン、JavaScript Object Notation）は、JavaScriptにおけるオブジェクトの表記法をベースとした軽量なデータ記述言語(Wikipediaから）
- ともに、Web技術の発展の中で開発された言語で、柔軟にデータを表現でき、多くのプログラミング言語で利用可能
- XML文書をDocument Object Model(DOM)でアプリケーションから利用可能、検索機能も充実

データベース(XML やJSON）

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<condition>
  <common>
    <runMode>1</runMode>
    <Gatherer>
      <numData>1000</numData>
    </Gatherer>
    <Monitor>
      <updateHz>1</updateHz>
      <displayType>2D</displayType>
      <displayData>Raw Data</displayData>
      <displayMode>
        <setLogX>0</setLogX>
        <setLogY>0</setLogY>
        <setLogZ>0</setLogZ>
      </displayMode>
            |
            |
</condition>
```

JSON

```
{"condition":
  {"common":
    {"runMode":1,
     "Gatherer":
       {"numData":1000},
     "Monitor":
       {"updateHz":1,
      "displayType":"2D",
      "displayData":"RawData",
      "displayMode":
        {"setLogX":0,
         "setLogY":0,
         "setLogZ":0},
            |
            |
}
```

デバッグ

- GDB
  - GNUソフトウェア・システムで動く標準のデバッガ
- Valgrind
  - メモリリークの検出、キャッシュのプロファイイリング（ヒット率）などを行うためのプログラミングツール

GDB

Segmentation faultの検出を行う手法

```
% g++ -g –o demo demo.cpp
% ulimit -c unlimited
% ./demo
segmentation fault

% ls
demo.cpp core.27980

% gdb ./demo core.27980
……
6   a[100000]=0;
(gdb) where
xxxxx in main() at demo.cpp:6
```

demo.cpp

```
#include <iostream>
#include <stdlib.h>

int main(void) {
  int* a = new int[10];
  a[100000]=0;
  delete [] a;
  return 0;
}
```

Valgrind

- 左図のプログラムはmallocとnew int[10]で動的なメモリの確保を行っているが、そのメモリの解放を行っていない

t.cpp

```
#include <iostream>
#include <stdlib.h>
void f(void) {
  int* x = (int*)malloc(10*sizeof(int));
  x[100]=0;
  //  free(x);
}
int main(void) {
  f();
  int* a = new int[10];
  a[100]=10;
  //  delete [] a;
  return 0;
}
```

% valgrind --tool=memcheck --leak-check=full ./t

```
==16970== Memcheck, a memory error detector
==16970== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==16970== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==16970== Command: ./t
==16970==
==16970== Invalid write of size 4
==16970==    at 0x400762: f() (t.cpp:5)
==16970==    by 0x400776: main (t.cpp:10)
==16970==  Address 0x4c2e068 is 0 bytes after a block of size 40 alloc'd
==16970==    at 0x4A05E1C: malloc (vg_replace_malloc.c:195)
==16970==    by 0x400755: f() (t.cpp:4)
==16970==    by 0x400776: main (t.cpp:10)
==16970==
==16970== Invalid write of size 4
==16970==    at 0x40078F: main (t.cpp:12)
==16970==  Address 0x4ffe9b0 is not stack'd, malloc'd or (recently) free'd
==16970==
==16970==
==16970== HEAP SUMMARY:
==16970==     in use at exit: 80 bytes in 2 blocks
==16970==  total heap usage: 2 allocs, 0 frees, 80 bytes allocated
==16970==
==16970== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==16970==    at 0x4A05E1C: malloc (vg_replace_malloc.c:195)
==16970==    by 0x400755: f() (t.cpp:4)
==16970==    by 0x400776: main (t.cpp:10)
==16970==
==16970== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2
==16970==    at 0x4A062CA: operator new[](unsigned long) (vg_replace_malloc.c:264)
==16970==    by 0x400780: main (t.cpp:11)
==16970==
==16970== LEAK SUMMARY:
==16970==    definitely lost: 80 bytes in 2 blocks
==16970==    indirectly lost: 0 bytes in 0 blocks
==16970==      possibly lost: 0 bytes in 0 blocks
==16970==    still reachable: 0 bytes in 0 blocks
==16970==         suppressed: 0 bytes in 0 blocks
==16970==
==16970== For counts of detected and suppressed errors, rerun with: -v
==16970== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 4 from 4)
```

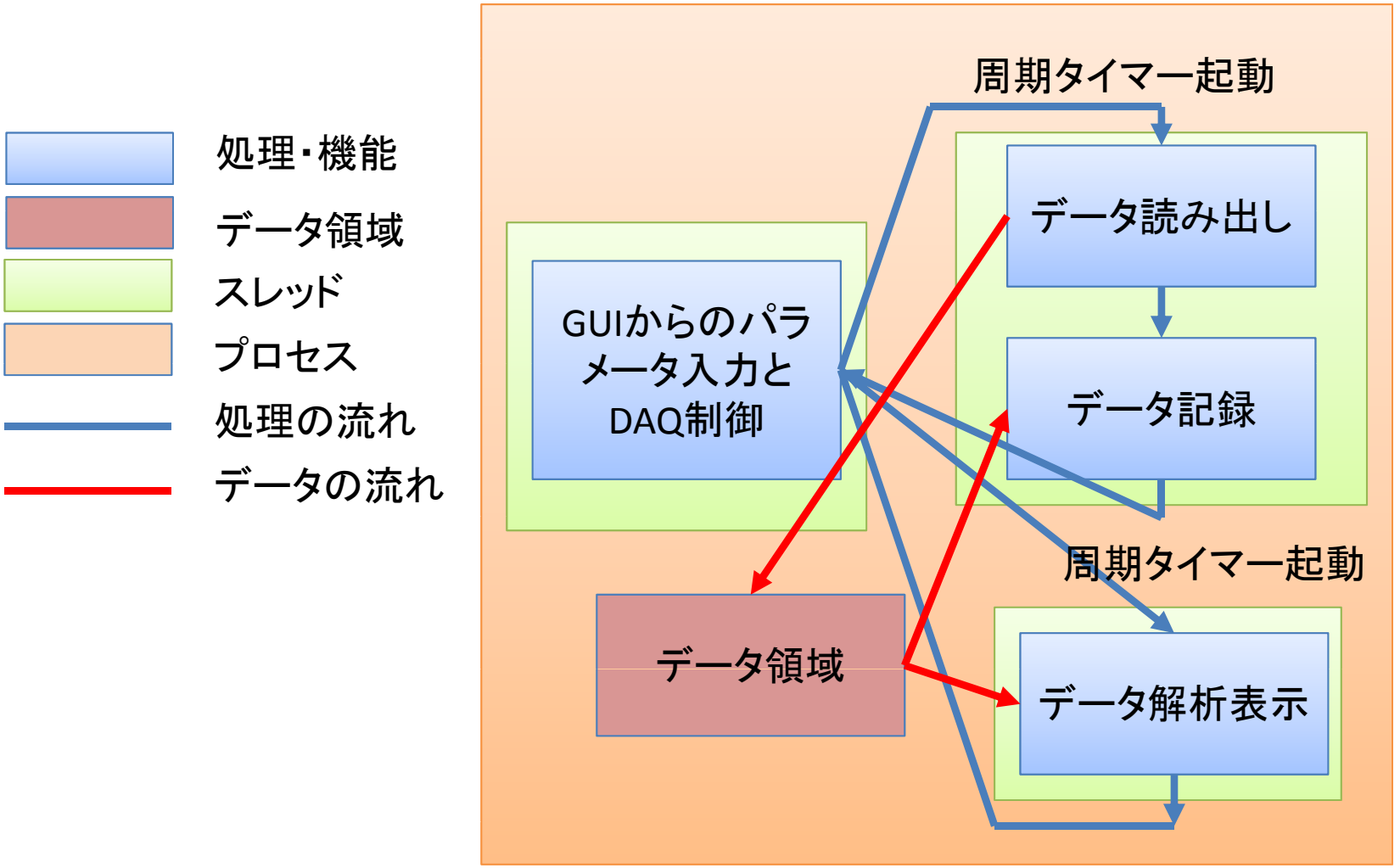
最も簡単なデータ収集法(まとめ）

- １つのプログラムですべての機能を実現
  - シングルプロセス・シングルスレッド
- パラメータやデータは各機能要素から見ると共有メモリ上にある
- 処理は随時処理で並行処理はどこにもない
  - データやパラメータの変更に伴う同期処理・排他処理は不要
- デバッグが容易
- しかし、DAQ操作性・性能・スケーラビリティーの面で限界がある

GUIを用いた簡単なデータ収集法

- GUIを導入すると、ボタンなどでイベントドリブンのプログラミングが要求されるので、「最も簡単なプログラミング」モデルは利用できない
- データ読み出し・記録・解析・表示・DAQ制御を１つのスレッドで処理することはできなくなる。
- これらの機能要素を複数のプロセスで動作させようとすると、途端にパラメータやデータはそれぞれの機能要素プログラムの中で、共有されなくなる。そこでスレッドを多重にする方法が取られる。
- スレッド多重化の方法にはいくつかあるが、スレッドは動作させるとメインスレッドとは独立に勝手に動作するためその制御法（例えば独自のステートマシンを実装）を確立する必要がある。
- 最も簡単な制御法は、タイマーを使って定期的にスレッドを起動させることである。

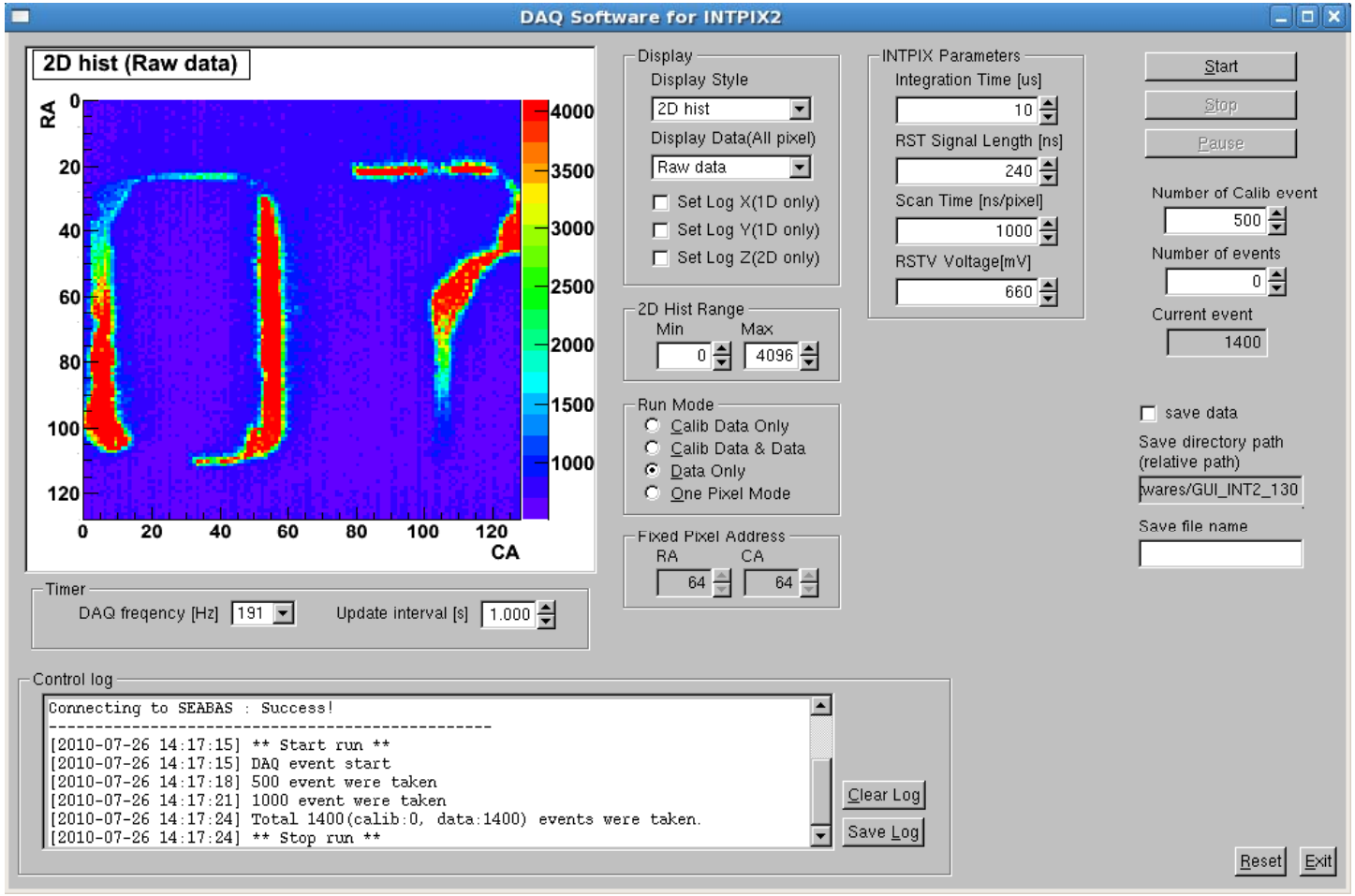
GUIを用いた簡単なデータ収集法



GUI

- C++を用いたROOTによるGUI
- Pythonを用いたTkinterによるGUI
- HTML/CSS/JavaScriptを用いたWebによるGUI

C++を用いたROOTによるGUI



From Hirose-san(SOI group)

C++を用いたROOTによるGUI

```
TApplication theApp(“app”, &argc, argv);
    |
MainHorzFrame = new TGHorizontalFrame(this,700,500);
Canv = new MyCanvas(MainHorzFrame,500,500);
Canv->CanvLayout();
MainHorzFrame->AddFrame(Canv, new TGLayoutHints(kLHintsExpandX | kLHintsExpandY, 5,5,3,1));
Opt = new MyOption(MainHorzFrame,175,500);
Opt->OptLayout();
MainHorzFrame->AddFrame(Opt, new TGLayoutHints(kLHintsLeft,5,5,7,4));
Para = new MyParameters(MainHorzFrame,175,500);
Para->ParaLayout();
MainHorzFrame->AddFrame(Para, new TGLayoutHints(kLHintsLeft,5,5,7,4));
Cont = new MyCtrl(MainHorzFrame,175,500);
Cont->CtrlLayout();
MainHorzFrame->AddFrame(Cont, new TGLayoutHints(kLHintsLeft,5,5,8,4));
AddFrame(MainHorzFrame, new TGLayoutHints(kLHintsExpandX | kLHintsExpandY,2,2,2,2));
SubHorzFrame = new TGHorizontalFrame(this,900,200);
Log =  new MylogEdit(SubHorzFrame, new TGString("Control log"));
Log->EdtLayout();
SubHorzFrame->AddFrame(Log, new TGLayoutHints(kLHintsLeft | kLHintsExpandY ,5,5,7,7));
exit = new TGTextButton(SubHorzFrame,"&Exit");
SubHorzFrame->AddFrame(exit, new TGLayoutHints(kLHintsRight | kLHintsBottom,5,5,7,7));
reset = new TGTextButton(SubHorzFrame,"&Reset");
AddFrame(SubHorzFrame, new TGLayoutHints(kLHintsExpandX,2,2,2,2));
    |
MapWindow();
    |
theApp.Run();ここでプロセスはイベント待ちになる
```

C++を用いたROOTによるGUI

下記は、データ収集・記録と解析表示のためのコード

```
utime = new TTimer(); 解析表示用
dtime = new TTimer();データ収集・記録用
utime->SetTime( (int)(Canv->GetUpdInt()->GetNumber()*1000));
dtime->SetTime(10);
utime->Connect("Timeout()", "MyMainFrame",this, "UpdateTimer()");
dtime->Connect("Timeout()", "MyMainFrame",this, "DAQTimer()");
```

```
void RunDAQ() {
  utime->TurnOn()
  dtime->TurnOn()
}
void StopDAQ() {
  utime->TurnOff()
  dtime->TurnOff()
}
```

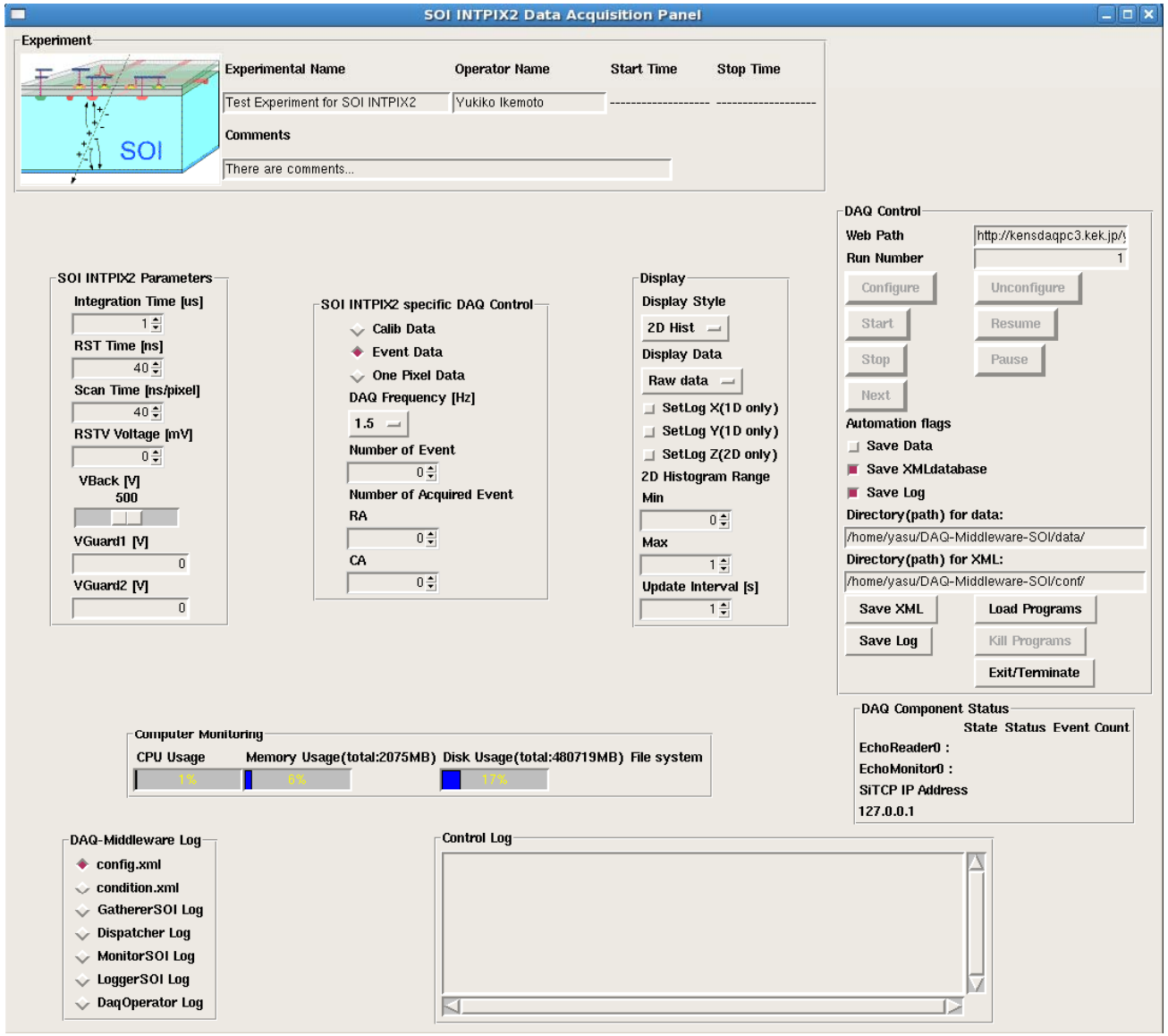
Startボタン

Stopボタン

```
void DAQTimer(){
  int selectret = sitcp->tcpselect(100);
  if(selectret == 1) { // Data Taking
    SiTCPから1フレーム分のデータを読み出し
    バッファに詰める
  } else if(selectret == 0) { // from Keyboard…

  } else if(selectret == -1) { // error…

  }
}
```

```
void UpdateTimer() {
TCanvas *fCanvas = Canv->GetECanvas()->GetCanvas();
fCanvas->cd();

if(DisStr_Id==0) h2[HstCnt_Id]->Fill(j,127-i,data[i][j]);
else if(DisStr_Id==1) h1[HstCnt_Id]->Fill(data[i][j]);

if(DisStr_Id==0){
    h2[HstCnt_Id]->Draw("COLZ");
    revAxis->Draw();
}else if(DisStr_Id==1){
    h1[HstCnt_Id]->Draw();
}
```

PythonをTkinterによるGUI



開発中

PythonをTkinterによるGUI用いた

```
def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid(sticky=N+S+E+W)
        self.parseConfigXML()  config.xmlからパラメータを取得
        self.daqmwcom = DAQMWCom()  DAQ操作用コマンド
        self.initTimer()
        self.make_cond_xml = MakeConditionXML(self)
        self.save_xml = SaveXML(self)
        self.setupSoiModule()  SOI moduleの初期化
        self.createWidgets()  パネルのセットアップ
        self.daqmwcom.setURLBase(self.daq_ctrl.getWebPath())
        self.make_cond_xml.setMaster(self)
        self.make_cond_xml.setPath(self.daq_ctrl.getXmlPath())
        self.save_xml.setMaster(self)
        self.compmon_update.start()

if __name__=='__main__':
    app = Application()
    app.master.geometry("-0+0")  # located at upper right
    app.master.title("SOI INTPIX2 Data Acquisition Panel")
    app.mainloop()  ここでプロセスはイベント待ちになる
```

```
def createWidgets(self):
パネルの生成
        self.exp = ExperimentPanel(self)
        self.ctrl_log = ControlLogPanel(self)
        self.daq_ctrl = DAQControlPanel(self)
        self.daqmw_log = DAQMWLogPanel(self)
        self.params_intpix = ParamsSoiIntpixPanel(self)
        self.disp = DisplayControlPanel(self)
        self.daq_ctrl_intpix = DAQControlSoiIntpixPanel(self)
        self.daq_status = DAQStatusPanel(self)
        self.compmon = ComputerMonitorPanel(self)
外部インスタンスの取得
        self.ctrl_log.setMaster(self)
        self.compmon.setMaster(self)
        self.daq_ctrl.setMaster(self)
        self.daqmw_log.setMaster(self)
        self.params_intpix.setMaster(self)
        self.daq_ctrl_intpix.setMaster(self)
        self.daq_status.setMaster(self)
パネルの配置
        self.exp.grid(row=0, column=0, columnspan=3)
        self.params_intpix.grid(row=1, column=0)
        self.daq_ctrl_intpix.grid(row=1, column=1)
        self.disp.grid(row=1, column=2)
        self.daq_ctrl.grid(row=1, column=3)
        self.compmon.grid(row=2, column=0, columnspan=3)
        self.daq_status.grid(row=2, column=3)
        self.daqmw_log.grid(row=3, column=0)
        self.ctrl_log.grid(row=3, column=1, columnspan=3)
```

PythonをTkinterによるGUI用いた

Startボタン

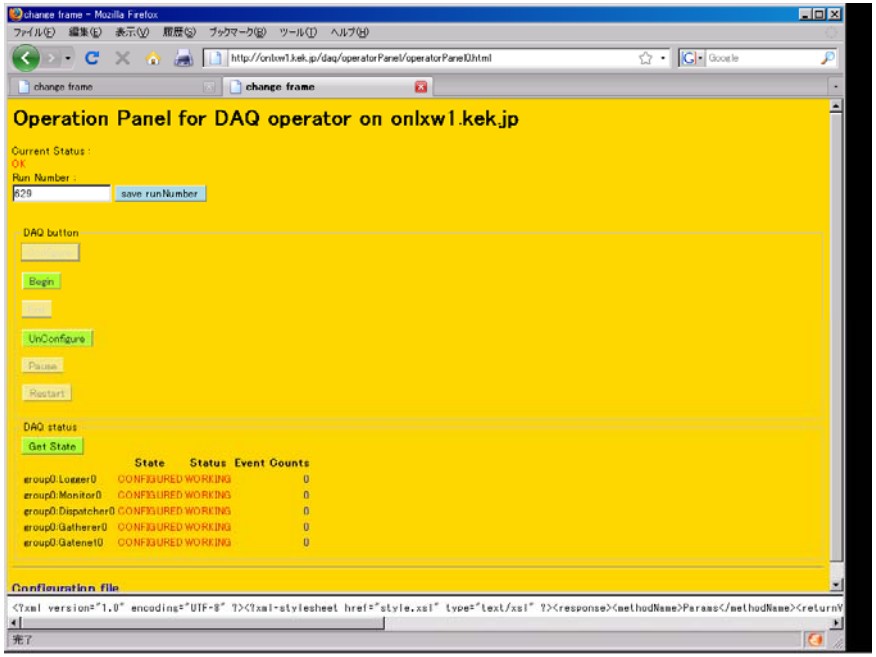
```
def actionDAQStart(self):
    self.exp.setStartTime()
    self.daqmwcom.start(str(self.getRunNumber()))  DAQ Opeartorへstartコマンド送信
    state = self.daqmwcom.getLog('state')      DAQ OpeartorからLog情報を受信
    if(self.checkState('RUNNING')):
        self.ctrl_log.putLog("DAQ components are not running...")
        return -1
    else:
        self.ctrl_log.putLog("start: ok")  Control Log  パネルへメッセージを書き込む
        self.daq_state = "RUNNING"

    self.daq_status.setCompLog()   Log情報を画面に表示
    self.daqctrl_update.start()   DAQステータスのアップデートタイマスタート
```

Stopボタン

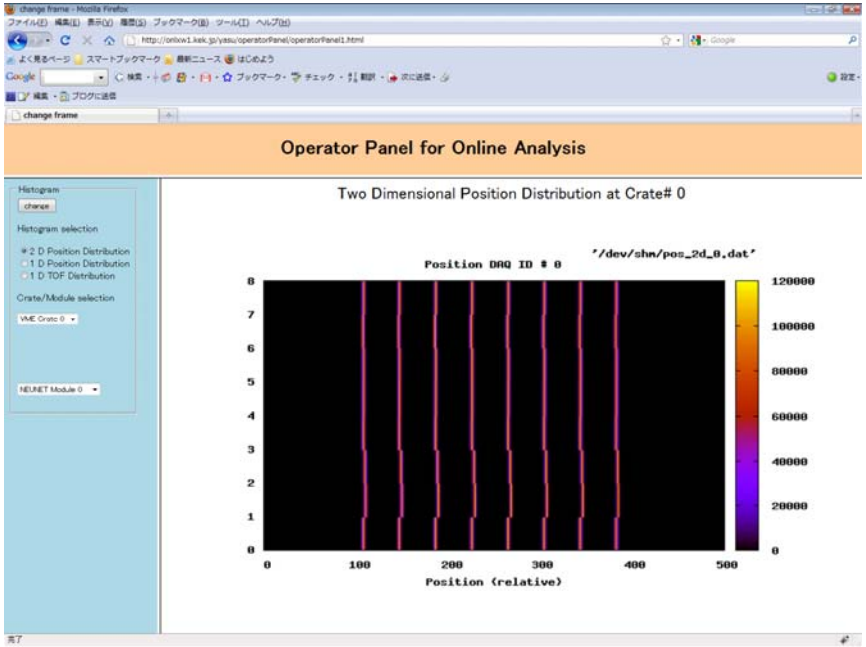
```
def actionDAQStop(self):
    self.daqctrl_update.cancel()   DAQステータスのアップデートタイマをキャンセル
    self.exp.setStopTime()
    self.daqmwcom.stop()      DAQ Opeartorへstopコマンド送信
    state = self.daqmwcom.getLog('state')      DAQ OpeartorからLog情報を受信
    if(self.checkState('CONFIGURED')):
        self.ctrl_log.putLog("DAQ components are not stopped...")
        return -1
    else:
        self.ctrl_log.putLog("stop: ok")   Control Log  パネルへメッセージを書き込む
        self.daq_state = "CONFIGURED"
    self.daq_status.setCompLog()   Log情報を画面に表示
```

HTML/CSS/JavaScriptを用いたWebによるGUI

DAQ制御のためのGUI



解析表示のためのGUI



HTML/CSS/JavaScriptを用いたWebによるGUI

HTML

```
<body onLoad="getNodeName();getRunNumber();getstate1('./daq.py/Log');">
<h1 id="nodename"></h1>
    |
<fieldset>
<legend>DAQ button</legend>
    |   startボタンの定義
<form name="myform2" action="./daq.py/Begin" target="status"  method="post", onlsubmit="return false">
<input type="hidden" name="cmd" />
<input id="beginId" type="image" name="begin" src="../parts/beginbuttonlight.jpg" alt="Begin"
onclick="change2Running();getRunNumber()" />
</form>
    |
</fieldset>

<fieldset>
<legend>DAQ status</legend>
<div id="DAQStatus"></div>
</fieldset>
```

HTML/CSS/JavaScriptを用いたWebによるGUI

JavaScript

```
function getstate1(url) {
  var msec = (new Date()).getTime();
  new Ajax.Request(url, {
    method: "get",
    parameters: "cache="+msec,
    onCreate:function(httpObj){ },
    onSuccess:function(httpObj){
      var res = httpObj.responseXML; サーバからXML文書の取得、以下それを解析する
        |
      var compNames = res.getElementsByTagName('compName');
      text="";
      var len = compNames.length;
      for(i=0; i<length;i+=len) {   以下、動的に生成するHTMLを作成
        text +="<table><tr><th></th><th>State</th><th>Status</th><th>Event Counts</th></tr>";
      }
      for(j = 0; j< len; j++) {
            text +="<tr><td>";
            text += compNames[i+j].firstChild.nodeValue;
            text += "</td><td class='compState'>";
            text += states[i+j].firstChild.nodeValue;
            text += "</td><td class='compStatus'>";
            text +=compStatus[i+j].firstChild.nodeValue;
            text +="</td><td class='eventCount'>";
            text += eventnums[i+j].firstChild.nodeValue;
            text +="</td></tr>";
       }
       text +="</table>";
   $("DAQStatus").innerHTML = text;
   }});
}
```

```
function getNodeName() {
    $("nodename").innerHTML = "Operation Panel for DAQ operator on "+location.hostname;
 }
```

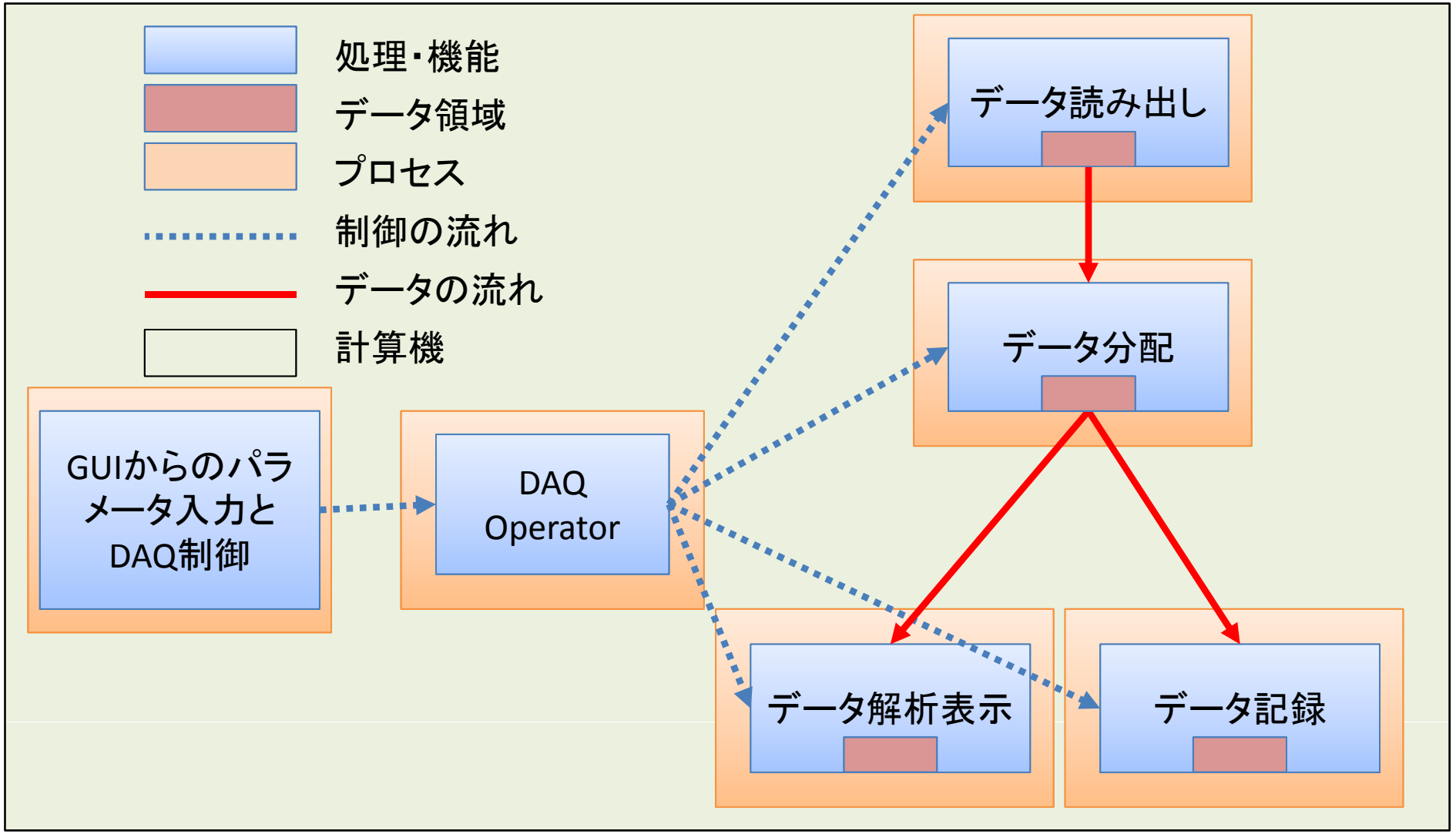
GUIを用いた簡単なデータ収集法(まとめ)

- GUIのメインスレッドはボタンなどのイベントで動作するプログラム
- マルチスレッドだが1つのプロセスであるので、パラメータ・データは各機能から見ると共有メモリ上にある
- スレッドが平行して動作するので、パラメータやデータの同期処理や排他処理が必要
- タイマーでデータ収集・記録・解析表示を行えば、スレッドの制御は単純になり複雑な仕組みは不要
- しかし、読み出しや解析の規模が大きくなるとこの方法では限界がある(スケーラビリティーや性能の限界)

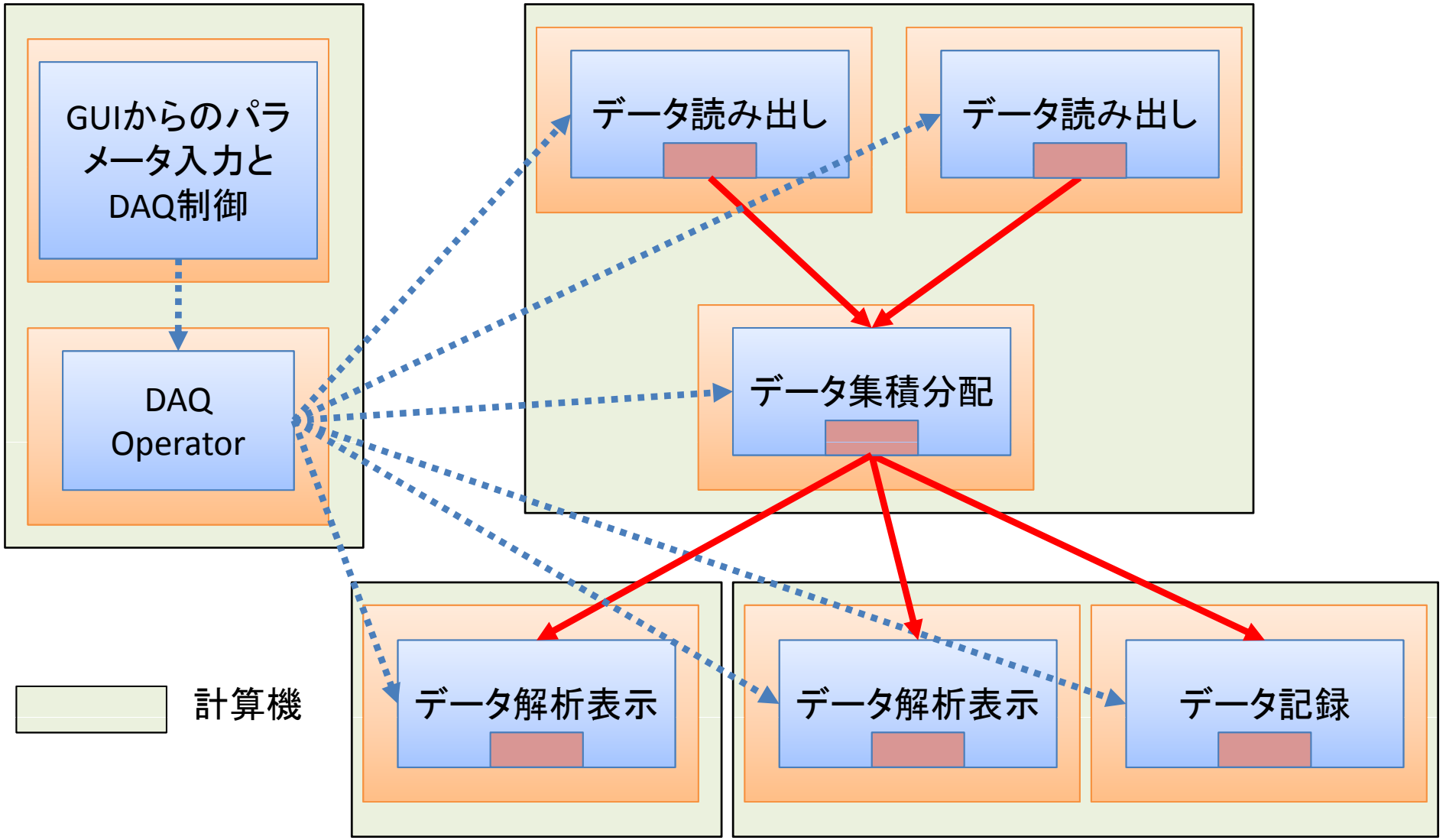
DAQ-Middlewareを用いたデータ収集法

- データ読み出し・記録・解析表示・DAQ制御の機能要素を多重にすること（つまり読み出しを1つではなく複数で並行して読みだすようにする）でスケーラビリティーや性能向上をめざすことができる
- 「最も簡単なデータ収集」や「GUIを用いた簡単なデータ収集」のモデルではスケーラビリティーや性能向上をめざすことはできない
- 現在求められているのは、複数の計算機を用いたデータ収集フレームワークである
- データ収集・データ記録・データ解析表示がそれぞれの機能ごとに複数のプロセスが複数の計算機上に存在し、それらをDAQ制御機能がコントロールするという仕組みが要求される。
- それぞれの機能に共通した仕組みを用意することで、コンポーネント構造を共通にすることが可能となる。それはコマンド・ステータス機能、データパス機能、さらにデータベース機能を実現する仕組みである。

DAQ-Middlewareを用いたデータ収集法



DAQ-Middlewareを用いたデータ収集法

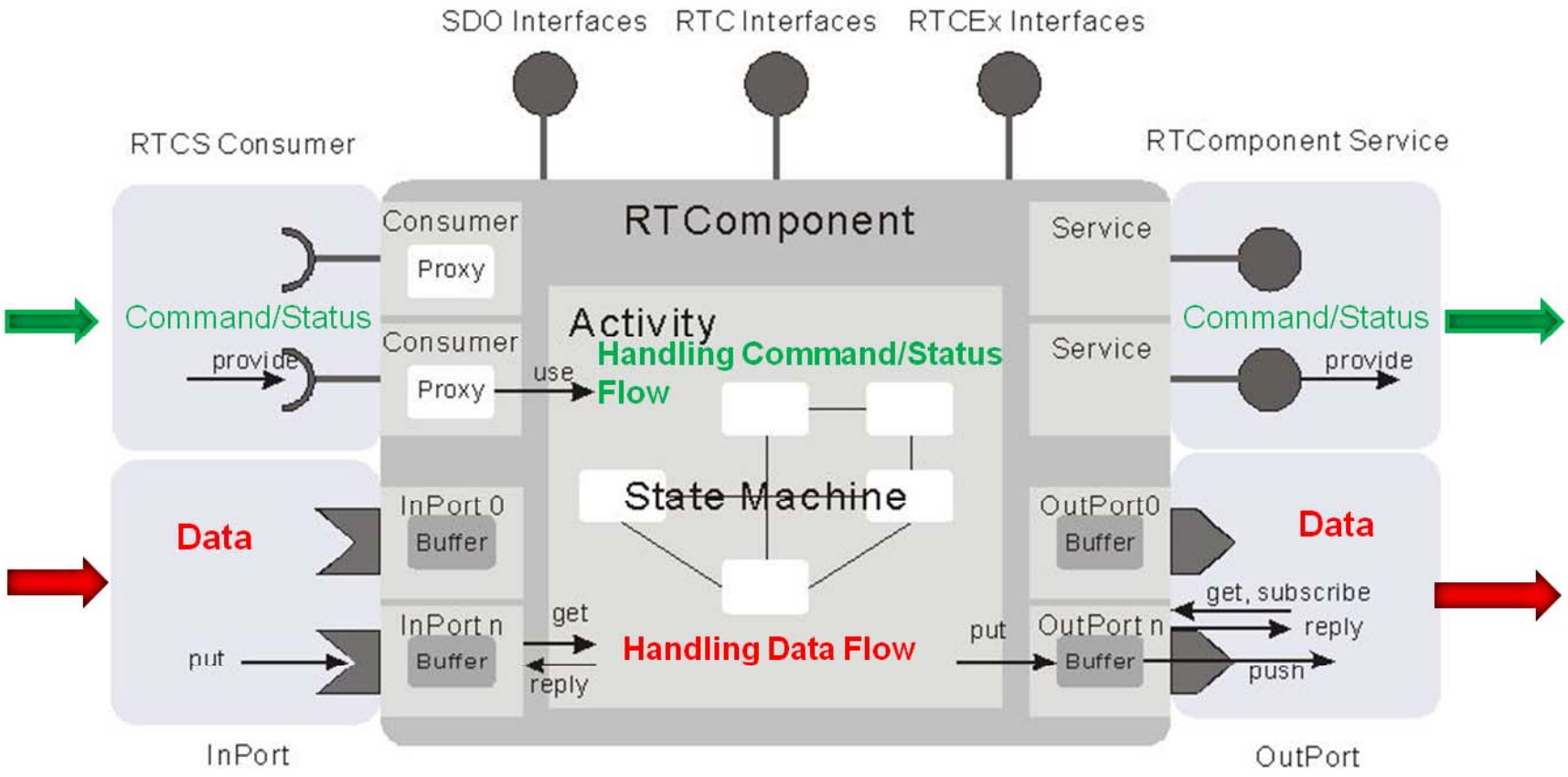


DAQ-Middlewareを用いたデータ収集法

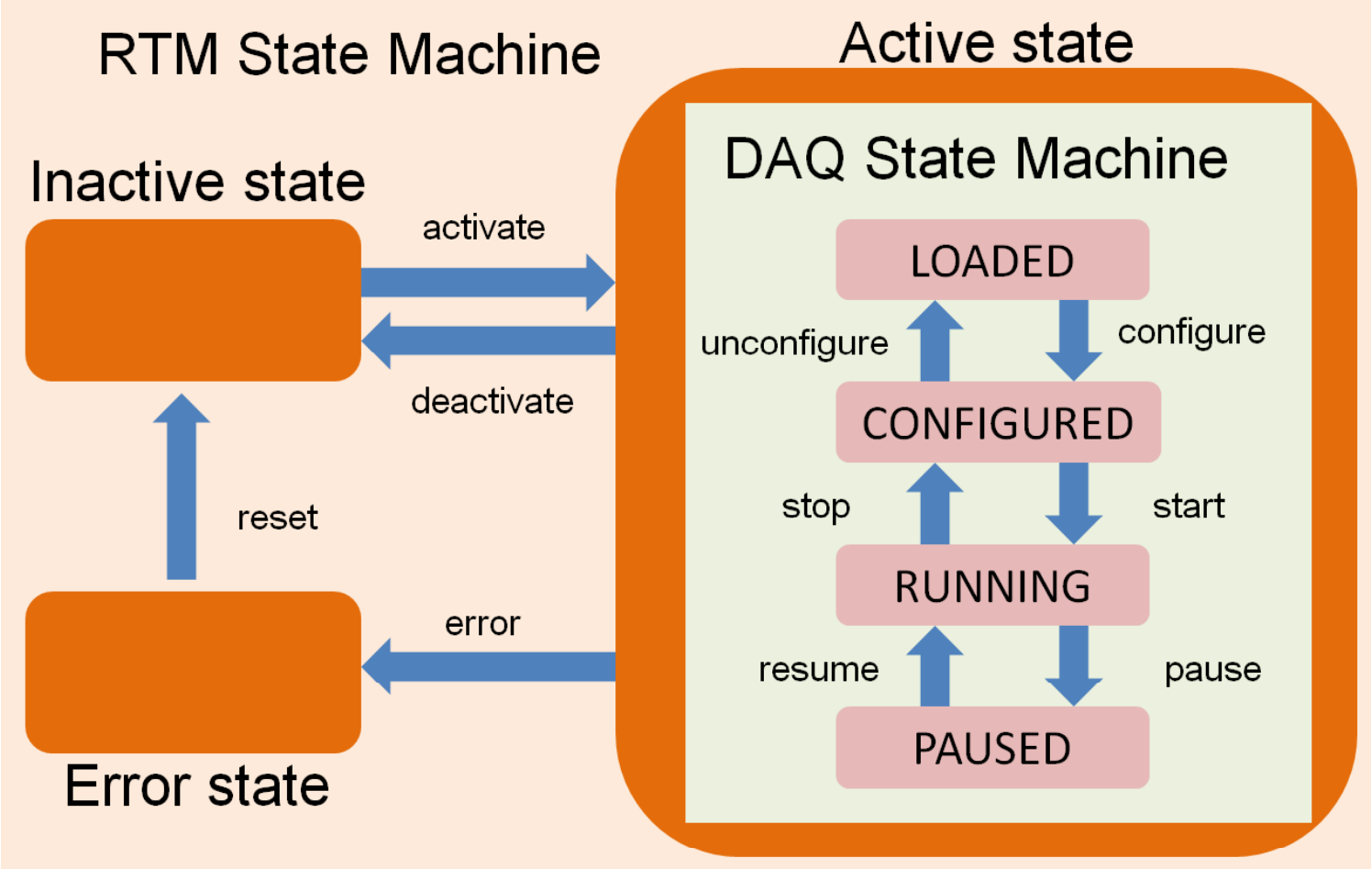
- DAQ-Middlewareの基本機能
  - DAQコンポーネントを基本単位とし、DAQ Operatorという特殊なDAQコンポーネントによりDAQ制御される
  - Configuration&Condition database (XML/JSON)によるシステムおよび実験条件の記述
  - Configuration databaseに基づくDAQコンポーネントの起動
- DAQコンポーネントの基本的機能
  - コマンド・ステータス機能とデータパス機能
  - 自立型DAQコンポーネント（単純なclient/serverモデルのコンポーネントではなく、DAQ制御を受けながらも、常にデータパスの処理を独自に行うという性格を持つ）

DAQ-Middlewareを用いたデータ収集法

自立型DAQコンポーネントモデル



DAQコンポーネントのステート



Configuration&Condition Database

- Configuration database： XML
  - 計算機や機器の構成などランごとには変更しないがシステムを記述するためのパラメータ
- Condition database： XML(JSON)
  - ランごとに変化する可能性のある実験条件を記述するためのパラメータ
- これらのデータベースはコンポーネントを汎用化する(プログラムを変えることなくデータベースを変えるだけでシステムの構成や実験条件を変更できる)

プログラミングの実際(読み出し)

一回しか呼ばれない

configureコマンド時

```
daq_configure() { システム構成パラメータの設定
   ::NVList* paramList;
   paramList = m_daq_service0.getCompParams();
   parse_params(paramList);
}
```

startコマンド時

```
daq_start() {
   set_condition();  実験条件パラメータの設定
   connect_modules();  SiTCPモジュールへの接続
}
```

stopコマンド時

```
daq_stop() {
  disconnect_modules();   SiTCPモジュールの切り離し
}
```

繰り返し呼ばれる

RUNNING state時

```
daq_run() {
  stopコマンドが来ているか、
       終了条件が成立しているか？
       チェックする
  以前のOutPortへの転送が成功していなかったら
       転送を試みる
  複数のSiTCPモジュールからのデータを待つ
  到着したデータを読み出す
  読み出したデータをフォーマットする
  フォーマットされたデータをOutPortに転送する
  転送に成功したら取得イベント数をインクリメントする
}
```

DAQ-Middlewareを用いたデータ収集法（まとめ）

- スケーラビリティーや性能の面で優れている
  - 必要に応じて計算機の数やDAQコンポーネントのシステム構成を柔軟に変更できる（CPUボトルネックになったら計算機・CPUの数を増やしDAQコンポーネントの再構成を行う）
- データベースに基づくシステム記述や実験条件の記述により、DAQシステムの汎用化が実現できる（データベースを変えるだけでシステム構成や実験条件の変更に対処できる）
- 本来実装すべき機能要素であるパラメータ入力・データ読み出し・記録・解析・表示・DAQ制御の中身に集中できる（平行処理に伴う排他処理・同期処理などの複雑さから解放される）

まとめ

- いかなるデータ収集法においても、パラメータ（データベース）入力・データ読み出し・記録・解析表示・DAQ制御はDAQの機能要素である
- たくさんのプロセス・スレッドを用いた多数の計算機・プロセスの並行処理のデバッグは簡単ではないので、個々の機能要素を個別にデバッグし動作確認を行う(多数の並行処理のデバッグに確立された手法はない)
- スケーラビリティーや性能を求めるなら、しっかりとしたフレームワークの上にDAQ機能を埋め込んでゆくのがよい
- その際は、個々の機能はできるだけ単機能にし、単機能としてデバッグを済ませた後、フレームワークに組み込むこと
- DAQ-Middlewareを用いた開発手法では、単機能なDAQコンポーネントを個別にデバッグし、これらのDAQコンポーネントをDAQ-Middlewareのフレームワークの上につなぎ合わせて目的とするDAQシステムを完成させる

backup

GUIを用いた簡単なデータ収集法