KiNOKO

Version 2.1

Users Guide

2008 年 8 月 5 日版

このマニュアルは KiNOKO ウェブページ上の HTML 文書を印刷用に抜粋・編集したものです． ウェブ上の文書は随時更新されるので，最新情報についてはそちらを参照するようにしてください．また，完全なリファレンスなどもウェブ上にあります．

KiNOKO ウェブページ： http://www.awa.tohoku.ac.jp/~sanshiro/kinoko/

上記アドレスは将来変更される可能性があります．アクセスできない場合は「kinoko　データ収集」などのキーワードで検索してください．

目　次

概要

KiNOKO は，VME や CAMAC などを使用する高エネルギー実験・原子核実験のための汎用のデータ収集ソフトウェアです．シングルプロセスの小さなシステムから数十台の計算機を並列に使用する規模のシステムまでを，統一されたインターフェースでカバーします．また，内部は高度にモジュール化されており，高い可用性と再利用性を実現しています．システムの構築から細かな動作の指定までほとんど全てをスクリプト言語によりコントロールでき，多くの実験や測定にそのまま使用できます．

Home

主な特長

強力なスクリプト言語

KiNOKO は，文法独立な構文解析エンジンを備えています．この構文解析エンジンは各実験特有の設定などを記述するために KiNOKO のあらゆる部分で使用されており，このためユーザが KiNOKO のソースコードを編集する必要はほとんどありません．

再利用可能なデバイスインターフェース

CAMAC や VME などのデバイスにアクセスする部分のコードは，そのデバイスが持つ機能へのインターフェースのみを記述した独立したコードとして構成されています．そのデバイスがどのように使われるかとは完全に分離されているため，同じモジュールを使用するほかの実験や測定にそのまま使用できます．広く使用されているモジュールのいくつかは，すでにコードが実装され，KiNOKO の一部として提供されています．

分散オブジェクト環境による統合された分散処理環境

KiNOKO のシステムは，その下位層に，オブジェクトリクエストブローカ（ORB）を持っています．これは，ネットワーク上の別の計算機の別のプロセス内にあるオブジェクトを，自プロセス内のローカルオブジェクトであるかのように扱うことを可能にするシステムです．これにより， KiNOKO は分散環境上で並列動作するシステムであるにもかかわらず，複雑なネットワーク処理をほとんど意識する必要がなく，統合されたひとつのシステムのように振舞います．

コンポーネントによる実行時システム構築

KiNOKO の上位層は，コンポーネントと呼ばれる機能単位に分割されています．KiNOKO のコンポーネントは，実行時にスクリプトによるシステム構築を可能にするもので，具体的には，実行時におけるコンポーネントの配置や，メソッドの結合，内部オブジェクトの共有などが実装されています．各コンポーネントは機能単体を提供しており，それらを実行時に組み立てるという方法により，さまざまな目的・構成のシステムを簡単に構築することができます．さらに，独自のコンポーネントを作成することにより，実験特有の機能を追加したり，特殊な外部システムにインターフェースすることができるようになります．

自己記述性を持ったデータストリーム

データストリームを流れるデータは，同じストリームの先頭に来るデータデスクリプタによってその構造や意味が完全に記述されています．これにより，このデータを処理する他のコンポーネントが，特定のデータ構造を想定しない汎用の実装を行うことができるようになり，再利用可能な解析部品の開発が可能になります．また，これらの情報がデータファイルに記録されれば，データファイルの管理の手間が大幅に減り，管理ミスによるトラブルも減少します．

分離されたユーザインターフェース

すべてのユーザーインターフェースは，KiNOKO 本体から完全に分離されています．本体とユーザインターフェースの間の通信はソケット経由の簡単なテキストコマンドによってなされており，互いにライブラリ等をリンクする必要はありません．これにより様々な環境・言語でのユーザインターフェースの実装が可能になると共に，システム本体を環境依存性の強いグラフィクインターフェースから分離します．

分散システムからシングルプロセスまで対応するスケーラビリティ

ORB による透過的なネットワーク分散処理と，コンポーネントモデルによる実行時コンポーネント配置によって，システムを複数の計算機上に展開することは極めて容易です．新しい計算機の追加を行なう場合でも，その際必要なのはスクリプトを数行編集することだけです． 逆に，ORB によって，通常は別のプロセスにより行なわれる処理も一つのプロセスにまとめることもでき，これを利用してシングルプロセスのデータ収集プログラムなども実装されています．

SQL データベースとの連携

KiNOKO は，JDBC に似た形のデータベースインターフェースを含んでおり，これはスクリプトからも利用できます．これにより，スクリプト中に実験パラメータ(HV テーブル，測定器ジオメトリなど)を直接記述する必要がなくなり，スクリプトが簡素になって，パラメータの一元管理も可能になります．

その他

KiNOKO は，その最下層において，すべてのシステムコールをカプセル化しています．また，KiNOKO の本体部分は，このシステムコールの利用を除けば，C++の標準ライブラリのみで実装されています．これらのことは，ユーザインターフェースが分離されていることとあわせて，システムの移植性を向上させます．

KiNOKO のほとんどすべての機能は，あとから拡張できるように実装してあます．拡張ポイントおいて，インターフェースを実装したクラスを追加・登録することにより，新機能の追加を容易に行えます．また，KiNOKO のソースツリーは，可能な限り部分利用ができるように，多くの部分が独立になっています．実際，すでに多数の部分利用例があり，その中には高エネルギー実験とは関連がないものもかなりあります．

KiNOKO が提供するもの

以下は，KiNOKO が提供するものの一部です．

- 一部の VME / CAMAC コントローラの Linux 用デバイスドライバ
- 一部の VME / CAMAC モジュールへのインターフェースコード
- データ読みだしシーケンス構築スクリプト
- データバッファとデータストリーム
- オンラインアナリシスフレームワーク
- オンラインデータビューア
- 簡易オンラインデータストレージ
- コンポーネントフレームワークとコンポーネント間通信システム
- システムコントロールスクリプト
- コントロールパネルとコントロールパネル構築スクリプト
- システムロガー
- サマリジェネレータ

KiNOKO が提供しないもの

以下は，KiNOKO が提供しないものの一部です．

- 上記以外の VME / CAMAC デバイスドライバ
- 上記以外の VME / CAMAC モジュールへのインターフェースコード
- オンライン解析プログラム
- イベントビルダ
- 測定器を描画するイベントビューア
- 特化されたデータストレージ
- オフライン解析向けデータストレージ

インストール手順

Home

動作プラットフォーム

基本システム

UNIX System V Release 4 (SVR4) に"ほぼ"準拠した OS (Linux など)と C++ コンパイラのみで，データの収集やデータファイルの読み書きを含む Kinoko のほとんど全ての機能を利用することができます(ただし，GUI は含まれません).

オプションソフトウェア

基本システムの他，以下のソフトウェアが利用できると，Kinoko をより便利に活用できるようになります．特に，"強く推奨" と書かれているソフトウェアは，特別な理由が無い限りは，インストールしておくことを強く推奨します． なお，以降の説明では，"強く推奨" となっているソフトウェアはインストールされているものとします．

GTK+ ライブラリ [強く推奨]: http://www.gtk.org/
libxml ライブラリ [強く推奨]: http://xmlsoft.org/
　　Kinoko の GUI インターフェースが使用できるようになります．
　　「GTK+ 2.4 以上」が推奨です．基本的な部分は「GTK+ 1.2 と imlib の組合せ」でも動作します．
Perl [強く推奨]: http://www.perl.com/CPAN/
　　Kinoko に付属するいくつかのユーティリティが利用できるようになります．
　　Kinoko をウェブブラウザにより外部からコントロールできるようになります．
zlib ライブラリ [推奨]: http://www.gzip.org/zlib/
　　データ圧縮機能を利用できるようになります．
　　圧縮して記録されたデータを読む場合にもこのライブラリがリンクされている必要があります．
GNU/readline ライブラリ [推奨]: http://www.gnu.org/directory/readline.html
　　コマンドラインがかなり快適になります．これを利用するには，curses または ncurses ライブラリもインストールされている必要があります(普通はインストールされています).
ROOT [推奨]: http://root.cern.ch/
　　Kinoko が自動生成するスクリプトを利用してヒストグラムの表示等ができるようになります．
　　Kinoko がオンラインで作成したヒストグラム等を ROOT 形式で保存できるようになります．
　　インタプリタ版の KinokoScript で ROOT のファイルを読み書きできるようになります．
SQL データベース
　　システムのいろいろな場面でデータベースにアクセスできるようになります．以下は，現在サポートされているデータベースシステムのリストです．

- PostgreSQL (7.1.3 以上): http://www.postgresql.org/
- MySQL (3.23 以上): http://www.mysql.com/
- SQLite (2.7.2 以上): http://www.hwaci.com/sw/sqlite/

Web サーバ
　　動作中の Kinoko をウェブブラウザ(Internet Explorer を除く)からリモートで操作できるようになります．また，Kinoko が生成したランサマリをブラウザ経由で参照できるようになります．

- Apache http://www.apache.org/

XSLT プロセッサ
　　Kinoko が XML 型式で生成するランサマリなどのファイルから HTML などを自動生成させることができます．以下はフリーで利用できる XSLT プロセッサの一部です．

- libxslt (Gnome): http://xmlsoft.org/XSLT
- Xalan (Apache): http://xml.apache.org/

- XT (Java): http://www.blnz.com/xt/
- SAXON (Java): http://saxon.sourceforge.net/

[注意]

- 最近の Linux ディストリビューションでは，ライブラリを，プログラムの実行に必要な部分と，プログラムのコンパイルに必要な部分とを分割し，それぞれ別のパッケージとするものが増えています．Kinoko を使用するためには，プログラム開発用のパッケージが必要です．これらは，通常 XXX-devel などのパッケージ名になっています．
- 最近の Linux ディストリビューションでは，開発用のパッケージは，明示的に指定しないとインストールされないことが多いようです．
- Linux で Kinoko に付属のデバイスドライバを使用するためには，カーネルソースのパッケージが必要になることがあります．古い RedHat 系の Linux (FedraCore 以前) では，これは「カーネル開発」のカテゴリに含まれています．詳細については，各ドライバのマニュアルを参照してください．

コンパイル

パッケージを展開すると kinoko という名前のディレクトリが作成されます．すでに古いバージョンの Kinoko がインストールされている場合は，古い方のディレクトリの名前を変えておいてください．

1. Kinoko をインストールするディレクトリに移り，必要なら古い Kinoko の名前を変えておく

```
% cd kinokoをインストールするディレクトリ
% mv kinoko kinoko.old.日付
```

2. Kinoko のパッケージを展開する．

```
% gunzip kinoko.tar.gz
% tar xvf kinoko.tar
```

これにより，ディレクトリ kinoko が作成されます．

3. 環境にあわせてソースを構成(configure)する．

```
% cd kinoko/src
% ./configure-for LINUX-FULL
```

ここで，configure-for は，引数に指定された名前によりコンパイラやコンパイルオプションなどを適当に設定して configure を呼び出す簡単なシェルスクリプトです． 引数に LINUX-FULL を指定すると，g++ コンパイラを使って，利用可能な全てのライブラリを使用するようにソースを構成します．LINUX や LINUX-MIN を指定すれば使用するライブラリを減らして構成します．コンパイルエラーが起きる場合はこれらの構成を試してみてください．

現在のところ，以下の構成オプションが定義されています．

名前	コンパイラ	コンパイラオプション	ライブラリ
LINUX	g++	-Wall	zlib, readline, ROOT
LINUX-FULL	g++	-Wall	利用できるもの全て
LINUX-MIN	g++	なし	なし
AIX	xlC	なし	zlib, readline, ROOT
AIX-GCC	g++	なし	zlib, readline, ROOT

以下のように第 2 引数にコンパイラオプションを渡すこともでます.

```
% ./configure-for LINUX-FULL "-g -O2"
```

また, configure-for を使わずに, 手動でパラメータを設定して configure を直接使っても構いません(以下は csh/tcsh の場合の例).

```
% setenv CXX "g++"          # bash では export CXX="g++" とする
% setenv CPPFLAGS ""
% setenv CXXFLAGS "-Wall -g"
% setenv MAKE "gmake"
% ./configure --without-readline --with-PostgreSQL
```

[KDF4] Kinoko Version 2.0 では, 古い Kinoko との混在環境での問題を避けるため, デフォルトで KDF4 を使用しないようになっています(読み込みは可). もし KDF4 を使用したい場合, configure-for スクリプトの先頭付近にある --enable-kdf2 オプションを宣言している部分をコメントアウトしてください. また, configure を直接使う場合で KDF4 を使いたくない場合は, --enable-kdf2 オプションを指定してください.

4. コンパイル. PentiumIII で 10〜15 分ほどかかります.

```
% make
```

最後に, Make: successful と表示されればコンパイルは成功です.

デバイスドライバのインストール

使用するコントローラにあわせたデバイスドライバをインストールしてください.
Linux で東陽テクニカ製の CAMAC コントローラ CC/7x00 (PCI/ISA)を使用する場合
Linux で豊伸電子製の CAMAC コントローラ CCP (PCI/ISA) を使用する場合

Kinoko に付属している camdrv をインストールしてください. camdrv は kinoko/drv/camdrv にあります. camdrv のインストール手順については, camdrv ホームページ を参照してください. なお, camdrv は Kinoko 本体よりも頻繁に更新しリリースします. インストールの前に最新リリースを確認し, 必要に応じて Kinoko 付属のものと入れ替えてください.

以下に, 手順を簡単に示します. デバイス上のジャンパスイッチの設定なども必要なので, 注意してください.

```
% cd kinoko/drv/camdrv/Linuxバージョン-デバイス名
% make
% su
# make install
# exit
```

Linux で SBS Technologies (Bit3) 社製 VME インターフェース Model-617/618/620 を使用する場合

Kinoko に付属している vmedrv をインストールしてください. vmedrv は kinoko/drv/vmedrv にあります. vmedrv のインストール手順については, vmedrv ホームページ を参照してください. なお, vmedrv は Kinoko 本体よりも頻繁に更新しリリースします. インストールの前に最新リリースを確認し, 必要に応じて Kinoko 付属のものと入れ替えてください.

以下に，手順を簡単に示します．デバイス上のジャンパスイッチの設定なども必要なので，注意してください．

```
% cd kinoko/drv/vmedrv/Linuxバージョン-Bit3_617
% make
% su
# make install
# exit
```

FORCE 社製 CPU ボードで，ボード上の VME コントローラを使用する場合

CPU ボードに付属のデバイスドライバ(FRCvme)をインストールしてください．インストールの手順については，CPU ボードに付属のマニュアルを参照してください．

環境変数の設定

Kinoko をコンパイルすると，kinoko/src に kinoko-cshrc，kinoko-bashrc というファイルができます．これを自分のホームディレクトリにある .cshrc(または.tcshrc) か .bashrc に追加してください．

```
tcsh を使用している場合
% cat kinoko-cshrc >> $HOME/.cshrc

bash を使用している場合
$ cat kinoko-bashrc >> $HOME/.bashrc
```

以下に，生成される kinoko-cshrc の例を示しておきます．

```
### Kinoko Settings ###

setenv KINOKO_ROOT /usr/local/kinoko
setenv KINOKO_XTERM /usr/X11R6/bin/kterm
setenv KINOKO_RSH /usr/bin/ssh
setenv KINOKO_CONTROL_PORT_BASE 40000
setenv KINOKO_DATASTREAM_PORT_BASE 42000
setenv KINOKO_SHELL_PORT_BASE 45000
setenv KCOM_PATH ".:${KINOKO_ROOT}/bin"
setenv PATH "${PATH}:${KINOKO_ROOT}/bin"
```

クイックツアー

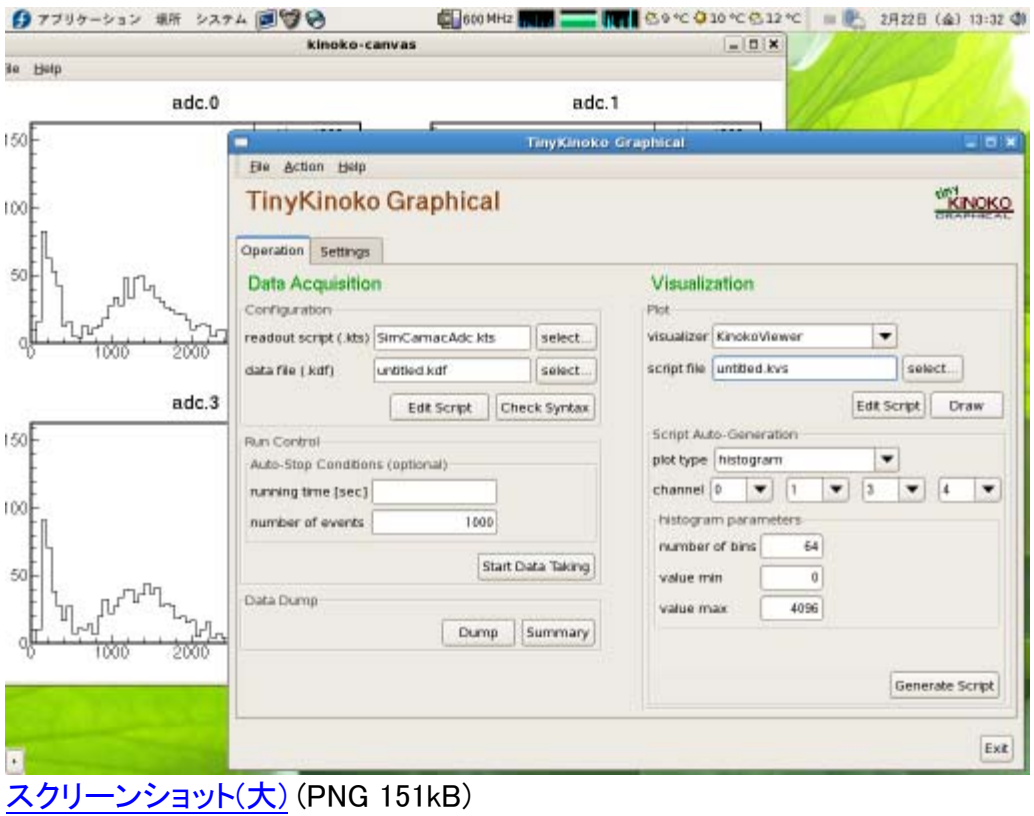
Home

とりあえず使ってみる

ディレクトリ kinoko/local/samples に，いくつかのサンプルスクリプトがあります．まず，CAMAC の ADC をソフトウェアでシミュレートする例を使って，Kinoko を一通り使ってみます．

TinyKinoko (Graphical 版)

TinyKinoko は，シングルプロセスで動作する軽量のデータ収集プログラムです．



スクリーンショット(大) (PNG 151kB)

使い方

1. ディレクトリ samples に移り，TinyKinoko を立ち上げてください．

```
% cd kinoko/local/samples
% tinykinoko-graphical
```

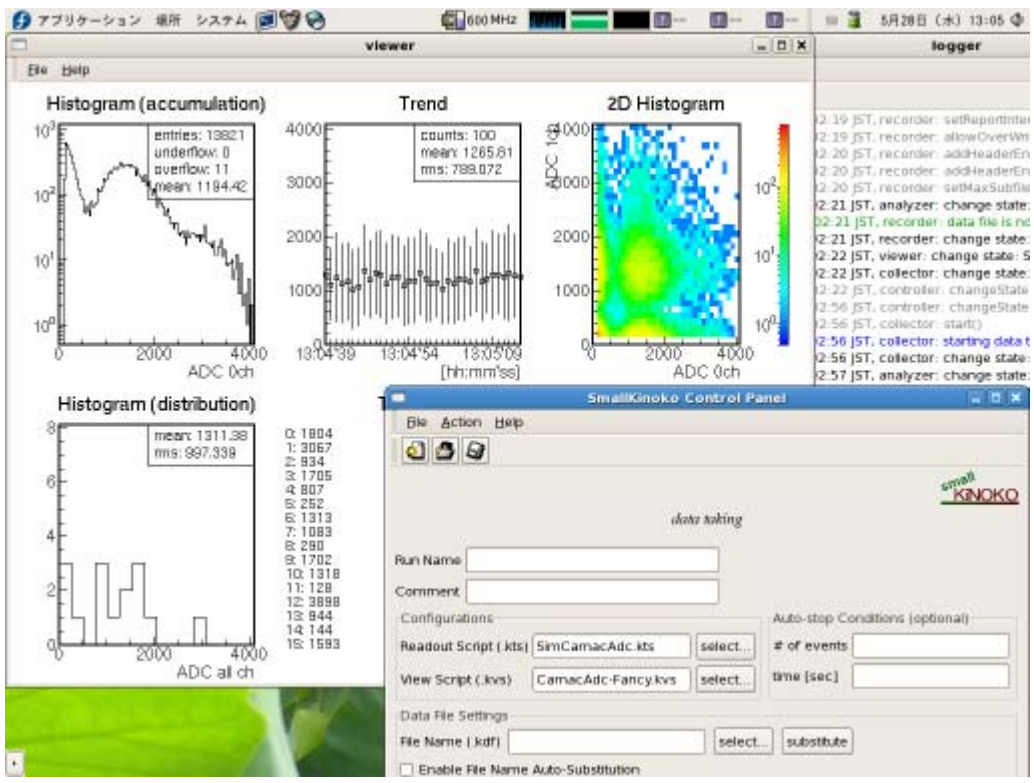
2. 表示されたウィンドウの Configuration および Run Control セクションの各フィールドに，以下の値を設定してください．

   Readout Script (.kts): **SimCamacAdc.kts**
   Data File (.kdf): **test01.kdf**
   number of events: **1000**

3. Run Control セクションの [Start Data Taking] ボタンをクリックしてください．データ収集を開始します．データ収集は，十数秒で終了します．
4. Data Dump セクションの [Dump] ボタンをクリックすると，エディタを起動し，得られたデータを表示します．また，[Summary] ボタンにより，イベント数や平均などの統計情報を表示します．
5. Plot セクションの Script File Name フィールドに test01.kvs と設定し，右下にある [Generate Script] ボタンを押してください．これにより描画スクリプトが自動生成されます．その後 [Draw] ボタンをクリックするとヒストグラムが描画されます．

SmallKinoko

SmallKinoko は，リアルタイムのデータ表示などができる，マルチプロセスのデータ収集システムです．ただし，単一の PC 上で動作し，ネットワーク上の分散並列処理などはできません．



スクリーンショット(大) (PNG 150kB)

使い方

1. ディレクトリ samples に移り，SmallKinoko を立ち上げてください．

```
% cd kinoko/local/samples
% smallkinoko
```

2. しばらくするといくつかのウィンドウが表示されます．この中の，タイトルバーにSmallKinoko Control Panel と表示されているウィンドウ(コントロールパネル)の入力フィールドに，以下の値を設定してください．

   Readout Script (.kts): **SimCamacAdc.kts**
   View Script (.kvs): **CamacAdc.kvs** または **CamacAdc-Fancy.kvs**
   File Name (.kdf): (空欄のまま)

3. 次に，コントロールパネル下部にある [Construct] ボタンをクリックしてください．
4. タイトルバーにlogger と書かれたウィンドウ(ロガー)には，各コンポーネントからのログメッセージが表示されます．各ログメッセージの先頭に E (エラー) または P (パニック)が無いことを確認してください．ログメッセージの表示が止まって，E や P

が無ければ，コントロールパネルの [Start] ボタンをクリックしてください．データ収集を開始します．もし，E または P がある場合は，コントロールパネルの [Quit] ボタンをクリックして，すぐに終了してください．

5. あきたら，コントロールパネルの [Stop] ボタンでデータ収集を終了し，[Quit] ボタンで終了してください．

準備

実際にデータを取り始める前に，作業ディレクトリを作成してください．どこでも良いのですが，kinoko ディレクトリの下に置きたいのならば，実験固有のファイルを置くための場所 kinoko/local があるので，ここを使ってください．この例では，このディレクトリの下に，trial01 というディレクトリを作成し，その下で作業することにします．

```
% cd kinoko/local
% mkdir trial01
% cd trial01
```

なお，クイックツアーで使用しているスクリプト類は全て kinoko/local/tutorials/QuickTour に置いてあるので，ここからコピーすることもできます．

TinyKinoko を使う

TinyKinoko は，シングルプロセスで動作するデータ収集プログラムです．リアルタイムのデータ表示などはできませんが，読み出しスクリプト(．ktsスクリプト)に基づくデータ読み出しなどはできます．この部分は他の kinoko と共通なので，特に初期の動作テスト等に役に立ちます．

ここでは，東陽テクニカの CAMAC コントローラ (CC/7x00) を使った CAMAC システムに，林栄精器の電荷積分型 ADC (RPC-022) を使い，チャンネル 0 からチャンネル 3 のデータを読み出してみます．なお，ADC はステーション 3 に入っており，ゲート入力後 LAM を発生するように設定されているとします．

以下のようなファイルを書き，Trial01.kts という名前で保存してください(行番号は後の説明のためのものです．これは入力しないでください)．

```
1: /* Trial01.kts */
2:
3: datasource Trial01
4: {
5:     int station_number = 3;
6:     long readout_channels = #0..#3;
7:
8:     CamacCrate crate;
9:     CamacController controller("Toyo-CC7x00");
10:     CamacModule adc("Rinei-RPC022");
11:
12:     crate.installController(controller);
13:     crate.installModule(adc, station_number);
14:
15:     on trigger(adc) {
16:         adc.read(readout_channels);
17:         adc.clear();
18:     }
19: }
```

若干の文法拡張がしてありますが，基本的な部分は C++ と同じなので，理解しやすいと思います．詳しくは，使用方法の章で説明します．

ユーティリティ ktscheck により，スクリプトの構文エラーをチェックできます．

```
% ktscheck Trial01.kts
```

UNIX の伝統に従い，エラーがない場合には何も出力されません．エラーがあれば，行番号とともにメッセージが表示されます(以下の例では，Rinei-RPC022 を Rinei-RPC002 としてしまっている)．

```
% ktscheck Trial01.kts
script exception: line 10: unknown camac module: Rinei-RPC002
%
```

エラーがないことを確認したら，このスクリプトを指定して，TinyKinoko でデータを読み出してみます．

```
% tinykinoko Trial01.kts trial01.kdf 100
```

ここで，第 2 引数はデータファイル名，第 3 引数は読み出すイベント数です．イベント数を省略したら，Ctrl-cを押すまで実行を続けます．また，--time オプションにより，時間を指定して実行することもできます．プログラムを実行する前に，ADC のゲートにちゃんと信号が入るようにしておいてください(適当な信号源がない場合は，10～100Hz 程度のクロックをゲートに入れるだけでも良いです)．

もし，すでに同名のデータファイルが存在している場合，tinykinoko はエラーを出して停止します．その場合は古いデータファイルを削除するか，強制上書きオプション -f を指定してください．

うまく動作したでしょうか．ちゃんと終了すれば，trial01.kdf というファイルができているはずです．script exception というエラーが出た場合は，スクリプトの文法エラーです．上の例とよく見比べて，修正してください．メッセージが出ずに固まってしまった場合は，以下の点を確認してください．

- ゲートにちゃんと信号が来ているか．ゲート信号幅は適切か．
- ADC は LAM を出すように設定されているか(RPC-022 はジャンパ設定が必要です)．
- CAMAC コントローラは LAM を受け取ったとき割り込みを発生させるようになっているか(CC/7000 ではジャンパ設定が必要です)．
- デバイスドライバはちゃんと動作しているか(ドライバのテストプログラムで確認してください)．
- モジュールが故障していないか(トラブルの原因の大半はこれです)．

データファイルを読む

得られたデータ(trial01.kdf)は様々な情報を含んだバイナリファイルです．これをテキストに直す一番簡単な方法は，以下のように kdfdump ユーティリティを使うことです．

```
% kdfdump trial01.kdf -
```

これにより以下のような出力が得られるはずです．

```
# Creator: tinykinoko
# DateTime: 2008-05-28 13:18:54 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/trial01
# ScriptFile: Trial01.kts

# EventTime: 1211948334
# EventTime: 1211948334

adc 0 1970
adc 1 1043
adc 2 1505
```

```
adc 3 339
# EventTime: 1211948334

adc 0 1216
adc 1 1411
adc 2 972
adc 3 183

adc 0 1521
adc 1 1510
adc 2 885
adc 3 1369
# EventTime: 1211948335

adc 0 173
adc 1 2022
adc 2 1016
adc 3 704
(以下省略)
```

から始まる行はコメントです．データは，左のカラムから，セクション名，アドレス，データです．この例の場合，セクション名はスクリプト中でモジュールにつけた名前(スクリプト 10 行目)，アドレスはADCのチャンネルです．イベントの区切りに空行が挿入されています．また，# EventTime には，時刻情報が UNIX の time(2) 形式で1秒ごとに記録されています．

ユーティリティ kdftable を使うと，もう少し解析しやすい形に整形できます．（ただし，kdftable で整形できるのは，各イベントのデータの構造が表形式で表現できる形で固定されている場合のみです．フラッシュADCやマルチヒットTDCなどのデータは，kdftableでは整形できません．）

```
% kdftable trial01.kdf -
# Creator: tinykinoko
# DateTime: 2008-05-28 13:18:54 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/trial01
# ScriptFile: Trial01.kts

# Fields: index time adc.0 adc.1 adc.2 adc.3
# StartTime: 1211948334

0 0 1970 1043 1505 339
1 0 1216 1411 972 183
2 1 1521 1510 885 1369
3 1 173 2022 1016 704
4 2 1632 1518 1075 1951
(以下省略)
```

ここで，最初のカラムはイベント番号，2番めは開始時刻からの経過秒数です．ヘッダの StartTime フィールドには開始時刻が UNIX の time(2) 形式で書かれています．

コメントや空行は解析プログラムの中で簡単に読み飛ばせると思いますが，もし邪魔なら，UNIX 標準のコマンド grep で取り除くことができます．

```
% kdftable trial01.kdf | grep "^[0-9]"
0 0 1970 1043 1505 339
1 0 1216 1411 972 183
2 1 1521 1510 885 1369
3 1 173 2022 1016 704
4 2 1632 1518 1075 1951
(以下省略)
```

プログラムの中から直接 KDF ファイルを読む方法については，使用方法の章で説明します．

TinyKinoko グラフィカルフロントエンド

TinyKinoko 用のグラフィカルフロントエンド tinykinoko-graphical を使うと，上記のデータ収集を GUI 上で行うことができます．さらに，tinykinoko-graphical は Kinoko の描画システムや GNUPLOT/ROOT と連動し，得られたデータのヒストグラムなどを表示することもできます．この際，これらの描画ツールでデータファイルを読み，解析するためのスクリプトを自動生成するので，これを雛型にして描画スクリプトや解析スクリプトを作成すると便利です(ハードコーディングが多いですが…)．

スクリーンショット (PNG 151kB)

データを収集するには，Configuration セクションのフィールドにファイル名などをセットし，Run Control セクションの [Start] ボタンを押してください．端末ウィンドウ(xterm など)が開き，その中で tinykinoko が実行されます．このウィンドウで Ctrl-c を押すことにより，データ収集を停止させることができます．

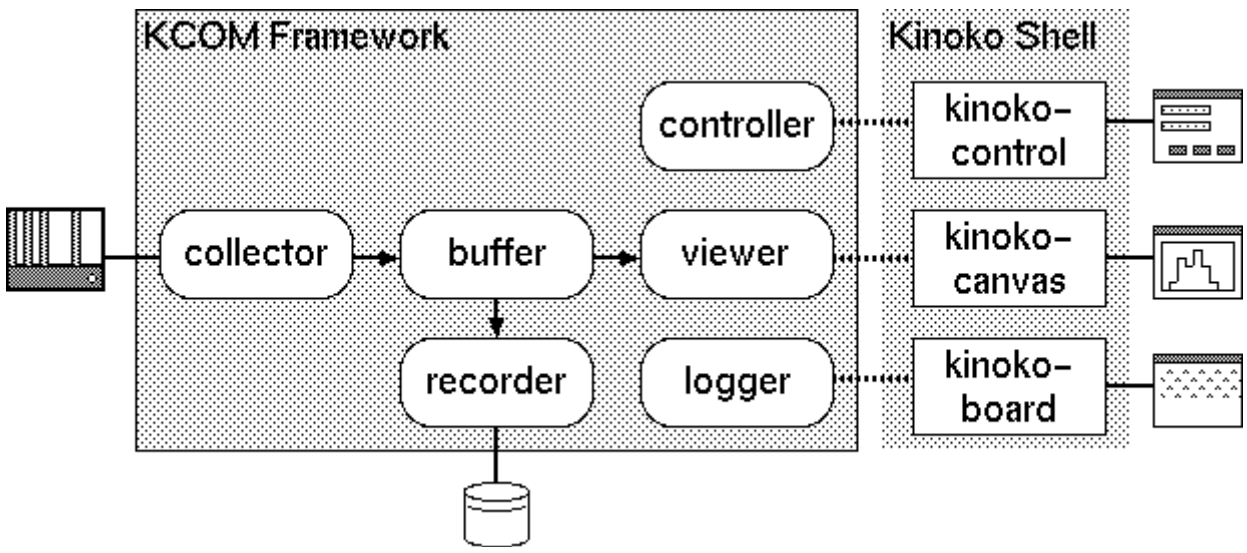
得られたデータは，Data Dump セクションの [Dump] ボタンで表示することができます．また，[Summary] ボタンで，チャンネルごとのイベント数や平均などの統計情報を表示できます．

Plot セクションのフィールドにパラメータを設定し，[Generate Script] ボタンを押すと，描画スクリプトを自動生成します．同じセクションの [Draw] ボタンにより，描画ツールを起動して，そのスクリプトを実行します．描画ツールの選択は，Plot セクションの Visualizer フィールドで指定します．

ROOT/GNUPLOT へのインターフェース部分には，内部で kdftable ユーティリティを使用しています．したがって，これらの機能が使えるのは，得られたデータが kdftable で扱える場合のみです．Kinoko の描画システムである KinokoViewer を選択した場合は，kdf ファイルが直接読み込まれるので，この制限はありません．

SmallKinoko を使う

SmallKinoko は，単一の PC 上で動作する(ネットワーク機能を持たない)マルチプロセスのデータ収集システムです．データを読み出す Collector，共有メモリを管理する Buffer，リアルタイムにデータを表示する Viewer，データを記録する Recorder，ログを記録する Logger，およびユーザコントロールを伝達する Controller の各コンポーネントから構成されます． このうち Controller，Logger，Viewer はユーザインターフェース(KinokoShell)との通信チャネルを持ち，それぞれ kinoko-control，kinoko-board，kinoko-canvas の各ウィンドウと接続されます．



使用する前に，Viewer コンポーネントが使用するスクリプト(ビュースクリプト)を作成します． 以下のようなファイルを書き，Trial01.kvs という名前で保存してください(拡張子は .kvs です．.kts などと混乱しないように注意してください)．また，Tinykinoko Graphical で KinokoViewer を指定して雛型を作成することもできます．

```
 1: /* Trial01.kvs */
 2:
 3: display Trial01
 4: {
 5:     Histogram histogram_adc_01("ADC ch 01", 128, 0, 4096);
 6:     Histogram histogram_adc_all("ADC all channels", 128, 0, 4096);
 7:     Tabular tabular_adc_all("ADC values");
 8:     Trend trend_event_rate("Event Rate", 0, 5000, 60);
 9:
10:     analysis {
11:         DataElement adc_01("adc", 1);
12:         DataElement adc_all("adc");
13:
```

```
14:         histogram_adc_01.fill(adc_01);
15:         histogram_adc_all.fillOne(adc_all);
16:         tabular_adc_all.fillOne(adc_all);
17:         trend_event_rate.fill(adc_01);
18:     }
19:
20:     on every (1sec) {
21:         histogram_adc_01.draw();
22:         histogram_adc_all.draw();
23:         tabular_adc_all.draw();
24:         trend_event_rate.drawCounts();
25:     }
26: }
```

このスクリプトも，だいたい想像したとおりに動きます．詳細については，使用方法の章で説明します．

ビュースクリプトに対するスクリプトチェックのユーティリティは kvscheck です．ktscheck と同様に，エラーがなければ何も表示されません．

```
% kvscheck Trial01.kvs
```

すでに tinykinoko などで取ったデータがあるならば，kinoko-viewer ツールを使用して描画内容をチェックすることができます．

```
% kinoko-viewer trial01.kdf Trial01.kvs | kinoko-canvas
```

スクリプトの準備ができたら，smallkinoko を起動します．

```
% smallkinoko
```

しばらくすると，いくつかのウィンドウが表示されます．ウィンドウのタイトルバーには，それぞれ，[SmallKinoko Control Panel]，[logger]，[viewer] とかかれているはずです．これらが，それぞれ Controller，Logger，Viewer の各コンポーネントと接続された KinokoShell のウィンドウです．

コントロールパネルウィンドウの各フィールドに，以下の値を設定してください．

```
Readout Script (.kts): Trial01.kts
View Script (.kvs): Trial01.kvs
Data File (.kdf): trial01.kdf
```

できたら，コントロールパネルの [Construct] ボタンを押してください．各コンポーネントがスクリプトの解読をし，データストリームを構築していきます．コンポーネントが出すログメッセージは，ロガーウィンドウに表示されます．以下は，ロガーに表示されるメッセージの例です．

```
N 05 Mar 2001 06:35:13 JST, collector: change state: ComponentReady
N 05 Mar 2001 06:35:14 JST, recorder: change state: ComponentReady
N 05 Mar 2001 06:35:14 JST, viewer: change state: ComponentReady
N 05 Mar 2001 06:35:14 JST, buffer: change state: Constructing
N 05 Mar 2001 06:35:15 JST, buffer: change state: DataTaking
D 05 Mar 2001 06:39:16 JST, collector: setReadoutScript(): trial01.kts
D 05 Mar 2001 06:39:16 JST, viewer: setViewScript(): trial01.kvs
D 05 Mar 2001 06:39:16 JST, recorder: setDataFile(): smallkinoko.kdf
D 05 Mar 2001 06:39:17 JST, collector: setSink: buffer (buffer) @127.0.0.1, shm: 100, msg: 101
```

```
D 05 Mar 2001 06:39:18 JST, recorder: setSource(): buffer (buffer) @127.0.0.1, shm: 100, msg: 101
D 05 Mar 2001 06:39:19 JST, viewer: setSource(): buffer (buffer) @127.0.0.1, shm: 100, msg: 101
N 05 Mar 2001 06:39:19 JST, collector: change state: Connecting
D 05 Mar 2001 06:39:19 JST, collector: connecting sink...
D 05 Mar 2001 06:39:20 JST, collector: connected
(以下省略)
```

各ログメッセージの最初の文字がログレベルを表しています. ログレベルの意味は以下のとおりです.

D: Debug	デバッグ情報	無視してください.
N: Notice	情報	見てるとおもしろいかも.
R: Remarkable	注目	超新星爆発など.
W: Warning	警告	実行に問題はないが, データに問題が生じるかもしれない.
E: Error	エラー	正常な実行が続けられない.
P: Panic	パニック	実行が続けられず, かつ終了処理もできない. 次回の起動前にクリーンアップが必要かもしれない.

もし, エラー (E または P) が表示された場合, [Quit] ボタンを押してシステムを終了させてください. ほとんどの場合, スクリプトのエラーが原因です.

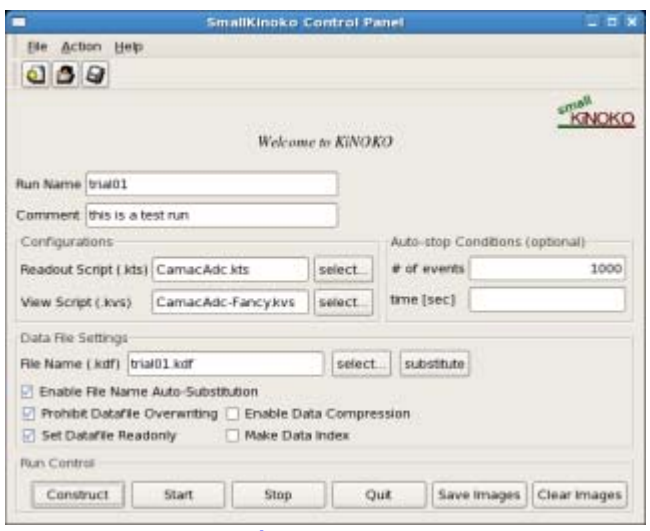
ここまで正常に来たら, [start] ボタンを押してデータ収集開始です. 一秒ごとにヒストグラムが更新され, 伸びていくのが見られます.

スクリーンショット (PNG 150kB)

あきたら, [Stop] ボタンを押して, 次に [Quit] を押し, 終了してください. 終了処理のため, [Quit] を押してからターミナルにプロンプトが戻るまで若干時間がかかります(数秒程度). 終了処理が完了する前に Ctrl-c などを押さないようにしてください(リソースが開放されない可能性があります).

SmallKinoko を活用する

SmallKinoko のコントロールパネルには, コメントを記入したりデータファイルに関するオプションを設定するなどの様々なフィールドがあります.



スクリーンショット(大) (PNG 48kB)

コントロールパネルに記述された RunName や Comment の内容は, [Construct] をした際に, そのままデータファイルのヘッダに記録されます. ヘッダの内容は, kdfdump などで表示させることができます.

[Auto-stop Conditions] にある #-of-events や time を設定すると, 指定した数のイベントを処理した時や指定した時間が経過した時に自動でランを停止させることができます. この機能を使用しない場合は, これらのフィールドを空欄のままにしておいてください.

データファイルを指定するフィールドのとなりにある [substitute] ボタンを押すと, データファイル名に含まれる特殊文字を置換できます. 何度もデータを取る場合, いちいちデータファイル名を変えなくて済むので便利です.

- "#数字" があると, 数字の部分をインクリメントします. # だけで数字がなければ, # が #1 になります.

```
run-#.kdf --> run-#1.kdf
run-#0100.kdf --> run-#0101.kdf
```

- "$d 数字" があると, 数字を削除して, $d を $d 日付 に置き換えます.

```
run-$d.kdf --> run-$d051207.kdf  (2005 年 12 月 7 日)
run-$d051206.kdf --> run-$d051207.kdf
```

- "$t 数字" があると, 数字を削除して, $t を $t 時間 に置き換えます.

```
run-$t.kdf --> run-$t092311.kdf  (9 時 23 分 11 秒)
run-$t092311.kdf --> run-$t092543.kdf
```

なお, いろいろな条件で取った kdf ファイルがたくさんある場合は, kdflist コマンドでそれらを一覧できます.

```
% kdflist run-*.kdf
run-#1.kdf   sanshiro   18528   2005-12-07 23:35:55 JST   physics run #1
run-#2.kdf   sanshiro    9788   2005-12-07 23:36:23 JST   pedestal run #1
run-#3.kdf   sanshiro   18712   2005-12-07 23:36:50 JST   physics run #2
run-#4.kdf   sanshiro    9880   2005-12-07 23:37:19 JST   pedestal run #2
```

[Data File Settings] にある Enable File Name Auto-Substitution を選択すると, このファイル名前置換を [Construct] のときに自動で行うようになります.

Prohibit Datafile Overwriting は, データファイルの上書きを禁止するオプションです. 通常, すでに存在するデータファイル名を指定すると, 警告のポップアップがでますが, このオプションが設定されていると, ポップアップで上書きを選択しても, エラーになって上書きができないようになります.

Set Datafile Readonly を選択すると, 取ったデータのデータファイルの属性を書き込み不可に設定します(444: -r--r--r-- にする). Kinoko はファイルの属性を一切変更しないので, ファイルはシステムレベルで保護されることになります.

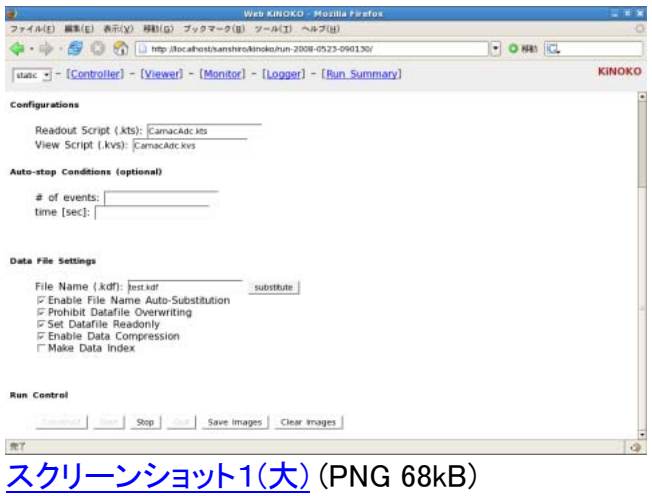
Enable Data Compression を選択すると, zlib を用いたファイル圧縮が行なわれます. ファイルはかなり小さくなりますが, ファイルのランダムアクセスなどの一部の機能が使えなくなる場合があります.

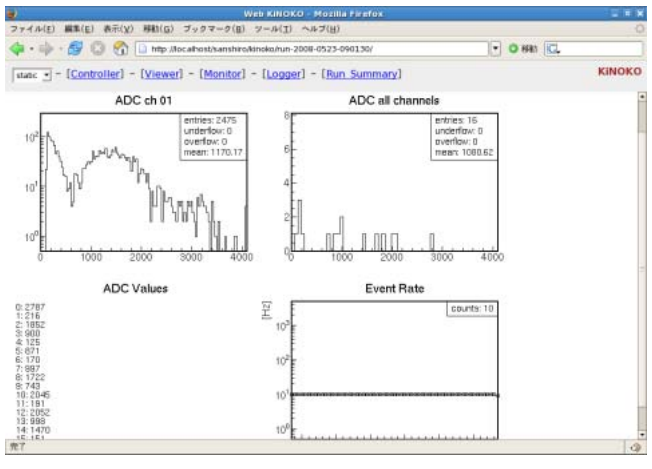
Make Data Index を選択すると, インデクスファイルが別に生成され, これを用いたランダムアクセスができるようになります. ただ, 通常は指定する必要はないと思います(インデクスは後から生成させることもできます).

これらのオプションは [Construct] ボタンをクリックする前に設定されている必要があります.

Web-KiNOKO を使う

Web-KiNOKO を使うと, 実行中の Kinoko をウェブブラウザからコントロールしたり, ビューアの絵やロガーのメッセージをリアルタイムに確認したりできるようになります. また, 生成されたウェブページを保存しておくことにより, ランサマリとして利用できます.





スクリーンショット1(大) (PNG 68kB)
スクリーンショット2(大) (PNG 41kB)

Web-KiNOKO は，比較的新しめの標準的なウェブブラウザで動作します．Firefox 1.5 および Safari 3.0 で動作確認を行いました．おそらく，Opera 9 でも動作すると思います．Internet Explorer では動作しません．また，ブラウザでは，JavaScript が利用可能に設定されている必要があります．

Web-KiNOKO では，それ自体としてはアクセス制限の機能を提供していません．必要に応じて，ウェブサーバの認証機能などを設定してください．

現時点では，Web-KiNOKO のセキュリティが十分でない可能性があります．ファイアーウォールの内側だけで使用するか，慎重なアクセス制限を設定した状態で使用してください．

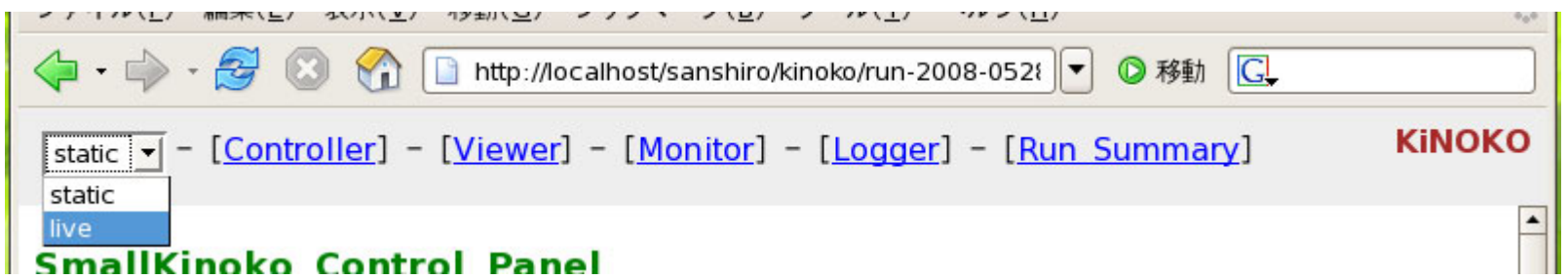
Web-KiNOKO を使用するには，ウェブサーバ（apache など）を動作させて，そこから参照できるディレクトリを用意する必要があります．このディレクトリでは CGI が実行できるようにしておいてください．kinoko/scripts に dump-http-request.cgi というサンプル CGI があるので，これをそのディレクトリに置くことにより，CGI が実行できるようになっているか確認できます．ここでは，/home/kinoko/public_html/webkinoko が Web-KiNOKO 用に設定されているディレクトリだとします．

上記の設定がされていれば，SmallKinoko のスタート時に --web-dir または --web-root オプションで Web-KiNOKO 用のディレクトリを指定するだけで Web-KiNOKO を使用できるようになります．SmallKinoko は --web-dir で指定されたディレクトリに必要なファイル類をコピーし，ウェブサーバ経由で参照できるようにします．--web-dir ではなく，--web-root でディレクトリを指定すると，SmallKinoko はそのディレクトリの下に run-日付 という名前のディレクトリを新規に作成し，そこを使用します．--web-root のほうが毎回違うディレクトリを指定する煩わしさがなく，また，過去のランの履歴を保存し一覧できるようになるので便利です．

```
% smallkinoko --web-root=/home/kinoko/public_html/webkinoko
```

環境変数 KINOKO_WEB_ROOT を指定しておくと，毎回 --web-root を指定しなくても Web-KiNOKO の設定が自動で行われます．ただし，この場合は SmallKinoko を実行するたびに全てのファイルが保存され続けてしまうので，ファイルの管理に注意が必要です．

--web-dir または --web-root を設定して SmallKinoko を立ち上げたら，そのディレクトリをウェブブラウザでアクセスしてみてください．--web-root で指定した場合は run-日付-時間 というディレクトリができているので，そこをクリックします．Web-KiNOKO のページの上部には，以下のようなメニューがあります．



一番左側のプルダウンメニューでは，動作モードを設定します．static モードでは，ページのロード時にのみ更新を行い，その後の自動更新は行いません．逆に live モードでは，コントロールパネルの値，ビューアの絵，ロガーのメッセージ，モニタの数値などが適宜更新されていきます．実行中の Kinoko に対しては live モードが，過去のランに対しては static モードが適しています．実行中の場合でも，自動更新が邪魔な場合は static モードにすることにより更新を止めることができます．Safari 3.0 では，今のところロガーの自動更新が動作しないので，ロガーだけは static モードを使用してください．

モード選択の右側には，ページ選択メニューがあり，それぞれコントロールパネルやビューアなどを選べます．モードを変更した場合は，もう一度ページ選択ボタンでページを選択しなおし，モード変更を反映させてください．

ネットワークの状態によっては，コントロールパネルからの操作に時間がかかることがあります．このときボタンを複数回クリックするとエラーメッセージなどが出てうっとおしいことになるので，コントロールパネルのボタンを操作したら，次の更新が終わるまで（通常1秒程度）はボタンなどを操作しないようにしてください．また，SmallKinoko 本体がポップアップウィンドウを表示してユーザの操作待ちになった場合，Web-KiNOKO の操作もできなくなってしまうので注意してください（Web-KiNOKO ではポップアップは表示されません）．

一番右側にある Run Summary は，ランの実行時間やコメント，使用したスクリプトなどをまとめて表示します．このページに関しては，実行中はあまり意味をもちません．実行後に過去のランの条件を参照するのに便利です．

オートパイロットを使う

オートパイロットを使うと，コントロールパネルに対するマウスやキーボードの操作を自動化できます．これにより，一定時間ごとのランのスタート・ストップや，条件を変えながらのランのくり返しなどが自動で行えるようになります．

以下は，[START] ボタンと [STOP] ボタンを操作して，1 時間ごとに 10 分間のデータを取る動作を 10 回繰り返す例です．以下の内容のファイルを auto-startstop.kcms という名前で作成してください．

```
int run_count = 0;

// 以下のブロックが 1 時間ごとに実行される
on every (1 hour)
{
    // invoke で，ボタンを押したのと同じ動作をする．ここでは，[START]
    invoke start();

    // after 文は，実行を指定時間遅れさせる．10 分後に [STOP]
    after (10 min) invoke stop();
}

// 以下のブロックが，[STOP] 後次の操作ができるようになったとき実行される
on transition (from "stopping" to "system_ready")
{
    // くり返しが 10 回目なら，[QUIT] で終了する
    run_count++;
    if (run_count == 10) {
        invoke quit();
    }
}
```

SmallKinoko を起動し，スクリプトやデータファイル名前を指定して，[Construct] をしてください．次に，コントロールパネルのメニューバーの [Action]-[LoadScript] でファイル選択ダイアログを開いて，auto-startstop.kcms を選択してください．これにより，オートパイロットスクリプトがロードされ，実行されます．

この例で作成されるデータファイルには，10 回分のランのデータが格納されます．これを分離するには，コマンド kdfseparate を使います．

```
% ls *.kdf
foo.kdf
% kdfseparate foo.kdf
making foo-1.kdf...
making foo-2.kdf...
(略)
making foo-10.kdf...
% ls *.kdf
foo.kdf    foo-1.kdf    foo-2.kdf    foo-3.kdf    foo-4.kdf
(略)
```

次はもう少し複雑な例です．ここでは，1 時間ごとに 10 分間の通常ランと，その後 1 分間のペデスタルランを行います．通常ランとペデスタルランでは異なったスクリプトを使うことにし，データファイルも別々にします．

```
int run_count = 0;
bool is_physics_run;

on every (1 hour)
{
    run_count++;
```

```
    // ポップアップでオートパイロットが中断しないようにデータファイルを削除しておく
    string data_file = "foo-physics-" + run_count + ".kdf";
    system("rm -f " + data_file);

    // スクリプト名やデータファイル名を設定
    // <> 演算子でコントロールパネルの入力フィールドの値にアクセスできる
    <readout_script>.setValue("SimCamacAdc.kts");
    <view_script>.setValue("CamacAdc.kvs");
    <data_file>.setValue(data_file);
    <message>.setValue("AUTOPILOT: taking physics data...");

    is_physics_run = 1;
    invoke construct();
}

// [Construct] 後，準備ができたら [START] する
// 同時に，[STOP] もスケジュールしておく
on transition (from "constructing" to "system_ready")
{
    invoke start();

    if (is_physics_run) {
        after (10 min) invoke stop();
    }
    else {
        after (1 min) invoke stop();
    }
}

// [STOP] 後，通常ランなら次のペデスタルランの設定をする
on transition (from "stopping" to "system_ready")
{
    if (! is_physics_run) {
        if (run_count == 10) {
            invoke quit();
        }
        else {
            <message>.setValue("AUTOPILOT: idling...");
        }
        return;
    }

    string data_file = "foo-pedestal-" + run_count + ".kdf";
    system("rm -f " + data_file);

    <readout_script>.setValue("SimCamacAdc.kts");
    <view_script>.setValue("CamacAdc.kvs");
    <data_file>.setValue(data_file);
    <message>.setValue("AUTOPILOT: taking pedestal data...");

    is_physics_run = 0;
    invoke construct();
}
```

オートパイロット実行中にポップアップなどが出て実行が中断しないように注意してください．あまり安全ではありませんが，この例では，新しくファイルを作成する前に，同名のファイルを削除しています．

この例には，[Construct] ボタンの操作も含まれているので，オートパイロットスクリプトのロードは，[Construct] を行う前(すなわち SmallKinoko を立ち上げた直後)に行います．後は，オートパイロットがシステムの終了まで自動で行います．

この例のように，起動直後からロードできるスクリプトは，smallkinoko のコマンド引数で起動時に指定することもできます．

```
% smallkinoko --autopilot = auto-periodic.kcms
```

コントロールパネルをカスタマイズする

ここでは，コントロールパネルをカスタマイズして，ADC のステーション番号と読み出すチャンネルをコントロールパネルから指定できるようにしてみます．以下は，ここで作成するコントロールパネルの外観です．ステーション番号やチャンネルの入力フィールド追加されています．

スクリーンショット(大) (PNG 30kB)

コントロールパネルの構成は，コントロールパネルスクリプト (KCML スクリプト) により記述されています．これは，今までとは異なり，XML を用いたスクリプトになっています．ここでは，SMallKinoko で使っているコントロールパネルスクリプトをベースにして，それを改造することにします(ただし，通常の SmallKinoko のコントロールパネルは多機能で複雑すぎるので，ここでは最小機能を実装している SmallKinoko Lite を使います)．SmallKinoko のスクリプト(とそこで使っている画像)を以下のように自分の作業ディレクトリにコピーしてください．

```
% cp $KINOKO_ROOT/scripts/SmallKinoko-Lite.kcml ./Trial02.kcml
% cp $KINOKO_ROOT/scripts/SmallKinoko.xpm .
```

Trial02.kcml を，以下のように編集してください．太字の行が追加する部分です．

```
 1: <?xml version="1.0"?>
 2:
 3: <KinokoControlPanel label="SmallKinoko Control Panel">
 4:   <HSpace/><Image file="SmallKinoko.xpm"/>
 5:   <VSpace/>
 6:
 7:   <HSpace/><Label name="message" label="Welcome to the Kinoko World." font="times-italic" size="14"/><HSpace/>
 8:   <VSpace/>
 9:
 0:   <EntryList>
11:     <Entry name="readout_script" label="ReadoutScript (.kts)" option="file_select"/>
12:     <Entry name="view_script" label="ViewScript (.kvs)" option="file_select"/>
13:     <Entry name="data_file" label="DataFile (.kdf)" option="file_select"/>
14:   </EntryList>
15:   <VSpace/>
16:
17:   <Frame label="ADC Configuration">
18:     <Entry name="station" label="Station" width="50" selection="1 2 3 4"/>
19:     <Label label="Ch: "/>
20:     <RadioButton name="enable_ch0" label="0"/>
21:     <RadioButton name="enable_ch1" label="1"/>
22:     <RadioButton name="enable_ch2" label="2"/>
23:     <RadioButton name="enable_ch3" label="3"/>
24:     <RadioButton name="enable_ch4" label="4"/>
25:     <RadioButton name="enable_ch5" label="5"/>
26:     <RadioButton name="enable_ch6" label="6"/>
27:     <RadioButton name="enable_ch7" label="7"/>
28:   </Frame>
29:   <NewLine/>
30:
```

```
31:   <Frame name="run_control" label="Run Control">
32:     <ButtonList>
33:       <Button name="construct" label="Construct" enabled_on="stream_ready system_ready"/>
34:       <Button name="start" label="Start" enabled_on="system_ready"/>
35:       <Button name="stop" label="Stop" enabled_on="data_taking"/>
36:       <Button name="clear" label="Clear" enabled_on="system_ready data_taking"/>
37:       <Button name="quit" label="Quit" enabled_on="stream_ready system_ready error"/>
38:     </ButtonList>
39:   </Frame>
40: </KinokoControlPanel>
```

このスクリプトが生成するコントロールパネルの外観は，コマンド kcmlcheck により確認することができます．

```
% kcmlcheck Trial02.kcml
```

次に，読み出しスクリプトの側を，コントロールパネルに設定された値を取得し，それを使うように変更します．Trial01.kts で，ステーション番号やチャンネルをハードコーディングしている部分(5 行目 と 6 行目)を削除し，以下の太字の部分を追加してください．ここで，getRegistry() が，コントロールパネルが設定したパラメータを取得する関数です．

```
1: /* Trial02.kts */
2:
3: datasource Trial02
4: {
5:     int station = getRegistry("control/station");
6:     int readout_channels;
7:     for (int ch = 0; ch < 16; ch++) {
8:         if (getRegistry("control/enable_ch" + ch) == "1") {
9:             readout_channels |= #ch;
10:        }
11:    }
12:
13:    CamacCrate crate;
14:    CamacController controller("Toyo-CC7x00");
15:    CamacModule adc("Rinei-RPC022");
16:
17:    crate.installController(controller);
18:    crate.installModule(adc, station_number);
19:
20:    on trigger(adc) {
21:        adc.read(readout_channels);
22:        adc.clear();
23:    }
24: }
```

作成した KCML スクリプトを引数にして，SmallKinoko を立ち上げてください．作成したコントロールパネルを持ったシステムが起動するはずです．

```
% smallkinoko Trial02.kcml
```

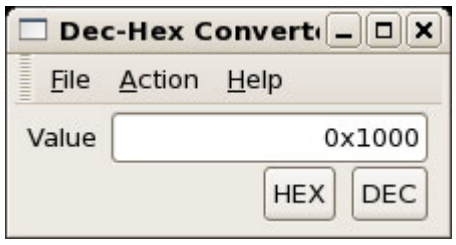
[Construct] する前に，ReadoutScript を Trial02.kts に変更するのを忘れないように注意してください． なお，この例のように，特定の読みだしスクリプトを使用することを想定する場合，KCML の <Constant>タグを使うことにより，KCML スクリプトの中にハードコーディングすることもできます．

```
10:   <EntryList>
11:     <Entry name="view_script" label="ViewScript (.kvs)" option="file_select"/>
12:     <Entry name="data_file" label="DataFile (.kdf)" option="file_select"/>
13:   </EntryList>
14:   <Constant name="readout_script" value="Trial02.kts"/>
```

コントロールパネルで独立アプリケーションを作る

コントロールパネルを記述する KCML スクリプトの中で <Script> タグを使うことにより，オートパイロットで使用した KCMS スクリプトの内容を直接 KCML の中に書くことができます．KCML スクリプトは kcmlcheck ツールで独立に動作させることができるので，これを利用することにより，Kinoko システムとは独立に動作するスタンドアローンのグラフィカルアプリケーションを作成することができます．実際，tinykinoko-graphical はこの仕組みを使って実装されています．

以下は，kcmlcheck を使って作成した例です．入力フィールドの値を10進数または16進数に変換します．



このプログラムは，1つの入力フィールドと2つのボタン，および改行や空白などのレイアウトウィジェットから構成されています．ソースコードは以下のようになっています．ボタンウィジェットの on_click 属性でボタンがクリックされたときに呼び出される関数を指定し，<Script> の中でその関数を定義しています．

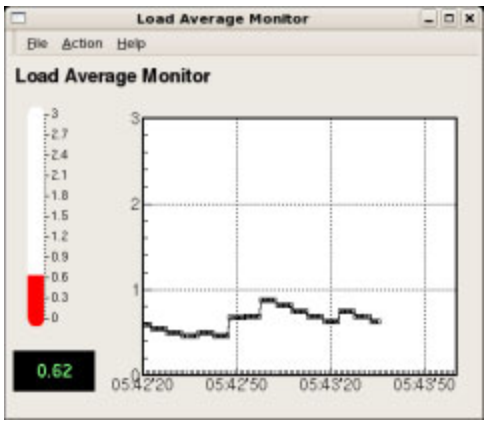
```
<?xml version="1.0"?>

<KinokoControlPanel label="Dec-Hex Converter">

 <Entry name="value" label="Value" alignment="right"/>
 <NewLine/>
 <HSpace/>
 <Button label="HEX" on_click="convertToHex"/>
 <Button label="DEC" on_click="convertToDec"/>

 <Script>
 <!--
  void convertToHex() {
       int value;
       try {
          value = (int) eval(<value>.getValue());
          <value>.setValue("0x" + hex(value));
       }
       catch {
          openPopup("ERROR", "OK", "invalid dec value");
       }
  }

  void convertToDec() {
       int value;
       try {
          value = (int) eval(<value>.getValue());
          <value>.setValue(value);
       }
       catch {
          openPopup("ERROR", "OK", "invalid hex value");
       }
  }
 -->
 </Script>
</KinokoControlPanel>
```

入力フィールドの値を getValue() で取得した後に eval() していますが，これは入力が計算式のときにその処理を行うためです．これにより，例えば 2 * 0x100 のような入力を処理できるようになっており，簡易電卓としても使用できます（上記プログラム中の int へのキャストを float にすれば関数電卓になります）．

このプログラムは kcmlcheck で実行できます.

```
% kcmlcheck DecHexConverter.kcml
```

次はもう少し複雑な例です. UNIX コマンドの uptime を呼び出してロードアベレージを取得し, Kinoko Version 2 より導入された VisualWidget と Viewlet を使ってロードアベレージの現在値と推移をグラフィカルに表示します.



スクリーンショット(大) (PNG 11kB)

ソースコードは以下のようになっています. uptime コマンドの出力形式によっては getLoadAverage() 関数の中身を修正する必要があるかもしれません. 文法の詳細については使用方法の章で説明します.

```
<?xml version="1.0"?>

<KinokoControlPanel label="Load Average Monitor">
 <Label label="Load Average Monitor" font="helvetica-bold" size="14"/><HSpace/>
 <VSpace/>

 <Box>
  <HSpace/>
  <VisualGuage name="guage" min="0" max="3" foreground="red" wideness="20"/>
  <HSpace/>
  <NewLine/>
  <VisualDisplay name="display" height="40" width="80" fg="lightgreen" bg="black" align="center"/>
 </Box>

 <Plot name="plot" width="370" height="300" xmin="0" xmax="100" ymin="0" ymax="3"/>

 <Script>
 <![CDATA[
on startup() {
      reset();
}

on every (1 sec) {
    update();
}

int start_time;
list x, y;

void reset() {
      start_time = time();
      x = {};    y = {};
      <plot>.setTimeTick(start_time, "%H:%M'%S", "sec", 30);
}
```

```
void update() {
        int time = time();
        double value = getLoadAverage();

        if (time - start_time > 100) {
            reset();
        }
        x <+>= time;
        y <+>= value;

        <guage>.setValue(value);
        <display>.setValue(value);
        <plot>.draw(x - start_time, y);
}

double getLoadAverage() {
        string uptime = shell("uptime");
        $match = (uptime =~ m/[Ll]oad [Aa]verage: ([0-9¥.]+)/);
        return $match[1];
    }
]]>
</Script>

</KinokoControlPanel>
```

異常終了後のクリーンアップ

もし, Kinoko が異常終了して, その結果システムにゴミ(共有メモリなどのシステム資源や生き残りプロセス)が残り, 次に起動しなくなった場合(通常, File already exists などのエラーメッセージが出ます), 以下の手順によりクリーンアップをおこなってください. ただし, この手順は, オペレーティングシステムによって微妙に異なります. ここでは Linux 2.x での手順を示します.

メッセージキューの削除

SmallKinoko は, 2 つのメッセージキューを使用します. 現在使用中のメッセージキューは, コマンド ipcs で確認できます.

```
% ipcs
(途中省略)
------ メッセージキュー --------
キー       msqid     所有者     権限      使用バイト数メッセージ
0x01039817 0         sanshiro  600       0          0
0x6503802e 1         sanshiro  600       0          0
```

kinoko 以外がメッセージキューを使用していることはあまりないのですが, 見分け方としては, 所有者が自分で, パーミッションが 600 となっていることです. 削除は, ipcrm コマンドに msqid を指定して行います.

```
% ipcrm msg 0
% ipcrm msg 1
```

この瞬間に, 多くの Kinoko プロセスがエラーメッセージを出して異常終了するはずです.

共有メモリの削除

SmallKinoko は，1 つの共有メモリを使用します．現在使用中の共有メモリは，コマンド ipcs で確認できます．

```
% ipcs
------ シェアードメモリセグメント --------
キー       shmid     所有者     権限      バイト     nattch   状態
0x5b037c7f 3         root      644       1048576    0        対象
0x00000000 260       root      644       98304      16       対象
0x00000000 1157      sanshiro  777       65536      2        対象
0x00000000 1158      sanshiro  777       65536      2        対象
0x00000000 1159      sanshiro  777       65536      2        対象
(途中省略)
0x00000000 71365     sanshiro  777       65536      2        対象
0x00000000 71366     sanshiro  777       65536      2        対象
0x6403802e 73159     sanshiro  600       33554432   4
(以下省略)
```

GNOME などのデスクトップ環境を使用していると，多くの共有メモリが使用されます．kinoko が使用している共有メモリの見分け方としては，所有者が自分で，パーミッションが 600 となっていて，サイズが 33554432 (または指定したサイズ)となっていることです．削除は，ipcrm コマンドに shmid を指定して行います．

```
% ipcrm shm 73159
```

生き残りプロセスの削除

まず，表示されている Kinoko 関係のウィンドウは全て閉じてください．つぎに，Kinoko 関連の生き残りプロセスを探します．

```
% ps -ef | grep inoko
sanshiro  1259     1  0 06:34 ttyp2    00:00:00 wish /home/sanshiro/work/kinoko/bin/kinoko-control-panel
kinoko.awa.tohoku.ac.jp 10001
sanshiro  1273     1  0 06:34 ttyp2    00:00:00 KinokoController-kcom controller port: 10002
sanshiro  1277     1  0 06:35 ttyp2    00:00:00 KinokoCollector-kcom collector file: smallkinoko.log
sanshiro  1303  1285  0 07:23 ttyp5     00:00:00 grep inoko
```

ここで，grep をするときに inoko で探していることに注意してください．これは，kinoko の最初の K が大文字と小文字の両方があるためです．kill コマンドにより，これらを削除します．

```
% kill 1259
% kill 1273
% kill 1277
```

Kinoko のユーティリティに，grep による検索と kill を一度に行う簡単なスクリプト killoff があります．これを使うと，上記の処理は以下のように行うことができます．

```
% killoff inoko
sanshiro  1259     1  0 06:34 ttyp2    00:00:00 wish /home/sanshiro/work/kinoko/bin/kinoko-control-panel
kinoko.awa.tohoku.ac.jp 10001
sanshiro  1273     1  0 06:34 ttyp2    00:00:00 KinokoController-kcom controller port: 10002
sanshiro  1277     1  0 06:35 ttyp2    00:00:00 KinokoCollector-kcom collector file: smallkinoko.log
commit? (y/n) > y
kill 1259
kill 1273
kill 1277
%
```

一応，キーワード kcom, korb でも grep してプロセスが生き残っていないか確かめてください．以下のプロセスが生き残っている可能性があります．

```
kcom-manager
kcom-registry-server
korb-broker
korb-nameserver
```

どうしてもうまくいかない場合

オペレーティングシステムを再起動してください．全てのゴミは自動で削除されます(再起動後，必要なドライバ類を組み込むのを忘れないように注意してください)．

使用方法

Home

KiNOKO の構造

[TinyKinoko, SmallKinoko のみを使う場合はこの章は読み飛ばして構いません]

KCOM フレームワークとコンポーネント

コンポーネント

Kinoko のシステムは，コンポーネントと呼ばれる単機能部品の組み合わせにより構築されます．コンポーネントには，ハードウェアをコントロールし，データを取得するもの(Collector)やデータを記憶装置に記録するもの(Recorder)，データをヒストグラムなどにして画面に表示するもの(Viewer)やユーザのコントロールをシステムに通知するもの(Controller) などがあります．また，ユーザが独自のコンポーネントを作成し，それをシステムに追加することもできます．多くのコンポーネントは，独自のスクリプト言語をもっていて，その動作を実行時にカスタマイズできるようになっています．

一つのコンポーネントは，一つの UNIX プロセスとして実装されていますが，他のコンポーネントと連携して動作できるようにするために，以下の 4 つの特別なインターフェースを備えています．

- プロパティ
- イベントとイベントスロット
- オブジェクトのエクスポートとインポート
- グローバルレジストリ

プロパティとは，他のコンポーネントから非同期に参照できるコンポーネント内部の値です．オブジェクトでいう内部変数に相当します．他のコンポーネントから値を読むことはできますが，変更することはできません．

イベント/イベントスロットは，コンポーネント間の同期的メッセージ交換メカニズムで，オブジェクトでいうメソッド呼び出しに相当します．すなわち，コンポーネントは，他のコンポーネントにイベントを投げることができ，イベントを受け取ったコンポーネントはそれに応じて処理を開始することができるということです．

オブジェクトのエクスポートとインポートの機能を使うことにより，コンポーネントは内部のオブジェクトを共有することができます．例えば，システムのログを記録するコンポーネントである Logger コンポーネントは，内部に通常のオブジェクトである Logger オブジェクトを持っていて，これをエクスポートしています．他のコンポーネントはこれをインポートし，別コンポーネントのオブジェクトであることを気にせずに，このロガーオブジェクトを使うことができるようになっています．

レジストリは，特定のコンポーネントに属さないパラメータを管理するためのグローバルなパラメータデータベースです．全てのコンポーネントから非同期に参照できるようになっています．

KCOM フレームワーク

これらのコンポーネント間の通信メカニズムは，全てネットワーク透過的に振舞います．すなわち，コンポーネントは複数の計算機上に分散配置でき，そしてそのことを全く意識せずに連携して動作できるということです．

ネットワーク透過的な同期・非同期コンポーネント間通信を実現するためには，コンポーネント間通信を媒介し，コンポーネントをコントロールするインフラストラクチャが必要です．これが KCOM フレームワークです．Kinoko のシステムが立ち上げられると，まずはじめに KCOM フレームワークのコアである KCOM マネージャプロセスが起動されます．続いて，マネージャプロセスはシステムで使用する全ての計算機上にサービスプロセス(broker)を走らせ，それらとの通信経路を確立します．broker は，自分の計算機上に共有メモリやメッセージキューなどのリソースを確保し，コンポーネントが通信できる環境を整えます．これらのインフラストラクチャの構築が終了すると，broker は，KCOM マネージャからの指示により，コンポーネントプロセスを起動し，準備した通信環境に接続させます．コンポーネント間の通信は，基本的に全てこれらの broker とマネージャにより仲介されます．

起動するコンポーネントの種類や数，配置する計算機などの情報はスクリプト(KcomScript)により記述されます．また，コンポーネント間共有オブジェクトの結合なども，このスクリプトに記述されます．コンポーネントのイベントやスロットは，コンポーネントどうしで直接接続することもできますが，これはコンポーネントの独立性や単機能性を損なうことになるので，通常は行ないません．そのかわり，コンポーネントイベントは全て KcomScript で取得し，コンポーネントのイベントスロットの呼び出しも KcomScript から行なうようにします．

以下は，KcomScript の例です．詳細については，KcomScript の章を参照してください．

```
1: // コンポーネント定義の取りこみ
2: import KinokoController;
3: import KinokoCollector;
4: import KinokoLogger;
5:
6: // コンポーネントの宣言と配置
7: component KinokoController controller("localhost", "port: 10000");
8: component KinokoCollector collector("daq01");
9: component KinokoLogger logger("localhost");
10:
11: // 共有オブジェクトの割り付け
12: assign logger.logger => controller.logger;
13: assign logger.logger => collector.logger;
14:
15: // イベント/スロットの結合
16: on controller.start()
17: {
18:     collector.start();
19: }
20:
(以下省略)
```

プライベート入出力チャネルと KinokoShell

コンポーネントは，KCOM フレームワークの通信チャネルとは別に，もう一つの入出力チャネルを持っています．これはプライベート入出力チャネルとよばれ，KcomScript の指定により，標準入出力や端末ウィンドウ，ネットワークポートなどに接続させることができます．上記の例では，7 行目で，controller コンポーネントのプライベート入出力チャンネルをネットワークポートの 10000 番に指定しています．

プライベート入出力チャネルは，通常，ユーザインターフェースへの接続に使われます．例えば，Controller コンポーネントはコントロールパネルへの接続にプライベート入出力チャネルを使用し，Logger はロガーウィンドウとの接続に，Viewer は描画ウィンドウとの接続に使用します．

Kinoko では，プラットフォーム独立性を保つために，そのコア部分(KinokoKernel)では，プラットフォーム依存性の強いグラフィカルユーザインターフェースを直接使用することを避けています．代わりに，プライベート入出力チャネルを使い，外部のユーザインターフェースプログラムに接続するという方法を取っています．このような，通信チャネルを介して KinokoKernel と接続し，ユーザインターフェースを担うプログラムを KinokoShell と呼びます．

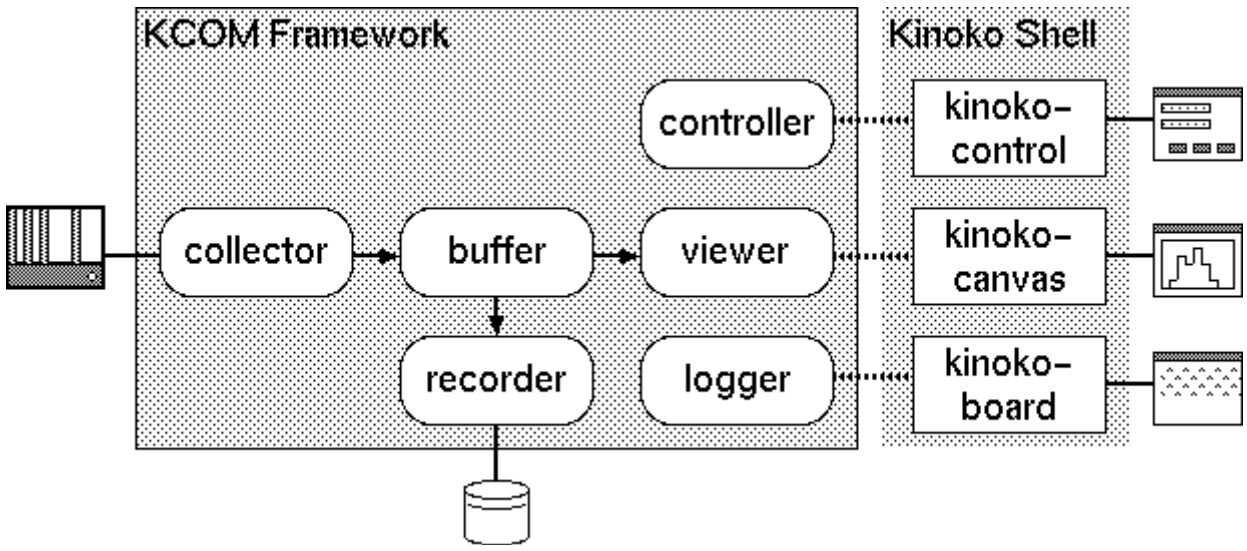
現在標準で用意されている KinokoShell には以下のものがあります．

kinoko-control
: KinokoController コンポーネントなどに接続するためのもので，入力フィールドやボタンなどを配置したコントロールパネルを表示し，ユーザの入力を伝えます．コントロールパネルのデザインは，XML を利用したスクリプト (KCML Script; Kinoko Control-panel Modeling Language) で記述されます．

kinoko-board
: KinokoLogger コンポーネントなどに接続するためのもので，コンポーネントからテキスト描画コマンドを受け取り，それをウィンドウ内の領域に描画します．

kinoko-canvas
: KinokoViewer コンポーネントなどに接続するためのもので，コンポーネントからグラフィクス描画コマンドを受け取り，それをウィンドウ内の領域に描画します．

以下は，SmallKinoko におけるコンポーネントと KinokoShell の接続の構造です．



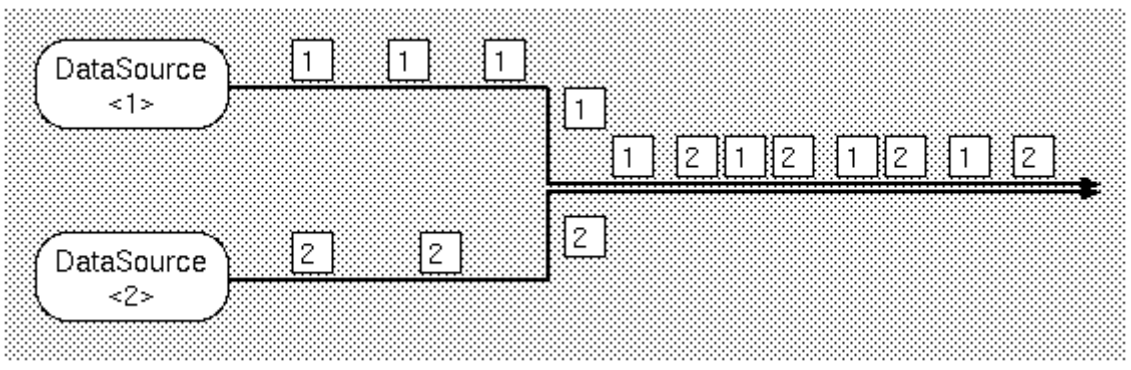
データストリーム

データストリームとデータソース

Kinoko のデータは，データパケットと呼ばれる固まりごとに転送されます．1 つのパケットは，典型的には，1 つのモジュールからの 1 回の読み出しで得られたデータになります．ただし，1 つのモジュールからの読み出しを複数のパケットに分割したり，複数のモジュールからの読み出しを 1 つのパケットにまとめたりもできます．さらに，あるパケットをもとに，計算処理を行なって，あたらしいパケットを生成したりなどもできるようになっています．

データパケットの転送系は，データストリームと呼ばれます．データストリームの開始点となる，データパケットを生成する場所を，Kinoko ではデータソースと呼びます．ハードウェアからのデータ読み出しを行なう Collector コンポーネントは，データソースの典型的な例です．Kinoko のシステムにおいて，全てのデータソースはユニークな名前を持ち，また，システムによりユニークな識別番号(DataSourceId)が割り振られます．

Kinoko では，データストリームは自由に結合させたり分岐させたりすることができます．ストリームが結合されると，結合後のストリームには複数のデータソースからのデータパケットが混在することになります．混在するデータパケットを識別するために，全てのデータパケットの先頭には，DataSourceId が記録されています．



データストリームコンポーネントタイプ

データストリームを構成するコンポーネントは，そのストリームへの接続の形態により，以下の種類に分類されます．

- ストリームソース (KinokoCollector, KinokoReproducer など)
- ストリームシンク (KinokoRecorder, KinokoViewer など)
- ストリームパイプ (KinokoTransporter, MyDataProcessor など)
- データバッファ (KinokoBuffer のみ)

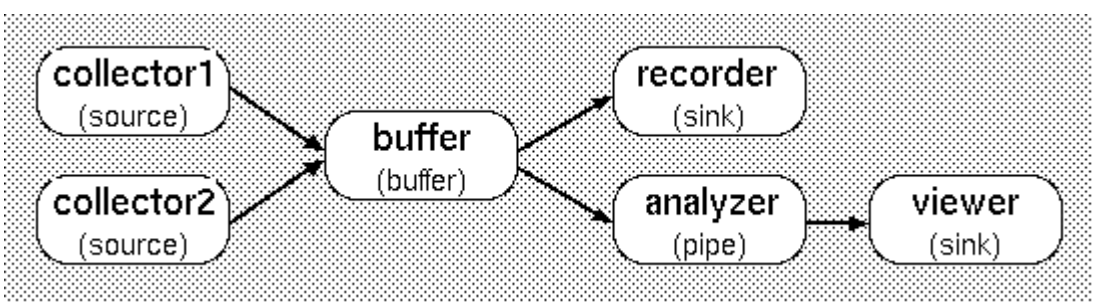
ストリームソースは，データソースをもち，データパケットを生成するコンポーネントです．ストリームシンクは，反対に，データパケットを受け取るだけで，下流側のストリームを持ちません．Recorder や Viewer などがストリームシンクの代表例です．これらはデータストリームに対してはパケットを受け取るだけで，パケットを送り出すことはしません．

ストリームパイプは，データストリームからデータパケットを受け取って，必要なら何らかの処理を行ない，下流のデータストリームにパケットを送り出すタイプのコンポーネントです．オンラインのデータ処理コンポーネントがこの例です．また，Kinoko には，なにもしないでそのまま送り出すコンポーネント KinokoTransporter があります．

データバッファは，ストリーム中では特殊なコンポーネントです．UNIX の共有メモリとメモリマネージャから構成され，その名前のとおり，ストリーム中でバッファとして機能するものです．共有メモリを使用しているため，多くの場面で効率良く動作します．バッファには以下のような用途があります．

- データフローレートの緩急を吸収する
- ストリーム下流の処理能力の変動を吸収する
- 複数のストリームからのデータを集める(ストリームを結合させる)
- 複数のストリームにデータを分配する(ストリームを分岐させる)

ストリームパイプは一つの入力ストリームと一つの出力ストリームしか持てないことに注意してください．ストリームを結合・分岐させるときは，常にバッファを使用します．以下は，2 つのデータソースからのデータストリームを，バッファを介して結合させ，さらにそこで 2 つのデータストリームに分岐させている例です．



バッファは，ストリームに対して受動的なコンポーネントで，入って来たデータパケット全てを，何の処理も加えずに全ての出力ストリームに分配します．また，共有メモリを使用しているため，ネットワーク越しのアクセスができません．ネットワーク越しにバッファに接続したい場合は，バッファ側の計算機に Transporter コンポーネントを入れるようにしてください(このあたりの仕様は将来改善する予定です)．

データストリームコンポーネントの配置と接続

データストリームコンポーネントの配置や接続も，通常のコンポーネントと同様に，KcomScript に記述して行ないます．ただし，KcomScript は汎用のコンポーネントフレームワークであるため，データストリーム構築に対する特別の機能はありません．ストリームの構築は，通常のコンポーネントのイベントスロットの呼び出しにより行ないます．

ストリームソースコンポーネントには，接続先を指定するためのスロット setSink() があります．同様に，ストリームシンクコンポーネントには，setSource() が，ストリームパイプには setSourceSink() があります．これらを接続する際には，両方向から接続先を指定する必要があります(ソース側もシンク側も setSink()/setSource() を行なう)．

バッファは受動的なコンポーネントなので，バッファ自体には接続先を指定するスロットはありません．ただし，接続要求を処理するために，はじめに start() を呼んでおく必要があります．

setSink()/setSource() で全てのコンポーネントの接続先を指定したら，connect() を呼んで接続を確立します．以下は，上記の配置例のストリーム接続を行なう KcomScript の例です(詳細については KcomScript の章で解説します)．

```
void connect()
{
    buffer.start();

    collector1.setSink(buffer);
    collector2.setSink(buffer);
    recorder.setSource(buffer);
    analyzer.setSourceSink(buffer, viewer);
    viewer.setSource(analyzer);

    collector1.connect();
    collector2.connect();
    recorder.connect();
    analyzer.connect();
    viewer.connect();
}
```

システムの終了時には，disconnect() を呼んでストリームの接続を解体します．

```
void disconnect()
{
    collector1.disconnect();
    collector2.disconnect();
    recorder.disconnect();
    analyzer.disconnect();
    viewer.disconnect();
}
```

コントロールパケット

データストリームを流れるものには，データパケットの他に，コントロールパケットと呼ばれる，特殊なパケットがあります．コントロールパケットは，主にランの開始や終了などを下流のコンポーネントに伝達するために使われるもので，以下の種類があります．

- RunBegin
- RunEnd
- RunSuspend
- RunResume

全てのデータソースは，ランの開始や終了の際にこれらのパケットをストリームに流します．下流のコンポーネントは，例えば RunEnd パケットを受け取ることにより，全てのデータの処理が終ったことを知ることができることになります．ストリームシンクでは，接続されているデータソースの数だけコマンドパケットを受け取ることに注意してください．例えば，2 つの Collector が接続された Recorder は，2 つの RunEnd パケットを受け取るまで処理を終了できないことになります．

ストリームデータフォーマット

データセクションとセクションタイプ

データストリームを流れるデータは，データセクションと呼ばれる単位に構造化されています． 特に指定せずにハードウェアからのデータ読み出しを行なう場合には，1 つのモジュールが 1 つのセクションに対応します．このとき，1 つのデータパケットには 1 つのモジュールからの 1 回の読み出し分のデータが記録されることになるため，1 つのデータパケットは 1 つのデータセクションのデータを1つを保持することになります．データセクションに対応するデータをセクションデータと呼びます．

全てのデータセクションは，データソース内でユニークなセクション名を持っています．また，システムによって，SectionId と呼ばれる識別番号が割り振られます．全てのセクションデータには，この SectionId が記録され，どのデータセクションのデータであるかが判別できるようになっています．SectionId は，データソースの中ではユニークな値になりますが，他のデータソースのデータセクションとは，同じ値が割り振られる可能性があります．DataSourceId と SectionId を両方使うことで，システム全体でユニークな識別番号となります．

データセクションは，そのセクションデータが保持するデータ形式に応じて，以下の種類があります．

indexed
: 可変長配列に対応するデータ構造です．セクションデータは可変個のエレメントデータから構成され，各エレメントデータは固定長のアドレス値とデータ値を持っています．CAMAC の ADC など，データが単純な整数値の集合からなる場合などに利用されます．以下は，indexed セクションデータの例です．

```
0 123    -- element data  -+
1 234    -- element data   |  -- section data
2 213    -- element data  -+

0 312
1 231
2 312
```

tagged
: 構造体に対応するデータ構造です．セクションデータは名前の付けられた固定数個のエレメントデータから構成され，各エレメントデータは固定長のデータ値を持ちます．indexed と似た構造のデータに対し，各エレメントデータの意味を強調したい場合に使用します．以下は，tagged セクションデータの例です．

```
pmt_0 123    -- element data  -+
pmt_1 234    -- element data   |  -- section data
pmt_2 213    -- element data  -+

pmt_0 312
pmt_1 231
pmt_2 312
```

block
: 構造のない(構造の分からない)データブロックを保持するデータ構造です．可変サイズのデータブロックを保持でき．データサイズのみが管理されます．内部にバッファを持ったモジュールからのデータなどで使われます．

nested
: 任意のセクションを任意数保持できるものです．nested のセクション自体を保持することもできます．ネストにより，複雑なデータ構造を表現できるようになっています．

データデスクリプタ

各データソースは，全てのデータパケットの送出に先立って，データデスクリプタと呼ばれる特殊なパケットを送り出します．データデスクリプタには，そのデータソースのデータソース名および DataSourceId，そのデータソースから送られるパケットのデータセクションの構造とセクション名および SectionId などが記録されています．データパケット自体には DataSourceId と SectionId しか記録されていないので，データパケット内のデータの解釈には，このデータデスクリプタの情報が必要になります．

また，データデスクリプタには，データソースアトリビュートと呼ばれる「名前・値」ペアを記録するフィールドもあります．これは，測定系のパラメータなど，データ自体には含まれない情報を記録するために使われます．

以下は，データデスクリプタの例です．詳細については，ここではあまり気にしないでください．

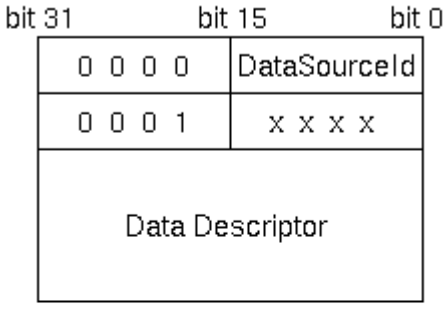
```
// CAMAC の ADC と TDC を一つづつ使った単純な例
datasource "CamacAdcTdc"<1025>
{
    attribute creator = "KinokoCollectorCom";
    attribute creation_date = "2003-04-13 17:40:23 JST";
    attribute script_file = "CamacAdcTdc.kts";
    attribute script_file_fingerprint = "edddd900";

    section "adc"<256>: indexed(address: int-8bit, data: int-24bit);
    section "tdc"<257>: indexed(address: int-8bit, data: int-24bit);
}
```

```
// いろいろと複雑なことをした例
datasource "MyComplicatedMeasurement"<5>
{
    attribute creator = "KinokoCollectorCom";
    attribute creation_date = "2003-04-13 17:47:43 JST";
    attribute script_file = "MyComplicatedMeasurement.kts";
    attribute script_file_fingerprint = "5f36e1a0";
    attribute setup_version="1.0";

    section "event"<6>: nested {
        section "event_info"<3>: tagged {
            field "time_stamp": int-32bit;
            field "trigger_count": int-32bit;
        }
        section "adc"<4>: indexed(address: int-8bit, data: int-24bit);
        section "tdc"<5>: indexed(address: int-8bit, data: int-24bit);
    }

    section "hv_monitor"<7>: indexed(address: int-16bit, data: int-16bit);
}
```

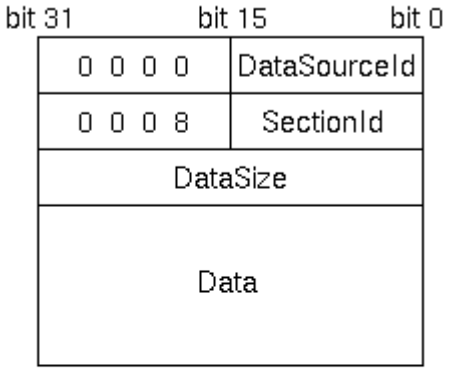
パケット構造

参考までに，各パケットの構造を以下に示します．ほとんどの場合，ユーザはパケットの物理的構造を意識する必要はありません．全てのパケットには DataSourceId が，データパケットにはそれに加えて SectionId が記録されているという点のみ留意しておいてください．

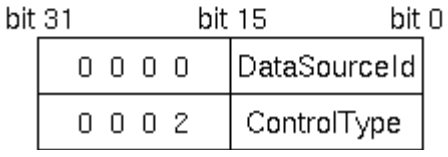
Data Descriptor Packet



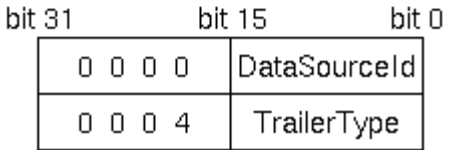
Data Packet



Cotrol Packet



Trailer Packet



バイトオーダは，パケットを生成した計算機のバイトオーダになります．最初の 32bit ワードに 16bit の DataSourceId が記録されているので，最初のワードの値と 0x0000ffff のビットアンドを取ることにより，バイトオーダを判別することができるようになっています．

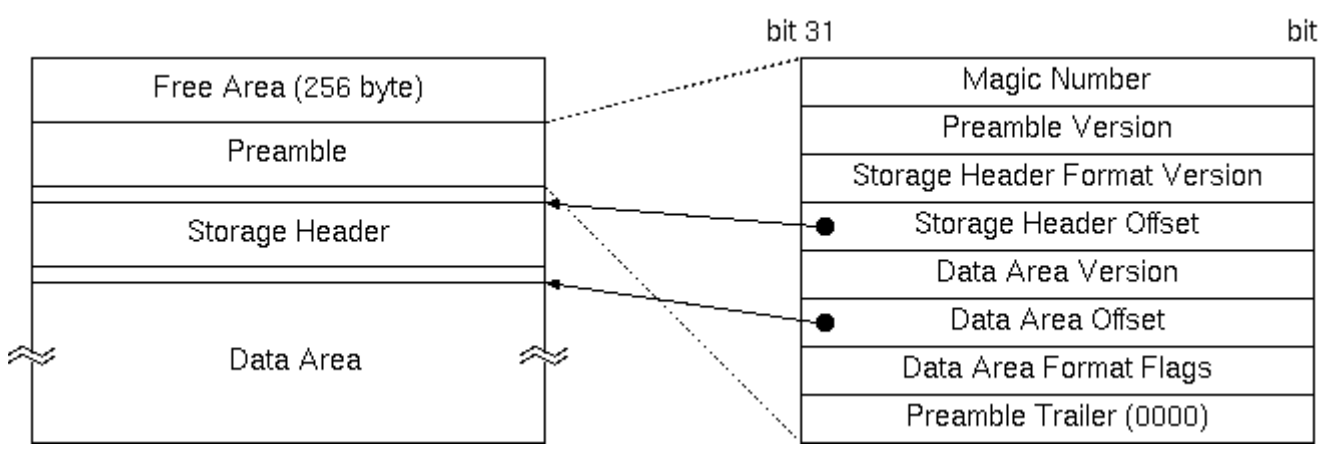
KDF ファイル

Kinoko では，簡易データストレージとして，KDF 形式と呼ばれるフォーマットのデータファイルを生成します．KDF には，全てのデータに加えて，データを取ったときの測定条件や使用したスクリプトファイル名など，必要な情報のほとんどが記録されています．また，バイトオーダの自動変換や，データ圧縮などの機能も持っています．オンラインデータパケットをほぼそのままの形でファイルに記録するため，比較的高速に動作します．

一方で，KDF は，オンライン用の簡易データストレージとして設計されているため，ファイル中のデータのランダムアクセスなどがあまり得意ではありません．このため，巨大ファイルを大量に扱うような実験において，オフラインアナリシスと共有するようなデータストレージに KDF を使用することは適切ではありません．そのような実験では，しばしばその実験用にデザインされたデータストレージシステムが別に準備されることが多いので，Kinoko ではユーザコンポーネントとしてそのストレージ用の Recorder を作成することが想定されています．

ファイル構造

参考として，KDF ファイルの構造を以下に示します．ただし，パケット構造と同様に，ユーザは KDF ファイル構造の詳細を意識しなければならないことはほとんどないはずです．



ファイル先頭の 256 byte は，自由に使える領域で，KDF を読むツールはこの部分を完全に無視します．通常は，KDF ファイルに関する簡単な説明がプレインテキストで書かれていて，cat などで KDF ファイルを見たときに，kdfdump などのツールを使うように促すメッセージが表示されるようになっています．

次のプリアンブル(Preamble)には，データファイル自体に関するパラメータが記録されています． 具体的には，ファイルのマジックナンバやデータフォーマットのバージョン，続くヘッダやデータブロックなどのファイルオフセットなどです．バージョン番号は 16bit のメジャーバージョンと 16 bit のマイナバージョンから構成され，下位互換性のある変更に関してはマイナバージョンだけが変更されることになっています．これにより，古いプログラムで新しいデータを読む場合でも，互換性が許す限りデータを正しく扱えるように

なっています(もちろん，非互換性が検出された場合にはエラーとなります)．さらに，データファイルを構成する各ブロック毎にバージョン番号を持たせることにより，一部が非互換の場合でも互換の部分は読むことができるようにしてあります．

ストレージヘッダ(Storage Header)には，外部から渡されたランパラメータなどが，テキスト形式で記録されています．以下は，ストレージヘッダの例です．

```
[Header]
{
        Creator="KinokoRecorderCom";
        DateTime="2003-04-14 10:47:24 JST";
        UserName="sanshiro";
        Host="kinoko.awa.tohoku.ac.jp";
        Directory="/home/sanshiro/work/kinoko/local/test";
        RunName="test001";
        Comment="this is a test";
}
```

データ領域(Data Area) は，データストリームのデータパケットが，ほぼそのままの形で記録されています．ただし，パケット自体にはパケットサイズが明示的に記録されていないことと，次のパケットのサイズをそのパケットを読む前に知りたいという情況が多いことから，各パケットデータの直前に，そのパケットのサイズがリトルエンディアンの 4byte 整数で記録されています．パケットの中身のバイトオーダは，異なったデータソースからのパケットが混在しているため，パケットごとに判断する必要があります．

KDF サブファイル

OS によるファイルサイズの上限の問題を避けるためと，コピーや管理などの便宜のため，Kinoko は KDF ファイルが大きくなると，それを複数のサブファイルに分割します．ファイルの分割はパケットの区切りで行なわれますが，その際単純に次のファイルを作成してそこへ記録を続けるだけなので，2 つめ以降のサブファイルにはプリアンブルやヘッダなどは含まれません．このため，通常のツールでサブファイルだけを読もうとすると，inconsistent magic number などのエラーが出て，正しく扱うことができません．

```
% ls -la
-r--r--r--   1 daq      kamland  268436820 Apr 16 10:01 run2459.kdf
-r--r--r--   1 daq      kamland  268436868 Apr 16 10:06 run2459.kdf.1
-r--r--r--   1 daq      kamland  268435480 Apr 16 10:11 run2459.kdf.2
-r--r--r--   1 daq      kamland  268438236 Apr 16 10:16 run2459.kdf.3
% kdfdump run2459.kdf.2
ERROR: TKinokoKdfStorage::ReadPreamble(): inconsistent magic number
%
```

KDF はオンライン用の簡易ストレージとして設計されているため，サブファイル自体には意味上の構造はなく，それを単体で読むことは想定されていません．ただし，デバッグやオンサイトアナリシスなどの目的のため，若干のサポートはあります．

kdfextract ユーティリティを使うと，先頭の KDF ファイルからファイルヘッダ，ストレージヘッダおよび RunBegin パケットを抽出できます．同時に，kdfextract は対応する RunEnd パケットを生成し，別のファイルに書き出します．これらを利用することにより，サブファイルをひとつの完結したファイルとして処理を行うことができるようになります．

```
% kdfextract run2459.kdf
making run2459-RunBegin.kdf...
making run2459-RunEnd.kdf...
% cat run2459-RunBegin.kdf run2459.kdf.2 run2459-RunEnd.kdf > run2459-2.kdf
% kdfdump run2459-2.kdf
```

この作業をする際にオリジナルのデータファイルを上書き削除しないように十分注意してください．全てのデータファイルにあらかじめ書き込み禁止属性を設定しておくことを勧めます．

KDF には raw アクセスモードがあり，これによりプリアンブルやストレージヘッダなどを無視して，直接データパケットを読むことができます．また，プリアンブルなどがないファイル(2 番目以降のサブファイル)でも，データパケットのみに直接アクセスしてその内容を読み出すことができます．

```
% kdfcheck --raw run2459.kdf.2
```

各パケットの識別にはその DataSourceId や SectionId の解釈が必要ですが，読み出しスクリプト等においてこれらを指定した値に固定することができます．これにより，データデスクリプタを読まなくても，パケットを識別し，データブロックを取得することが可能になります．

ただし，これらは全ていわば裏技で，Kinoko のエラーチェックやパラメータ管理などの機能を全て無視していることに注意してください．

スクリプトの基本

Kinoko では，システムの構成を記述したり，各コンポーネントの動作をカスタマイズしたりするために，スクリプトを使用します． 使われるスクリプト言語の文法は，その用途によりいろいろと拡張されていますが，基本となる部分は共通です．ここでは，この基本部分 (KinokoScript) について説明します．

KinokoScript の処理系として，kinoko-script コマンドがあります．スクリプトファイルを引数に kinoko-script を実行すると，そのスクリプトを実行します．

```
% cat hello.kino
// println() は引数の値を表示して改行する組み込み関数
println("hello, world.");

% kinoko-script hello.kino
hello, world.
%
```

kinoko-script を引数なしで実行すると，プロンプトを表示し，1 行づつ対話的に実行できます．

```
% kinoko-script
> println("hello, world.");
hello, world.
> 1+2*3;
7     // 実行の結果，値があるときはそれが表示される
> for (int i = 0; i < 3; i++) {
? println(i + ": hello");  // 文が完結していないときは，プロンプトが ? になる．
? }
0: hello
1: hello
2: hello
> .q            // Ctrl-d または .q で終了
%
```

スクリプトを実行した後にプロンプトで対話的に処理を続けたい場合は，--interactive または -i オプションを指定します．

```
% kinoko-script hello.kino -i
hello, world.
> sqrt(2);
1.41421
> .q
%
```

KinokoScript の文法は C/C++ をベースに構成してあり，基本的な部分は共通です．以下は，C/C++ との相異点を中心に説明します．

型と変数

値, 型とリテラル

整数 0 123 0x123abc
実数 3.14 1.23e10 1.23e+10 1.23e-10
虚数 1i 123i 3.14i 1.23e10i 1.23e+10i 1.23e-10i
文字列 "hello world" "123"
真偽値 true false
リスト { 1, 2, "hello" }

虚数は実数または整数の末尾に i を付けて表現します. 複素数のリテラルはありませんが, 実数と虚数を組み合わせて表現できます.

```
// + 演算子で複素数を作る
println(log(-1+1i));   // --> 0.346574+2.35619i

// 引数が実数なので実数演算を行おうとしてエラーになる
println(sqrt(-2.0));   // --> ERROR: invalid argument

// 虚数の 0 を付けると複素数で演算を行う
println(sqrt(-2+0i));   // --> 1.41421i
```

i 単体では変数となってしまい, 虚数の表現にはならないことに注意してください. 虚数単位のリテラルは 1i (いちアイ) です.

以下に文字列の使い方の例を示します.

```
// + 演算子による結合
println("hello" + " " + "world");   // --> hello world

// 数値は文字列に変換されてから結合される
println("pi = " + 3.1415);          // --> pi = 3.1415

// 数値表現の文字列は直接数値にキャストできる
println((int) "123" * 2);          // --> 246

// sizeof 演算子で長さを取得
println(sizeof("hello world"));     // --> 11

// [] 演算子で文字配列としてアクセス
println("hello world"[6]);          // --> w
```

以下はリストの使い方の例です.

```
// [] 演算子による要素アクセス
println({123, 456, "hello"}[1]);   // --> 456

// [] 演算子の中はリストでも良い
println({123, 456, "hello"}[{1,2}]);   // --> { 456, "hello" }

// リストどうしの演算は各要素ごとの演算になる
println({1, 2, "hello"} + {3, 4, " world"});   // --> {4, 6, "hello world"}

// リストとスカラの演算. これも要素ごとになる
println(2 * {0, 1, 2} + 1);   // --> {1, 3, 5}

// ほとんどの数学関数はリストに対して要素ごとに作用する
println(exp({0, 1, 2}));   // --> { 1, 2.71828, 7.38906 }
```

```
// リストの結合には専用の演算子を用いる
println({1, 2} <+> {3, 4});   // --> {1, 2, 3, 4}

// リストに対して集合演算
println({1, 2, 3, 4, 6, 12} <&> {1, 2, 4, 8, 16});   // --> { 1, 2, 4 }
```

リテラルではありませんが，規則的な整数や実数のリストは，リスト生成演算子 [] を使って簡単に生成することができます．

```
println([0, 5]);       // --> { 0, 1, 2, 3, 4 }
println([0, 10, 2]);   // --> { 0, 2, 4, 6, 8 }
println([5, 0]);       // --> { 5, 4, 3, 2, 1 }
println([0, 5] ** 2);  // --> { 0, 1, 4, 9, 16 }
```

型宣言名

int/long	整数: int と long は同じもので，処理系の long 型
float/double	実数: float と double は同じもので，処理系の double 型
complex	複素数
string	文字列
bool	真偽値
void	値が存在しないことを示す特殊な型
pointer	ポインタ
variant	バリアント: 任意の型の値を保持できる
list	リスト: 複数のバリアントをまとめたの

以下はバリアント変数の使い方の例です．

```
// バリアント変数に整数を保持させる
variant a = 123;
println(typeof(a));   // --> int

// a の値は整数なので，整数の加算が行われる
println(a + 456);   // --> 579

// 同じ変数に文字列を保持させる
a = "123";
println(typeof(a));   // --> string

// こんどは文字列に対する加算が行われる
println(a + 456);   // --> 123456

// リストも保持できる
a = { 123, "123" };
println(typeof(a));   // --> list
println(a + 456);     // --> { 579, "123456" }
```

バリアント変数は，$ をつけることにより宣言せずに使うことができます．この変数のスコープは常にグローバルです．

```
for ($i = 0; $i < 10; $i++) {
    $sum += $i;
}

// この $sum は {} の中の $sum と同じもの
println($sum);   // --> 45
```

リストは，通常の整数インデクスの他に，文字列キーを使ってアクセスすることもできます．

```
list range;
range{"min"} = 0;
range{"max"} = 10;
println(range);         // --> { "min" => 0, "max" => 10 }
println(keys(range));   // --> { "min", "max" }
println(range{"max"}); // --> 10
```

演算子と式

以下の演算子が C/C++ と同様に使用できます．

結合順位	結合規則	シンボル
2	後置単項	++ --
4	前置単項	+ - ! ~ * & ++ -- delete
6	左から右	* / %
8	左から右	+ -
10	左から右	<< >>
12	左から右	< <= > >=
14	左から右	== !=
16	左から右	&
18	左から右	^
20	左から右	|
22	左から右	&&
24	左から右	||
26	右から左	= += -= *= /= %=

以下の演算子は C/C++ にもありますが，動作が若干異なります．

結合順位	結合規則	シンボル	備考
0	特殊	new	配列の生成時でもコンストラクタ引数を渡すことができる
4	前置単項	sizeof	配列・リスト・文字列に対して長さを返す

以下の演算子が独自に追加されています．

結合順位	結合規則	シンボル	動作
0	特殊	**	冪乗を計算
2	後置単項	!	階乗を計算
4	前置単項	keys	連想リストのキーのリストを返す
4	前置単項	typeof	値の型名を返す
4	前置単項	#	指定されたビットが立った整数を生成
8	左から右	<+>	リストまたはスカラ値を結合して新しいリストを生成する
15	左から右	..	指定された範囲のビットが立った整数を生成
16	左から右	<&>	リストの要素に対して集合演算を行う
26	右から左	<+>=	リストにリストまたはスカラ値を追加する

冪乗演算子 ** が通常の 2 項演算子ではなく特殊演算子になっているのは，-3**2 が -9 となるようにするためです．

文と制御構造

以下の文が C/C++ と同様に使用できます.

名称	構文
複文	{ 文リスト }
空文	;
式文	式;
if-文	if (条件式) 文
while-文	while (条件式) 文
for-文	for (初期化文; 条件式; 式) 文
continue-文	continue;
break-文	break;
return-文	return 式^{opt};

以下の文は C/C++ と似ていますが, 若干異なった振舞いをします.

名称	構文	備考
throw-文	throw 式;	引数の式は省略できない
例外処理文	try 文 catch(引数宣言) 文	catch 節は 一つの try に対し一つだけ. catch 節の引数に例外オブジェクトを代入できないとエラーになる
例外処理文	try 文 catch 文	catch 節引数は省略可

以下の文は C/C++ にはないものですが, Perl などの多くのスクリプト言語にあるものと同様のものです.

名称	構文	動作
foreach-文	foreach (パラメータ宣言; リスト式) 文	パラメータにリストの各要素値を順に代入しながら文の実行を繰り返す

関数定義その他エントリ

エントリとは, スクリプトの最外構造で, 実行の開始点などを定義します. 通常, 文はエントリの内部に記述されます. C 言語型の関数はエントリの代表的な例です.

直接実行開始点としては使われないものの, エントリと同列の位置に記述される文法要素もあります. 例えば, 大域変数の宣言などがその例です. これらは, KinokoScript では, 擬似エントリと呼ばれます. ほとんどの擬似エントリは, あるエントリが実行される前に自動で実行されます.

関数定義エントリ

構文: *戻り値型 関数名* (*引数リスト*) { *文* }

関数を定義する. 基本的に C と同じ構文だが, 引数が無い場合は引数リストは空にしておく. キーワード void は引数リストには使用できない.

include 疑似エントリ

構文 include *"ファイル名"*;

指定されたファイルをその位置に取りこむ.

文 疑似エントリ

どの実行エントリにも属さない文を定義する. これらの文は, エントリが実行される直前に, 定義された順で自動的に実行される.

組み込みクラスと組み込み関数

入出力関係

組込み関数

void print(string line)
　　引数に与えられた文字列を標準出力に書き出す.
void printLine(string line) / void println(string line)
　　引数に与えられた文字列を標準出力に書き出し, 改行する.
void putByte(int byte) / void putc(int byte)
　　引数に与えられた値を標準出力に1バイトとして書き出す.
string getLine() / string getln()
　　標準入力から一行読み込み, その文字列を返す. 返される文字列は改行文字を含む. EOF の場合, 空文字列を返す.
int getByte() / string getc()
　　標準入力から1バイトを読み, その値を返す. EOF の場合, -1 を返す.
string readLine() / string readln()
　　getLine() と同じ機能だが, GNU/readline ライブラリがリンクされている場合, 入力に行編集機能が利用できる.

InputFile クラス

```
InputFile file("source.cc");
string line;
int line_count = 1;

while (line = file.getLine()) {
    println(line_count + ": " + line);
    line_count++;
}
```

InputFile(string file_name)
　　コンストラクタ. ファイルを入力モードでオープンする.
string getLine() / string getln()
　　ファイルから一行読み込み, その文字列を返す. 返される文字列は改行文字を含む. EOF の場合, 空文字列を返す.
string getByte() / string getc()
　　ファイルから1バイトを読み, その値を返す. EOF の場合, -1 を返す.

OutputFile クラス

```
OutputFile file("sine_table.dat");
double pi = acos(-1);
for (double x = 0; x < 2*pi; x += pi/100) {
    file.println(x + " " + sin(x));
}
```

OutputFile(string file_name)
　　コンストラクタ. ファイルを出力モードでオープンする.
void print(string line)
　　引数に与えられた文字列をファイルに書き出す.
void printLine(string line) / void println(string line)
　　引数に与えられた文字列をファイルに書き出し, 改行する.
void putByte(int byte) / void putc(int byte)
　　引数に与えられた値を標準出力に1バイトとして書き出す.

InputPipe クラス

```
InputPipe pipe("zcat mydata.dat.gz");
while ($line = pipe.getLine()) {
    //...
}
```

- コンストラクタ引数のコマンドを実行し，その出力を取り込みます．
- InputFile クラスの全てのメソッドが同じ型式で利用できます．

OutputPipe クラス

- コンストラクタ引数のコマンドを実行し，このオブジェクトの print()/println() 出力をコマンドへの入力に結びつけます．
- OutputFile クラスの全てのメソッドが同じ型式で利用できます．

書式化

組込み関数

string dec(int value, int width)
　　引数に与えられた整数値を10進表記の文字列に変換する
string hex(int value, int width)
　　引数に与えられた整数値を16進表記の文字列に変換する
string bin(int value, int width)
　　引数に与えられた整数値を2進表記の文字列に変換する
string fixed(float value, int precision)
　　引数に与えられた実数値を固定小数点表記の文字列に変換する
string scientific(float value)
　　引数に与えられた実数値を科学表記の文字列に変換する

Formatter クラス

Formatter クラスは数値などから書式化した文字列を生成するためのもので， C++ の ostream と同様の機能を同様の形式で提供します(C++ の挿入演算子 << は put() メソッドになります)．

```
// Formatter で出力を整形する
Formatter formatter;
string line = formatter.put("pi=").setPrecision(10).put(3.14159265).flush();
println(line);   //--> pi=3.14159265

// 上記全てを 1 行に書くこともできる
println(Formatter().put("pi=").setPrecision(10).put(3.14159265).flush());
```

Formatter& put(int value)
Formatter& put(float value)
Formatter& put(complex value)
Formatter& put(string value)
　　引数の値を内部バッファに挿入する．
string flush()
　　内部バッファの内容を文字列にして返す．
Formatter& setWidth(int width)
　　次に挿入される値の表示幅の最小値を指定する．
Formatter& setPrecision(int precision)
　　浮動小数点数を表示する際の表示精度を指定する．
Formatter& setFill(string character)
　　表示幅を埋めるための文字を指定する．
Formatter& setBase(int base)
　　整数を表示する際の基数を指定する．

Formatter& hex()
　　整数を表示する際の基数を 16 進数に設定する.
Formatter& dec()
　　整数を表示する際の基数を 10 進数に設定する.
Formatter& fixed()
　　浮動小数点数を固定小数点型式で表示するように設定する. setPrecision() で設定する精度は小数点以下の桁数の最大値となる.
Formatter& scientific()
　　浮動小数点数を科学型式で表示するように設定する. setPrecision() で設定する精度は桁数の最大値となる.

Scanner クラス

Scanner クラスは C++ の istream に似た機能を提供します(C++ の抽出演算子 >> は get() メソッドになります).

```
// Scanner で入力行を数値に分解する
string line;
double x, y, z;
while (line = getln()) {
    Scanner scanner(line);
    if (scanner.get(x).get(y).get(z).good()) {
        //...
    }
}

// ちょっとだけ簡単な書き方
string line;
double x, y, z;
while (line = getln()) {
   if (Scanner(line).get(x).get(y).get(z).good()) {
        //...
    }
}
```

Scanner(string value)
　　コンストラクタ. 引数に渡された文字列を内部バッファに取りこむ.
Scanner& load(string value)
　　引数に渡された文字列を内部バッファに取りこむ.
Scanner& get(int& value)
Scanner& get(float& value)
Scanner& get(complex& value)
Scanner& get(string& value)
Scanner& get(variant& value)
　　内部バッファをスキャンして, 対応する文字列から値を取りだす.
Scanner& getLine(string& line, string terminator = ″¥n″)
　　terminator 引数に指定された終端文字までを内部バッファから読み出し, line に返す. 終端文字はバッファからは取り除かれるが, line に返される文字列には含まれない.
Scanner& skipWhiteSpaca()
　　内部バッファの先頭から連続する空白文字を取り除く.
Scanner& setBase(int base) / Scanner& setbase(int base)
　　整数を読み出す際の基数を指定する.
int isGood() / int good()
　　前回の get() に成功した場合は 1 を, 失敗した場合は 0 を返す.
int lastGetCount() / int gcount()
　　前回の get() で読みこんだ文字数を返す.

数学関数

数学関数

–	float sin(float x)	complex sin(complex x)	list sin(list x)
–	float cos(float x)	complex cos(complex x)	list cos(list x)
–	float tan(float x)	complex tan(complex x)	list tan(list x)
–	float asin(float x)	–	list asin(list x)
–	float acos(float x)	–	list acos(list x)
–	float atan(float x)	–	list atan(list x)
–	float atan2(float y, float x)	–	list atan2(list y, list x)
–	float exp(float x)	complex exp(complex x)	list exp(list x)
–	float log(float x)	complex log(complex x)	list log(list x)
–	float log10(float x)	complex log10(complex x)	list log10(list x)
–	float sqrt(float x)	complex sqrt(complex x)	list sqrt(list x)
int abs(int x)	float abs(float x)	float abs(complex x)	list abs(list x)
–	float arg(float x)	float arg(complex x)	list arg(list x)
int real(int x)	float real(float x)	float real(complex x)	list real(list x)
int imag(int x)	float imag(float x)	float imag(complex x)	list imag(list x)

変換関数

float trunc(float x) / list trunc(list x)
float round(float x) / list round(list x)
float ceil(float x) / list ceil(list x)
float floor(float x) / list floor(list x)

乱数生成

float srand(int seed)
float rand()

リスト処理関数

統計演算

int length(list x)
float min(list x)
float max(list x)
float sum(list x)
float mean(list x)
float deviation(list x) / float rms(list x)

リスト演算

list delta(list x)
　　リストの差分をとる
list sigma(list x)
　　リストの和分をとる
list divide(list x, int n)
　　リストを指定長さごとに分割する

リスト処理

list zeros(int n)
　　値が全て 0 の長さ n のリストを生成する
list ones(int n)
　　値が全て 1 の長さ n のリストを生成する
int count(list x)
　　引数のリストから bool にキャストして true になる要素数を数える
list find(list x)
　　引数のリストから bool にキャストして true になる要素のインデクスのリストを返す

文字列処理関数

これらの文字列処理関数は，基本的に Perl の同名の関数と同じ動作をします．

string substr(string str, int offset, int length)
string substr(string str, int offset)
: str の部分文字列(offset から length 文字)を取りだす．length が省略された場合文字列の末尾までが取りだされる．length が負の場合は末尾から −length 文字を残すように長さが調節される．

int index(string str, string substr, int offset=0)
: str の中の offset 以降で substr が最初に出現する位置を返す．substr が存在しない場合は −1 を返す．

string chop(string str);
: str の末尾の 1 文字を str から削除し，削除した文字を返す．

int chomp(string str);
: str の末尾にある改行文字を str から削除し，削除した文字数を返す．

その他 組込み関数

シェル・プロセス制御・環境変数

int system(string command)
: シェルを起動し，引数のコマンドをシェルから実行する．コマンドの実行が終了するまで system() を呼び出したスクリプトは実行を停止する．

string shell(string command)
: シェルを起動し，引数のコマンドをシェルから実行し，コマンドの標準出力を文字列にして返す．コマンドの実行が終了するまで shell() を呼び出したスクリプトは実行を停止する．

int execute(string path, string ...)
: 第一引数に指定されたプログラムを子プロセスとして実行する．関数は任意の数の引数をとり，それらは子プロセスの起動引数になる．子プロセスの起動に成功した場合はそのプロセス ID を，失敗した場合は −1 を返す．

int wait() / int wait(int process_id)
: 引数に指定したプロセス ID を持つ子プロセスが終了するまで実行を停止する．子プロセスがすでに終了している場合は，関数は即座にリターンする．引数に指定できる子プロセスは execute() 関数で実行した子プロセスのみ．引数を省略した場合，いずれかの子プロセスが終了するまで実行を停止する．すでに終了し，ゾンビになっている子プロセスが存在する場合，関数は即座にリターンする．

int kill(int process_id, int signal_id = SIGTERM)
: 引数に指定したプロセス ID をもつプロセスに引数に指定したシグナルを送る．

void sleep(int time_sec) / void sleep(int time_sec, int time_usec)
: 引数に渡された時間が経過するまでスクリプトの実行を停止する．

string getEnvironmentVariable(string name)
: 引数に渡された名前の環境変数を取得する．環境変数が定義されていない場合は，空文字列を返す．

void setEnvironmentVariable(string name, string value)
: 引数に指定された名前の環境変数に引数に指定された値を設定する．

時刻

int time()
: 現在時刻を UNIX 時間で返す．

int timeOf(int year, int month, int day, int hour, int minute, int second)
: 引数に指定された時刻を UNIX 時間で返す．

string localTime() / string localTime(string time_format)
: 現在時刻を time_format に従って整形し，その文字列を返す．time_format が指定されていない場合，そのシステムのデフォルトのフォーマットで整形する．

string localTimeOf(int time_value) / string localTimeOf(int time_value, string time_format)
: 引数に指定された時刻を time_format に従って整形し，その文字列を返す．time_format が指定されていない場合，そのシステムのデフォルトのフォーマットで整形する．時刻の表現は UNIX 時間．

プログラム引数

int numberOfArguments()
: プログラムに渡された引数の数を返す．第 1 引数はスクリプト名．

string getArgumentOf(int index)
: プログラムに渡された引数のうち index 番目のものを返す．第 1 引数はスクリプト名．

int numberOfParameters()
: プログラムに渡された引数のうち，オプション(- からはじまるもの)と第一引数(スクリプト名)を除いたもの(パラメータ)の数を返す．

string getParameterOf(int index)
: プログラムに渡された引数のうち，オプション(- からはじまるもの)と第一引数(スクリプト名)を除いたもの(パラメータ)の index 番目のものを返す．

int isOptionSpecified(string option_name)
　　プログラムに渡された引数のうち，*-option_name*, *--option_name*, *-option_name=value*, *--option_name=value* となっているものがあれば真を返す．
string getOptionValueOf(string option_name)
　　プログラムに渡された引数のうち，*-option_name=value*, *--option_name=value* となっているものがあれば，その *value* の値を返す．
bool IsSwitchSpecified(string switch_character)
　　プログラムに渡された引数のうち，*-switch_character* としてあたえられた文字があれば，真を返す．

ファイルシステム

void makeDirectory(string directory_name)
　　新しいディレクトリを作成する．
string currentDirectory(void)
　　カレントディレクトリ名を返す．
void changeDirectory(string directory_name)
　　引数のディレクトリをカレントディレクトリに設定する．

その他

variant eval(string expression)
　　引数の文字列に書かれた式を KinokoScript の文法にしたがって評価し結果を返す．expression 内では，eval() が呼ばれたときに有効な変数を参照できる．

正規表現

正規表現の基本的な構文は以下のとおりです．

文字列値 =~ m/*パターン*/*オプション*
文字列値 !~ m/*パターン*/*オプション*
文字列値 =~ s/*パターン*/*置換文字列*/*オプション*

m/*パターン*/ は =~ 演算子の左辺の値に対して正規表現マッチを行い，マッチした文字列のリストを返します．先頭の m は省略可能です．!~ は，=~ と同等のマッチを行いますが，マッチしなかったときに true を，マッチしたときに false を返します． s/*パターン*/*置換文字列*/ は =~ 演算子の左辺の値に対して正規表現マッチを行い，マッチした部分を *置換文字列* で置き換えます．=~ 演算子の左辺の値が左辺値の場合，置換後の文字列が代入されます．

オプションに指定できる文字と意味は以下のとおりです．

g グローバルマッチを行う
i 大文字小文字を区別しない

以下にいくつかの例を示します．

単純なマッチ

```
string text = "cosine";
if (text =~ m/sin/) {
    println("matches");
}
```

サブマッチ

```
string text1 = "cosine";
list result1 = (text1 =~ m/(.+)(sin(.*))/);
println(result1);  // --> { "cosine", "co", "sine", "e" }

string text2 = "tangent";
list result2 = (text2 =~ m/(.+)(sin(.*))/);
println(result2);  // --> { }  （マッチしないときは空リストを返す）
```

置換

```
string text = "She sells sea shells.";
text =~ s/sea/C/;
println(text);      // --> "She sells C shells"
```

グローバルマッチオプションをつけると，マッチした全ての文字列を置換します．

```
string text = "She sells sea shells by the sea shore.";
text =~ s/sea/C/g;
text =~ s/shore/show/;
println(text);      // --> "She sells C shells by the C show."
```

今のところ，置換文字列中でのサブパターンの使用はできません．

分解

```
// split はパターンで区切られたテキストを要素に分解する
string line = "022-(217)-6727";
list elements = split(line, /[-()]+/);
println(elements)   // --> { "022", "217", "6727" }

// パターンを省略すると空白文字(列)で区切る
string line2 = "hello kinoko world";
list elements2 = split(line2);
println(elements2)   // --> { "hello", "kinoko", "world" }
```

SQL データベースインターフェース

方法 1: 普通のやり方

```
{
    // データベースオブジェクトの作成とコネクションの確立
    string driver_name = "PostgreSQL";
    string database_name = "E364";
    Database db(driver_name, database_name);

// 問い合わせの実行
string query = "select * from hv_table where hv > 2000";
QueryResult* result = db.executeSql(query);

// フィールド名の表示
int number_of_columns = result->numberOfColumns();
for (int i = 0; i < number_of_columns; ++i) {
        print(result->fieldNameOf(i) + " ");
}
println();
println("---");

// データの表示
while (result->next()) {
        for (int i = 0; i < number_of_columns; ++i) {
            print(result->get(i) + " ");
        }
        println();
}
```

```
    // 結果オブジェクトは問い合わせの処理が終るたびに delete する
    delete result;
}
```

方法 2: 簡単なやり方 その 1

```
{
    Database db("PostgreSQL", "E346");

    // 問い合わせの結果が 1 行 1 カラムの場合は getValueOf() で値を直接取得できる
    for (int channel = 0; channel < 32; channel++) {
        string query = "select hv from hv_table where channel=" + channel;
        int hv = db.getValueOf(query);
    }
}
```

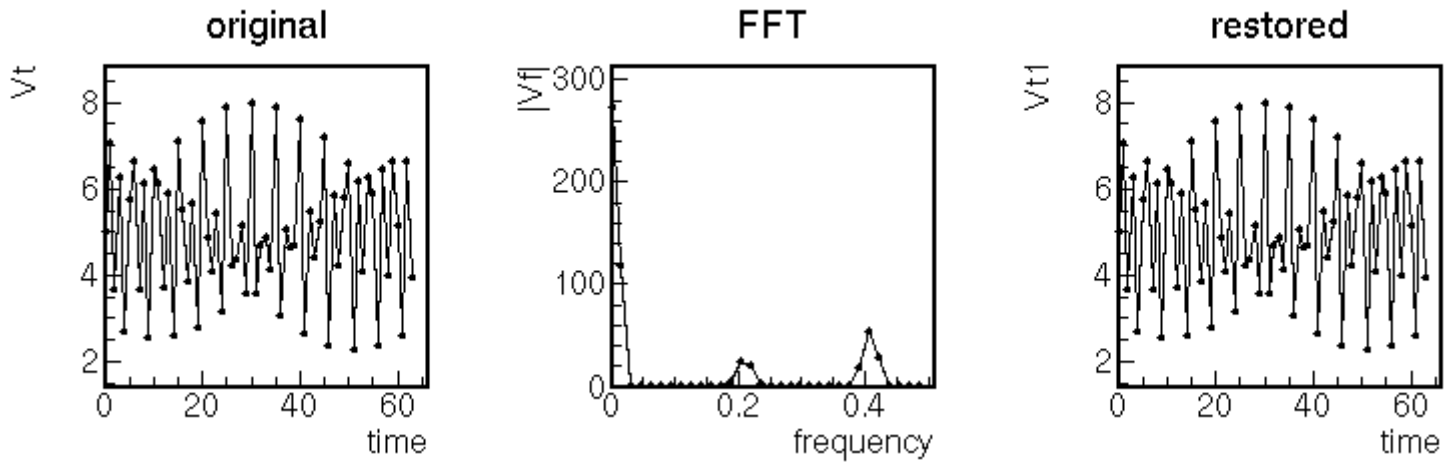
方法 3: 簡単なやり方 その 2

```
{
    Database db("PostgreSQL", "E346");

    // 問い合わせを実行し, 結果の各行に対してループ
    // ループ中では, "@フィールド名" で現在行のフィールドの値にアクセスできる
    sql[db] "select channel hv from hv_table" {
            println(@channel + ": " + @hv);
    }
}
```

ディジタル信号処理

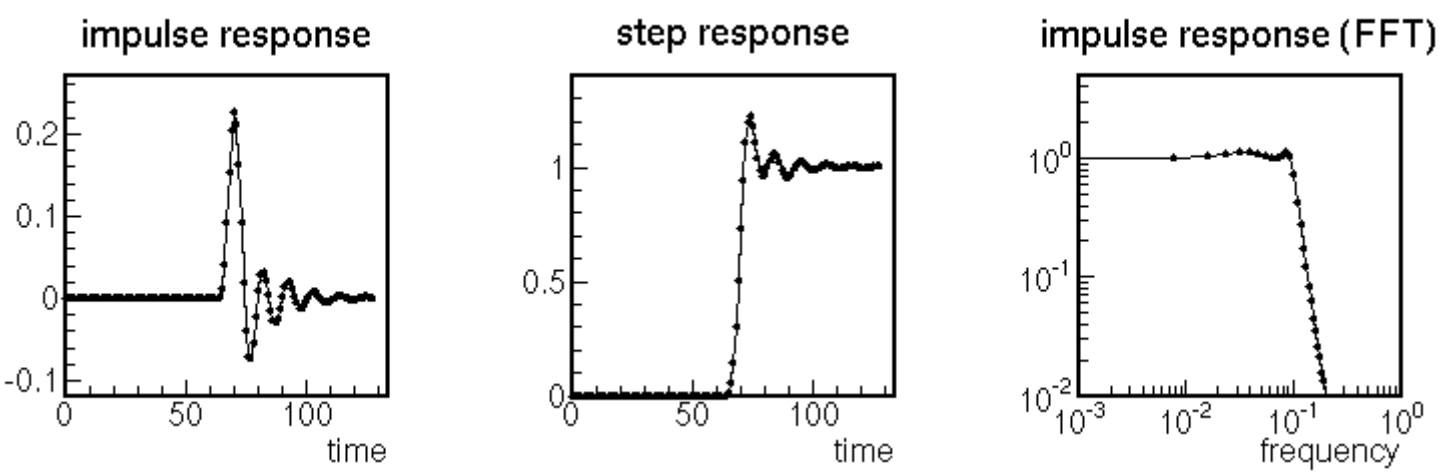
高速フーリエ変換 (FFT) の例

```
// 長さ 64 のサンプルデータ列を作る //
int n = 64;
list t = [0, n];
list Vt = sin(84.0 * t/n) + 2*sin(164.0 * t/n) + 5.0;    // 適当な関数でデータを作る
println(Vt);  // --> グラフ

// 窓関数を作り, 適用する //
Vt *= hammingWindow(sizeof(Vt));

// フーリエ変換:結果は複素数なので絶対値を表示する //
list Vf = fft(Vt);
println(abs(Vf));  // --> グラフ

// 逆変換でもとに戻してみる:結果は複素数なので実部を表示する(虚部は0のはず) //
// ついでに窓関数も戻しておく(窓によっては 0 除算に注意) //
list Vt1 = ifft(Vf);
Vt1 /= hammingWindow(sizeof(Vt1));
println(real(Vt1));  // --> グラフ
```



ディジタルフィルタの例

```
// インパルス入力とステップ入力を作る //
int sample_length = 2048;
list impulse = zeros(sample_length/2) ones(1) zeros(sample_length/2-1);
list step = zeros(sample_length/2) ones(sample_length/2);

// チェビシェフフィルタを生成 //
double cutoff_frequency = 0.1;
int number_of_poles = 4;
double max_ripple = 0.1;
ChebychevFilter filter(cutoff_frequency, number_of_poles, max_ripple);

// インパルス応答を計算する //
filter.applyTo(impulse);
println(impulse);  // --> グラフ

// ステップ応答を計算する //
filter.applyTo(step);
println(step);  // --> グラフ

//  インパルス応答のフーリエ変換 //
list f = fft(impulse);
println(abs(f));  // --> グラフ
```



組み込み関数

list fft(list x)

引数のリストに対して高速フーリエ変換を行う. 引数は complex または float のリスト, 戻り値は complex のリスト. 引数のリストの長さが2の冪乗でない場合, 2の冪乗になるまで後ろに 0 が加えられたものに対して計算される. 戻り値のリストの長さは引数のリストの長さと同じ(0 を加えた場合は加えた後の長さ)で, 通常の FFT の慣習に従った順序で並べられている. すなわち, 最初の要素が周波数 0 (DC) に対応し, 中央の点がナイキスト周波数. その先は負のナイキスト周波数から 0 (DC) へ向かって並べられる(入力が実数の場合は正の周波数部分の折り返しとなる).

list ifft(list x)

引数のリストに対して高速逆フーリエ変換を行う. 引数および戻り値の意味は fft() に準じる.

list parzenWindow(int data_length)
list hanningWindow(int data_length)
list hammingWindow(int data_length)
list welchWindow(int data_length)
list blackmanWindow(int data_length)
　　ウィンドウ関数. 引数に指定された長さのリストにして返す.
float sinc(float x) / list sinc(list x)
　　sinc() 関数

組み込みクラス

ディジタルフィルタは以下の組み込みクラスで実装されています. 周波数はサンプル間隔を 1 として規定されています. したがって, ナイキスト周波数は 0.5 に対応します.

MovingAverageFilter(float cutoff_freq)
WindowedSincFilter(float cutoff_freq, int kernel_length)
SinglePoleFilter(float cutoff_frequency)
ButterworthFilter(float cutoff_freq, int n_of_poles)
ChebychevFilter(float cutoff_freq, int n_of_poles, float max_ripple)

読み出しスクリプト

読み出しスクリプトは, データ読み出しの手順を表現する内部オブジェクトの構造(読み出しシーケンス)を構築するために, KinokoCollector や tinykinoko などによって使用されるスクリプトです. 基本スクリプトに加え, 以下の文法要素が追加されています.

- datasource エントリ
- on 文
- when 文
- unit 文
- attribure 文
- invoke 文
- terminate / skip 文
- VmeCrate / VmeController / VmeModule クラス
- CamacCrate / CamacController / CamacModule クラス
- SoftwareModule クラス
- ReadoutChannelList クラス
- Register クラスとオーバロードされた演算子
- DataRecord クラス
- getRegistry() / setRegistry() 関数
- readRegistry() / writeRegistry() 関数
- suspend() 関数
- echo() 関数
- scriptFileName() 関数

読み出しスクリプトはシステムの起動時に一度だけ実行され, 読み出しシーケンスを構築します. 読み出しシーケンスとは, デバイスにアクセスしてデータの読み書きを行う一連の手続きを表現した内部オブジェクトの構造です. 読み出しシーケンスはシーケンスの開始条件とともに記述され, データ収集中に開始条件が成立するとそのたびに毎回実行されます. スクリプトの実行とシーケンスの実行を混同しないように注意してください.

クレート, コントローラ, モジュール

読み出しスクリプトにおいて, まずはじめに行なうことは, 使用するデバイスの宣言と, 対応するドライバの関連付け, およびそれらの接続を記述することです.

使用するデバイスの宣言は, 通常のオブジェクトと同じ構文でおこないます. VME, CAMAC, およびソフトウェアの各モジュール, コントローラ, クレートに対して, それぞれ VmeModule/CamacModule/SoftwareModule, VmeController/CamacController, および VmeCrate/CamacCrate クラスがあります(ソフトウェアモジュールにはコントローラおよびクレートはありません). モジュールおよびコントローラには, その実際のデバイスに対応したドライバ(KinokoModuleDriver, KinokoControllerDriver) のドライバ名をコンストラクタの第 1 引数に指定します.

```
datasource CamacAdc {
    CamacCrate camac_crate;
    CamacController camac_controller("Toyo-CC7x00");
    CamacModule adc("Rinei-RPC022");
    CamacModule tdc("Rinei-RPC060");

    VmeCrate vme_crate;
    VmeController vme_controller("SBS-620");
    VmeModule latch("SIS_3600");
    VmeModule interrupter("Rinei-RPV130");
```

利用できるモジュールドライバおよびコントローラドライバの一覧は，コマンド kinoko-lsmod で表示させることができます．

```
% kinoko-lsmod
VME Controllers:
    SBS_617:
    SBS_618:
    SBS_620:
    Kinoko_Vmedrv:
    Null:

CAMAC Controllers:
    Toyo_CC7x00:
    Hoshin_CCP:
    Kinoko_Camdrv:
    Null:

VME Modules:
    Generic_MemoryA16D16: VmeMemory (Generic_VmeMemory)
    Generic_MemoryA16D32: VmeMemory (Generic_VmeMemory)
    (中略)
    Hoshin_V004: VmeScaler (Hoshin_V004)
    Rinei_RPV130: VmeIORegister (Rinei_RPV130)
    Rinei_RPV160: VmeFADC (Rinei_RPV160)
    SIS_3600: VmeLatch (SIS_3600)
    SIS_3601: VmeOutputRegister (SIS_3601)
    SIS_3801: VmeScaler (SIS_3801)

CAMAC Modules:
    Generic_Standard: CamacModule (Generic_CamacModule)
    Rinei_RPC022: CamacQADC (Rinei_RPC022)
    Rinei_RPC060: CamacTDC (Rinei_RPC060)
    Rinei_RPC081: CamacFADC (Rinei_RPC081)
    (中略)
```

ここで，コロンの左側がドライバ名，右側がそのモジュールの種別と型番です．ドライバ名は，原則として "製造者_型番" となっています．ドライバ名中のアンダースコアはスクリプト中でハイフンに置き換えることができます．

使用するデバイスを宣言したら，次にそれらの接続を記述します．CAMAC ならどのモジュールをどのステーションに入れるか，VME ならベースアドレスと，必要なら IRQ および割り込みベクタを指定します．これを行なうのが，VmeCrate/CamacCrate の installController()/installModule() メソッドです．

```
int station;
camac_crate.installController(camac_controller);
camac_crate.installModule(adc, station = 10);
camac_crate.installModule(tdc, staion = 11);

int address, irq, vector;
vme_crate.installController(vme_controller);
vme_crate.installModule(latch, address = 0x01000000);
vme_crate.installModule(interrupter, address = 0x8000, irq = 3, vector = 0xfff0);
```

8bit の割り込みベクタを使用する場合，installModule() に指定する割り込みベクタの上位 8bit を 0xff で埋めてください (0xf0 -> 0xfff0 など).

ここまでで，デバイスにアクセスする準備が整いました.

読み出しシーケンスとシーケンス開始条件

セットアップの記述が終了したら，次に読み出し手順の記述を行ないます.

Kinoko では，データ収集時におけるスクリプト解釈のオーバーヘッドを避けるため，スクリプトの解釈はデータ収集開始前に一度だけ行ない，そのときに読み出し手順に対応した内部オブジェクトの構造(読み出しシーケンス)を組み立てるようにしています．読み出しシーケンスとは，シーケンスアクションと呼ばれる単純な操作(1 モジュールの読み出し，クリア，レジスタアクセスなど)のリストです．データ収集が開始されたあとは，このシーケンスが繰り返し実行されることになります．スクリプトの実行とシーケンスの実行を混同しないように注意してください.

シーケンスは，データ収集中に「シーケンス開始条件」が成立することにより実行されます．シーケンス開始条件には，例えばモジュールからの読み出し要求や，タイマによる指定時間経過の通知などがあります．一つのスクリプト中に複数のシーケンスを記述することができます．全てのシーケンスはそれぞれのシーケンス開始条件を持ちます．スクリプトでは，シーケンスはシーケンス開始条件とともに記述されます.

シーケンスおよびその開始条件の記述は，on 文によって行ないます． on 文の先頭に開始条件を指定し，その後にシーケンス構築文を続けます． シーケンス構築文では，シーケンスアクション生成メソッドという特殊なメソッドを呼ぶことにより，シーケンスにシーケンスアクションを追加していきます.

以下は，CAMAC の ADC からデータを読んでクリアする読み出しスクリプトの例です． ここでは，ADC の LAM によりシーケンスを開始するようにしています．シーケンス構築文では，CamacModule クラスの持っているシーケンスアクション生成メソッド read() と clear() を使って，それぞれ READ アクションと CLEAR アクションを生成し，シーケンスに追加しています.

```
datasource CamacAdc
{
    CamacCrate crate;
    CamacController controller("Toyo-CC7x00");
    CamacModule adc("Rinei-RPC022");

    int station;
    crate.installController(controller);
    crate.installModule(adc, station = 10);

    on trigger(adc) {        // ADC の LAM でシーケンスを開始する
        adc.read(#0..#3);    // ADC のチャンネル 0 から 3 を読み出すシーケンスアクションを生成
        adc.clear();         // ADC をクリアするシーケンスアクションを生成
    }
}
```

このスクリプトにより生成されるシーケンスは，コマンド ktscheck で表示させることができます(細かい部分はあまり気にしないでください).

```
% ktscheck --show-sequence CamacAdc.kts
trigger 2:
        .sequence
        ->      SingleRead      adc, 0:1:2:3
        ->      CommonControl   adc, CLEAR
        .end
```

以下は，input_register からのトリガを待って，adc 4 枚の読み出しとクリアを行う例です．スクリプトの実行と構築されるシーケンスの関係を強調するために，若干複雑なスクリプトにしてあります.

```
// 読み出す ADC のリスト
list adc_list = { &adc0, &adc1, &adc2, &adc3 };

on trigger (input_register) {
    for (int i = 0; i < sizeof(adc_list); i++) {
        adc_list[i]->read(#0..#3);
        adc_list[i]->clear();
    }
}
```

これにより構築されるシーケンスは以下のようになります.

```
% ktscheck --show-sequence foo.kts
trigger 2:
        .sequence
        ->      SingleRead      adc0, 0:1:2:3
        ->      CommonControl   adc0, CLEAR
        ->      SingleRead      adc1, 0:1:2:3
        ->      CommonControl   adc1, CLEAR
        ->      SingleRead      adc2, 0:1:2:3
        ->      CommonControl   adc2, CLEAR
        ->      SingleRead      adc3, 0:1:2:3
        ->      CommonControl   adc3, CLEAR
        .end
```

スクリプトの実行によりループが展開されていることに注意してください. なお, このスクリプトから生成されるシーケンスは, 以下のスクリプトから生成されるシーケンスと厳密に同等になります.

```
on trigger (input_register) {
    adc0.read(0x000f);
    adc0.clear();
    adc1.read(0x000f);
    adc1.clear();
    adc2.read(0x000f);
    adc2.clear();
    adc3.read(0x000f);
    adc3.clear();
}
```

つまり, 前者のように記述しても, 実行時のオーバーヘッドには全く差が生じないことになります.

逆に, 以下のようにしても, 期待どおりの動作はしません.

```
int count = 0;
on trigger(input_register) {
    count++;
    if (count > 1000) {
        scaler.read(#0);
        scaler.clear();
        count = 0;
    }
}
```

これは, 以下の記述と厳密に同等です ("スクリプト" は一度だけ実行され, その時 if 文の条件は成立しないから).

```
on trigger(input_register) {
    ;
}
```

シーケンス実行時に変数の値を参照したい場合は, 以下のようにシーケンスレジスタを使用する 必要があります. 詳細は, シーケンスレジスタの項を参照してください.

```
Register count;
on trigger(input_register) {
    count += 1;
    when (count > 1000) {
        scaler.read(#0);
        scaler.clear();
        count = 0;
    }
}
```

参考に，このスクリプトから生成されるシーケンスを表示させてみると，以下のようになります．

```
% ktscheck --show-sequence foo.kts

trigger 2:
        .sequence
        ->      OperateRegister [0x861a348], 1, ADD
        ->      .sequence conditional ([0x861a348] > 3e8)
        ->      ->      SingleRead      scaler, 0
        ->      ->      CommonControl   scaler, CLEAR
        ->      ->      OperateRegister [0x861a348], 0, LOAD
        ->      .end
        .end
```

シーケンスアクション生成メソッド

モジュールオブジェクト (CamacModule / VmeModule / SoftwarModule) には，以下のシーケンスアクション生成メソッドが定義されています．ただし，モジュールによっては，これらの一部のみが実装されていることがあります．詳細については，各モジュールの KinokoModuleDriver リファレンスを参照してください．

void read(ChannelList channel_list)
void read(int channel_bits)
: 引数に指定された各チャンネルからデータを 1 ワードづつ読み出し，indexed 型のデータとしてデータストリームに送り出す．単純な ADC, TDC やスケーラなどで利用される．

void tagRead(ChannelList channel_list)
: 引数に指定された各チャンネルからデータを 1 ワードづつ読み出し，引数の ChannelList に指定されているタグを使って tagged 型のデータとし，データストリームに送り出す．

void sequentialRead(ChannelList channel_list)
void sequentialRead(int channel_bits)
: 引数に指定された各チャンネルからデータを複数ワード読み出し，indexed 型のデータとしてデータストリームに送り出す．読み出されるデータワード数はモジュールに依存する．FADC や マルチヒット TDC などで利用される．

void blockRead()
void blockRead(int address)
void blockRead(int address, int size)
: モジュールからデータブロックを読み出し，block 型のデータとしてデータストリームに送り出す．FIFO やバッファを持つモジュールの読み出しに向いている．

void clear()
void clear(int channel)
: モジュール上のデータをクリアする．実際の振舞いはモジュール依存．

void enable()
void enable(int channel)
: モジュールまたはモジュール上の指定されたチャンネルをエネーブルする．実際の振舞いはモジュール依存．

void disable()
void disable(int channel)
: モジュールまたはモジュール上の指定されたチャンネルをディスエーブルする．実際の振舞いはモジュール依存．

void writeRegister(int address, Register register)
: モジュール上の指定されたアドレスに値を書き込む．

void readRegister(int address, Register register)
: モジュール上の指定されたアドレスから値を読み出し，引数のレジスタに格納する．

void waitData()
void waitData(int/Register timeout)
void waitData(int/Register timeout, Register response)
モジュールでデータが利用可能になるまで実行を停止する.
timeout が指定されている場合は timeout 秒を上限にデータを待つ.
データが来れば response レジスタに 1 を, 来なければ 0 を返す.

上記の他, モジュールによって独自のアクション生成メソッドが定義されていることがあります. 詳細については, 各モジュールの KinokoModuleDriver リファレンスを参照してください.

CamacController には以下のアクション生成メソッドがあります.

void initialize()
CAMAC クレートに Z を発行する.
void clear()
CAMAC クレートに C を発行する.
void setInhibition()
CAMAC クレートに I を設定する.
void releaseInhibition()
CAMAC クレートの I をクリアする.

CamacModule には以下のアクション生成メソッドがあります.

void transact(int F, int A)
指定された F と A で CAMAC アクションを実行する.
void transact(int F, int A, int data)
void transact(int F, int A, Register data)
指定された F, A, data で CAMAC アクションを実行する. data がレジスタなら, 結果を data に書き戻す.
void transact(int F, int A, int data, Register Q, Register X)
void transact(int F, int A, Register data, Register Q, Register X)
指定された F, A, data で CAMAC アクションを実行し, Q および X レスポンスを引数の Q, X レジスタに返す. data がレジスタなら, 結果を data に書き戻す.

CAMAC の read() などでは, Q レスポンスが帰ってこないときはデータが存在しないとしてデータストリームには全く何も送り出しません. もし, Q がないということを, 特別なデータ値で示したいならば, setInvalidDataValue() 関数を使ってそのことを指定してください. これにより, Q レスポンスの有無にかかわらず, read() は常に値を送り出すようになります.

void setInvalidDataValue(int value)
Q レスポンスがないときのデータ値を指定する

シーケンス開始条件とトリガハンドリング

シーケンス開始条件には, 以下ように トリガ, トラップ, コマンド の 3 つの種類があります.

トリガ
構文: on trigger （*デバイス*）
デバイスからのサービスリクエスト(後述)でシーケンスを実行する. サービスリクエストの詳細はデバイスに依存するが, 一般には, CAMAC なら LAM, VME なら割り込みなどになる.
トラップ
構文: on *トラップ名*
ランの開始や終了などの, ある特定のタイミングでシーケンスを実行する. 現在定義されているトラップは, run_begin / run_end / run_suspend / run_resume の 4 つ.
コマンド
構文: on command （*コマンド名, パラメータリスト*^{opt}）
ユーザのコントロールなど, 外部コマンドによりシーケンスを実行する.

on trigger の引数に渡されるデバイスは, 読み出し開始などの要求をシステムに通知できる機能を持ったものでなければなりません. これらの要求通知は, 通常は CAMAC なら LAM を使って, VME ならば割り込みやモジュール上のレジスタのフラグなどを使って実現されます.

モジュールの読み出し要求がシステムに通知されるようにするためには, 対応する KinokoModuleDriver が, サービスリクエスタと呼ばれるインターフェースを実装している必要があります. サービスリクエスタとは, デバイスの読み出し要求等をシステムに通知するためのインターフェースです. デバイスにより読み出し要求の形式が異なるため, それに対応して, サービスリクエスタにも以下のようないくつかの形式があります.

WaitForSeriveceRequest()
　　サービス要求が出されるか，指定した時間が経過するまで実行をブロックする．
IsRequestingService()
　　サービス要求を出しているか調べる．
EnableSignalOnServiceRequest()
　　サービス要求の際にシグナルを発行するように設定する．

これらのうちどのインターフェースが実装されるかはモジュールおよびコントローラの仕様によります．

同時に接続されるサービスリクエスタの数や，それらに実装されているインターフェースの種類に応じて，Kinoko は以下のうちから適当なハンドリングスキームを選択します．

シングルトリガソース
　　WaitForServiceRequest() を呼び出す．
ポーリングループ
　　一定時間(1ms)ごとに各デバイスの IsRequestingService() を呼ぶ．
　　ポーリングの順はラウンドロビン方式．
シグナル割り込み & ポーリング
　　EnableSinalOnServiceRequest() をしてから，シグナルを待ち，シグナルが来るか一定時間(1sec)が経過したらポーリングを行なう．

datasource 内に on trigger が 1 つだけの場合，常にシングルトリガソースが選択されます．on trigger が 2 つ以上の場合，全てのサービスリクエスタがシグナルの発行をサポートしている場合はシグナル割り込み&ポーリングが，そうでない場合はポーリングループが選択されます．これらのハンドリングスキームによってシステムのパフォーマンスおよび割り込み応答の速さが大きく影響されることに注意してください．

いくつかの組み合わせについて，利用できるサービスリクエスタのインターフェースを以下に示します．

VME モジュール + vmedrv
　　モジュールが VME 割り込みをサポートしている場合，WaitForServiceRequest() はドライバ内部で VME 割り込みから変換された PCI 割り込みを待つ．
　　モジュールが VME 割り込みをサポートしている場合，シグナル割り込みが利用できる．
　　IsRequestingService() の実装はモジュール依存．通常利用できる．
CAMAC モジュール + camdrv (東陽 CC/7x00)
　　WaitForServiceRequest() はドライバ内部で LAM から変換された PCI の割り込みを待つ．
　　IsRequestingService() は TestLAM (F8) を実行し Q レスポンスを見る．
　　シグナル割り込みは(今のところ)利用できない．
CAMAC モジュール + camdrv (豊伸 CCP)
　　WaitForServiceRequest() はドライバ内部で LAM を待つループを実行する．
　　IsRequestingService() は TestLAM (F8) を実行し Q レスポンスを見る．
　　シグナル割り込みは利用できない．
ソフトウェアモジュール
　　WaitForServiceRequest() は条件が成立するまでインターバルタイマーでスリープする．
　　IsRequestingService() の実装はモジュール依存．タイマー系(IntervalTimer, OneshotTimer など) では通常利用できる．
　　シグナル割り込みは利用できない．

ポーリングの間隔は Kinoko により適切に設定されますが，setPollingInterval() 関数によってポーリング間隔を指定することができます．また，もし Kinoko が割り込みを選択する場合に割り込みを使用したくない場合，forcePolling() 関数を呼び出すことにより割り込みの使用を回避することができます．

void setPollingInterval(int interval_usec)
　　ポーリングの間隔をマイクロ秒単位で指定する．
void forcePolling()
void forcePolling(int interval_usec)
　　割り込み使用の可否にかかわらずポーリングを使用するように指定する．

タイミングコントロール

on run_begin などのトラップを使うと，データ収集における特定のタイミングでシーケンスを実行させることができます．

```
on run_begin {
    // スタート直後に，他の全てのシーケンスに先だって実行される
    output_register.outputPulse(#0);
}

on trigger(adc) {
    adc.read(#0..#3);
    adc.clear();
}
```

現在実装されているトラップには，以下のものがあります．

run_begin
　　データ収集開始直後にシーケンスを実行する．
run_end
　　データ収集終了直前にシーケンスを実行する．
run_suspend
　　データ収集が一時中断されるとき，その直前にシーケンスを実行する．
run_resume
　　中断されていたデータ収集が再開されるとき，その直後にシーケンスを実行する．

また，on command を使用すると，KCOM イベントなどの外部イベントにより起動されるシーケンスを記述することもできます．on command の詳細については，外部イベントインターフェースの項を参照してください．

IntervalTimer などのソフトウェアモジュールを使用すると，一定時間間隔でくり返し実行するシーケンスや，一定の時間経過後に実行するシーケンスを記述することができます．

```
int interval_sec = 0;
int interval_usec = 100000;  // 100ms
SoftwareModule timer("IntervalTimer");

on run_begin {
    // くり返し周期を指定する
    timer.setInterval(interval_sec, interval_usec);
}

on trigger(timer) {
    adc.blockRead();
}
```

```
SoftwareModule timer("OneShotTimer");

on run_begin {
    // 3 秒後にシーケンスを実行するようにする
    timer.setInterval(3);
    timer.start();
}

on trigger(timer) {
    output_register.outputLevel(#0);
}
```

数ミリ秒から数秒程度の短いディレイをシーケンスに挿入するには，suspend() シーケンスアクション生成関数を用います．複数のシーケンスが同時に実行されることはないので，どこかで suspend() をすると，そのデータソース全体がその時間だけ停止することになります．

```
SoftwareModule timer("OneShotTimer");

on run_begin {
    output_register.outputLevel(#0);

    // ここで 1 秒間実行を停止する
    suspend(1, 0);
}

on trigger(adc) {
    adc.blockRead();
}
```

suspend() シーケンスアクション生成関数のインターフェース宣言は以下のとおりです.

void suspend(int sec, int usec)
　　指定した時間シーケンスの実行を停止する.

シーケンスレジスタ

シーケンスレジスタとは, シーケンス中において変数のように扱えるデータ領域です. シーケンス実行時に値の評価・操作が行えますが, 通常の演算子は使用できず, 行える演算も限られています. また, 保持できる値は整数のみです.

```
Register reg;

on run_begin {
    reg = 0;
}

on trigger(timer) {
    reg += 1;
    output_register.writeRegister(address = 0, reg);
}
```

現在利用できる演算は以下のとおりです.

register = int_value / register = register
　　左辺のレジスタに右辺の値を代入する.
register += int_value / register += register
　　左辺のレジスタに現在の値と右辺の値の和を代入する.
register −= int_value / register −= register
　　左辺のレジスタに現在の値と右辺の値の差を代入する.
register *= int_value / register *= register
　　左辺のレジスタに現在の値と右辺の値の積を代入する.
register /= int_value / register /= register
　　左辺のレジスタに現在の値と右辺の値の商を代入する.
register %= int_value / register %= register
　　左辺のレジスタに現在の値と右辺の値の剰余を代入する.
register &= int_value / register &= register
　　左辺のレジスタに現在の値と右辺の値のビット論理積(bitwise AND)を代入する.
register |= int_value / register |= register
　　左辺のレジスタに現在の値と右辺の値のビット論理和(bitwise OR)を代入する.
register ^= int_value / register ^= register
　　左辺のレジスタに現在の値と右辺の値のビット排他的論理和(bitwise eXclusive OR)を代入する.
register <<= int_value / register <<= register
　　左辺のレジスタの値を右辺の値だけ左にビットシフトさせる.
register >>= int_value / register >>= register
　　左辺のレジスタの値を右辺の値だけ右にビットシフトさせる.

スクリプトのデバッグなどの用途のために，レジスタの値を画面に表示するアクション dump() があります．

```
datasouce RegisterTest {
    SoftwareModule timer("IntervalTimer");
    Register trigger_count;

    on trigger(timer) {
        trigger_count += 1;
        trigger_count.dump();
    }
}
```

echo() アクションを使っても，レジスタの値を表示させることができます．

```
datasouce RegisterTest {
    SoftwareModule timer("IntervalTimer");
    Register trigger_count;

    on trigger(timer) {
        trigger_count += 1;
        echo("trigger counts are: ", trigger_count);
    }
}
```

これらスクリプトを tinykinoko で実行すると，プログレスバーの表示などと干渉して画面が乱れるので，--quiet(または-q)オプションを指定してください．

```
% tinykinoko RegisterTest.kts foo.kdf --quiet
trigger_count: 1
trigger_count: 2
trigger_count: 3
^C
%
```

条件付きシーケンス実行

when 文を使うと，シーケンスレジスタの値に応じて，特定のシーケンスのみを実行するようにすることができます．

```
Register reg;
on trigger(input_register) {
    input_register.readRegister(address = 0, reg);
    when (reg == 0x0001) {
        adc_01.read();
    }
    when (reg == 0x0002) {
        adc_02.read();
    }
}
```

レジスタに対して利用できる関係演算子(レジスタ関係演算子)は以下のとおりです．演算子の意味や結合準位は通常の関係演算子と同じですが，左辺はレジスタ，右辺は整数値またはレジスタでなければなりません．また，戻り値は特殊な「レジスタ論理値」型となります．この型の値は when 文の条件式でのみ意味を持ちます．

register == value
: レジスタの値が value に等しいとき true を返す．

register != value
: レジスタの値が value に等しくないとき true を返す．

register < valeue
: レジスタの値が value よりも小さいとき true を返す．

register <= value
: レジスタの値が value より小さいか等しいとき true を返す．

register > value
: レジスタの値が value より大きいとき true を返す.

register >= value
: レジスタの値が value より大きいか等しいとき true を返す.

register & value
: レジスタの値と value のビット論理積 (bitwise AND) が 0 でないとき true を返す.

register ^ value
: レジスタの値と value のビット排他的論理和 (bitwise eXclusive OR) が 0 でないとき true を返す.

さらにレジスタ論理値に対して，以下の論理演算子を利用できます．戻り値はおなじくレジスタ論理値型です(ネストして使用できます).

! register_bool
: 論理反転：register_bool の値が false のとき true を返す.

register_bool && register_bool
: 論理積：右辺値と左辺値がともに true の場合 true を返す.

register_bool || register_bool
: 論理和：右辺値と左辺値のどちらかが true の場合 true を返す.

データレコード

データレコードは，レジスタの値をデータストリームに送るためのオブジェクトです． 複数のレジスタに名前を付けて tagged 型のデータとしてストリームに送り出します.

```
DataRecord run_header_record;
DataRecord event_count_record;
Register event_count;

on run_begin {
    discriminator.writeRegister(address = 0, threshold_00);
    discriminator.writeRegister(address = 1, threshold_01);

    run_header_record.fill("Threshold-00", threshold_00);
    run_header_record.fill("Threshold-01", threshold_01);
    run_header_record.send();

    event_count = 0;
}

on trigger(adc) {
    adc.read(#0..#3);

    event_count += 1;
    event_count_record.fill("EventCount", event_count);
    event_count_record.send();
}
```

fill() メソッドの第３引数にデータ幅（1 から 32）指定できます．省略した場合は 32 となります．第４引数には kdfdump などのツールで表示したときの表示フォーマットを指定できます．指定形式は C の printf() と同様です.

```
    run_header_record.fill("address", address, 24, "%08x");
```

時刻の取得

シーケンスアクション生成関数 readTime() は，シーケンス実行時に時刻を取得し，引数のレジスタにその値を格納します.

```
Register time, event_number;
DataRecord event_info;

on trigger(adc) {
    adc.read(channel_list);
```

```
    readTime(time);
    event_number.add(1);
    event_info.fill("time", time);
    event_info.fill("event_number", event_number);
    event_info.send();
}
```

readTime() は，第 2 引数にレジスタを与えれば，そこにマイクロ秒の時間を返します．

```
    Register time_sec, time_usec;
    readTime(time_sec, time_usec);
    event_info.fill("time_sec", time_sec);
    event_info.fill("time_usec", time_usec);
```

IntervalTimer や OneShotTimer などのタイマ系ソフトウェアモジュールは，read() シーケンスアクションにより，UNIX 時間をデータとしてストリームに送り出します．

```
SoftwareModule timer("IntervalTimer");

on trigger(timer) {
    timer.read(#0);
    adc.blockRead();
}
```

KiNOKO Version 2.0 からは，TimePacket が導入され，Collector など Kinoko の標準ツールで取得したデータには全て秒精度のタイムスタンプが付加されるようになりました．したがって，上記のような方法で時刻を記録する必要はほとんどなくなったはずです．

外部イベントインターフェース

[TinyKinoko, SmallKinoko ではこの機能は利用できません]

KCOM イベントなどの外部イベントに応じてシーケンスを起動するには，on command 文を使用します．on command 文はパラメータを取り，最初のパラメータにイベント名，必要ならそれ以降に引数を指定します．引数は，すでに宣言されたレジスタでなければなりません．

```
Register threshold;
on command("setThreshold", threshold) {
    discriminator.writeRegister(address = 0, threshold);
}
```

また，invoke 文を使うことにより，KCOM イベントなどの外部イベントを発行することができます．

```
on trigger(adc) {
    adc.read(#0..#3);

    Register event_count;
    event_count.add(1);
    when (event_count > 100) {
        invoke clear();
        event_count.load(0);
    }
}
```

外部レジストリインターフェース

読み出しスクリプトから KCOM レジストリなどの外部レジストリにアクセスするためのインターフェースとして，getRegistry()/setRegistry() と readRegistry()/writeRegistry() があります．前者は通常の関数で，スクリプト実行時にレジストリにアクセスします．後者はシーケンスアクション生成関数で，実際にレジストリにアクセスするのはシーケンス実行時になります．

スクリプト実行時に変数に読む

```
string run_type = getRegistry("control/run_type");
if (run_type == "calibration") {
    on run_begin {
        pulser.start();
    }
}
```

シーケンス実行時にレジスタに読む

```
on run_begin {
    Register threshold;
    readRegistry("control/threshold", threshold);
    discriminator.setThreshold(threshold);
}
```

データソース ID とセクション ID の指定

通常，データソース ID や セクション ID は，Kinoko によって自動で割り当てられますが，これらの値がユーザの指定した値に固定されていると解析プログラムなどが作成しやすくなることがあります． これを行なうために，スクリプト中でこれらの値を指定できるようになっています．

データソース ID を指定するには，datasource エントリのデータソース名の直後で，"<>" で囲ってデータソース ID を記述します．セクション ID は，そのデータを読み出すモジュールオブジェクトのコンストラクタにおいて，引数によってセクション名とともに指定します．

```
datasource CamacAdc<3>
{
    CamacCrate crate;
    CamacController controller("Toyo-CC7x00");
    CamacModule adc("Rinei-RPC022", "adc", 7);

    crate.installController(controller);
    crate.installModule(adc, 2);

    on trigger(adc) {
        adc.read(#0..#15);
        adc.clear();
    }
}
```

特に指定しなければ，Kinoko の自動割り当てによるデータソース ID の値は 1024 以上に，セクション ID の値は 256 以上になります．手動による指定では，これらより小さい値を使うようにしてください．また，0 は特別な意味を持つので，使わないようにしてください．

ネストデータセクションと unit 文

通常，一つの読み出しアクションにつき一つのデータセクションが生成されますが，unit 文を使うと，複数の読み出しアクションをまとめて一つの nested セクションを生成することができます．

```
on trigger(adc) {
    unit event {
        adc.read(#0..#3);
        tdc.read(#0..#3);
    }
}
```

データソースと同様に，セクション名の直後で "<>" で囲むことにより，セクション ID を指定することができます．

```
    unit event<12> {
```

データソースアトリビュート宣言

attribute 文を使用すると，スクリプトからデータソースアトリビュートを追加することができます．

```
datasource CamacAdc
{
    attribute setup_version = "1.0";

    CamacCrate crate;
    CamacController controller("Toyo-CC7x00");
    CamacModule adc("Rinei-RPC022");

    crate.installController(controller);
    crate.installModule(adc, 2);

    on trigger(adc) {
          adc.read(#0..#15);
          adc.clear();
    }
}
```

ビュースクリプト

ビュースクリプトは，データの処理手順やヒストグラム等の表示手順などを表現する内部オブジェクトの構造(アナリシスシーケンスとビューシーケンス)を構築するために，KinokoViewer などによって使用されるスクリプトです． 基本スクリプトに加え，以下の文法要素が追加されています．

- display エントリ
- analysis 文
- on 文
- invoke 文
- DataElement クラス
- Histogram / Histogram2d / Trend / Wave / Map / Tabular クラス
- Grid / Placer クラス
- PlainTextViewRepository / XmlViewRepository / RootViewRepository クラス
- setRegistry()/getRegistry() 関数

アナリシスシーケンスとビューシーケンス

ビュースクリプトも読み出しスクリプトと同様にシーケンスの構造を記述するものですが，ビュースクリプトでは，データの処理に関するシーケンス(アナリシスシーケンス)と，表示要素(ビューオブジェクト)に対するシーケンス(ビューシーケンス)の 2 つを記述します．アナリシスシーケンスは，対象とするデータを受け取るたびに実行され，ヒストグラムのフィルやデータ条件に基づくユーザへの通知などを行います． 一方，ビューシーケンスは，run_begin/run_end などの特定の条件や外部からのコマンド，あるいはタイマなどにより起動され，ビューオブジェクトの表示や保存などを行います．

アナリシスシーケンス

アナリシスシーケンスはキーワード analysis で記述します．アナリシスシーケンスは，処理対象のデータパケットを受け取るたびに実行されます．analysis キーワードの直後に，処理対象データのデータソース名を指定します．

```
Histogram histogram_adc_01("ADC ch 01", 256, 0, 4096);

// データソース CamacAdc からのデータでヒストグラムをフィルする
analysis ("CamacAdc") {
    DataElement adc_01("adc", 1);
    histogram_adc_01.fill(adc_01);
}
```

システムにデータソースが一つしかない場合(SmallKinoko を使っている場合など)，以下のように，データソース名を省略することができます．データソースが複数あって，データソース名が省略されている場合，処理対象となるデータソースは未定義です．

```
Histogram histogram_adc_01("ADC ch 01", 256, 0, 4096);

analysis {
    DataElement adc_01("adc", 1);
    histogram_adc_01.fill(adc_01);
}
```

ビューシーケンス

ビューシーケンスは，シーケンス開始条件とともにキーワード on で記述します．シーケンス開始条件には，以下のものがあります．

on every (*time* sec)
: time に指定された時間ごとにシーケンスが繰り返し実行されます．

on construct / on destruct
: システム構築直後または終了処理直前にシーケンスが実行されます．

on run_begin / on run_end
: ラン開始直前またはラン終了直後にシーケンスが実行されます．

on run_suspend / on run_resume
: ランが中断した直後または再開する直前にシーケンスが実行されます．

on pre_clear / on post_clear
: ビューアの表示がクリアされる直前または直後にシーケンスが実行されます．

以下は，ビューシーケンスの記述例です．

```
// 1 秒毎にヒストグラムを画面に描画
on every(1 sec) {
    histogram_adc_01.draw();
}

// ラン終了時にヒストグラムをファイルに保存する
PlainTextViewRepository repository("foo");
on run_end {
    histogram_adc_01.save(repository);
}
```

DataElement クラス

DataElement クラスは，アナリシスシーケンスにおいて，データストリーム中の特定のデータエレメントを指定するために使われるものです．

```
Histogram histogram_adc_01("ADC ch 01", 256, 0, 4096);

analysis {
    // セクション名 "adc", アドレス 1 を指定
    DataElement adc_01("adc", 1);

    // データエレメントを使ってヒストグラムをフィル
    histogram_adc_01.fill(adc_01);
}
```

指定するデータセクションの型 (indexed, tagged, nested) に対応して，以下のようにいくつかのコンストラクタがあります．

```
// indexed セクション "adc" の アドレス 1 のデータ
DataElement adc_01("adc", 1);

// taggged セクション "monitor" のタグ "high_gain" のデータ
DataElement monitor_high("monitor", "high_gain");

// nested セクション "pmt" 中の indexed セクション "adc" の アドレス 1 のデータ
DataElement pmtadc_01("pmt:adc", 1);

// セクション "adc" の全てのデータ
DataElement adc("adc");
```

ビューオブジェクト

ヒストグラムなど，ビューシステムにおいてデータをある側面から要約し，画面への表示やファイルへの保存などをできるようにしたオブジェクトをビューオブジェクトと呼びます(ただし，画面への表示のみに着目してデータと直接関係しない特殊なビューオブジェクトもあります)．以下は，現在 Kinoko で利用できるビューオブジェクトです．

ヒストグラム (Histogram)
: 普通の 1 変数のヒストグラム．

2 元ヒストグラム (Histogram2d)
: 普通の 2 変数ヒストグラム．Scatter/Color/Box などの形で平面上に描画される．

トレンド (Trend)
: 値の時間変化に着目したもの．データのタイムスタンプ（時刻情報）をもとに，一定時間間隔ごとのデータ数・和・平均・分散など計算し，表示する．トリガレートなどの統計量や，HV 値や温度などの連続変化量など．

波形 (Wave)
: FADC データなど，一つのチャンネル(アドレス)の 1 イベント分のデータが複数のデータエレメントで構成される場合の，インデクスに対するデータ値の形．FADC なら波形になる．

マップ (Map)
: 一つのイベントが複数チャンネルのデータから構成される場合の，各チャンネルのデータの値．指定した位置に，データ値に対応した色の円が描画される．例えば，これを電子回路の各チャンネルの位置に対応するように並べれば，各チャンネルのアクティビティを表示できる．

表 (Tabular)
: データのテキスト表示．tagged 形式のデータなど，データ要素に名前が付いていれば，「名前：値」形式で表示される．

絵 (Picture)
: データ表示にはかかわらない．固定した文字や図形，イメージなどを表示するためのもの．

これらのビューオブジェクトにデータを読ませるには，アナリシスシーケンスを構築し，そこにデータ読みこみアクションを追加します．

上記のビューオブジェクトには，このために以下の 2 つのアナリシスシーケンスアクション生成メソッドがあります．

void fill(DataElement data_element)
　　指定されたデータエレメントに対応するデータを読む．
void fillOne(DataElement data_element)
　　ビューオブジェクトがデータをホールドしていなければ，指定されたデータエレメントに対応するデータを 1 イベント分だけ読む．ホールドしているデータは，ビューシーケンスの clear() アクションで開放される．

ビューオブジェクトにより，上記のうち一方のみが意味をもつ場合があることに注意してください．例えば，トレンドビューは，データの経時変化に注目しているので，fillOne() は通常意味を持ちません．逆に，表(Tabular)で fill() を行なうと，clear() するまでのデータが全て保持されてしまい，メモリを大量に消費するとともに，表示の際に行があふれてしまいます．

アナリシスシーケンスでデータを読ませたビューオブジェクトは，ビューシーケンスで画面に表示したりファイルに保存したりできます．また，保持しているデータを開放して，次の fillOne() でデータを読めるようにするための clear() もビューシーケンスのアクションです．以下は，上記のビューオブジェクトが共通で持っているビューシーケンスアクション生成メソッドです．

void draw()
　　保持しているデータを画面に描画する．
void clear()
　　保持しているデータを消去する．
void save(ViewRepository repository)
　　保持しているデータをリポジトリ(ファイルなど)に保存する．

ほとんどのビューオブジェクトでは，上記のシーケンスアクション生成メソッドに加えて，以下の通常のメソッドが利用できます．

void setAxisTitle(string x_title, string y_title)
　　軸のタイトルを設定する
void setYScaleLog() / void setYScaleLinear()
　　縦軸をログスケールやリニアスケールにする
void setDisplayStatistics(string stat_name, ...)
　　表示する統計値を指定する
void setColor() / void setFont() / void setTextAdjustment()
　　図形要素を描く際のプロパティを設定する
void putLine(float x0, float y0, float x1, float y1)
void putRectangle(float x0, float y0, float x1, float y1)
void putCircle(float x, float y, float radius)
void putText(float x, float y, string text)
　　ビューオブジェクトの表示領域に線分や文字などの図形要素を描く．座標系はビューオブジェクトの座標(スクリーン座標ではない)．円は，座標系の縦横比に応じて，通常は楕円になる．
void putImage(float x, float y, string file_name)
　　ファイルからイメージを読み，ビューの表示領域に描画する．座標系はビューオブジェクトの座標(スクリーン座標ではない)．現在のところ表示できるフォーマットは XPM のみ．

これらは通常のメソッドのため，スクリプト実行時(システム構築時)に一度しか実行されないことに注意してください．

以下に，各ビューオブジェクトごとの詳細を示します．

ヒストグラム (Histogram)

```
Histogram histogram("ADC ch 0", 256, 0, 4096);

analysis {
    DataElement adc00("adc", 0);
    histogram.fill(adc00);
}

on every(1sec) {
    histogram.draw();
}
```

Histogram(string title, int nbins, float min, float max)
　　コンストラクタ．引数でビンの取り方を指定する．
void setReferenceHistogram(ViewRepository repository, string name, float tolerance_sigmas=3)
　　参照ヒストグラムを設定する．参照ヒストグラムは面積をノーマライズした上でビンごとに tolerance_sigmas に指定された幅の灰色バンドで重ね書きされる．

ヒストグラムビューでは，fill() と fillOne() の両方を使うことができます．複数チャンネルを持ったデータエレメントで fillOne() を使うと，イベントごとのデータ値の分布図を作ることができます(全チャンネルのイベントごとのヒット時間分布など)．

2 元ヒストグラム (Histogram2d)

```
Histogram2d histogram("TDC v.s. ADC", 256, 0, 4096, 256, 0, 4096);

analysis {
    DataElement adc00("adc", 0);
    DataElement tdc00("tdc", 0);
    histogram.fill(adc00, tdc00);
}

on every(1 sec) {
    histogram.draw();
}
```

Histogram2d(string title, int x_nbins, float x_min, float x_max, int y_nbins, float y_min, float y_max)
　　コンストラクタ．引数でビンの取り方を指定する．
void fill(DataElement data_element_x, DataElement data_element_y)
void fillOne(DataElement data_element_x, DataElement data_element_y)
　　データを読みこむアナリシスシーケンスアクション生成メソッド．Histogram2d の fill() のみ引数に DataElement を 2 つとる．
void draw(string type = "color")
　　引数 type で描画の形式を指定できる．現在有効な type は以下のとおり．

- color: 色を使って値を表現する．
- scatter: 各ビンで，値に比例した数の点を描く．
- box: 値に比例した面積の長方形を描く．

2 元ヒストグラムビューでは，ヒストグラムビューと同様に，fill() と fillOne() の両方を使うことができます．

トレンド (Trend)

```
Trend trend("Trigger Rate", 1024, 0, 100);

analysis {
    DataElement adc00("adc", 0);
    trend.fill(adc00);
}

on run_begin {
    trend.setOperationRange(10, 30);
    trend.enableAlarm("trigger rate out of range")
}

on every(1 sec) {
    trend.drawCounts();
}
```

Trend(string title, float min, float max, int depth, int tick_width = 1)
　　コンストラクタ．min, max は縦軸の範囲，depth は表示する時間幅(秒)，tick_width はデータ数や平均を計算する時間幅(秒)
void draw()
void drawCounts() / void drawSum() / void drawMean() / drawDeviation()
　　時間ビンごとに統計値を計算し，描画する．
void setTimeTick(string format, string unit="", int step=0);
　　時間軸を時刻表示とし，その表示形式を設定する．format は UNIX の date(1) コマンドと同形式，unit と step で間隔を指定する．unit は "sec", "min", "hour" など．
　　例えば，setTimeTick("%H:%M", "min", 10) とすると，10分間隔で hh:mm 形式のラベルが付けられる．

void setOperationRange(float lower_bound, float upper_bound, int number_of_points_to_judge=2)
値の「正常値」の範囲を設定する. この範囲はヒストリプロットに描画され, また, enableAlarm() が設定されている場合, 値が正常値から連続してはずれた場合, アラームを発行する.

アラームイベントは, 最終点を除くデータ点が number_of_points_to_judge 個連続して範囲内に入っている状態から number_of_points_to_judge 個連続して範囲外に出たときに発行される. したがって, 正常値外で推移している間に繰り返し発行されることはないし, 少数点の飛び出しや復帰でアラームが発行されることもない.

void enableAlarm(string message)
データ値が setOperationRange() で設定した「正常値」外の値になったときに, アラームイベントを発行するようにする. KCOM フレームワークでは, これは "alarm(string message)" の形の KCOM イベントとなる. この message には, enableAlarm() の引数の message がそのまま渡される.

void disableAlarm()
アラームイベントの発行を抑止する.

setTimeTick() で時刻表示を設定しなかった場合, 横軸の時間表示は開始からの経過秒数となります.

setOperationRange(), enableAlarm(), disableAlarm() はビューシーケンスアクションです. ビューシーケンス中で, 範囲を変更したり, アラームの設定・解除を行なうことができます.

ヒストリビューでは, fillOne() は使用できません.

波形 (Wave)

```
Wave wave_latched("FADC Waveform (Latched)", 0, 4096, 0, 1024);
Wave wave_averaged("FADC Waveform (Averaged)", 0, 4096, 0, 1024);

analysis {
    DataElement fadc00("fadc", 0);
    wave_latched.fillOne(fadc00);
    wave_averaged.fill(fadc00);
}

on every(3 sec) {
    wave_latched.draw();
    wave_averaged.draw();
    wave_latched.clear();
}

on every (10 sec) {
    wave_averaged.clear();
}
```

Wave(string title, float x0, float x1, float y0, float y1)
コンストラクタ. x0, x1 は横軸の範囲で, データインデクス値. y0, y1 は縦軸で, データ値. 10bit 4ksample の FADC のデータを全て表示するなら, (x0, x1, y0, y1) = (0, 4096, 0, 1024) となる.

ウェーブビューでは, fill() と fillOne() の両方を使うことができます. clear() せずに新しいデータを fill() すると, 保持されるデータは各サンプルごとの平均値となります. すなわち, fillOne() を使うと一つの波形を表示し, fill() を使うと平均化された波形を表示することになります.

マップ (Map)

```
Map map("ADC hit map", 0, 1, 0, 16, 0, 4096);

for (int ch = 0; ch < 16; ch++) {
    map.addPoint(ch, 0.5, ch + 0.5);
}
map.setPointSize(3);

analysis {
    DataElement adc("adc");
    map.fillOne(adc);
}
```

```
on every(1 sec) {
    map.draw();
    map.clear();
}
```

Map(string title, float x0, float x1, float y0, float y1, float z0, float z1)
　　コンストラクタ. x0, x1, y0, y1 は描画点を指定するための座標系, z0, z1 はデータ値の表示範囲.
void addPoint(int address, float x, float y)
　　データアドレス address のデータを位置 (x, y) に表示するように指定する.
void setPointSize(float point_radius)
　　データを表示する点を描画する大きさを指定する.

マップビューでは, fill() は使用できません.

表 (Tabular)

```
Tabular tabular("Laser Intensity Monitor");

analysis {
    DataElement monitor("monitor_adc");
    tabular.fillOne(monitor);
}

on every(1 sec) {
    tabular.draw();
    tabular.clear();
}
```

Tabular(string title, int number_of_columns = 1)
　　コンストラクタ. 指定したカラム数で領域を分割する.

タブラビューでは, fill() は使用できません.

絵 (Picture)

```
Picture picture;

picture.putImage(0, 0, "KinokoLogo.xpm");

on clear {
    picture.draw();
}
```

Picture()
Picture(string title)
Picture(string title, float x0, float x1, flaot y0, float y1)
　　コンストラクタ. 引数で座標系を指定する. 省略した場合は, (0, 1, 0, 1) となる.

ピクチャビューは, 表示領域だけを持った特殊なビューオブジェクトで, データをフィルすることはできません. putImage() などで絵や文字を表示することを意図しています. ファイルのイメージを描画する場合, draw() は時間がかかることが多いので, draw() アクションは on every() シーケンスではなく on clear() シーケンスに入れるようにしてください.

レイアウトオブジェクト

レイアウトオブジェクトは内部に他のビューオブジェクトを格納できる特殊なビューオブジェクトです. それ自身は何も表示しませんが, 保持しているビューオブジェクトを自分の領域内に適当に配置して表示されるようにします. 配置の仕方により, いくつかの種類のレイアウトオブジェクトがあります.

グリッド (Grid)

グリッドは，領域を等間隔の表に区切り，そこにビューを並べていきます．ビューは，グリッドに追加された順で，まず左から右に，次に上から下に，配置されていきます．

Grid()
　　コンストラクタ．表の列数・行数はグリッドオブジェクトによって適当に決定される．
Grid(int number_of_columns)
　　コンストラクタ．引数で表の列数を指定する．行数はグリッドオブジェクトによって適当に決定される．
void put(View view)
　　グリッドにビューを追加する．

プレーサ (Placer)

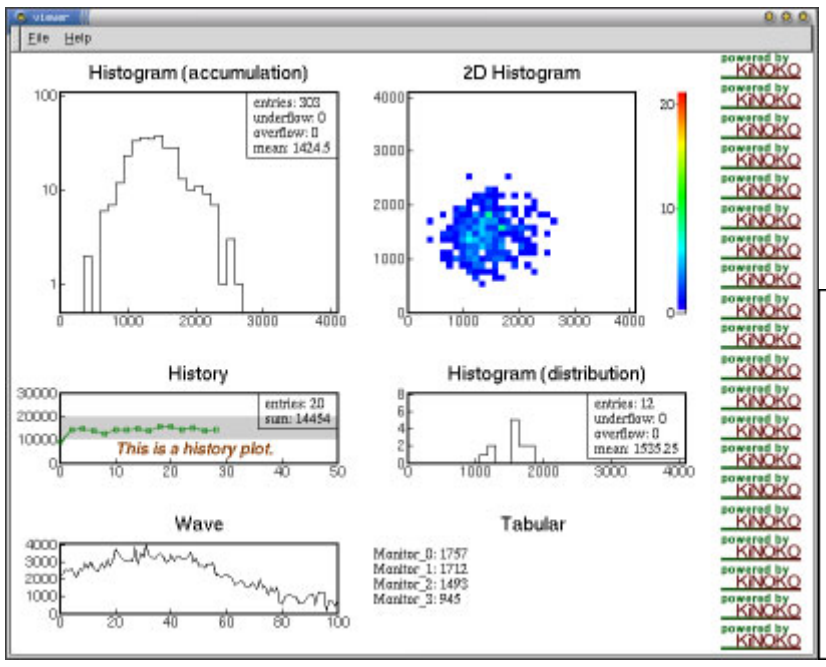
プレーサは，ビューオブジェクトをユーザの指定した位置と大きさで配置します．位置指定に用いる座標系は，横向きが x で右が正，縦向きが y で下が正となります．通常のビューオブジェクトの座標系とは上下が逆になっているので注意してください．

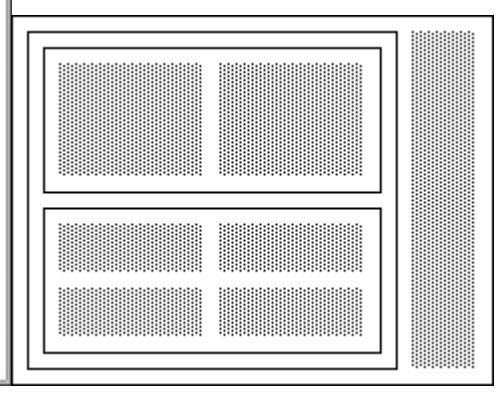
Placer()
　　コンストラクタ．座標系は左上が (0, 0) で右下が (1, 1) となる．
Placer(double x0, double x1, double y0, double y1)
　　コンストラクタ．座標系は左上が (x0, y0) で右下が (x1, y1) となる．
void put(View view, double x0, double x1, double y0, double y1)
　　プレーサ内の指定した位置にビューを配置する．

レイアウトオブジェクト自体もビューオブジェクトなので，レイアウトオブジェクトの中にレイアウトオブジェクトを配置することができます．グリッドとプレーサを組み合わせて使うことにより，複雑なレイアウトを比較的単純に作ることができます．

キャンバスは，ルートグリッドとよばれるグリッドをひとつ持っていて，どのレイアウトオブジェクトにも配置されていないビューオブジェクトを自動的にこのグリッドに配置します．レイアウトオブジェクトを 1 つ作って全てのビューオブジェクトをこのレイアウトオブジェクトに配置した場合でも，このレイアウトオブジェクト自体はルートグリッドに配置されることになります．

以下はレイアウトオブジェクトを組み合わせて使った例です．ルートグリッドにプレーサを置き，その中にカラム数 1 のグリッドを入れ，さらにその中に 2 つのグリッドを入れています．右端のイメージはプレーサで直接座標を指定しました．





```
Placer placer(0, 1.2, 0, 1);
Grid grid(1);
Grid grid_upper(2);
Grid grid_lower(2);

placer.put(grid, 0, 1, 0, 1);
grid.put(grid_upper);
grid.put(grid_lower);

grid_upper.put(histogram_adc00);
grid_upper.put(histogram_2d);

grid_lower.put(history_adc01);
grid_lower.put(histogram_adc);
```

```
    grid_lower.put(wave_fadc00);
    grid_lower.put(tabular_adc);

    placer.put(picture, 1, 1.2, 0, 1);
```

ビューリポジトリ

ビューリポジトリを使うと，描画したヒストグラムやグラフなどをファイルに保存することができます．保存するファイルフォーマットにより，以下のビューリポジトリがあります．

PlainTextRepository
: 数値をそのままテキストで書き出した型式です．リポジトリごとにディレクトリが作成され，さらにその中に一度の保存で一つのファイルが作成されます．基本的に空白区切りで数字を書き出した型式ですが，パラメータなどはファイルの先頭で「# 名前： 値」の形で記述されます．

XmlRepository（未完成）
: XML 型式で保存します．XML とは，タグにより構造化したテキストの型式です．全てのビューオブジェクトが一つのファイルに保存されます．

RootRepository
: ROOT 型式で保存します．全てのビューオブジェクトが一つのファイルに保存されます．
RootRepository を使用するためには，Kinoko が --with-root オプション付きでコンパイルされている必要があります．

以下は，データ収集中 1 時間ごとと，データ収集の終了時に，ヒストグラムを ROOT 型式で保存する例です．

```
display CamacAdc {
    Histogram histogram_adc_01("ADC ch 01", 128, 0, 4096);

    analysis {
        DataElement adc_01("adc", 1);
        histogram_adc_01.fill(adc_01);
    }

    on every (1 sec) {
        histogram_adc_01.draw();
    }

    // ROOT 型式のビューリポジトリの作成 //
    RootViewRepository repository("histogram_adc01.root");

    // 1 時間ごとにヒストグラムを保存 //
    on every (3600 sec) {
        histogram_adc_01.save(repository);
    }

    // データ収集終了時にヒストグラムを保存 //
    on run_end {
        histogram_adc_01.save(repository);
    }
}
```

1 つのビューオブジェクトを同じリポジトリに何度も保存することができます． 1 つのビューオブジェクトが同じリポジトリに 2 回以上保存されると，最後のもの以外にはリビジョン番号が付加されます．上記の例では，histogram_adc_01;0 histogram_adc_01;1, ... という名前で保存されることになります（リビジョン番号の付け方はリポジトリの種類により変わることがあります）．

ビューリポジトリは，大量のデータを高速に保存するのには適していません．データはあくまでデータファイル(データストレージ)に保存し，ビューリポジトリには，興味深いイベントが起きた場合や適当な間隔ごとに絵を保存するなどの目的で使用してください．

外部イベントインターフェース

[TinyKinoko, SmallKinoko ではこの機能は利用できません]

KCOM イベントなどの外部イベントに応じてビューシーケンスを起動するには，on command 文を使用します．on command 文は引数にイベント名をとります．

```
on command("initialize") {
    histogram.clear();
}
```

また，アナリシスシーケンス中で invoke 文を使うことにより，KCOM イベントなどの外部イベントを発行することができます．

```
analysis {
    DataElement trigger_rate("scaler", 0);
    when (trigger_rage > 1000) {
        invoke tooHighTriggerRate();
    }
}
```

外部レジストリインターフェース

ビュースクリプトから KCOM レジストリなどの外部レジストリにアクセスするためのインターフェースとして，getRegistry()/setRegistry() 関数があります．これは通常の関数で，スクリプト実行時にレジストリにアクセスします．

```
double nbins = getRegistry("control/histogram_nbins");
double min = getRegistry("control/histogram_min");
double max = getRegistry("control/histogram_max");

Histogram histogram("histogram", nbins, min, max);
```

シーケンス実行時にレジストリにアクセスする機能はいまのところ実装されていません．

コントロールパネルスクリプト

コントロールパネルスクリプト(KCML スクリプト)は，ユーザのコントロールをシステムに伝達するためのインターフェースであるコントロールパネルの構成を記述するためのスクリプトです．コントロールパネルには，ウィジェットとよばれる部品を配置することができます．ウィジェットには，入力フィールドやチェックボタンなどの入力ウィジェット，ボタンなどのアクションウィジェット，そしてフレームやスペーサなどのレイアウトウィジェットなどがあります．

入力ウィジェットには名前が付けられ，その名前によってウィジェットの値がシステムの他の部分から参照できるようになります．また，アクションウィジェットは，ユーザの操作によって，システムへイベントを発行したり，コントロールパネルスクリプト中に記述されたアクションを実行したりすることができます．

コントロールパネルスクリプトは，Kinoko の他のスクリプトと異なり，XML を用います．その構造は HTML のフォーム + JavaScript に似ています．<Script> タグを用いると，その中に Kinoko の基本スクリプトを埋め込むことができます．

簡単な例

まずはデータ収集システムとは関係のない簡単なアプリケーションを作成して，コントロールパネルスクリプトの概要を見てみます．左側の行番号は説明のためのものなので，入力しないでください．

```
1: <?xml version="1.0"?>
2:
3: <KinokoControlPanel label="The Graphical Unix Shell">
4:
5:     <Label label="command:"/>
6:     <Entry name="command"/>
7:     <Button label="execute" on_click="execute_command"/>
```

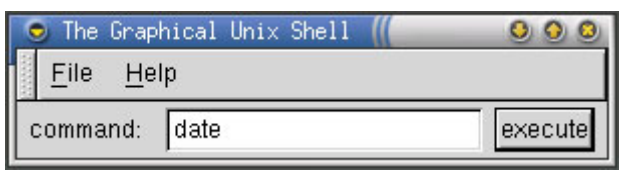
```
 8:
 9:    <Script>
10:    <![CDATA[
11:      void execute_command() {
12:          string command = <command>.getValue();
13:          println("> " + command);
14:          system(command);
15:      }
16:    ]]>
17:    </Script>
18:
19: </KinokoControlPanel>
```

1行目は，このファイルが XML 形式であることを宣言するための XML の文法です．KCML スクリプトは全体を <KinokoControlPanel> で囲みます（3行目と19行目）．5行目から7行目はウィジェット，<Script> タグで囲まれた9行目から17行目は標準 Kinoko スクリプトの埋め込みです．10行目と16行目はその中身の Kinoko スクリプトが XML 文書として解釈されないようにするための XML の文法です．12 行目では特殊な文法が使われていますが，このように <> で囲むことにより，埋め込みスクリプトからウィジェットを参照することができます．

このスクリプトは，標準ユーティリティ kcmlcheck で実行することができます．

```
% kcmlcheck GraphicalUnixShell.kcml
```

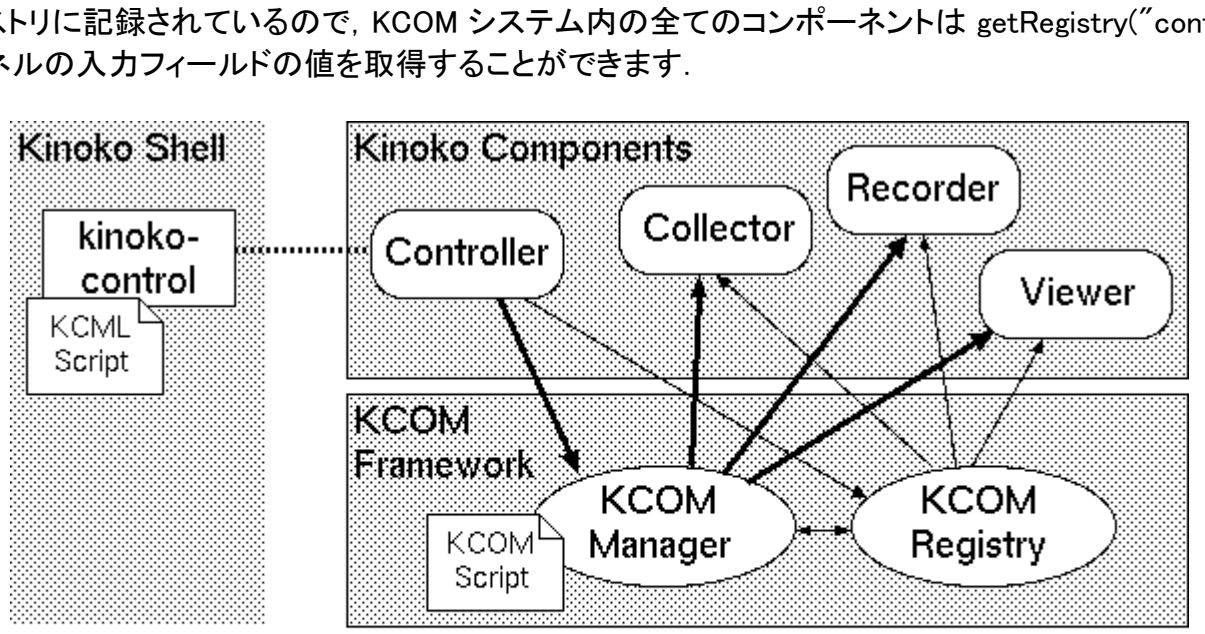
実行すると，以下のようなウィンドウが表示されます．入力フィールドに任意の UNIX コマンドを書いて，execute ボタンを押すと，それが実行されます．終了するには，メニューから File > eXit を選んでください．



KCOM システムインターフェース

KCOM システムへコマンドを送るコンポーネントである KinokoController コンポーネントは，シェルとして通常 kinoko-control を接続します．kinoko-control は，コントロールパネルスクリプト（KCML スクリプト）を読み，スクリーンにコントロールパネルを描画し，ユーザの入力を待ちます．

ローカルアクション（埋め込みスクリプト関数の呼び出し）に結びつけられていないボタンがクリックされると，kinoko-control は，そのときの入力ウィジェットの値全てとクリックされたボタンの名前（name 属性の値）を KinokoController へ通知します．KinokoController は，入力ウィジェットの値をレジストリ（/control 以下）に記録し，ボタンの名前をそのまま KCOM のイベント名として KCOM イベントを発行します．このイベントにより KCOM スクリプト（後述）の on controller.XXX() が呼び出されます．入力フィールドの値が全てレジストリに記録されているので，KCOM システム内の全てのコンポーネントは getRegistry("control/*field_name*") によりコントロールパネルの入力フィールドの値を取得することができます．



以下は，SmallKinoko で使われている KCML スクリプトと KCOM スクリプトからの抜粋です．ここでは，コントロールパネルの [construct] ボタンが押されたら，KCOM の on controller.construct() が呼ばれるようになっており，その中でコントロールパネルの入力フィールドに記述されている読み出しスクリプトやビュースクリプトのファイル名やデータファイル名をレジストリから取得するようになっています．

[KCML スクリプト]

```
<EntryList>
   <Entry name="readout_script" label="ReadoutScript (.kts)" option="file_select"/>
   <Entry name="view_script" label="ViewScript (.kvs)" option="file_select"/>
   <Entry name="data_file" label="DataFile (.kdf)" option="file_select"/>
</EntryList>

<VSpace/>

<Frame name="run_control" label="Run Control">
   <ButtonList>
      <Button name="construct" label="Construct" enabled_on="stream_ready system_ready"/>
      <Button name="start" label="Start" enabled_on="system_ready"/>
      <Button name="stop" label="Stop" enabled_on="data_taking"/>
      <Button name="clear" label="Clear" enabled_on="system_ready data_taking"/>
      <Button name="quit" label="Quit" enabled_on="stream_ready system_ready error"/>
   </ButtonList>
</Frame>
```

[KCOM スクリプト]

```
on controller.construct()
{
   controller.changeState("constructing");

   (中略)

   string readout_script = getRegistry("control/readout_script");
   string view_script = getRegistry("control/view_script");
   string data_file = getRegistry("control/data_file");

   (中略)

   collector.setReadoutScript(readout_script);
   viewer.setViewScript(view_script);
   recorder.setDataFile(data_file);

   (中略)

   controller.changeState("stream_ready");
}
```

コントロールパネルはステートをもっており，これは KCOM の changeState() で設定されます．KCML のボタンウィジェットで enabled_on 属性を指定することにより，そのボタンを操作できるステートを指定できます．この仕組みにより，システムの状況に応じて適正な操作のみをユーザが操作可能なようにすることができます．

入力ウィジェット

現在のところ，以下の入力ウィジェットがあります．これらの値は，ボタンが押されたときに，KinokoController コンポーネントを介してレジストリに記録さます．また，埋め込みスクリプトから getValue() や setValue() を使って値の参照や変更ができます．

エントリ (<Entry>)

テキスト入力フィールドです．以下の属性値を指定できます．

- name (必須): 値を取得するときにキーとなる名前
- label: エントリフィールドの横に描かれる文字列
- selection: 空白区切りで選択可能な値を並べる．これを指定するとプルダウンメニューになる
- width: 入力フィールドの表示幅
- alignment: 内部の文字列の配置．right または left
- on_focus: このエントリがフォーカスされたときに呼び出す埋め込みスクリプトの関数名
- on_blur: このエントリがフォーカスからはずれたときに呼び出す埋め込みスクリプトの関数名

チェックボタン (<CheckButton>)

項目選択のためのチェックボタンです．選択されている場合は値 1 を，されていない場合は値 0 をとります．以下の属性値を指定できます． テキスト入力フィールドです．以下の属性値を指定できます．

- name (必須): 値を取得するときにキーとなる名前
- label: チェックボタンの横に描かれる文字列

ラジオボタン (<RadioButton>)

項目選択のためのラジオボタンです．選択されている場合は値 1 を，されていない場合は値 0 をとります．通常，後述の RadioButtonList によりグループ化し，その中のひとつだけが選択可能なようにします． 以下の属性値を指定できます．

- name (必須): 値を取得するときにキーとなる名前
- label: ラジオボタンの横に描かれる文字列

定数 (<Constant>)

画面には何も表示されませんが，定数値を保持するウィジェットです．KCML スクリプトの中で値をハードコーディングしたい場合に用います． 以下の属性値を指定できます．

- name (必須): 値を取得するときにキーとなる名前
- value (必須): 値

表示ウィジェット

表示ウィジェットは，値の入力にはかかわりませんが，画面上に文字や絵を表示するウィジェットです．通常定数値ですが，埋め込みスクリプトの setValue() メソッドによりその表示内容を変えることもできます．

ラベル (<Label>)

文字列を表示します． 以下の属性値を指定できます．

- name: 埋め込みスクリプトなどから参照するための名前
- label: 表示する文字列
- font: 表示に使用するフォント
- fontsize: フォントサイズ

イメージ (<Image>)

ファイルから画像を読み表示します．今のところ XPM 形式のみ表示できます． 以下の属性値を指定できます．

- file (必須): 画像ファイル名

アクションウィジェット

アクションウィジェットは，クリックなどにより KinokoController コンポーネントを介して KCOM へイベントを送り出したり，埋め込みスクリプトの関数を呼び出したりするウィジェットです．on_click 属性が指定されていれば埋め込みスクリプトを実行し，なければ KinokoController へアクションを送り出します．

ボタン (<Button>)

ボタンです. クリックによりアクションを実行します. 以下の属性値を指定できます.

- name: クリックにより発生させるアクション名 (KCOM イベント名)
- label: ボタン上に表示する文字列
- image: ボタン上に表示する画像. 現在のところ XPM のみ.
- tip: ツールチップに表示させる文字列
- enabled_on: ボタンを使用可能にするステートのリスト. 空白区切りで指定する
- on_click: クリックにより実行する埋め込みスクリプトの関数名

レイアウトウィジェット

レイアウトウィジェットは, それ自体は画面に表示されないが, 他のウィジェットの並びなどを制御するウィジェットです. レイアウトウィジェットによる指定がない限り, ウィジェットは左から右へ左詰めで並べられていきます.

改行 (<NewLine>)

次のウィジェットが一段下の左端に置かれるようにします

スペーサ (<HSpace> / <VSpace>)

HSpace は横方向, VSpace は縦方向のスペースを挿入します. HSpace は余ったスペースを埋めるように横へ伸びるので, HSpace の右に置かれたウィジェットは右詰めに, HSPace で挟まれたウィジェットは中央揃えで配置されるようになります.

ボックス (<Box>)

複数のウィジェットをまとめてひとつのウィジェットとしてレイアウトされるようにします. スペーサと併用すると便利です.

フレーム (<Frame>)

複数のウィジェットを線で囲みラベルを付けます. 以下の属性値を指定できます.

- label: 囲みにつけるラベル文字列

リスト (<ButtonList> / <RadioButtonList> / <CheckButtonList> / <EntryList>)

複数の同種ウィジェットを一列に整列させて表示します. ButtonList はボタンを横揃えで同じ大きさに並べます. その他は縦揃えで並べます.

テーブル (<Table> / <Cell>)

Table は Cell でまとめられたウィジェットを縦横に揃えた表に配置します.

Table では以下の属性値を指定できます.

- width: 横方向のセルの数
- height: 縦方向のセルの数. 足りない場合は自動延長される
- homogeneous: "true" を指定するとセルの幅を揃えて配置する

Cell では以下の属性値を指定できます.

- top: テーブルの中のセルの縦方向座標
- left: テーブルの中のセルの横方向座標
- width: テーブルの中で使う横方向のセル数
- height: テーブルの中で使う縦方向のセル数

ノートブック (<Notebook> / <Page>)

タブのついた複数のパネル(ページ)を作成します.

Notebook では, getValue() / setValue() により表示ページの取得や設定ができます. Notebook では以下の属性値を指定できます.

- name: 埋め込みスクリプトなどから参照するための名前
- tab: タブを表示させる位置:top, bottom, left, right, none

Page では以下の属性値を指定できます.

- label: タブに表示させる文字列

ツールバー (<Toolbar>)

ツールバーを作成します. ツールバーの中には Button と HSpace を配置できます.

メニュー

<Menu> タグを利用することにより, KCML コントロールパネルのメニューバーに新しいメニューを追加することができます. メニューの中には <MenuItem> でメニュー項目を並べます. MenuItem の振舞は Button と基本的に同等です.

Menu では以下の属性値を指定できます.

- label (必須): メニューに表示させる文字列

MenuItem では以下の属性値を指定できます.

- name: クリックにより発生させるアクション名(KCOM イベント名)
- label: ボタン上に表示する文字列
- enabled_on: メニュー項目を使用可能にするステートのリスト. 空白区切りで指定する
- on_click: クリックにより実行する埋め込みスクリプトの関数名

埋め込み Kinoko スクリプト

KCML の中で <Script> タグを使うことにより, Kinoko の基本スクリプトを記述することができます. この中の関数を on_click などにより呼び出すことができ, また, このスクリプト中からウィジェットの値を操作できます. KCML 埋め込みスクリプト (KCMS; Kinoko Control-panel Manipulation Script) では, 基本スクリプトに加え, 以下の文法要素が追加されています.

- ControlWidget クラス
- lookupWidget() 関数
- saveWidgetValues() / loadWidgetValues() 関数
- currentState() 関数
- <> 演算子
- on 文
- after 文
- invoke 文

ボタンウィジェットなどの on_click 属性, エントリウィジェットなどの on_focus および on_blur 属性に KCMS の関数名を指定しておけば, それらのウィジェットの対応する操作が行われたときに指定した関数が呼び出されます.

KCMS スクリプト中では, lookupWidget() 関数によりウィジェットをオブジェクトとして参照できます. lookupWidget() の引数には, ウィジェットの name 属性の値を指定します.

- ControlWidget lookupWidget(string widget_name)

lookupWidget() により取得したウィジェットオブジェクトに対して, 以下の操作ができます.

getValue()
: ウィジェットの値を参照する. 入力ウィジェットでは入力値, その他はウィジェットによって異なる

setValue()
: ウィジェットに値を設定する.

enable()
: ウィジェットを操作可能に設定する

disable()
　　ウィジェットを操作不可能に設定する
getAttribute()
　　ウィジェットの属性値を取得する. ただし, 取得できる属性値はウィジェットにより異なる
setAttribute()
　　ウィジェットの属性値を設定する. ただし, 設定できる属性値はウィジェットにより異なる

<> 演算子は lookupWidget() 関数のショートカットです. lookupWidget(name) と <name> は厳密に同等です.

saveWidgetValue() / loadWidgetValue() 関数により, 現在のウィジェット値をファイルに保存したりファイルから読み出して各ウィジェットに設定したりできます.

- void saveWidgetValues(string file_name);
- void loadWidgetValues(string file_name);

ファイル名は任意ですが, 普通は拡張子に .kcv (Kinoko Control Values) を用います. ファイル形式は XML です.

invoke 文により, KCMS スクリプトから KinokoController を経由して KCOM イベントを発行することができます.

- invoke イベント;

```
void start() {
    println("start at: " + localTime();
    invoke start();
}
```

on 文により, 一定のタイミングで文を実行したり, ステートの変化や KCOM イベントの発行をトラップできます.

on startup { 文リスト }
on shutdown { 文リスト }
　　開始時および終了時に呼び出される
on every (TIME) { 文リスト }
　　TIME に指定した時間間隔で呼び出される. TIME の形式は「数値 単位」で, 単位には sec min hour day のいずれかを指定する
on transition(from FROM to TO) { 文リスト }
on transition(from FROM) { 文リスト }
on transition(to TO) { 文リスト }
on transition() { 文リスト }
　　指定したステートの変化で呼び出される. from や to は省略でき, その場合は全てのステートが指定されたのと同等になる
on invocation(NAME) { 文リスト }
on invocation() { 文リスト }
　　指定の KCOM イベントが発行されたときに呼び出される

after 文により, 指定の時間の遅延の後で関数または KCOM イベントの発行を行うように指定できます. 構文は以下のとおりです.

- after (TIME) call 関数;
- after (TIME) invoke イベント;

TIME の時間指定形式は on 文と同様です.

```
on transition(to "running") {
    after (10 min) call stop();
}
```

currentState() 関数により, 現在のステートを取得できます.

- string currentState()

ビジュアルウィジェットとビューレット

ビジュアルウィジェットは，グラフィクスを使って値をより視覚的に表示するもので，Label などにより数値のみを表示するよりもかなり表現力があります．ビューレットはビューアウィンドウ中で描画されていたプロットなどをコントロールパネル上に表示するウィジェットです．

現在のところ，ビジュアルウィジェットもビューレットも， Web-KiNOKO によるブラウザ上での表示には対応していません．

(<VisualDisplay>)

(<VisualLight>)

(<VisualIndicator>)

(<VisualGuage>)

(<VisualMeter>)

(<VisualButton>)

(<VisualLightButton>)

(<VisualSlider>)

(<Canvas>)

(<Plot>)

コンポーネント配置結合スクリプト

[TinyKinoko, SmallKinoko のみを使う場合はこの章は読み飛ばして構いません]

コンポーネント配置結合スクリプト(KCOM スクリプト)は，システムを構成する各コンポーネントの計算機への配置や，コンポーネントイベント/スロットの結合，共有オブジェクトの割り付けなどを記述するために，Kinoko のシステムプロセス(kcom-manager)によって使用されるスクリプトです．KCOM スクリプトでは，基本スクリプトに加えて，以下の文法要素が追加されています．

- import 文
- component 文
- assign 文
- asynchronous 文
- after 文
- exit 文
- on エントリ
- setTimeout() / enableTimeout() / disableTimeout() 関数
- setRegistry() / getRegistry() 関数

KCOM スクリプトは読み出しスクリプトやビューオブジェクトとは異なり，普通のインタプリタとして動作します(シーケンスは使用しません)．すなわち，全ての変数や式は，実行時に毎回値が評価されるということです．

KCOM スクリプトにより，以下のようなことができます．

- コンポーネントを指定した計算機上で実行させる．
- コンポーネントのプライベート入出力チャネルを端末ウィンドウやソケットに結合させる．
- コンポーネント内部の公開オブジェクトを別のコンポーネントと共有させる．
- コンポーネントに対しイベントを発行する(イベントスロットを実行させる)．
- コンポーネントが発行するイベントを取得し，それに対して処理を行なう．
- コンポーネントのプロパティ値を取得する．
- レジストリの値を取得・変更する．

もちろん，Kinoko の基本スクリプトが持つ機能も全て利用できます(データベースアクセスなど)．また，コンポーネントの入出力チャネルに接続させる KinokoShell のプロセス(キャンバスやコントロールパネルなど)の実行も通常 KCOM スクリプト中から行ないます．

KCOM スクリプトの大まかな構造は以下のようになります.

```
// コンポーネント定義の取りこみ
import コンポーネント型名;

// コンポーネントの宣言と配置
component コンポーネント型名 コンポーネント名("ホスト名","入出力チャネル");

// 共有オブジェクトの割り付け
assign コンポーネント名.オブジェクト名 => コンポーネント名.オブジェクト名;

on startup()
{
    // 開始処理
}

on shutdown()
{
    // 終了処理
}

// イベント/スロットの結合
on コンポーネント名.イベント名(イベント引数)
{
    コンポーネント名.イベント名(イベント引数);
}
```

コンポーネントのインターフェース宣言

各コンポーネントのインターフェース定義は, コンポーネントのプログラムを単体で引数なしで実行することにより見ることができます. 以下は, Kinoko のデータ読み出しコンポーネントである KinokoCollector のインターフェース定義を表示させたものです.

```
% KinokoCollector-kcom
//
// KinokoCollectorCom.kidl
//
// Kinoko Data-Stream Component
// Collector: DAQ front-end process
//
// Author: Enomoto Sanshiro
// Date: 22 January 2002
// Version: 1.0

component KinokoCollectorCom {
    property string host;
    property string stream_type;
    property string state;
    property int port_number;
    uses KinokoLogger logger;
    emits dataAcquisitionFinished();
    accepts setSource();
    accepts setSink();
    accepts setSourceSink();
    accepts connect();
    accepts construct();
    accepts destruct();
    accepts disconnect();
    accepts halt();
    accepts quit();
    accepts start();
```

```
    accepts stop();
    accepts setReadoutScript(string file_name, string datasource_name);
    accepts setMaxEventCounts(int number_of_events);
    accepts setRunLength(int run_length_sec);
    accepts executeCommand(string command_name, int parameter);
    accepts disable();
}

%
```

ここで，emits はこのコンポーネントが発行するイベントを，accepts はこのコンポーネントが受け取ることのできるイベントを，それぞれ宣言しています．また，property は外部から参照できるパラメータ(プロパティ)を宣言しています．uses はこのコンポーネントが必要とする外部オブジェクトで，システムを構成するコンポーネントのいずれかにより提供されなければなりません．この例では KinokoLogger というクラスのオブジェクトを必要としています．

Kinoko の標準コンポーネントのうち，KinokoLogger クラスのオブジェクトを提供するのは KinokoLogger コンポーネントです．以下は KinokoLogger コンポーネントのインターフェース定義を表示させたもので，この中で provides によって，KinokoLogger クラスのオブジェクト logger を，他のコンポーネントから参照できる公開オブジェクトと宣言しています．

```
% KinokoLogger-kcom
//
// KinokoLoggerCom.kidl
//
// Kinoko System Component
// Logger: Logbook Writer
//
// Author: Enomoto Sanshiro
// Date: 22 January 2002
// Version: 1.0

component KinokoLoggerCom {
    property string host;
    provides KinokoLogger logger;
    emits processRemarkable(string message);
    emits processWarning(string message);
    emits processError(string message);
    emits processPanic(string message);
    accepts quit();
    accepts startLogging(string file_name);
    accepts writeDebug(string message);
    accepts writeNotice(string message);
    accepts writeRemarkable(string message);
    accepts writeWarning(string message);
    accepts writeError(string message);
    accepts writePanic(string message);
}

%
```

KCOM スクリプトの先頭では，import 文によってこのコンポーネント宣言を取り込み，KCOM スクリプト内でコンポーネントとして使用できるようにします（が今のところ何もしていません）．

上記の Collector および Logger コンポーネントを使用するためには，以下のように記述します．

```
import KinokoCollector;
import KinokoLogger;
```

コンポーネントの宣言と配置

import により KCOM スクリプトからコンポーネントを使えるようにした後は，component 宣言によりそれらのコンポーネントを計算機上に実際に配置します．また，このコンポーネントを KinokoShell に接続する場合には，同時に KinokoShell の接続先も指定します．以下は，Collector と Logger をそれぞれ host_daq と host_hub に配置し，Logger コンポーネントのプライベート入出力チャネルを KinokoShell である kinoko-board にソケット経由で接続した例です．kinoko-board への接続を指定する前に，execute() 関数により kinoko-board プログラムを起動しておきます．

```
string host_daq = "daq.kinoko.ac.jp";
string host_hub = "hub.kinoko.ac.jp";
int port_log = 30000;

execute("kinoko-board", "--client", host_hub, port_log);

component KinokoCollector collector(host_daq);
component KinokoLogger logger(host_hub, "port: " + port_log);
```

オブジェクトの結合

次に，assign 文により，logger コンポーネントがエクスポートしている logger オブジェクトを Collector コンポーネントにインポートします．

```
assign logger.logger => collector.logger;
```

ただし，この例に限っては，collector コンポーネントが Logger 型のオブジェクトを必要としており，Logger 型オブジェクトを提供するコンポーネントが KCOM システム上にひとつしか存在していないので，上記の結合は自動で行われ，明示的に書く必要はありません．Logger 型オブジェクトをエクスポートするコンポーネントが複数存在する場合は，assign 文を使って明示的に指定する必要があります，

イベントとスロットの結合

コンポーネントが emit したイベントは，KCOM スクリプトの on 文で受け取ることができます．on 文には，特定のコンポーネントからの特定のイベントだけを受け取る on ComponentName.EventName(...) {} 形式と，発行コンポーネントによらずイベント名だけを指定する on .EventName(...) {} 形式があります． 以下は，Recorder 型の recorder コンポーネントが発行する finishRecording() イベントだけを受け取る例です．

```
on recorder.finishRecording() {
    // データの記録が完了したのでここで終了処理を開始する
}
```

以下は，processError(string message) イベントを，発行元にかかわらず受け取る例です．

```
on .processError(string message) {
    println("ERROR: ", message);
    // エラー処理をする
}
```

on 文の後ろの {} の中に，イベントを受け取ったときの処理を書きます．この中で他のコンポーネントのイベントスロットにイベントを送ることができるので，これによりコンポーネント間でイベントとスロットを間接的に接続することができます．以下は，controller コンポーネントの start() イベントを，collector コンポーネントの start() イベントに間接的に接続した例です．{} の中には複数の文をかけるので，同時に reporter コンポーネントにも write() イベントを送っています．

```
on controller.start() {
    collector.start();
    reporter.write("start_time", localTime());
}
```

ここで，collector コンポーネントは start() イベントを，reporter コンポーネントは write(string, string) イベントを，それぞれ accept できる必要があります．

プロパティの参照

コンポーネントのプロパティは，オブジェクトと同じ形式で，ドット演算子．を使って読み出すことができます．ただし，変更はできません．コンポーネントのプロパティ値を変更した場合は，イベントを経由して行うようにしてください．

```
on .stop() {
    string collector_state = collector.state;
    if (collector_state != "running") {
        println("The collector component is not running");
        return;
    }
```

レジストリの参照

レジストリはひとつの KCOM システム全体で共有されるパラメータの集合です．ツリー構造で管理され，パラメータ名はツリーにそって / で区切ったものになります（UNIX のファイル名と同じ構造）．このレジストリ値は読みだしスクリプト，ビュースクリプトおよびコントロールパネルスクリプトからもアクセスでき，設定値などを共有するために使われます．

KCOM スクリプトからレジストリ値にアクセスするには，getRegistry() / setRegistry() 関数を使うだけです．それぞれ，以下のような定義になっています．

```
string getRegistry(string name)
void setRegistry(string name, string value)
```

oneway 呼び出し，非同期実行とタイムアウト

KCOM スクリプトからコンポーネントにイベントを投げた場合，KCOM スクリプトの実行はコンポーネントがイベントの処理を終了するまで停止します（同期実行）．一部のコンポーネントの不具合により，システム全体が停止するのを防ぐために，コンポーネントからの返信待ちにタイムアウトを設定できます．

```
void setTimeout(int length_sec)
        コンポーネントへイベントを送ってからの返信待ち時間の上限を設定する
void enableTimeout()
        タイムアウト処理を行う
void disableTimeout()
        タイムアウト処理を行わない
```

もし設定した時間を経過してもコンポーネントからの返信がない場合，そのコンポーネントは正常な動作をしてないと判断し，システムから切り離されます（画面には component XXX dismissed と表示されます）．切り離されると，それ以降のそのコンポーネントへのアクセスは全てスキップされます．

ユーザからの入力を受け取るイベントなど，イベントの種類によっては返信までに大きく時間がかかるものもあります．そのようなものがタイムアウトで切り離されないようにするために，返信までタイムアウト以上の時間がかかるイベントを投げるときには，disableTimeout() により一時的にタイムアウト処理を止めるようにしてください．

```
disableTimeout();

// ポップアップウィンドウを開くので，返信まで時間がかかる //
string decision = controller.openQueryPopup(
        "Warning", "Yes No",
        "file exists: " + data_file + " -- overwrite?"
);

enableTimeout();
```

場合によっては，イベントごとに同期を取る必要がないこともあります．このようなときは，同期をとらなくて良い処理をまとめて実行することにより，システムの処理性能を大幅に向上させることができます．KCOM スクリプトでは，asynchronous 文により，非同期に同時実行できるイベントをまとめることができます．

```
asynchronous {
    collector.construct();
    analyzer.construct();
    recorder.construct();
    viewer.construct();
}
```

asynchronous ブロックに入ると，リプライを待たずに次々とイベントを発行します．最後に，全てのイベントの処理が終わるのを待って，asynchronous ブロックが終了します．

さらに，イベントの種類によっては，そもそもリプライを待つ必要が全くないものもあります．このようなときは oneway 文によりリプライ不要と宣言することによって，システムの効率を大きく向上させることができます．

```
oneway {
    reporter.write("run-name", run_name);
    reporter.write("comment", comment);
    reporter.write("date", localTime());
    reporter.write("log-file", log_file_name);
    reporter.write("buffer-size", buffer_size);
    reporter.write("data-file", data_file);
    reporter.write("script", readout_script, "target=collector", "type=kts");
    reporter.write("script", view_script, "target=viewer", "type=kvs");
}
```

この例では，他のコンポーネントは reporter の処理を待つ必要が全くないので，oneway によりリクエストだけ出しておいて先に進むようにしています．

実行タイミングの制御

KCOM の on 文では，KCOM イベント取得時での呼び出しに加えて，特定のタイミングで文を実行させるように指定することもできます．

```
on startup() {
    // ここは KCOM スタート直後に呼び出される
}

on shutdown() {
    // ここは KCOM 終了直前に呼び出される
}

on every(10 sec) {
    // ここは KCOM 動作中 10 秒ごとに呼び出される
}
```

on every() の括弧内には時間を単位付きで指定します．単位には，sec min hour day が使用できます．

また，after 文を使うことにより，指定した時間の経過後に指定の関数を呼び出すことができます．

```
on startup() {
    after (1 hour) call suggestFinish();
    println("Let's finish in one hour.");
}

void suggestFinish() {
    println("It's about time to finish.");
}
```

after () call ... の括弧内には時間を単位付きで指定します．単位には，sec min hour day が使用できます． 指定時間に達するまでの間は，通常の KCOM 処理が行われます．

システムの終了

KCOM システムは，exit 文に達することにより終了します．

```
on controller.quit() {
    exit;
}
```

SmallKinoko の構造と拡張方法

[TinyKinoko, SmallKinoko のみを使う場合はこの章は読み飛ばして構いません]

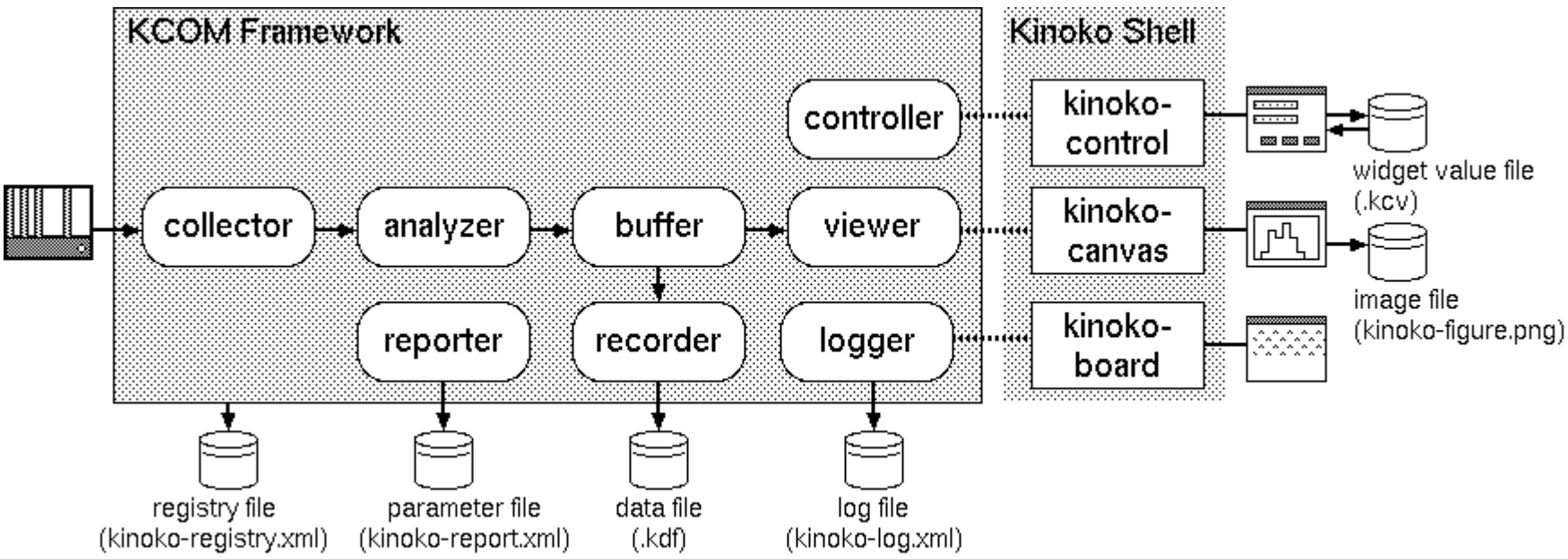
SmallKinoko は，ひとつの PC 上でひとつの Collector コンポーネントを使用する構成を仮定して書かれた KCOM と KCML のセットです．どちらも，kinoko/scripts にあり，ファイル名は SmallKinoko.kcom と SmallKinoko.kcml です．Kinoko と SmallKinoko の違いは自分で作った KCOM を使うかどうかだけで，SmallKinoko を起動することと，SmallKinoko.kcom を指定して Kinoko を起動することとは同等です．

```
% kinoko $KINOKO_ROOT/scripts/SmallKinoko.kcom
```

したがって，独自の分散並列システムを構築する場合でも，SmallKinoko.kcom をもとにすると作成が容易になります（現時点ではデータストリームの接続が手作業でかなり煩わしくなっていますが，このへんは将来改良予定です）．

SmallKinoko の構造

SmallKinoko には Collector，Recorder，Viewer がひとつづつあり，これらが Buffer を介して接続されています．さらに Collector と Buffer の間には何もしない Transporter コンポーネントが analyzer という名前で挿入されており，これをこれを独自のデータアナライザコンポーネントに置き換えることにより，容易に自分のコンポーネントを組み込めるようになっています（1行の変更）．また，Collector が Buffer に直接接続されていないことにより，Collector だけを別 PC 上で走らせるという変更も極めて容易です（これも1行の変更）．SmallKinoko には，これらデータストリームコンポーネントに加えて，ユーザ操作を KCOM に伝達する Controller，ログを記録する Logger およびパラメータを保存してランサマリを生成する Reporter の各コンポーネントがひとつづつ配置されています．



Recorder が生成するデータファイル（KDF ファイル）に加えて，SmallKinoko はいろいろなファイルを生成します．Logger，Reporter および KinokoCanvas は，それぞれログメッセージ，ランパラメータおよびビューアの絵をファイルに保存します．これらは後からランサマリを作成するときや，Web-KiNOKO などで使用されます．KCOM フレームワーク内のレジストリサーバは，レジストリの内容を定期的にファイルに書きだします．レジストリには，コンポーネントの動作状態などの情報が記録されているので，このファイルを読むことによりシステムの動作状況を把握することができます．この機能は Web-KiNOKO のシステムモニタで使用されており，将来 SmallKinoko にもこの機能を利用したシステムモニタが導入される予定です．コントロールパネルでは，ウィジェットの値をファイルに書き出したりファイルから読み込んだりすることができ，これは起動時のデフォルト値のロードや Web-KiNOKO で使われています．

SmallKinoko.kcom において，コンポーネント宣言は以下のようになっています．

```
import KinokoCollector;
import KinokoBuffer;
import KinokoRecorder;
import KinokoViewer;
import KinokoController;
import KinokoLogger;
import KinokoReporter;
import KinokoTransporter;

component KinokoController controller(host, ui_control);
component KinokoReporter reporter(host);
component KinokoCollector collector(daq_host);
component KinokoBuffer buffer(host);
component KinokoRecorder recorder(host);
component KinokoViewer viewer(host, ui_view);
component KinokoLogger logger(host, ui_log);
component KinokoTransporter analyzer(host);
```

ここで，host や ui_control, ui_view, ui_log などは環境変数や起動時引数などから決定される文字列です．デフォルトでは host と daq_host は同じ中身ですが，ここを変えることにより，Collector コンポーネントを他の PC に配置することができます．

Controller，Viewer および Logger に接続される KinokoShell のプロセスも，KCOM スクリプトの先頭付近で起動されます．

```
execute("kinoko-control --client " + host + " " + port_control);
execute("kinoko-board --client " + host + " " + port_log);
execute("kinoko-canvas --client " + host + " " + port_view);
```

on startup() では，ロガー，レポーターおよびコントロールパネルのスタート，データストリームの接続(connect() 関数の呼び出し)を行います．この際にエラーが検出されなければ，コントローラのステートを stream_ready に設定することにより，コントロールパネル上の Construct および Quit ボタンを操作できるようにします．

```
on startup()
{
    // KCOM タイムアウトの初期設定
    setTimeout(kcom_timeout_sec);

    // ロガーとレポーターのスタート
    logger.startLogging(log_dir + "/" + log_file_name);
    reporter.open(log_dir + "/" + report_file_name);

// KCMLを指定し，コントロールパネルをスタート
controller.open(kcml_script);
if (web_dir != "") {
    // Web-KiNOKO を使う場合…
        controller.loadScript(web_dir + "/kweb-export-kcv.kcms");
        controller.execute("setWebDirectory", web_dir);
}
if (kcms_script != "") {
        println("loading " + kcms_script + "…");
        controller.loadScript(kcms_script);
}

// コンポーネントの内部状態をレジストリに記録するように指定する
collector.enableConditionMonitor();
analyzer.enableConditionMonitor();
recorder.enableConditionMonitor();
viewer.enableConditionMonitor();
buffer.enableConditionMonitor();
```

```
    // コンポーネント単体の準備ができたのでデータストリームを接続
    controller.changeState("component_ready");
    connect();

    // エラーがなければユーザがパラメータを設定して Construst できるようにする
    if (error_count == 0) {
        controller.changeState("stream_ready");
    }
    else {
        controller.setWidgetValue("message", "error: click [quit] to shutdown");
        controller.changeState("error");
    }
}
```

Web-KiNOKO 用に，ビューアの絵とレジストリ値を5秒ごとに保存するように on every が使われています．

```
on every (5 sec) {
    saveRegistry(log_dir + "/" + registry_file_name);
    if ((web_dir != "") && (is_running)) {
        viewer.saveImage(web_dir + "/" + figure_file_name);
    }
}
```

データストリーム接続を確立する connect() 関数の中身は以下のようになっています．受動的に動作する Buffer をまずスタートし，次に全てのコンポーネントに接続相手を指定してから connect() イベントを送り出します．効率を良くするために，非同期で実行できるブロックが asynchronous 指定されています．

```
void connect()
{
    buffer.start();

    asynchronous {
        collector.setSink(analyzer);
        analyzer.setSourceSink(collector, buffer);
        recorder.setSource(buffer);
        viewer.setSource(buffer);
    }

    asynchronous {
        collector.connect();
        analyzer.connect();
        recorder.connect();
        viewer.connect();
    }
}
```

ここまでで，コンポーネントが配置され，データストリームが接続されて，ユーザのコマンド待ちの状態になりました．ユーザは コントロールパネル上にパラメータ（読みだしスクリプトやデータファイル名など）を設定し Construct をクリックするか，Quit をクリックすることができます．KCML スクリプトにより，Construct ボタンがクリックされると KCOM に construct イベントが通知され，Quit ボタンがクリックされると quit イベントが通知されるようになっています(Button ウィジェットの name 属性が指定されているだけ)．

KCOM が construct イベントを受け取ると，KCOM スクリプトの on .construct() が呼び出されます．ここでは，construct() 関数を呼び出してエラーチェックの後ステートを system_ready に設定します．すでに construct されている場合には，construct() 関数を呼び出す前に destruct() 関数を呼び出します．

```
on .construct()
{
    if (is_constructed) {
        destruct();
        is_constructed = false;
    }

    controller.changeState("constructing");
    controller.setWidgetValue("message", "constructing...");
```

```
    if (construct() && (error_count == 0)) {
        controller.changeState("system_ready");
        controller.setWidgetValue("message", "ready to start data acquisition");
        is_constructed = true;
    }
    else {
        controller.changeState("stream_ready");
        controller.setWidgetValue("message", "construction failed: try again.");
    }
}
```

以下は，construct() 関数の中身です．コントロールパネル上のパラメータをレジストリ経由で取得して，レポーターに記録し，コンポーネントに設定して，最後に各コンポーネントをコンストラクトしています．

```
int construct()
{
    // コントロールパネルのパラメータをレジストリから取得
    string run_name = getRegistry("control/run_name");
    string comment = getRegistry("control/comment");
    readout_script = getRegistry("control/readout_script");
    view_script = getRegistry("control/view_script");
    data_file = getRegistry("control/data_file");

    // レポーターにパラメータを記録
    oneway {
        reporter.write("run-name", run_name);
        reporter.write("comment", comment);
        reporter.write("date", localTime());
        (中略)
    }

    // コンポーネントにパラメータを設定
    oneway {
        collector.setReadoutScript(readout_script);
        viewer.setViewScript(view_script);
        recorder.setDataFile(data_file);
        (中略)
    }

    // 各コンポーネントをコンストラクト
    asynchronous {
        collector.construct();
        analyzer.construct();
        recorder.construct();
        viewer.construct();
    }

    return 1;
}
```

これで，データ収集を開始する準備が整いました．ユーザはコントロールパネルの start をクリックすることができ，これにより KCOM の on .start() が呼ばれ，KCOM Script ではこの中で Collector の start() を呼び出します．

停止とシャットダウンは，基本的には上記手順の逆順ですが，停止には注意する必要があります．フロントエンドの Collector に Stop を送っても，データストリームにデータが残っている場合は，これらが全て処理されるのを待たなければならないからです．

Recorder はストリームから送られてくる RunBegin パケットと RunEnd パケットを数えており，RunBegin と同じ数の RunEnd を受け取ると finishRecording() イベントを発行します．SmallKinoko ではこれを利用して，全てのデータ処理が終了するまで Destruct できないようにしています．

```
on .stop()
{
    controller.changeState("stopping");
    controller.setWidgetValue("message", "stopping...");
    stop();
}

void stop()
{
    collector.stop();
}

on recorder.finishRecording()
{
    controller.changeState("system_ready");
    controller.setWidgetValue("message", "ready to start data acquisition");
    saveRegistry(log_dir + "/" + registry_file_name);
}
```

データ収集の停止は，コントロールパネルからのユーザの操作だけではなく，指定時間の経過や，ディスクスペースの不足などによっても引き起こされます．SmallKinoko では，これらのイベント受け取った際も stop() を呼ぶようになっています．

```
// 指定時間の経過など
on collector.dataAcquisitionFinished()
{
    controller.changeState("stopping");
    controller.setWidgetValue("message", "stopping...");
    stop();
}

// ディスクスペース不足
on recorder.diskSpaceExhausted()
{
    logger.writeError("disk space exhausted");

    if (is_running) {
        controller.changeState("stopping");
        controller.setWidgetValue("message", "stopping...");
        stop();
    }
}
```

コントロールパネルの quit が押されたら，スタートアップ時と逆順で終了処理をし，最後に exit して KCOM 自体も終了します．

```
on .quit()
{
    controller.changeState("shutdown");
    controller.setWidgetValue("message", "shutting down...");

    if (is_constructed) {
        is_constructed = false;
        destruct();
    }

    disconnect();
    quit();

    sleep(1);

    exit;
}
```

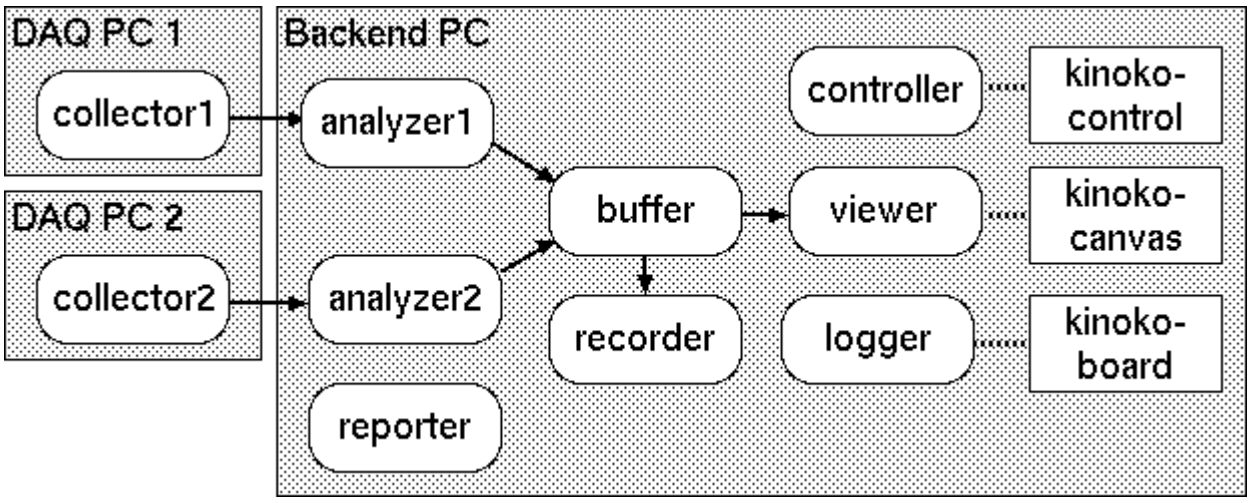
destruct() 関数，disconnect() 関数および quit() 関数の中身については $KINOKO_ROOT/scripts/SmallKinoko.kcom を参照してください．基本的に起動手順の逆ですが，Buffer については，特に能動的な動作をしないため，最後に quit() しているだけです．

SmallKinoko の拡張例１：Collector コンポーネントを2つ使う

SmallKinoko 拡張の例として，Collector コンポーネントを2つ使うように SmallKinoko.kcom を書き換えてみます．ここで作成する KCOM および KCML スクリプトは kinoko/local/tutorials/KcomScript にあります．SmallKinoko.kcom との diff をとることにより，必要な変更が一覧できます．

```
% diff -c SmallKinoko.kcom DualCollector.kcom
```

この例では，2つの Collector コンポーネントを別々のフロントエンド PC に配置することにします．2つの Collector からのストリームの合流には Buffer を使用します．現在のところ，Buffer は共有メモリを直接使用しているため，ネットワーク越しでの直接の接続はできないことに注意してください．このため，それぞれの Collector にひとつづつ Transporter コンポーネントが接続されています（名前は analyzer）．



SmallKinoko とのいちばん大きな違いは，Collector と Transporter を2つづつ使う点です．これらの component 宣言は以下のようになります．

```
component KinokoController controller(host, ui_control);
component KinokoReporter reporter(host);
component KinokoCollector collector1(daq_host_1);
component KinokoCollector collector2(daq_host_2);
component KinokoBuffer buffer(host);
component KinokoRecorder recorder(host);
component KinokoViewer viewer(host, ui_view);
component KinokoLogger logger(host, ui_log);
component KinokoTransporter analyzer1(host);
component KinokoTransporter analyzer2(host);
```

ここで，daq_host_1 および daq_host_2 にはフロントエンド PC のホスト名を適切に設定しておきます．

connect() 関数の中身は，ストリーム接続構成を反映して，以下のようになります．

```
void connect()
{
    buffer.start();

    asynchronous {
        collector1.setSink(analyzer1);
        collector2.setSink(analyzer2);
        analyzer1.setSourceSink(collector1, buffer);
        analyzer2.setSourceSink(collector2, buffer);
        recorder.setSource(buffer);
        viewer.setSource(buffer);
    }
```

```
    asynchronous {
        collector1.connect();
        collector2.connect();
        analyzer1.connect();
        analyzer2.connect();
        recorder.connect();
        viewer.connect();
    }
}
```

ふたつの Collector で別々の読みだしスクリプトを使うこともできますが，ここではひとつのスクリプトの異なったデータソースを使うことにします．ひとつの kts スクリプトの中には複数の datasource エントリを作成でき，大域変数や関数などを共有することができるので，ひとつのシステムとして動作するものの読みだしスクリプトはこの例のようにひとつにまとめておくと便利で，また，管理も楽になります．

construct() 関数のデータソースを指定する部分は以下のようになります．

```
int construct()
{
    readout_script = getRegistry("control/readout_script");
    view_script = getRegistry("control/view_script");
    data_file = getRegistry("control/data_file");
    string datasource1 = getRegistry("control/datasource1");
    string datasource2 = getRegistry("control/datasource2");

    // 中略

    oneway {
        collector1.setReadoutScript(readout_script, datasource1);
        collector2.setReadoutScript(readout_script, datasource2);
        viewer.setViewScript(view_script);
        recorder.setDataFile(data_file);
        // 中略
    }

    asynchronous {
        collector1.construct();
        collector2.construct();
        analyzer1.construct();
        analyzer2.construct();
        recorder.construct();
        viewer.construct();
    }

    return 1;
}
```

これに合わせて，コントロールパネルの方もデータソースを指定できるように変更します．ここでは，複雑さを避けるために SmallKinoko-Lite.kcml をベースにしていますが，SmallKinoko.kcml をベースにしても基本的に同様です．

多くの場合，複数の PC を使用するような大きなシステムでは読み出しスクリプトは固定するか，少なくともデータソース名は決まっている（crate01 など）ので，ここで行うようにコントロールパネルから変更できるようにする必要はあまり多くないと思います．スクリプト名やデータソース名を固定していい場合には，KCML の Constant ウィジェットを使うか KCOM に直書きするのが簡単です．

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="kweb-control-panel.xsl"?>

<KinokoControlPanel label="Kinoko Control Panel">
    <HSpace/><Label name="message" label="Welcome to KiNOKO." font="times-italic"/><HSpace/>
    <VSpace/>
```

```
<EntryList>
   <Entry name="readout_script" label="ReadoutScript (.kts)" option="file_select"/>
   <Entry name="view_script" label="ViewScript (.kvs)" option="file_select"/>
   <Entry name="data_file" label="DataFile (.kdf)" option="file_select"/>
</EntryList>
<VSpace/>
<Frame label="Datasource Names">
   <EntryList>
      <Entry name="datasource1" label="Collector 1"/>
      <Entry name="datasource2" label="Collector 2"/>
   </EntryList>
</Frame>
<NewLine/>

<Frame name="run_control" label="Run Control">
   <ButtonList>
      <Button name="construct" label="Construct" enabled_on="stream_ready system_ready"/>
      <Button name="start" label="Start" enabled_on="system_ready"/>
      <Button name="stop" label="Stop" enabled_on="data_taking"/>
      <Button name="clear_images" label="Clear" enabled_on="system_ready data_taking"/>
      <Button name="quit" label="Quit" enabled_on="stream_ready system_ready error"/>
   </ButtonList>
</Frame>
</KinokoControlPanel>
```

start() では，両方の Collector をスタートします．stop() も同様です．これにより，Buffer より下流では RunBegin と RunEnd がそれぞれふたつづつ送られるようになります．Recorder の finishRecording() は，このような状況でも適切に動作します．

```
void start()
{
   collector1.start();
   collector2.start();
   // 中略
}

void stop()
{
   collector1.stop();
   collector2.stop();
   // 中略
}
```

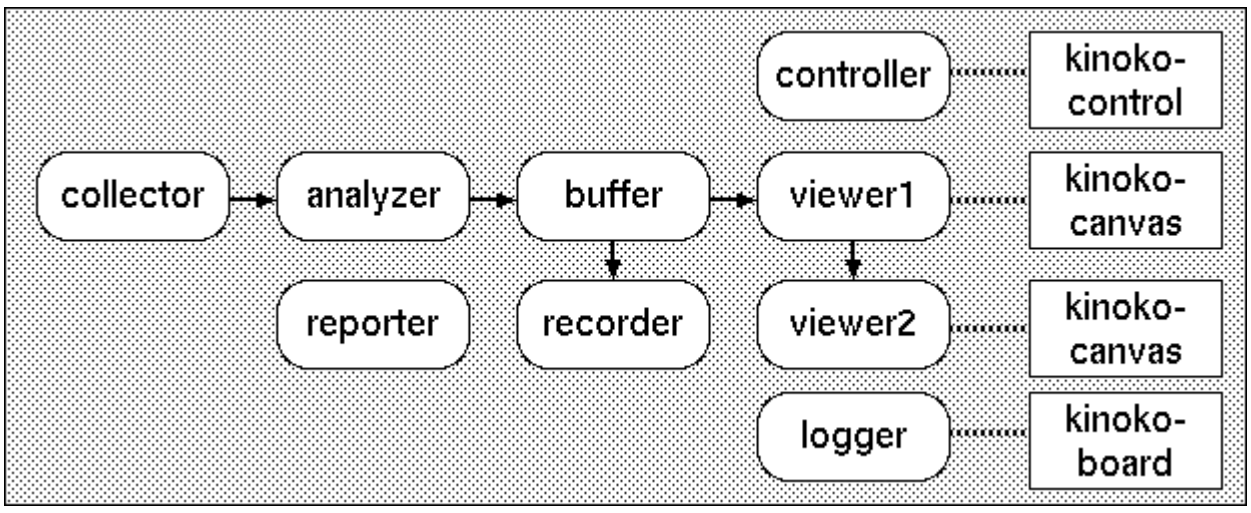
作成した KCOM スクリプトは kinoko コマンドで起動します．

```
% kinoko DualCollector.kcom
```

SmallKinoko の拡張例２：Viewer コンポーネントを２つ使う

次に，Viewer コンポーネントを2つ使用して多くのプロットを表示できるようにする例を見てみます．ここで作成する KCOM および KCML スクリプトは kinoko/local/tutorials/KcomScript にあります．

Viewer を2つ使用する場合でも，Collector の例と同様に，Buffer をストリームの分岐に使用することができます．しかし，ここでは，ChainedStreamSink の機能を使って Viewer コンポーネントをチェーン上に一列に接続する構成をとることにします． システム構成にもよりますが，Buffer はシステムのボトルネックになりやすいため，多数の StreamSink コンポーネント（Viewer や Recorder）を使用する場合はこのような構成にするほうが性能が向上することが多いようです．また，全体的な性能が不足した場合，StreamSinkChain の一部を別 PC に移動することも容易にできるという利点もあります（component 宣言部分の引数を変えるだけ）．



Viewer が2つあるため, kinoko-canvas も2つ用意します. 接続用のポートも別々に割り当てる必要があります. KCOM スクリプトの先頭付近は以下のようになります.

```
int port_control = port_base;
int port_log = port_base + 1;
int port_view1 = port_base + 2;
int port_view2 = port_base + 3;

string ui_control = "port: " + port_control;
string ui_log = "port: " + port_log;
string ui_view1 = "port: " + port_view1;
string ui_view2 = "port: " + port_view2;

execute("kinoko-control --client " + host + " " + port_control);
execute("kinoko-board --client " + host + " " + port_log);
execute("kinoko-canvas --client " + host + " " + port_view1);
execute("kinoko-canvas --client " + host + " " + port_view2);

component KinokoController controller(host, ui_control);
component KinokoReporter reporter(host);
component KinokoCollector collector(daq_host);
component KinokoBuffer buffer(host);
component KinokoRecorder recorder(host);
component KinokoViewer viewer1(host, ui_view1);
component KinokoViewer viewer2(host, ui_view2);
component KinokoLogger logger(host, ui_log);
component KinokoTransporter analyzer(host);
```

connect() 関数は以下のようになります. この例のようにストリームシンクの Viewer をチェーン接続する場合は, ストリームパイプの場合と同様に setSourceSink() を使って接続先を指定します.

```
void connect()
{
    buffer.start();

    asynchronous {
        collector.setSink(analyzer);
        analyzer.setSourceSink(collector, buffer);
        recorder.setSource(buffer);
        viewer1.setSourceSink(buffer, viewer2);
        viewer2.setSource(viewer1);
    }

    asynchronous {
        collector.connect();
        analyzer.connect();
        recorder.connect();
        viewer1.connect();
        viewer2.connect();
    }
}
```

ビューアが2つあるので，ここでは2つのビュースクリプトを使うことにします（前章の例と同様に，ひとつのスクリプトの異なる display エントリを指定することもできます）．construct() 関数の中身は以下のようになります．

```
int construct()
{
    string readout_script = getRegistry("control/readout_script");
    string view_script_1 = getRegistry("control/view_script_1");
    string view_script_2 = getRegistry("control/view_script_2");
    string data_file = getRegistry("control/data_file");
    // 中略

    oneway {
        collector.setReadoutScript(readout_script);
        viewer1.setViewScript(view_script_1);
        viewer2.setViewScript(view_script_2);
        recorder.setDataFile(data_file);
      // 中略
    }

    asynchronous {
        collector.construct();
        analyzer.construct();
        recorder.construct();
        viewer1.construct();
        viewer2.construct();
    }

    return 1;
}
```

これに合わせて，コントロールパネルも2つのビュースクリプトを指定できるように変更します．

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="kweb-control-panel.xsl"?>

<KinokoControlPanel label="Kinoko Control Panel">
    <HSpace/><Label name="message" label="Welcome to KiNOKO." font="times-italic"/><HSpace/>
    <VSpace/>

    <EntryList>
        <Entry name="readout_script" label="ReadoutScript (.kts)" option="file_select"/>
        <Entry name="view_script_1" label="ViewScript 1 (.kvs)" option="file_select"/>
        <Entry name="view_script_2" label="ViewScript 2 (.kvs)" option="file_select"/>
        <Entry name="data_file" label="DataFile (.kdf)" option="file_select"/>
    </EntryList>
    <VSpace/>

    <Frame name="run_control" label="Run Control">
        <ButtonList>
            <Button name="construct" label="Construct" enabled_on="stream_ready system_ready"/>
            <Button name="start" label="Start" enabled_on="system_ready"/>
            <Button name="stop" label="Stop" enabled_on="data_taking"/>
            <Button name="clear_images" label="Clear" enabled_on="system_ready data_taking"/>
            <Button name="quit" label="Quit" enabled_on="stream_ready system_ready error"/>
        </ButtonList>
    </Frame>
</KinokoControlPanel>
```

以上で終了です．これで，描画ウィンドウが2つあるシステムが使用できるようになりました．

```
% kinoko DualViewer.kcom
```

前章の Collector コンポーネントを増やして複数の計算機に分散させる例と組み合わせたり，analyzer を独自のデータ解析コンポーネントと置き換えたりすることにより，現段階でもかなり複雑なシステムを組み上げることができるようになっています．

モジュールドライバの作成

Kinoko でハードウェアを操作しデータを読むためには，そのモジュールの動作を Kinoko に教えるためのコード，すなわちモジュールドライバ (KinokoModuleDriver) が必要です．Kinoko のパッケージにはすでにいくつかのモジュールドライバのコードが付属しており，使用できるモジュールの一覧は kinoko-lsmod コマンドで一覧できます．CAMAC モジュールに関しては，Generic-CamacModule でそのまま使用できるものも多くあります．詳細については，リファレンス > モジュールドライバリファレンス の章を参照してください．

現在サポートされていないデバイスを使用する場合は，モジュールドライバを独自に作成して Kinoko に組み込む必要があります．Kinoko は VME および CAMAC を使用するためのモジュールドライバフレームワークを提供しており，これを用いればハードウェアの詳細部分をかなり簡略化できます．一方で，製品付属のライブラリや外部ソフトウェアパッケージを使用することも可能で，これにより GPIB や データ収集 PCI カードなどのデバイスからデータを読み出すことも，イベントジェネレータのようなシミュレーションパッケージから疑似データを取得することもできるようになります．

簡単な例

ここでは，Linux カーネルの乱数生成機構である /dev/random を仮想的なデバイスと見て，ここから読み出しを行って乱数を生成するモジュールドライバを作り，それを Kinoko に追加してみます．この例の完全なコードは kinoko/local/tutorials/ModuleDriver/SoftwareModule にあります．

/dev/random は，いろいろなデバイスドライバなどの動作（環境ノイズ）をもとに疑似ではない乱数を生成するカーネルインターフェースで，/dev/random 自体は普通のキャラクタ型デバイスのようにアクセスできます．read() をすれば乱数をかえしてきます．もし，乱数の生成に十分な「エントロピー」がなければ，必要な環境ノイズが得られるまで read() はブロックされます．この振舞いは，データ収集デバイスの読み出しとデータ待ちに似ています．/dev/random の詳細については，Linux のマニュアルページ random(4) を参照してください．

この例では，以下のファイルを作成します．

- module-DevRandom.cc: モジュールドライバ本体
- module-DevRandom.hh: モジュールドライバヘッダ
- DevRandom-test.cc: モジュールドライバを単体テストするプログラム
- DevRandom.kts: 作成したモジュールドライバをテストするための読み出しスクリプト
- Makefile: メイクファイル

モジュールドライバは全て，間接的に TRoomModule の派生クラスとなります．VME および CAMAC のモジュールドライバはそれぞれ TRoomVmeModule および TRoomCamacModule から派生させ，その他のデバイスは現時点では TRoomSoftwareModule から派生させます．これらは全て TRoomModule の派生クラスです．これらのクラスの詳細については，拡張方法 の モジュールドライバを書く の章を参照してください．

モジュールドライバのインターフェースの中で，最低限実装しなければならないのが，コンストラクタ，デストラクタと Clone() メソッドです．Clone() では，自分自身と同じ型のインスタンスをひとつ生成して，そのポインタを返すようにします．

```
TSoftwareModule_DevRandom::TSoftwareModule_DevRandom(void)
: TRoomSoftwareModule("DevRandom", "Software_DevRandom")
{
    // /dev/random 用のファイルハンドル．無効な値に初期化
    _DeviceFile = -1;
}

TSoftwareModule_DevRandom::~TSoftwareModule_DevRandom()
{
}

TRoomSoftwareModule* TSoftwareModule_DevRandom::Clone(void)
{
    //自分と同じ型のインスタンスを作成してポインタを返す
    return new TSoftwareModule_DevRandom();
}
```

これに加えて，以下のメソッドをオーバーライドしておくと，Kinoko がいろいろな場面でデータサイズ等を自動管理し，最適化します．

```
int TSoftwareModule_DevRandom::NumberOfChannels(void) throw(THardwareException)
{
    // モジュールのチャンネル数．この例ではいくつでも良い
    return 16;
}

int TSoftwareModule_DevRandom::AddressBitLength(void)
{
    // チャンネル番号を格納するのに必要なビット数
    return 4;
}

int TSoftwareModule_DevRandom::DataBitLength(void)
{
    // データエレメントをひとつ格納するのに必要なビット数
    return 16;
}
```

Initialize() メソッドと Finalize() メソッドをオーバーライドしておくと，それぞれデータ収集開始直前と終了直後に呼び出されるようになります．ここでは，このタイミングで /dev/random をオープンおよびクローズをするようにします．

```
int TSoftwareModule_DevRandom::Initialize(int InitialState) throw(THardwareException)
{
    // /dev/random をリードオンリーでオープン
    _DeviceFile = open("/dev/random", O_RDONLY);

    // エラーチェック
    if (_DeviceFile < 0) {
        throw THardwareException(
            "TSoftwareModule_DevRandom::Initialize()",
            "unable to open file: /dev/random"
        );
    }

    return 0;
}

int TSoftwareModule_DevRandom::Finalize(int FinalState) throw(THardwareException)
{
    // /dev/random をクローズ
    if (_DeviceFile >= 0) {
        close(_DeviceFile);
        _DeviceFile = -1;
    }

    return 0;
}
```

データの読み出しを行えるようにするためには，Read() メソッドをオーバーライドします．戻り値はデータのサイズで，失敗したら例外 THardwareException を投げます．データが存在しない場合は例外を投げるのではなく，0 を返すようにします．

```
int TSoftwareModule_DevRandom::Read(int Address, int &Data) throw(THardwareException)
{

    // U16bit は RoomConfig.hh で宣言されている符号なし 16bit 整数
    // 他に U8bit と U32bit がある
    U16bit Buffer;

    // /dev/random から 16bit をひとつ読み出す
    int Result = read(_DeviceFile, &Buffer, sizeof(Buffer));
```

```
    // エラーチェック
    if (Result < 0) {
        if (errno == EINTR) {
            // シグナル割り込みで中断された．やりなおし
            return this->Read(Address, Data);
        }
        else {
            throw THardwareException(
                "TSoftwareModule_DevRandom::Read()", strerror(errno)
            );
        }
    }
    Data = Buffer;

    return Result;
}
```

ここまでで，読み出しスクリプトから read(), taggedRead() および sequentialRead() が使えるようになっています（sequentialRead() は上記 Read() メソッドが 0 を返すまで繰り返し Read() を呼び出すので，この例では修正が必要です）．

読み出しスクリプトで，このモジュールをシーケンス開始条件に使う(on trigger に使う)ためには，HasData() メソッドをオーバライドしてデータが読み出し可能かどうかを Kinoko に伝えるようにします．データが存在すれば true を，そうでなければ false を返すようにします．同時に，この例では必要ありませんが，モジュール上のデータをクリアするメソッド Clear() もオーバライドしておきます．一般には，Clear() で HasData() の条件もクリアし，次のデータの到着を待つようにします．

```
bool TSoftwareModule_DevRandom::HasData(int Address) throw(THardwareException)
{
    // poll() システムコールにより，/dev/random から即時にデータが読み出し可能か調べる

    struct pollfd fds;
    fds.fd = _DeviceFile;
    fds.events = POLLIN;

    if (poll(&fds, 1, 0) < 0) {
        throw THardwareException(
            "TSoftwareModule_DevRandom::HasData()", strerror(errno)
        );
    }

    return fds.revents & POLLIN;
}

int TSoftwareModule_DevRandom::Clear(int Address) throw(THardwareException)
{
    return 0;
}
```

ここまでで，読み出しスクリプトでこのモジュールを on trigger の条件に使うことと，waitData() を使うことができるようになりました．この時点でのこのモジュールドライバのクラス定義は以下のようになっています．

```
class TSoftwareModule_DevRandom: public TRoomSoftwareModule {
  public:
    TSoftwareModule_DevRandom(void);
    virtual ~TSoftwareModule_DevRandom();
    virtual TRoomSoftwareModule* Clone(void);
    virtual int NumberOfChannels(void) throw(THardwareException);
    virtual int AddressBitLength(void);
    virtual int DataBitLength(void);
    virtual int Initialize(int InitialState = 0) throw(THardwareException);
    virtual int Finalize(int FinalState = 0) throw(THardwareException);
    virtual bool HasData(int Address = -1) throw(THardwareException);
    virtual int Clear(int Address = -1) throw(THardwareException);
    virtual int Read(int Address, int &Data) throw(THardwareException);
  protected:
    int _DeviceFile;
};
```

最後に，このモジュールドライバを Kinoko に自動登録するためのトリッキーなコードを module-DevRandom.cc の先頭付近に追加します．

```
// static なオブジェクトなので，プログラムロードと同時に初期化される...はず．
static TRoomSoftwareModuleCreator Creator(
    "DevRandom", new TSoftwareModule_DevRandom()
);
```

make をすると，モジュールドライバ module-DevRandom.o と，それを単体テストする DevRandom-test が生成されます．モジュールドライバの開発過程で毎回 Kinoko を再構築し，Kinoko を起動しているとかなり面倒なので，このような単体テストコードを使用すると便利です．このソース DevRandom-test.cc は，先頭付近のデバイス名を変えるだけで他のモジュールドライバへの転用が容易なように書かれています．

```
% make
% ./DevRandom-test
```

単体テストで問題がなければ，これを Kinoko に組込みます(データレートが低い場合はマウスをぐりぐり動かしてください)．このソースコードを Kinoko のソースツリーに組み込んで全体を再コンパイルしても良いのですが，ここではモジュールドライバのみを kinoko/devices にコピーして，必要な部分だけ再コンパイルをするようにします．Makefile にはすでにこれを行うコードが書かれていますので，これを使います．

```
% make rebuild-kinoko
```

これで，Kinoko からこのモジュールドライバが使えるようになっているはずです．tinykinoko からテストスクリプトを実行して確認します．

```
% tinykinoko DevRandom.kts test.kdf 100
% kdfdump test.kdf
```

/dev/random は，乱数を生成するのに十分なハードウェアイベントが起こらないとデータを返しません．この例では，トリガがかからないことになります．なかなか先に進まない場合は，マウスを動かすなどしてハードウェアイベントをたくさん起こすようにしてください．

その他のインターフェース

上記の例は，Read() のみを行う非常に単純なものでした．モジュールドライバには，より複雑な状況で使われる様々なインターフェースがあります．以下にその例をいくつか示します．詳細については，拡張方法 の モジュールドライバを書く の章を参照してください．

- SequentialRead() と NextNumberOfDataElements() を実装すると，sequentialRead() の効率がかなり向上します．また，モジュールによっては，この方法でのみ読み出しができるような構造のものもあります．
- BlockRead() と NextDataBlockSize() を実装すると，ブロック読み出し (読み出しスクリプトの blockRead() アクション) が使用できるようになります．また，モジュールによっては，この方法でのみ読み出しができるような構造のものもあります．
- HasData() に加えて，WaitData() を実装しておくと，Kinoko が単純な構成での不要なポーリングループを避けられるようになり，効率が向上します．また，モジュールが割り込みなどを使用できる場合は，サービスリクエスタのインターフェースを実装しておくと，トリガの多面待ちなどの状況での性能がかなり向上します．
- モジュールとのデータのやりとりは，Read() 系に加えて，ReadRegister() および WriteRegister() が使用できます．これらが実装されていると，読み出しスクリプトからも同名のメソッドを介してモジュールの設定などができるようになります．
- 複雑な機能を持ったモジュールでは，MiscControl() インターフェースを実装することにより，一連のレジスタアクセスなどをまとめてひとつのアクションとすることができます．例えば，Kinoko に付属の LeCroy ADC/TDC Tester では，この機能により多数の設定アクションが利用できるようになっています．

VME および CAMAC

上記の例のような SoftwareModule では，ハードウェアインターフェースに関しては Kinoko からのサポートは一切ありませんでした．これに対し，VME や CAMAC を使う場合，TRoomVmeModule や TRoomCamacModule からモジュールドライバを派生させることにより，これらのクラスで定義されている多くのユーティリティメソッドを利用できるようになります．例えば，VME なら，メモリマッピングや DMA 転送などが，CAMACなら CAMAC サイクルの実行や LAM の操作などが，コントローラの種類によらず同じインターフェースで利用できます．これらの詳細については，拡張方法 の モジュールドライバを書く の章を参照してください．

データファイルの読み方 0 - ダンプツールを使う -

TinyKinoko や SmallKinoko などでとったデータは，KDF (Kinoko Data Format) 形式のファイルで保存されます．これはデータの他にヘッダやデータデスクリプタなどのさまざまな情報を含んだバイナリファイルで，kdfdump や kdftable などの標準ユーティリティでテキストに変換できます．また，kdfprofile や kdfcheck などを使うと，そのデータを取るときに使ったスクリプト名やその他のさまざまな設定を見ることができます．

kdfdump

kdfdump コマンドを使うと，KDF ファイルに書かれている全てのデータを表示させることができます．以下は，kdfdump コマンド出力の例です．

```
% kdfdump test01.kdf -
# Creator: tinykinoko
# DateTime: 2008-05-28 13:18:54 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/trial01
# ScriptFile: Trial01.kts

# EventTime: 1211948334
# EventTime: 1211948334

adc 0 1970
adc 1 1043
adc 2 1505
adc 3 339
# EventTime: 1211948334

adc 0 1216
adc 1 1411
adc 2 972
adc 3 183

adc 0 1521
adc 1 1510
adc 2 885
adc 3 1369
# EventTime: 1211948335

adc 0 173
adc 1 2022
adc 2 1016
adc 3 704
（以下省略）
```

kdfdump のコマンド書式は以下のようになっています．

```
kdfdump [オプション] データファイル名　[データソース名 [データセクション名 ...]]
```

第2引数のデータソース名に - を指定するとファイルの中の最初のデータソースが指定されたことになります．データソース名を省略すると全てのデータソースを表示します．第3引数のセクション名は表示対象を限定するもので，省略した場合は全てのセクションを表示します．セクション名は複数指定することができます．

kdfdump の出力は，以下のようになっています．

- 最初にストレージヘッダの内容が 行頭に # を付けて表示される
- イベント区切り (EventTrailer) の位置に空行が挿入される
- データソースが指定されていない場合は，パケットの先頭位置に　###[*データソース名*]### と書かれた行が挿入される
- 一行にひとつのデータエレメントが，セクション名，アドレス，データの順で空白区切りで表示される．
- 時間情報のあるパケットについては，# EventTime: *タイムスタンプ* が表示される．ここで，タイムスタンプは UNIX の time(2) の形式．

バイナリデータブロックのダンプ

block 型のデータについては，16進のダンプが表示されます．バイナリデータのまま出力したい場合は，--binary オプションを指定します．

```
% kdfdump --binary test01.kdf - > test01.out
```

ただし，ここで書き出されるのはデータ本体のみ（モジュールから読み出されたデータブロックそのもの）で，ストレージヘッダやセクション名などは含まれません．もしデータブロック本体だけではチャンネルやデータサイズなどが分からなくなってしまう場合は，--packet オプションを指定して，パケット全体をダンプします．

```
% kdfdump --packet test01.kdf - > test01.out
```

パケットは，データブロックの先頭に32ビットワードが3ワード付加された形をしており，この中にデータソース ID やセクション ID が書かれています．

bit 16	bit 0
0 0 0 0	DataSourceID
0 0 0 8	SectionID
DataSize (byte)	
DataArea : :	

データパケットは 32bit にアラインされていることに注意してください．つまり，DataArea のサイズが 4 バイトの整数倍でない場合，末尾に1バイトから3バイトが挿入されることになります．パケットのサイズは，データを格納できる最小の4の倍数バイトとなります．

データソース ID やセクション ID からデータソース名やセクション名を特定するには，kdfcheck ユーティリティを使い，その先頭部分を見ます．

```
% kdfcheck test01.kdf | more
% Preamble Version: 00010002
% Header Version: 00010001
% Data Area Version: 00010002
% Data Format Flags: 00000000

# Creator: tinykinoko
# DateTime: 2008-06-13 17:31:23 JST
# UserName: sanshiro
# Host: kinoko
# Directory: /home/sanshiro/kinoko/local/samples
# ScriptFile: CamacAdc.kts

[29373]--- 0, 388 bytes @1f0
Data Descriptor
    datasource "CamacAdc"<29373>
    {
        attribute creator = "tinykinoko";
        attribute creation_date = "2008-06-13 17:31:23 JST";
        attribute host_name = "kinoko";
        attribute user_name = "sanshiro";
        attribute script_file = "CamacAdc.kts";
        attribute script_file_fingerprint = "6355550a";

        section "adc"<257>: indexed(address: int-5bit, data: int-16bit);
    }

[29373]--- 1, 12 bytes @378
Command: 1 RUN_BEGIN, 1213345883 (13 Jun 2008 17:31:23 JST)
```

```
[29373]--- 2, 12 bytes @388
Command: 5 CLOCK_TICK, 1213345883 (13 Jun 2008 17:31:23 JST)
(以下省略)
```

これにより，データソース CamacAdc のデータソース ID は 29373，セクション adc のセクション ID は 257 であることが分かります．kdfdump でバイナリダンプして解析するのは一時的な用途を想定していますが，もし継続的行うのならば，読み出しスクリプトでデータソース ID やセクション ID を固定するようにしておくと便利です．

kdftable

kdfdump は KDF ファイル内のデータを全て表示することを目的としており，その出力内容はデータの中身によりいろいろと変化してしまいます（データ数やセクションの有無など）．もしデータの構造が単純なものなら，kdftable を使うとより扱いやすい形式の出力が得られます．

```
% kdftable test01.kdf
# Creator: tinykinoko
# DateTime: 2008-05-28 13:18:54 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/trial01
# ScriptFile: Trial01.kts

# Fields: index time adc.0 adc.1 adc.2 adc.3
# StartTime: 1211948334

0 0 1970 1043 1505 339
1 0 1216 1411 972 183
2 1 1521 1510 885 1369
3 1 173 2022 1016 704
4 2 1632 1518 1075 1951
(以下省略)
```

kdftable のコマンド書式は以下のようになっています．

```
kdftable [オプション] データファイル名　[データソース名 [データセクション名 ...]]
```

第2引数のデータソース名に - を指定するか，指定を省略するとファイルの中の最初のデータソースが指定されたことになります．第3引数のセクション名は表示対象を限定するもので，省略した場合は全てのセクションを表示します．セクション名は複数指定することができます．

kdftable の出力は，以下のようになっています．

- 最初にストレージヘッダの内容が 行頭に # を付けて表示される
- 表示対象のデータの名前が # Fields: の後ろに表示される
- スタートした時間が UNIX の time(2) 形式で # StartTime: の後ろに表示される
- index 型のデータについて，イベント番号，スタートからの経過秒数，データが1イベント1行で空白区切りで表示される
- tagged 型のデータについて，「# セクション名.フィールド名 データ」の形式で1エレメント1行で表示される

kdftable は，このように表形式で出力できるデータにのみ対応しています．多くの単純なモジュールで使えますが，フラッシュ ADC やマルチヒット TDC などには使えません．

自作の解析プログラムなどから kdftable の出力を読むためには，行ごとに読み込み，空行と # から始まる行を飛ばして，空白区切りで数値を読めばよいことになります．

```
// test01.out は kdftable の出力 (kdftable test01.kdf > test01.out)
string InputFileName = "test01.out";

ifstream InputFile(InputFileName.c_str());
```

```
if (! InputFile) {
        cerr << "ERROR: unable to open file: " << InputFileName << endl;
        exit(EXIT_FAILURE);
}

string Line;
int EventIndex, Time, FieldIndex, Data;

while (getline(InputFile, Line)) {
    // 空白行とコメント行をスキップ
    if (Line.empty() || (Line[0] == '#')) {
        continue;
    }

    istringstream Stream(Line);
    Stream >> EventIndex >> Time;
    for (FieldIndex = 0; Stream >> Data; FieldIndex++) {
        // ここで EventIndex, Time, FieldIndex, Data が揃っている
        // Histogram[FieldIndex]->Fill(Data) とか
    }
    // ここはイベントの区切り
  }
```

popen(3) 関数を使ってプログラムから kdftable を呼び出すようにすれば，毎回手動で kdftable を実行したりその結果をファイルに保存したりする必要がなくなり便利です．以下に例を示します（C と C++ が混ざって見苦しいことになっていますが…）．

```
#include <iostream>
#include <sstream>
#include <string>
#include <cstdio>
#include <cstdlib>

using namespace std;

string InputFileName = "test01.kdf";
static const int MaxLineLength = 1024;

int main(void)
{
    // kdftable を実行するパイプを作る
    string Command = "kdftable " + InputFileName;
    FILE* InputFile = popen(Command.c_str(), "r");
    if (InputFile == NULL) {
        cerr << "ERROR: unable to execute command: " << Command;
        exit(EXIT_FAILURE);
}

char Line[MaxLineLength];
int EventIndex, Time, FieldIndex, Data;

// popen() は C のファイル構造体を返すので仕方なく fgets() を使う
while (fgets(Line, sizeof(Line), InputFile) != NULL) {
        // 空白行とコメント行をスキップ
        if ((Line[0] == '¥n') || (Line[0] == '#')) {
            continue;
        }

        istringstream Stream(Line);
        Stream >> EventIndex >> Time;
        for (FieldIndex = 0; Stream >> Data; FieldIndex++) {
            // ここで EventIndex, Time, FieldIndex, Data が揃っている
            // Histogram[FieldIndex]->Fill(Data) とか
        }
```

```
        // ここはイベントの区切り
    }

    pclose(InputFile);

    return 0;
}
```

データファイルの読み方 1 - DataAnalyzer を使う -

ここでは，kdfdump などの変換ユーティリティを使わずに，直接 C++ のプログラムから KDF 形式のファイルを読む方法を説明します．ここで使用する例の完全なソースコードは kinoko/local/tutorials/DataAnalyzer にあります．

KDF ファイルを読む方法はいくつかありますが，ここではこのうち一番手軽な DataAnalyzer フレームワークを用いる方法を説明します．DataAnalyzer フレームワークを使うと，データ構造の詳細に煩わされずに統一的で手軽な方法でデータを読むことができます．CAMAC の ADC などで取ったデータを順に処理する場合などに適しています．

一方で，複数のデータソースを同時に扱ったり，未知のデータ構造に対して汎用の処理を行なったりなどの高度な処理をしようとすると，逆に使いにくい面もあります．このような場合は，後述する DataProcessor フレームワークを使うことになります．作成したプログラムを将来コンポーネント化したい場合も，DataProcessor フレームワークを使ってください．

簡単な例

データファイルの例として，1 枚の ADC モジュールを kinoko/local/samples にある CamacAdc.kts を使って読んだものを使います．このデータファイルの中身は以下のようになっています．

```
% kdfdump test01.kdf -
# Creator: tinykinoko
# DateTime: 2003-03-09 20:10:17 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/samples
# ScriptFile: CamacAdc.kts


adc 0 258
adc 1 363
adc 2 2073
adc 3 1114

adc 0 298
adc 1 705
adc 2 138
adc 3 1461

(以下省略)
```

```
% kdfcheck test01.kdf
% Preamble Version: 00010002
% Header Version: 00010001
% Data Area Version: 00010001
% Data Format Flags: 00000000

# Creator: tinykinoko
# DateTime: 2003-03-09 20:10:17 JST
# UserName: sanshiro
```

```
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/samples
# ScriptFile: CamacAdc.kts

[2269]--- 0, 308 bytes
Data Descriptor
    datasource "CamacAdc"<2269>
    {
        attribute creator = "tinykinoko";
        attribute creation_date = "2003-03-09 20:10:18 JST";
        attribute script_file = "CamacAdc.kts";
        attribute script_file_fingerprint = "6355550a";

        section "adc"<256>: indexed(address: int-16bit, data: int-16bit);
    }

(以下省略)
```

このデータファイルには，データソース CamacAdc のデータのみが入っていて，その中にはセクション adc のみがあることに注意してください．

はじめにやることは，Kinoko のデータ解析フレームワークである KinokoSectionDataAnalyzer クラスを継承して，自分のアナライザクラスを作り，そこで ProcessData() メソッドをオーバーライドすることです．

```
/* DataAnalyzer01.cc */

#include "KinokoKdfReader.hh"
#include "KinokoSectionDataAnalyzer.hh"

class TMyDataAnalyzer: public TKinokoSectionDataAnalyzer {
  public:
    TMyDataAnalyzer(void);
    virtual ~TMyDataAnalyzer();
    virtual int ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException);
  protected:
    int _EventCount;
};
```

ここではファイル中のデータを全て表示し，最後にイベント数を表示するようにします．メンバの _EventCount はイベント数を数えるためのものです．オーバーライドするメソッドの他に，コンストラクタとデストラクタも必要です．

次にこれらのメソッドの実装です．

```
static const char* SectionName = "adc";

TMyDataAnalyzer::TMyDataAnalyzer(void)
: TKinokoSectionDataAnalyzer(SectionName)
{
    _EventCount = 0;
}

TMyDataAnalyzer::~TMyDataAnalyzer()
{
    cout << endl;
    cout << _EventCount << " events were processed." << endl;
}

int TMyDataAnalyzer::ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException)
{
    _EventCount++;
```

```
    int Address, Data;
    while (SectionData->GetNext(Address, Data)) {
          cout << Address << " " << Data << endl;
    }

    return 1;
}
```

親クラスである TKinokoSectionDataAnalyzer のコンストラクタには，読みたいデータのセクション名を渡してください．これにより，そのセクションのデータオブジェクトを引数に ProcessData() が呼ばれるようになります．ProcessData() は，ひとつのデータパケットごと呼び出されます．

ProcessData() の中では，そのセクションデータオブジェクトから GetNext() で順にデータを読み出し，表示させます．GetNext() は，そのデータオブジェクトから読み出すデータがなくなると，false をかえします．

最後に，main() 関数を作成し，Kinoko のファイル読み出しをおこなうクラス TKinokoKdfReader のインスタンスを作成して，RegisterAnalyzer() で自分の Analyzer 登録します．TKinokoKdfReader のコンストラクタの引数にはデータファイル名を，RegisterAnalyzer() では読み出したいデータのデータソース名を指定します．この例のようにデータファイルにデータソースが一つしか含まれていない場合，データソース名には空文字列を指定できます．

```
static const char* DataSourceName = "";

int main(int argc, char** argv)
{
    if (argc < 2) {
          cerr << "Usage: " << argv[0] << " DataFileName" << endl;
       return EXIT_FAILURE;
    }
    string DataFileName = argv[1];

    TKinokoKdfReader* KdfReader = new TKinokoKdfReader(DataFileName);
    TKinokoDataAnalyzer* Analyzer = new TMyDataAnalyzer();

    try {
          KdfReader->RegisterAnalyzer(DataSourceName, Analyzer);
          KdfReader->Start();
    }
    catch (TKinokoException& e) {
          cerr << "ERROR: " << e << endl;
       return EXIT_FAILURE;
    }

    delete Analyzer;
    delete KdfReader;

    return 0;
}
```

このプログラムをコンパイルするためには，Kinoko のライブラリをリンクする必要があります． 標準ユーティリティ kinoko-config を使うと，必要なライブラリやインクルードパスの指定を自動化できます．以下に，kinoko-config を使用した Makefile の例を示します．

```
# Makefile for sample DataAnalyzer

CXX = $(shell kinoko-config --cxx)
FLAGS = $(shell kinoko-config --flags)
LIBS = $(shell kinoko-config --libs)

DataAnalyzer01: DataAnalyzer01.o
          $(CXX) -o $@ $@.o $(LIBS)

.cc.o:
          $(CXX) $(FLAGS) -c $<
```

このプログラムを実行すると，以下のようになります．

```
% ./DataAnalyzer01 test01.kdf
0 258
1 363
2 2073
3 1114
0 298
1 705
2 138
3 1461
(中略)

100 events were processed.
%
```

複数のセクションを読む

つぎに，もうすこし複雑な例として，複数のセクション(複数のモジュール)のデータの読み出しを行ないます．また，イベントとイベントの間に空行を表示させてみます．

前回と同様に，KinokoSectionDataAnalyzer を継承して，自分のアナライザクラスを作成し，ProcessData() をオーバーライドします．また，イベントデータの最後で空行を表示させるために，ProcessTrailer() もオーバーライドします．

```
/* DataAnalyzer02.cc */

class TMyDataAnalyzer: public TKinokoSectionDataAnalyzer {
  public:
    TMyDataAnalyzer(void);
    virtual ~TMyDataAnalyzer();
    virtual int ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException);
    virtual int ProcessTrailer(int TrailerValue) throw(TKinokoException);
  protected:
    int _AdcSectionIndex, _TdcSectionIndex;
};
```

次に，メソッドの実装です．今回は読み出すデータセクションが複数あるので，前回のように TKinokoDataSectionAnalyzer のコンストラクタでセクション名を指定するのではなく，以下のように AddSection() メソッドを使用します．AddSection() は戻り値にデータ種別を区別するための SectionIndex を返すので，その値をメンバ変数に保存しておきます．

ProcessData() では，AddSection() で取得した SectionIndex を使ってデータオブジェクトのセクションを区別します．

ProcessTrailer() では，トレイラの種別を確認して，イベントトレイラなら，イベント区切りの空行を出力します．

```
TMyDataAnalyzer::TMyDataAnalyzer(void)
{
    _AdcSectionIndex = AddSection("adc");
    _TdcSectionIndex = AddSection("tdc");
}

TMyDataAnalyzer::~TMyDataAnalyzer()
{
}

int TMyDataAnalyzer::ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException)
{
    const char* SectionName = "unknown";
    if (SectionData->SectionIndex() == _AdcSectionIndex) {
        SectionName = "adc";
    }
    else if (SectionData->SectionIndex() == _TdcSectionIndex) {
```

```
        SectionName = "tdc";
    }

    // 実は SectionName だけなら，以下のように取得することもできる.
    // string SectionName = SectionData->SectionName();

    int Address, Data;
    while (SectionData->GetNext(Address, Data)) {
        cout << SectionName << " " << Address << " " << Data << endl;
    }

    return 1;
}

int TMyDataAnalyzer::ProcessTrailer(int TrailerValue) throw(TKinokoException)
{
    if (TrailerValue == TKinokoDataStreamScanner::Trailer_Event) {
        cout << endl;
    }

    return 1;
}
```

main() および Makefile は，前回の例と同様です.

CamacAdcTdc.kts で取ったデータに対してこのプログラムを実行すると，以下のようになります.

```
% ./DataAnalyzer02 test02.kdf
adc 0 1234
adc 1 761
adc 2 1650
adc 3 1236
tdc 0 3055
tdc 1 2541
tdc 2 1737
tdc 3 2181

adc 0 1060
adc 1 1200
adc 2 1508
adc 3 1339
tdc 0 1014
tdc 1 2807
tdc 2 2037
tdc 3 2101

adc 0 1339
(以下省略)

%
```

ストレージヘッダを読む

ストレージヘッダの情報は，main() 関数中で作成した TKinokoKdfReader クラスのインスタンス KdfReader を使って以下のように取得できます．

```
TKinokoKdfReader* KdfReader = new TKinokoKdfReader(DataFileName);
TKinokoDataAnalyzer* Analyzer = new TMyDataAnalyzer();

// ストレージヘッダにアクセスする前にアナライザの登録を終らせておく
KdfReader->RegisterAnalyzer(DataSourceName, Analyzer);

string DateTime = KdfReader->StorageHeader()->ValueOf("DateTime");
cout << "DateTime: " << DateTime << endl;
```

StorageHeader() メソッドの呼び出しは，全ての RegisterAnalyzer() 呼び出しが終ってから行なうようにしてください．これは Kinoko の内部構造に起因する制約です．

TKinokoKdfReader の StorageHeader() メソッドは，TKinokoStorageHeader クラスのインスタンスへのポインタを返します．

TKinokoStorageHeader クラスには，各エントリの情報を取得するための以下のメソッドが定義されています．

TKinokoStorageHeader

unsigned NumberOfEntries(void) const;
　　ストレージヘッダに記録されているエントリの数を返す．
std::string EntryNameAt(unsigned Index) const;
　　Index 番目のエントリのエントリ名を返す．
std::string ValueAt(unsigned Index) const;
　　Index 番目のエントリの値を返す．
std::string ValueOf(const std::string& Name) const;
　　Name に指定されたエントリ名のエントリの値を返す．エントリが存在しない場合は空文字列を返す．

データソースアトリビュートを読む

データソースアトリビュートの情報は，main() 関数中で作成した TKinokoKdfReader クラスのインスタンス KdfReader を使って，以下のように取得できます．

```
TKinokoKdfReader* KdfReader = new TKinokoKdfReader(DataFileName);
TKinokoDataAnalyzer* Analyzer = new TMyDataAnalyzer();

// データソースにアクセスする前にアナライザの登録を終らせておく
KdfReader->RegisterAnalyzer(DataSourceName, Analyzer);

TKinokoDataSource* DataSource = KdfReader->DataDescriptor()->DataSource(DataSourceName);
if (DataSource != 0) {
    string Fingerprint = DataSource->GetAttributeOf("script_file_fingerprint");
    cout << "ScriptFileFingerprint: " << Fingerprint << endl;
}
```

DataDescriptor() メソッドの呼び出しは，全ての RegisterAnalyzer() 呼び出しが終ってから行なうようにしてください．これは Kinoko の内部構造に起因する制約です．

tagged section のデータを読む

今までの例のデータは全て indexed 型でした．ここでは，DataRecord や taggedRead() などで生成される tagged 型のデータを読みます．

tagged セクションでも，データの読み方には indexed と大きな違いはありません．唯一の違いは，ProcessData() 中の GetNext() で，GetNext(int& Address, int& Data) の代わりに GetNext(string& FieldName, int& Data) を使うことだけです．

```
static const char* SectionName = "adc";

int TMyDataAnalyzer::ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException)
{
    string SectionName = SectionData->SectionName();

    string FieldName;
    int Data;
    while (SectionData->GetNext(FieldName, Data)) {
        cout << SectionName << FieldName << " " << Data << endl;
    }

    return 1;
}
```

データブロックを読む

次に，block section のデータのように要素(データエレメント)に分けられないデータを読む場合や，indexed や tagged でもデータエレメントに分けずにまとめて読みたい場合に使える，データブロックの読み方を説明します．

ここでも，違いはやはり ProcessData() の中身だけです．セクションデータオブジェクトからデータ領域のポインタを取得するには，GetDataBlock() メソッドを用います．また，データブロックのサイズは，DataBlockSize() メソッドで取得できます．

以下は，データブロックを読んで，それを画面に 16 進ダンプする例です．

```
/* DataAnalyzer03.cc */

int TMyDataAnalyzer::ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException)
{
    void* DataBlock = SectionData->GetDataBlock();
    size_t DataBlockSize = SectionData->DataBlockSize();

    // 16 進ダンプ //
    cout << "[" << SectionData->SectionName() << "]";
    cout << hex << setfill('0');
    for (unsigned Offset = 0; Offset < DataBlockSize; Offset++) {
        if (Offset % 16 == 0) {
            cout << endl;
            cout << setw(4) << Offset << ":  ";
        }
        else if (Offset % 8 == 0) {
            cout << "  ";
        }
        cout << setw(2) << (int) ((unsigned char*) DataBlock)[Offset] << " ";
    }
    cout << dec << setfill(' ') << endl;

    return 1;
}
```

CamacAdc.kts で取ったデータに対してこのプログラムを実行すると，結果は以下のようになります，

```
% ./DataAnalyzer03 test03.kdf
[adc]
0000:  00 00 00 00 d2 04 00 00  01 00 00 00 f9 02 00 00
0010:  02 00 00 00 72 06 00 00  03 00 00 00 d4 04 00 00

[adc]
0000:  00 00 00 00 24 04 00 00  01 00 00 00 b0 04 00 00
```

```
0010:  02 00 00 00 e4 05 00 00  03 00 00 00 3b 05 00 00

[adc]
0000:  00 00 00 00 3b 05 00 00  01 00 00 00 1d 07 00 00
0010:  02 00 00 00 35 05 00 00  03 00 00 00 9d 06 00 00

(以下省略)
```

nested section のデータを読む

最後に，unit 文などで生成される nested セクションの読み方を説明します．といっても，プログラム自体には，今までのものとほとんど違いはありません．違いは，セクション名が nested のセクション名に対応したものになるだけです．

例として，以下の読み出しスクリプトで読んだデータを考えます．

```
on trigger(adc) {
    unit event {
        adc.read(#0..#3);
        tdc.read(#0..#3);
    }
}
```

このスクリプトで読んだデータは以下のようになっています．

```
% kdfdump test04.kdf -
# Creator: tinykinoko
# DateTime: 2003-03-09 21:57:05 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/samples
# ScriptFile: CamacAdcTdc.kts


event:adc 0 1736
event:adc 1 1492
event:adc 2 1676
event:adc 3 1086
event:tdc 0 2758
event:tdc 1 2408
event:tdc 2 1877
event:tdc 3 1682

(以下省略)
```

```
% kdfcheck test04.kdf
% Preamble Version: 00010002
% Header Version: 00010001
% Data Area Version: 00010001
% Data Format Flags: 00000000

# Creator: tinykinoko
# DateTime: 2003-03-09 21:57:05 JST
# UserName: sanshiro
# Host: kinoko.awa.tohoku.ac.jp
# Directory: /home/sanshiro/work/kinoko/local/samples
# ScriptFile: CamacAdcTdc.kts
```

```
[2403]--- 0, 436 bytes
Data Descriptor
   datasource "CamacAdcTdc"<2403>
   {
      attribute creator = "tinykinoko";
      attribute creation_date = "2003-03-09 21:57:05 JST";
      attribute script_file = "CamacAdcTdc.kts";
      attribute script_file_fingerprint = "a3ca80e0";

      section "event"<256>: nested {
          section "adc"<256>: indexed(address: int-32bit, data: int-32bit);
          section "tdc"<257>: indexed(address: int-32bit, data: int-32bit);
       }
   }

(以下省略)
```

すなわち，indexed 型のセクション adc, tdc が nested 型のセクション event の中に入っています．

このデータを読むのに必要な変更は，単にセクション名を nested の構造に対応させるだけです．

```
TMyDataAnalyzer::TMyDataAnalyzer(void)
{
    _AdcSectionIndex = AddSection("event:adc");
    _TdcSectionIndex = AddSection("event:tdc");
}
```

プル型データアナライザ

いままでのプログラムは，ProcessData() がフレームワーク側から呼ばれるタイプ(プッシュ型)のアナライザでした．状況によっては，データを読むループをユーザプログラム側で持ち，その中でデータを読み出す(プル型)ようにした方が便利な場合もあります．ここではそのための方法について簡単に考えてみます．

TKinokoKdfReader::Start() を呼び出すと，制御はフレームワークに渡ってしまい，全てのデータを処理し終わるまで返ってきません．TKinokoKdfReader::ProcessNext() は，TKinokoKdfReader::Start() と基本的に同じ処理をしますが，1 パケットの処理が終るごとにリターンするので，これを使うことにより，データに対するループをユーザプログラムの中に書くことができます．実際，以下のように，TKinokoKdfReader::ProcessNext() で空のループを組めば，その処理は TKinokoKdfReader::Start() と同等になります(処理スピードは少し変わります)．

```
while (KdfReader->ProcessNext()) {
   ;
}
```

ただし，一回の ProcessNext() の呼び出しごとにアナライザの ProcessData() が呼ばれるわけではないことに注意してください．ProcessNext() は 1 パケット分を処理しますが，パケットにはデータ以外のものや，他のデータソースや他のデータセクションのデータなどもあるからです．

これを使って，データパケットのループをユーザプログラム側に持ってきた例を以下に示します．

```
/* DataAnalyzer04.cc */

#include <cstdlib>
#include <iostream>
#include <string>
#include "KinokoKdfReader.hh"
#include "KinokoSectionDataAnalyzer.hh"

using namespace std;
```

```
// データは indexed 型のセクション adc に 4 つのエレメントがあるとする
static const char* DataSourceName = "";
static const char* SectionName = "adc";
static const int NumberOfDataElements = 4;


class TMyDataAnalyzer: public TKinokoSectionDataAnalyzer {
  public:
    TMyDataAnalyzer(TKinokoKdfReader* KdfReader);
    virtual ~TMyDataAnalyzer();
    virtual int ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException);
    virtual bool GetData(const int*& Data);
  protected:
    TKinokoKdfReader* _KdfReader;
    int _DataBuffer[NumberOfDataElements];
    bool _IsDataAvailable;
};


TMyDataAnalyzer::TMyDataAnalyzer(TKinokoKdfReader* KdfReader)
: TKinokoSectionDataAnalyzer(SectionName)
{
    _KdfReader = KdfReader;
    _IsDataAvailable = false;
}

TMyDataAnalyzer::~TMyDataAnalyzer()
{
}

// ここでは，データを読んで，内部バッファに格納する
int TMyDataAnalyzer::ProcessData(TKinokoSectionData* SectionData) throw(TKinokoException)
{
    int Address, Data;
    while (SectionData->GetNext(Address, Data)) {
          if (Address < NumberOfDataElements) {
              _DataBuffer[Address] = Data;
          }
    }

    _IsDataAvailable = true;

    return 1;
}

// 内部バッファにあるデータを返す
bool TMyDataAnalyzer::GetData(const int*& Data)
{
    // データがなければ，ファイルから読む
    while (! _IsDataAvailable) {
          if (! _KdfReader->ProcessNext()) {
              return false;
          }
    }

    Data = _DataBuffer;
    _IsDataAvailable = false;

    return true;
}
```

```
int main(int argc, char** argv)
{
    if (argc < 2) {
          cerr << "Usage: " << argv[0] << " DataFileName" << endl;
        return EXIT_FAILURE;
    }
    string DataFileName = argv[1];

    TKinokoKdfReader* KdfReader = new TKinokoKdfReader(DataFileName);
    TMyDataAnalyzer* Analyzer = new TMyDataAnalyzer(KdfReader);

    try {
          KdfReader->RegisterAnalyzer(DataSourceName, Analyzer);

        // データパケットに対するループ
          const int* Data;
          while (Analyzer->GetData(Data)) {
              for (int i = 0; i < NumberOfDataElements; i++) {
                    cout << i << " " << Data[i] << endl;
              }
              cout << endl;
          }
    }
    catch (TKinokoException& e) {
          cerr << "ERROR: " << e << endl;
        return EXIT_FAILURE;
    }

    delete Analyzer;
    delete KdfReader;

    return 0;
}
```

データファイルの読み方 2 - DataProcessor を使う -

KinokoSectionDataAnalyzer を使ってデータファイルを読むのは，手軽ですが，できることには限りがあります．例えば，複数のデータソースからのデータを同時に処理したり，未知のデータソース/データセクションのデータを読むなどという処理は，KinokoDataSectionAnalyzer では多少無理があります．この場合は，もう一つのデータ解析フレームワークである KinokoDataProcessor を使うことになります．

また，DataProcessor を使ったコードは，コンポーネントとしてオンラインストリームに統合することが極めて容易です(もともと DataProcessor はこの目的のために設計されました)．将来オンラインにする予定のデータ処理やオンラインとオフラインの両方で使用したいコードなどは，DataProcessor のフレームワークを使うほうが便利です．

DataProcessor はデータパケットの中身を直接ユーザに見せるため，indexed や tagged のデータをこのフレームワークで扱うことは，多少わずらわしい面がありますが，block 型のデータに関してはむしろ簡単になります．

簡単な例

DataProcessor は，オンラインでデータを処理するためのフレームワークとしてはじめに設計されました．そのため，オンラインストリームにおける 3 つの形態 (Source, Pipe, Sink) に応じて，DataProducer, DataProcessor, DataConsumer の 3 種類があります．ここでは，データを読むだけなので，DataConsumer を使用します．

以下は，データファイルに含まれる全データパケットの，DataSourceId と SectionId およびデータの 16 進ダンプを表示するプログラムです．

```
/* DataProcessor01.cc */

#include <iostream>
#include <iomanip>
#include "MushArgumentList.hh"
#include "KinokoDataProcessor.hh"
#include "KinokoStandaloneComponent.hh"

using namespace std;


class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(void);
    virtual ~TMyDataConsumer();
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
    virtual void OnReceiveTrailerPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
};



TMyDataConsumer::TMyDataConsumer(void)
{
}

TMyDataConsumer::~TMyDataConsumer()
{
}

void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    // DataSourceId と SectionId を取得 //
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);

    // データへのポインタとサイズを取得 //
    void* Data = TKinokoDataSectionScanner::DataAreaOf(DataPacket);
    size_t DataSize = TKinokoDataSectionScanner::DataSizeOf(DataPacket);

    // 16 進ダンプ //
    cout << "[" << DataSourceId << ":" << SectionId << "]";
    cout << hex << setfill('0');
    for (unsigned Offset = 0; Offset < DataSize; Offset++) {
        if (Offset % 16 == 0) {
            cout << endl;
            cout << setw(4) << Offset << ":  ";
        }
        else if (Offset % 8 == 0) {
            cout << "  ";
        }
        cout << setw(2) << (int) ((unsigned char*) Data)[Offset] << " ";
    }
    cout << dec << setfill(' ') << endl;
}

// イベント区切りなどで空行を表示する //
void TMyDataConsumer::OnReceiveTrailerPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    // DataSourceId とトレイラ種別を取得 //
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    int TrailerValue = TKinokoDataStreamScanner::TrailerValueOf(DataPacket);
```

```
    // 空行を表示 //
    cout << "[" << DataSourceId << ":" << TrailerValue << "]" << endl;
    cout << endl;
}


int main(int argc, char** argv)
{
    TMushArgumentList ArgumentList(argc, argv);

    TMyDataConsumer MyDataConsumer;
    TKinokoStandaloneDataConsumer StandaloneDataConsumer(&MyDataConsumer, "MyDataConsumer");

    try {
        StandaloneDataConsumer.Start(ArgumentList);
    }
    catch (TKinokoException &e) {
        cerr << "ERROR: " << e << endl;
    }

    return 0;
}
```

まず，KinokoDataConsumer を継承して，MyDataConsumer をつくり，OnReceiveDataPacket() をオーバーライドします．OnReceiveDataPacket() は，データパケットを受け取るたびに呼び出されるメソッドです．引数には，データパケット自体へのポインタと，データパケットのサイズが渡されます．

データパケットから DataSourceId を得るには，TKinokoDataStreamScanner の静的メソッド DataSourceIdOf() を用います．同様に，SectionId は，TKinokoDataSectionScanner の SectionIdOf() を使います．

TKinokoDataSectionScanner を使うと，パケット中のデータブロックへのポインタと，データブロックのサイズも取得することができます．

main() 関数で TMyDataConsumer のインスタンス MyDataConsumer を作成し，TKinokoStandaloneDataConsumer を使ってシステムと結びつけます．ここで，TKinokoStandaloneDataConsumer の代わりに TKinokoDataConsumerCom を使うと，そのままオンライン用のデータ解析コンポーネントとなります．TKinokoStandaloneDataConsumer にプログラム引数 (ArgumentList) を渡して Start() を呼べば，あとは全て Kinoko が処理をおこないます．

このプログラムのコンパイル手順は，データアナライザを使用する場合と基本的に同じです．

```
# Makefile for sample DataProcessor

CXX = $(shell kinoko-config --cxx)
FLAGS = $(shell kinoko-config --flags)
LIBS = $(shell kinoko-config --libs)

DataProcessor01: DataProcessor.o
        $(CXX) -o $@ $@.o $(LIBS)

.cc.o:
        $(CXX) $(FLAGS) -c $<
```

データデスクリプタを読む

DataProcessor は，データパケットをそのままの形で渡すので，その解釈には通常データデスクリプタが必要です(DataSourceId や SectionId が読み出しスクリプト等で固定されている場合は，この手順は省くことができます)．

TMyDataConsumer を作った際の親クラス TKinokoDataConsumer は，データデスクリプタオブジェクト _InputDataDescriptor を持っており，ここからデータデスクリプタの情報を得ることができます．また，TKinokoDataConsumer は，最初の RunBegin パケットを受け取ったときにメソッド OnRunBegin() を呼ぶので，これを継承してこの中でデータデスクリプタを処理することにします．

クラス定義は以下のようになります．メソッド OnRunBegin() を追加し，メンバで処理対象のデータソース・セクションの名前と ID を保持するようにします．

```
/* DataProcessor02.cc */

#include "KinokoDataSource.hh"
#include "KinokoDataSection.hh"

class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(string DataSourceName, string SectionName);
    virtual ~TMyDataConsumer();
    virtual void OnRunBegin(void) throw(TKinokoException);
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
  protected:
    string _DataSourceName, _SectionName;
    int _DataSourceId, _SectionId;
};
```

コンストラクタで処理対象のデータソース・セクションの名前を受け取ります．DataSouceId と SectionId は無効を示す負の値を入れておきます．

```
TMyDataConsumer::TMyDataConsumer(string DataSourceName, string SectionName)
{
    _DataSourceName = DataSourceName;
    _SectionName = SectionName;

    _DataSourceId = -1;
    _SectionId = -1;
}
```

OnRunBegin() メソッドでは，親クラスである TKinokoDataConsumer が保持している _DataDescriptor オブジェクトから DataSource オブジェクトを，DataSouce オブジェクトから DataSection オブジェクトを順に取得し，それぞれの ID を得ます．

```
void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    // 名前から DataSource オブジェクトを取得 //
    TKinokoDataSource* DataSource = _InputDataDescriptor->DataSource(_DataSourceName);

    if (DataSource != 0) {
        // DataSourceId を取得 //
        _DataSourceId = DataSource->DataSourceId();

        // 名前から DataSection オブジェクトを取得 //
        TKinokoDataSection* DataSection = DataSource->DataSection(_SectionName);

        if (DataSection != 0) {
            // SectionId を取得 //
            _SectionId = DataSection->SectionId();
        }
    }
}
```

OnReceiveDataPacket() では，受け取ったデータパケットの DataSourceId/SectionId を保持している処理対象の DataSouceId/SectionId と比較し，処理対象でない場合はスキップするようにします．

```
void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    // データパケットの DataSourceId を取得し，比較 //
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
        return;
    }

    // データパケットの SectionId を取得し，比較 //
    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _SectionId) {
        return;
    }

    （以下省略）
```

main() 関数では，MyDataConsumer に処理対象のデータソースおよびデータセクションの名前を渡すようにします．

```
int main(int argc, char** argv)
{
    TMyDataConsumer MyDataConsumer("CamacAdc", "adc");

    （以下省略）
```

TKinokoDataDescriptor は，データデスクリプタに書かれた全ての情報を保持しているオブジェクトです．基本的に各データソースの情報を保持する TKinokoDataSource のリストです．TKinokoDataSource は，さらに各セクションの情報を保持する TKinokoDataSection を保持しています．以下は，これらのクラスで利用できるメソッドのリストです．

TKinokoDataDescriptor

TKinokoDataSource* DataSource(long DataSourceId)
　　データソース ID から，データソースオブジェクトを検索し，見つかればそれを返す．見つからない場合は 0 を返す．
TKinokoDataSource* DataSource(const std::string& DataSourceName)
　　データソース名から，データソースオブジェクトを検索し，見つかればそれを返す．見つからない場合は 0 を返す．
const std::vector<TKinokoDataSource*>& DataSourceList(void)
　　保持している全てのデータソースオブジェクトのリストを返す．

TKinokoDataSource

long DataSourceId(void) const
　　データソース ID を返す．
const std::string& DataSourceName(void) const
　　データソース名を返す．
TKinokoDataSection* DataSection(int SectionId)
　　セクション ID から，データセクションオブジェクトを検索し，見つかればそれを返す．見つからない場合は 0 を返す．
TKinokoDataSection* DataSection(const std::string& SectionName)
　　セクション名から，データセクションオブジェクトを検索し，見つかればそれを返す．見つからない場合は 0 を返す．
bool GetAttributeOf(const std::string& Name, std::string& Value)
　　データソースアトリビュートを取得し，Value に入れる．アトリビュートが見つかれば true を，見つからなければ false を返す．

TKinokoDataSection

int SectionType(void) const
　　セクション型に対応する値を返す．以下の enum 値が定義されている．

- TKinokoDataSection::SectionType_Indexed
- TKinokoDataSection::SectionType_Tagged
- TKinokoDataSection::SectionType_Block
- TKinokoDataSection::SectionType_Nested

int SectionId(void) const
　　セクション ID を返す.
const std::string& SectionName(void) const
　　セクション名を返す.
TKinokoDataSource* DataSource(void) const
　　このセクションが属するデータソースのデータソースオブジェクトを返す.

indexed 型のデータを読む

indexed 型のデータを読むためには，データブロックの中身を解読する必要があります．データブロック内のフォーマットをカプセル化し，データブロックからデータエレメントを取り出すインターフェースを提供するものとして，TKinokoIndexedDataSectionScanner クラスがあります．あるセクションデータを読むためのスキャナオブジェクトは，対応するデータセクションオブジェクトにより生成されます．

以下の例では，OnRunBegin() 内でスキャナオブジェクトも取得し，それを OnReceiveDataPacket() で使っています．スキャナオブジェクトは，生成したデータセクションオブジェクトにより削除されるので，ユーザがこれを削除する必要はありません．

```
/* DataProcessor03.cc */

#include "KinokoIndexedDataSection.hh"

class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(string DataSourceName, string SectionName);
    virtual ~TMyDataConsumer();
    virtual void OnRunBegin(void) throw(TKinokoException);
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
  protected:
    string _DataSourceName, _SectionName;
    int _DataSourceId, _SectionId;
    TKinokoIndexedDataSectionScanner* _Scanner;
};
```

```
void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    TKinokoDataSource* DataSource = _InputDataDescriptor->DataSource(_DataSourceName);
    if (DataSource != 0) {
        _DataSourceId = DataSource->DataSourceId();
        TKinokoDataSection* DataSection = DataSource->DataSection(_SectionName);

        if (DataSection != 0) {
            _SectionId = DataSection->SectionId();

            // セクション型を確認し，スキャナオブジェクトを取得する //
            if (DataSection->SectionType() != TKinokoDataSection::SectionType_Indexed) {
                throw TKinokoException("invalid section type");
            }
            _Scanner = ((TKinokoIndexedDataSection*) DataSection)->Scanner();
        }
    }
}
```

```
void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
        return;
    }
```

```
    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _SectionId) {
        return;
    }

    // スキャナオブジェクトを使ってデータエレメントを取りだす //
    int NumberOfElements = _Scanner->NumberOfElements(DataPacket);
    int Address, Data;
    for (int Index = 0; Index < NumberOfElements; Index++) {
        _Scanner->ReadFrom(DataPacket, Index, Address, Data);
        cout << Address << " " << Data << endl;
    }
}
```

tagged 型のデータを読む

tagged 型のセクションを読む場合も，基本的に indexed 型の場合と同様に，Scanner を使用します．ただし，この場合は tagged セクション用の TKinokoTaggedSectionDataScanner になります．

```
/* DataProcessor04.cc */

#include "KinokoTaggedDataSection.hh"

class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(string DataSourceName, string SectionName);
    virtual ~TMyDataConsumer();
    virtual void OnRunBegin(void) throw(TKinokoException);
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
  protected:
    string _DataSourceName, _SectionName;
    int _DataSourceId, _SectionId;
    TKinokoTaggedDataSection* _DataSection;
    TKinokoTaggedDataSectionScanner* _Scanner;
};
```

```
void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    TKinokoDataSource* DataSource = _InputDataDescriptor->DataSource(_DataSourceName);
    if (DataSource != 0) {
        _DataSourceId = DataSource->DataSourceId();
        TKinokoDataSection* DataSection = DataSource->DataSection(_SectionName);

        if (DataSection != 0) {
            _SectionId = DataSection->SectionId();

            // セクション型を確認し，スキャナオブジェクトを取得する //
            // 後で使うため，データセクションオブジェクト自体もとっておく //
            if (DataSection->SectionType() != TKinokoDataSection::SectionType_Tagged) {
                throw TKinokoException("invalid section type");
            }
            _DataSection = (TKinokoTaggedDataSection*) DataSection;
            _Scanner = _DataSection->Scanner();
        }
    }
}
```

```
void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
          return;
    }

    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _SectionId) {
          return;
    }

    // スキャナオブジェクトを使ってデータエレメントを取りだす //
    // セクション構造にかかわる情報はデータセクションオブジェクトから取得していることに注意 //
    int NumberOfFields = _DataSection->NumberOfFields()
    int Data;
    for (int Index = 0; Index < NumberOfElements; Index++) {
        string FieldName = _DataSection->FieldNameOf(Index);
          _Scanner->ReadFrom(DataPacket, Index, Data);
          cout << FieldName << " " << Data << endl;
    }
}
```

あらかじめ処理対象のフィールド名が決まっている場合，OnRunBegin() で FieldIndex を取得しておくと処理が速くなります．

```
/* DataProcessor04b.cc */

void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    TKinokoDataSource* DataSource = _InputDataDescriptor->DataSource(_DataSourceName);
    if (DataSource != 0) {
          _DataSourceId = DataSource->DataSourceId();
          TKinokoDataSection* DataSection = DataSource->DataSection(_SectionName);

          if (DataSection != 0) {
              _SectionId = DataSection->SectionId();

              if (DataSection->SectionType() != TKinokoDataSection::SectionType_Tagged) {
                    throw TKinokoException("invalid section type");
              }
              _DataSection = (TKinokoTaggedDataSection*) DataSection;
              _Scanner = _DataSection->Scanner();

              // データセクションオブジェクトから FieldIndex を取得する //
              _EventNumberIndex = _DataSection->FieldIndexOf("event_number");
              _EventTimeIndex = _DataSection->FieldIndexOf("event_time");
          }
    }
}

void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
          return;
    }
```

```
    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _SectionId) {
          return;
    }

    // FieldIndex を指定してデータエレメントを取得する //
    int Data;
    _Scanner->ReadFrom(DataPacket, _EventNumberIndex, Data);
    cout << "event_number: " << Data << endl;
    _Scanner->ReadFrom(DataPacket, _EventTimeIndex, Data);
    cout << "event_time: " << Data << endl;
}
```

フィールドの数が増えてインデックスの管理が繁雑になった場合には，データ用変数を直接 Scanner に登録してしまう方法もあります(登録する変数の管理に注意を要します).

```
/* DataProcessor04c.cc */

void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    TKinokoDataSource* DataSource = _InputDataDescriptor->DataSource(_DataSourceName);
    if (DataSource != 0) {
          _DataSourceId = DataSource->DataSourceId();
          TKinokoDataSection* DataSection = DataSource->DataSection(_SectionName);

          if (DataSection != 0) {
              _SectionId = DataSection->SectionId();

              if (DataSection->SectionType() != TKinokoDataSection::SectionType_Tagged) {
                    throw TKinokoException("invalid section type");
              }
              _DataSection = (TKinokoTaggedDataSection*) DataSection;
              _Scanner = _DataSection->Scanner();

               // Scanner に変数を登録する //
               // 変数のスコープと寿命に注意 //
              _Scanner->MapVariable("event_number", &_EventNumber);
              _Scanner->MapVariable("event_time", &_EventTime);
          }
    }
}

void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
          return;
    }

    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _SectionId) {
          return;
    }

    // Update() をすると登録した変数に値が反映される //
    _Scanner->Update(DataPacket);
    cout << "event_number: " << _EventNumber << endl;
    cout << "event_time: " << _EventTime << endl;
}
```

TKinokoTaggedDataSetion で利用できるメソッドの一部を以下に示します.

bool HasField(const std::string& FieldName)
　　FieldName の名前を持ったフィールドが存在すれば true を, しなければ false を返す.
int FieldIndexOf(const std::string& FieldName) throw(TKinokoException)
　　FieldName の名前のフィールドのインデクスを返す. フィールドが存在しない場合は例外 TKinokoException を投げる.
std::string FieldNameOf(int FieldIndex) throw(TKinokoException)
　　FieldIndex のインデクスのフィールド名を返す. Index が範囲外の場合は例外 TKinokoException を投げる.
int NumberOfFields(void)
　　フィールドの数を返す.

nested 型のデータを読む

```
datasource CamacAdc
{
    CamacCrate crate;
    CamacController controller("Toyo-CC7x00");
    CamacModule adc("Rinei-RPC022");

    crate.installController(controller);
    crate.installModule(adc, 10);

    DataRecord event_info;
    Register event_number, event_time;

    on trigger(timer) {
        event_number += 1;
        readTime(event_time);

      // unit 文によりネスト型のパケットを生成
        unit event {
            adc.read(#0..#3);
            adc.clear();

            event_info.fill("event_number", event_number);
            event_info.fill("event_time", event_time);
            event_info.send();
        }
    }
}
```

```
datasource "CamacAdc"<3603>
{
    attribute creator = "tinykinoko";
    attribute creation_date = "2008-07-19 07:25:07 JST";
    attribute script_file = "NestedDataSection.kts";
    attribute script_file_fingerprint = "81d9e4ae";

    section "event"<257>: nested {
        section "adc"<258>: indexed(address: int-32bit, data: int-32bit);

        section "event_info"<259>: tagged {
            field "event_number": int-32bit;
            field "event_time": int-32bit;
        }
    }
}
```

```
/* DataProcessor05.cc */

#include <iostream>
#include "MushArgumentList.hh"
#include "KinokoDataSource.hh"
#include "KinokoDataSection.hh"
#include "KinokoIndexedDataSection.hh"
#include "KinokoTaggedDataSection.hh"
#include "KinokoNestedDataSection.hh"
#include "KinokoDataProcessor.hh"
#include "KinokoStandaloneComponent.hh"

using namespace std;


class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(void);
    virtual ~TMyDataConsumer();
    virtual void OnRunBegin(void) throw(TKinokoException);
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
  protected:
    int _DataSourceId;
    int _EventSectionId;
    TKinokoNestedDataSection* _EventSection;
    TKinokoNestedDataSectionScanner* _EventScanner;
    int _EventInfoSectionId;
    TKinokoTaggedDataSection* _EventInfoSection;
    TKinokoTaggedDataSectionScanner* _EventInfoScanner;
    int _AdcSectionId;
    TKinokoIndexedDataSection* _AdcSection;
    TKinokoIndexedDataSectionScanner* _AdcScanner;
};



TMyDataConsumer::TMyDataConsumer(void)
{
}

TMyDataConsumer::~TMyDataConsumer()
{
}

void TMyDataConsumer::OnRunBegin(void) throw(TKinokoException)
{
    string DataSourceName;
    string SectionName;
    TKinokoDataSource* DataSource;
    TKinokoDataSection* DataSection;

    DataSourceName = "CamacAdc";
    DataSource = _InputDataDescriptor->DataSource(DataSourceName);
    if (DataSource == 0) {
        throw TKinokoException("datasouce not found: " + DataSourceName);
    }
    _DataSourceId = DataSource->DataSourceId();

    SectionName = "event";
    DataSection = DataSource->DataSection(SectionName);
    if (DataSection == 0) {
        throw TKinokoException("section not found: " + SectionName);
    }
```

```
    _EventSectionId = DataSection->SectionId();
    _EventSection = (TKinokoNestedDataSection*) DataSection;
    _EventScanner = _EventSection->Scanner();

    // nested 型の DataSection, は DataSource と同様の方法で
    // 保持している DataSection を返す.
    SectionName = "event_info";
    DataSection = _EventSection->DataSection(SectionName);
    if (DataSection == 0) {
        throw TKinokoException("section not found: " + SectionName);
    }
    _EventInfoSectionId = DataSection->SectionId();
    _EventInfoSection = (TKinokoTaggedDataSection*) DataSection;
    _EventInfoScanner = _EventInfoSection->Scanner();

    SectionName = "adc";
    DataSection = _EventSection->DataSection(SectionName);
    if (DataSection == 0) {
        throw TKinokoException("section not found: " + SectionName);
    }
    _AdcSectionId = DataSection->SectionId();
    _AdcSection = (TKinokoIndexedDataSection*) DataSection;
    _AdcScanner = _AdcSection->Scanner();
}

void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    if (DataSourceId != _DataSourceId) {
        cout << "### Unexpected DataSource: ID=" << DataSourceId << endl;
        return;
    }

    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);
    if (SectionId != _EventSectionId) {
        cout << "### Unexpected DataSection: ID=" << SectionId << endl;
        return;
    }

    // nested 型ではネストしているセクションのパケットがデータ
    int DataSize = _EventScanner->DataSizeOf(DataPacket);
    U8bit* SubPacket = (U8bit*) _EventScanner->DataAreaOf(DataPacket);

    int SubPacketSize;
    int ProcessedSize = 0;
    while (ProcessedSize < DataSize) {
        int SectionId = TKinokoDataSectionScanner::SectionIdOf(SubPacket);

        if (SectionId == _EventInfoSectionId) {
            int NumberOfFields = _EventInfoSection->NumberOfFields();
            int Data;
            for (int Index = 0; Index < NumberOfFields; Index++) {
                string FieldName = _EventInfoSection->FieldNameOf(Index);
                _EventInfoScanner->ReadFrom(DataPacket, Index, Data);
                cout << FieldName << " " << Data << endl;
            }
        }
        else if (SectionId == _AdcSectionId) {
            int NumberOfElements = _AdcScanner->NumberOfElements(DataPacket);
            int Address, Data;
            for (int Index = 0; Index < NumberOfElements; Index++) {
                _AdcScanner->ReadFrom(DataPacket, Index, Address, Data);
                cout << Address << " " << Data << endl;
            }
```

```
        }
        else {
            cout << "### Unexpected DataSection: ID=" << SectionId << endl;
        }

        SubPacketSize = TKinokoDataSectionScanner::PacketSizeOf(SubPacket);
        ProcessedSize += SubPacketSize;
        SubPacket += SubPacketSize;
    }
}


int main(int argc, char** argv)
{
    TMushArgumentList ArgumentList(argc, argv);

    TMyDataConsumer MyDataConsumer;
    TKinokoStandaloneDataConsumer StandaloneDataConsumer(&MyDataConsumer, "MyDataConsumer");

    try {
        StandaloneDataConsumer.Start(ArgumentList);
    }
    catch (TKinokoException &e) {
        cerr << "ERROR: " << e << endl;
    }

    return 0;
}
```

データストリームの読み方 – DataProcessor をコンポーネントにする –

前章で説明した DataProcessor を使ってデータをファイルから読むプログラムが作成できていれば，これを実行中の Kinoko のデータストリームに接続して動作するコンポーネントにするのは極めて容易です．このために必要な変更は，main() の中の TKinokoStandaloneDataConsumer を TKinokoDataConsumerCom に変更し，TKcomProcess のインスタンスをひとつ作成してこれに渡すだけです．コンポーネントのデバッグは煩わしいことが多いので，このように一度スタンドアロンのプログラムを作成しテストしてからコンポーネント化するのが楽です．

以下は，前章の例 DataProcessor01 をコンポーネント化したものです． 行った変更は，インクルードファイルの追加を除き，わずか3行です．

```
/* MyDataProcessor-kcom.cc */

#include <iostream>
#include <iomanip>
#include "MushArgumentList.hh"
#include "KinokoDataProcessor.hh"
#include "KinokoDataProcessorCom.hh"
#include "KcomProcess.hh"

using namespace std;


class TMyDataConsumer: public TKinokoDataConsumer {
  public:
    TMyDataConsumer(void);
    virtual ~TMyDataConsumer();
    virtual void OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException);
};
```

```
TMyDataConsumer::TMyDataConsumer(void)
{
}

TMyDataConsumer::~TMyDataConsumer()
{
}

void TMyDataConsumer::OnReceiveDataPacket(void* DataPacket, long PacketSize) throw(TKinokoException)
{
    // DataSourceId と SectionId を取得 //
    int DataSourceId = TKinokoDataStreamScanner::DataSourceIdOf(DataPacket);
    int SectionId = TKinokoDataSectionScanner::SectionIdOf(DataPacket);

    // データへのポインタをサイズを取得 //
    void* Data = TKinokoDataSectionScanner::DataAreaOf(DataPacket);
    size_t DataSize = TKinokoDataSectionScanner::DataSizeOf(DataPacket);

    // 16 進ダンプ //
    cout << "[" << DataSourceId << ":" << SectionId << "]";
    cout << hex << setfill('0');
    for (unsigned Offset = 0; Offset < DataSize; Offset++) {
        if (Offset % 16 == 0) {
            cout << endl;
            cout << setw(4) << Offset << ":  ";
        }
        else if (Offset % 8 == 0) {
            cout << " ";
        }
        cout << setw(2) << (int) ((unsigned char*) Data)[Offset] << " ";
    }
    cout << dec << setfill(' ') << endl;
}


int main(int argc, char** argv)
{
    TMushArgumentList ArgumentList(argc, argv);

    TMyDataConsumer MyDataConsumer;
    TKinokoDataConsumerCom DataConsumerCom(&MyDataConsumer);
    TKcomProcess ComProcess(&DataConsumerCom);

    try {
        ComProcess.Start(ArgumentList);
    }
    catch (TKinokoException &e) {
        cerr << "ERROR: " << e << endl;
    }

    return 0;
}
```

KCOM の規約により，ファイル名は [コンポーネント名-kcom] でなければなりません． 作成したコンポーネントのインターフェース宣言は以下のようになります．

```
% ./MyDataProcessor-kcom
//
// KinokoDataConsumerCom.kidl
//
// Kinoko Data-Stream Component
// DataConsumer: User Data Process Component
```

```
component KinokoDataConsumerCom {
    property string component_type;
    property string host;
    property long condition_check_time;
    property long heart_beat_rate;
    property long vital_level;
    property string stream_type;
    property string state;
    property int port_number;
    uses KinokoLogger logger;
    accepts system();
    accepts enableConditionMonitor();
    accepts setSource();
    accepts setSink();
    accepts setSourceSink();
    accepts connect();
    accepts construct();
    accepts destruct();
    accepts disconnect();
    accepts halt();
    accepts quit();
}
```

このコンポーネントをシステムに組込みデータストリームに接続するには，[SmallKinoko の構造と拡張] で説明している方法で KCOM スクリプトを編集します．他のデータストリームコンポーネント同様，setSource() で接続元を指定してから connect() し，construct() をします．終了時には destruct() してから disconnect() し，quit() をします．

SmallKinoko.kcom を編集して MyDataProcessor を接続した例が kinoko/tutorial/DataAnalyzer/MyDataProcessor.kcom にあるので参考にしてください．SmallKinoko.kcom からの変更はわずか8行の追加です(viewer の後ろに同名のメソッドを加えているだけ)．SmallKinoko.kcom との diff をとると必要な変更を一覧できます．

```
% diff $KINOKO_ROOT/scripts/SmallKinoko.kcom MyDataProcessor.kcom
14a15
> import MyDataProcessor;
117a119
> component MyDataProcessor my_processor(host);
320a323
>       my_processor.setSource(buffer);
327a331
>       my_processor.connect();
416a421
>       my_processor.construct();
470a476
>       my_processor.destruct();
493a500
>       my_processor.disconnect();
505a513
>       my_processor.quit();
```

[SmallKinoko の構造と拡張] で説明しているように，KCOM スクリプトは kinoko コマンドで実行します．コンポーネント実行ファイルを Kinoko 標準でない場所に置く場合は，環境変数 KCOM_PATH でその場所を指定してください．コロン区切りで複数のパスを指定できます．

```
% echo $KCOM_PATH
.:/home/kinoko/kinoko/bin          # KCOM_PATH にはカレントディレクトリが含まれている
% kinoko MyDataProcessor.kcom
```

この例では，データストリームから受け取ったデータを単純に標準出力に書き出しているだけですが，これをデータストレージに書き出したり描画ツールに渡したりすることにより様々な応用ができます．受け取ったデータを加工して Kinoko に戻す方法は拡張方法の章で説明します．

camdrv

- CAMAC device driver for Linux 2.x -

English

camdrv は，Linux 上で東陽テクニカ製 CAMAC コントローラ CC-7700 / CC-7000 (PCI/ISA) および豊伸電子製 CAMAC コントローラ CCP (PCI/ISA) を使用するためのデバイスドライバです．完全なカーネルモードで動作するローダブルモジュールで，割り込み待ちなどの高度な機能も実装しています．さらに，広く使われている幾つかの CAMAC ライブラリと互換のライブラリも用意しましたので，既存のプログラムにほとんど手を加えずに Linux 環境に移行することができます．

作成には充分注意していますが，間違い等が含まれている可能性があります．動作の保証はできませんので，御了承下さい．なお，ライセンスは "LGPL Version 2.1" とします．LGPL2.1 の詳細については，配布パッケージに含まれている COPYING-LGPL2.1 ファイル を参照してください．

camdrv は，分散型汎用オンライン環境の構築を目指す KiNOKO プロジェクトの一部として作成されました．KiNOKO プロジェクトの詳細については，KiNOKO ホームページ を御覧下さい．

動作環境

現在利用可能なモデルと Linux のバージョンは以下のとおりです．

OS	CC/7700-PCI	CC/7700-ISA	CC/7000-ISA	CCP (PCI/ISA)	CCP (PCI-2703A)
Linux 2.0.x	○	○	○	×	×
Linux 2.2.x	○	○	○	○	×
Linux 2.4.x	○	○	○	○	○
Linux 2.6.x	○	○	○	○	○

- マルチクレート機能 (setcn()/CSETCR()) が利用できるのは Linux 2.4 以降用のもののみです．
- 豊伸電子の CCP は，ISA と PCI で同じドライバを利用できます．
- 豊伸電子の CCP で，「変換アダプタ」を利用するタイプのものは，CCP_PCI_2703 のドライバを使用してください．IO ポートの設定などは必要ありません．

- 2002 年 5 月以降に発売された豊伸電子の CCP-PCI (基板に PCI2 と書かれているもの) は，以下のドライバが camdrv と同じインターフェースで利用できるようです．

  http://psux1.kek.jp/~omata/linux_KODAQ/drv_hoshin_pci2/index.html

動作を確認した Linux ディストリビューションは，以下のとおりです．

kernel 2.6

Fedora Core 6
　　ドライバをコンパイルする前に，使用しているカーネルに対応する kernel-devel パッケージをインストールしてください．
Fedora Core 5
　　ドライバをコンパイルする前に，使用しているカーネルに対応する kernel-devel パッケージをインストールしてください．
Fedora Core 2 / 3 / 4
　　そのままコンパイルできます．

kernel 2.4

Fedora Core 1
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください.
Red Had Linux 8.0
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください.
Vine Linux 2.5
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/include に設定してください.
Red Had Linux 7.1J / 7.3
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください.

kernel 2.2

Red Had Linux 7.0J
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux/include に設定してください.
Red Had Linux 6.2 / 6.2J
　　そのままコンパイルできます.
Red Had Linux 6.1 / 6.1J
　　そのままコンパイルできます.
Kondara MNU/Linux 2000
　　そのままコンパイルできます.
Vine Linux 2.1.5
　　そのままコンパイルできます.
Vine Linux 2.0
　　そのままコンパイルできます.
Debian GNU/Linux (woody)
　　そのままコンパイルできます.
　　もしかしたら, kernel-headers パッケージをインストールする必要があるかもしれません.
Debian GNU/Linux (potato)
　　そのままコンパイルできます.
　　もしかしたら, kernel-headers パッケージをインストールする必要があるかもしれません.

ダウンロードとフィードバック

- ダウンロード
  - Version 1.0.1
  - Version 1.0.0
- フィードバック
  - 関連情報掲示板
  - メールによる問い合わせ

質問などは, なるべく掲示板のほうにおねがいします(匿名でかまいません). 同じ内容の質問が多くなっていますので, 御協力をおねがいします.

また, トラブルに関する質問の際は, 使用している Linux ディストリビューションの種類・バージョンやデバイス構成・型番, エラーメッセージや dmesg コマンドの出力など, なるべく詳細な情報を記述してください.

インストール

1. パッケージを展開し, ドライバソースのあるディレクトリに移動します.

```
% gunzip camdrv-?.?.?.tar.gz
% tar xvf camdrv-?.?.?.tar
% cd camdrv/Linux2.6_CC77pci
```

   ドライバソースのディレクトリ名は, 使用している Linux のバージョンや CAMAC コントローラのモデルにより適当に読みかえて下さい.

2. 2.4.x 以前のバージョンの Linux カーネルを使用している場合は，環境に合わせて，Makefile を編集する必要があります．Makefile の先頭付近にある以下の変数を設定してください．開発者がテストした環境での設定が上記の動作環境のリストに書いてありますので，参考にしてください．

   KERNEL_INCLUDE_DIR
   「カーネルヘッダファイル」のあるディレクトリです．動作中のカーネルがコンパイルされたときに使用されたヘッダファイルと厳密に同じものでなければなりません．
   一部のディストリビューションでは，kernel-headers パッケージに入っているカーネルヘッダが実際にカーネルのコンパイルに使用されたものと違う場合があります．このような場合は kernel-source パッケージをインストールし，その中のヘッダファイルを指定してください(カーネルソースのコンパイルは必要ありません)．
   自分でカーネルを再構築した場合は，その際に使用したソースのヘッダファイルのディレクトリを指定してください．

   USE_MODVERSIONS
   動作中のカーネルが MODVERSIONS を有効にしてコンパイルされたものなら 1 を，そうでない場合は 0 を指定してください．最近のほとんどのディストリビューションでは，MODVERSIONS が使われているようです(″1″ を指定する)．
   自分でカーネルの再構築を行なった場合は，そのときの設定に従ってください．
   現在のカーネルが MODVERSIONS を使っているかは，/proc/ksyms を cat してみて，表示されたカーネル関数にバージョン情報が付加されているかを見れば分かります．

   ```
   % cat /proc/ksyms | grep kmalloc
   c012e880 kmalloc_R93d4cfe6            <-- MODVERSION が使われている

   % cat /proc/ksyms | grep kmalloc
   c012e880 kmalloc                      <-- MODVERSION が使われていない
   ```

3. CC-7x00/ISA または CCP を使用している場合，I/O ポートおよび IRQ 番号の手動設定が必要になります (CCP では IRQ の設定は必要ありません)．空いている I/O ポートおよび IRQ 番号を調べ，モジュールのジャンパを設定して，Makefile を修正してください．Makefile の先頭に次のような記述があるので，この部分を書き換えます．

   ```
   IOPORT = 0x0c00
   IRQ = 10
   ```

   PCI デバイスが使用している I/O ポートや IRQ 番号などは，以下のいずれかのコマンドによって調べることができます．

   ```
   % /sbin/lspci -v
   % cat /proc/pci
   ```

   そのほか，完全ではありませんが，以下のコマンドは I/O ポートや IRQ の使用状況を調べるのに役立ちます．

   ```
   % cat /proc/ioports
   % cat /proc/irq
   % cat /proc/interrupts
   % cat /proc/stats
   ```

   情況によっては，BIOS による I/O ポートや IRQ の設定が有用(または必要)な場合もあります．

4. ドライバをコンパイルします．

   ```
   % make
   ```

5. root になって，ドライバをシステムに登録します．dmesg で成功したかを確認できます．

   ```
   % su
   # make install
   # dmesg
         いろいろなメッセージ
   camdrv: at 0x0c00 on irq 10 (major = 126).
   #
   ```

6. Linux をリブートしたときは，もう一度ドライバを登録しなければなりません．

   ```
   # make install
   ```

   リブート時に，自動でドライバを組み込む方法については，KiNOKO-DAQ Technical Tips を参照してください．

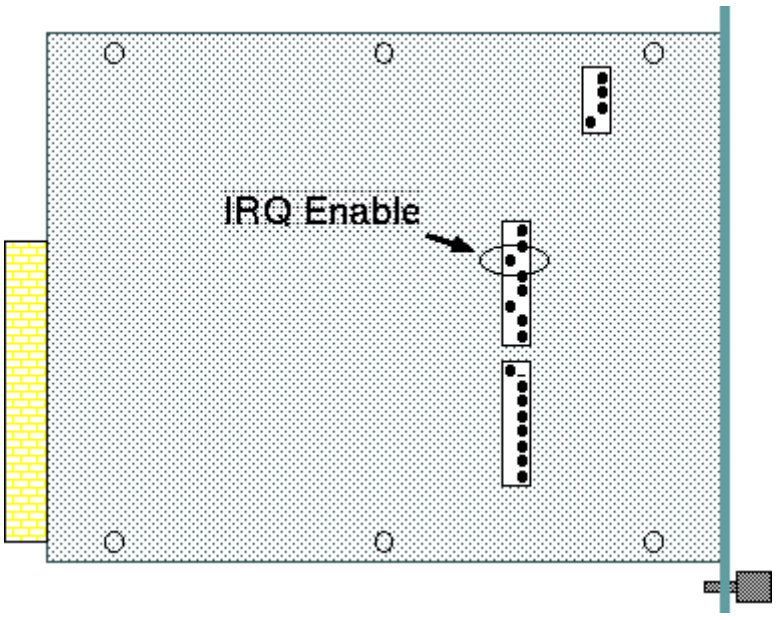
7. システムからドライバを取り除きたい場合，以下のようにします．

```
# make uninstall
# dmesg
        いろいろなメッセージ
camdrv: removed.
#
```

開発者が動作確認をしたバージョンよりも新しいバージョンの Linux を使用した場合，コンパイルやインストールに失敗するかもしれません．その場合は，可能な限り対処しますので，上記のフィードバックのリンクより開発者に連絡をください．

ハードウェア設定

CC-7000 を使用している場合，LAM の割り込み待ちを行なうために，コントローラ本体のジャンパスイッチを「割り込み ON」に設定する必要があります．また，動作モードが 16bit に設定されている必要があります(これは工場出荷時の設定値ですが，一応確認してください)．



なお，フロントパネルの REQ と G-IN を LEMO ケーブルで接続し，トグルスイッチを ON LINE 側にすることも忘れないように注意してください．

動作テスト

camdrv には，いくつかのテストプログラムが用意されています．これをコンパイルして，動作テストをおこないます． camdrv を展開したディレクトリで，make をしてください．

```
% cd ..
% make
```

ドライバ設定のテスト

コントローラに inhibit を送って，フロントパネルの I の LED が点灯するかをチェックします．

```
% ./inhibit_test
press ENTER to set inhibit...
press ENTER to release inhibit...
%
```

ENTER を押して，クレートコントローラの LED が点灯するかを確認してください． もし点灯しなかった場合，I/O ポートが正しく設定されていないことが考えられます． ジャンパの設定とドライバのパラメータが一致しているか，I/O ポートの割り当て範囲が他のボードと衝突していないかなどを確認してください (PCI 版の camdrv は I/O ポートの割当てを自動でおこないます)．

割り込み設定のテスト

LAM を発生させるモジュールをクレートに差し，割り込みを発生させます． 1Hz 程度で LAM を発生させ，正しく LAM を待っていることと，正しく LAM を拾っていることを確認してください． このテストプログラムは，3 秒以上 LAM が立たないと，timed out を返します．これも確認してください．

(LAM を発生させるために F26 が必要なモジュールを使う場合は，F26 の CAMAC アクションを初期化の直後で一回行うようにプログラムを編集する必要があります．)

```
% ./lam_test
Waiting LAM ...OK.
Waiting LAM ...OK.
Waiting LAM ...timed out.    <- データ入力を止めた
Waiting LAM ...timed out.
Waiting LAM ...OK.           <- ふたたび接続
(16 回繰り返す)
%
```

もし LAM が立っているにもかかわらず timed out する場合は，割り込み回線が正しく設定されていないことが考えられます．ジャンパの設定とドライバのパラメータが一致しているか，IRQ の設定値が他のモジュールと衝突していないかなどを確認してください (PCI 版の camdrv は IRQ の割当てを自動でおこないます).

ISA のコントローラを使用している場合，他のデバイスの影響で，割り込みがうまく処理されない場合があるようです．このようなときは，IRQ を別の値にして試してみて下さい.

もし，LAM が立っていないのに OK と表示されれば，それは camdrv のバグ(かモジュールの故障)です．その場合は，可能な限り対処しますので，上記のフィードバックのリンクより開発者に連絡をください.

CAMAC アクションのテスト

LAM を発生させるモジュールを用いて，LAM を立てたモジュールからの読みだしを行ないます．LAM は，コントローラの LAM フラグを見ます(従って，割り込み処理は必要ありません)．データの読みだしは，ファンクション F0 によって行ないます．アドレスは 0 から始め，Q レスポンスがある間インクリメントしていきます.

このテストプログラムは割り込みを使用しないので，割り込みの取得に失敗していても実行することができます．ただし，LAM を発生させるために F26 が必要なモジュールを使う場合は，F26 の CAMAC アクションを初期化の直後で一回行うようにプログラムを編集する必要があります.

```
% ./camaction_test
[00:04:00] 110
[00:04:01] 104
[00:04:02] 110
[00:04:03] 108
（しばらくつづく）
%
```

出力フォーマットは，"[イベント番号:モジュールアドレス(N):内部アドレス(A)] データ" です．ここでも，1Hz 程度の LAM でタイミングを確認してください．また，適当な信号を入力し，正しい値が読み込まれているかも確認してください．上の 2 つのテストに成功していて，ここで失敗したら，それはおそらく camdrv のバグ(かモジュールの故障)です．その場合は，可能な限り対処しますので，上記のフィードバックのリンクより開発者に連絡をください.

使用方法

camdrv は，汎用データ収集システム KiNOKO から利用すると便利です．詳細は，KiNOKO ホームページ を参照してください.

camdrv は，KiNOKO から利用すると便利ですが，広く使われている CAMAC ライブラリと互換のライブラリも用意しました.

KEK 標準 CAMAC ライブラリ (camlib) 互換ライブラリ

KEK が定めた CAMAC ライブラリの標準仕様に従ったものです.
使用するには，
- ヘッダファイル "camlib.h" をインクルードし，
- オブジェクトファイル "camlib.o" をリンクしてください.
- ソースは "camlib.c" です.

相違点は，
- ブロック処理はサポートしていません.
- リスト処理はサポートしていません.
- CWLAM(int mask) の mask はダミー引数で，LAM のマスキングはサポートしていません.

ライブラリの詳細は，
- KEKオンライン部のCAMACのページを参照してください.

東陽テクニカ製 DOS/WIN 用 CAMAC ライブラリ 互換ライブラリ

製品に付属する東陽テクニカ製のライブラリと同じ関数を実装したものです.
使用するには，
- ヘッダファイル "toyocamac.h" をインクルードし，
- オブジェクトファイル "toyocamac.o" をリンクしてください.
- ソースは "toyocamac.c" です.

相違点は,
ヘッダファイル名が違います:-).
SetIOP()/GetIOP() は必要ないので, ありません.
SetWait()/GetWait() はたぶん必要ないので, ありません.
SetLIN()/GetLIN() はサポートしていません.
RStat() はサポートしていません.
ブロック処理はサポートしていません.
ライブラリの詳細は,
製品付属のマニュアルを参照してください.
動作テストで用いたプログラム "camaction_test.c" がサンプルとして役立ちます.

既知の不具合

- CC/7700-ISA で, Q, X レスポンスが正しく取得できないことがある.
これは...ドライバの責任では...ないような...気が...しなくも...ない...

vmedrv

- VME device driver for Linux 2.x -

English

vmedrv は，Linux 上で SBS Technologies (Bit3) 社製 PCI-VME アダプタ (Model 616/617/618/620) を使用するためのデバイスドライバです．完全なカーネルモードで動作するローダブルモジュールで，プログラム I/O，メモリマップドアクセス(mmap())，VME 割り込み，DMA 転送など，VME アクセスに使われる全ての機能が使用可能です．

作成には充分注意していますが，間違い等が含まれている可能性があります．動作の保証はできませんので，御了承下さい．なお，ライセンスは "LGPL Version 2.1" とします．LGPL2.1 の詳細については，配布パッケージに含まれている COPYING-LGPL2.1 ファイル を参照してください．

vmedrv は，分散型汎用オンライン環境の構築を目指す KiNOKO プロジェクトの一部として作成されました．KiNOKO プロジェクトの詳細については，KiNOKO ホームページ を御覧下さい．

動作環境

現在実装されている機能と Linux のバージョンは以下のとおりです．

OS	PIO	mmap()	割り込み	DMA	Non-Block DMA	poll()	利用可能モデル
Linux 2.0.x	read()のみ	○	○	Read()のみ	-	-	SBS 617
Linux 2.2.x	○	○	○	○	-	-	SBS 617/618/620/620-3
Linux 2.4.x	○	○	○	○	-	-	SBS 616/617/618/620/620-3
Linux 2.6.x	○	○	○	○	○	○	SBS 616/617/618/620/620-3

動作を確認した Linux ディストリビューションは，以下のとおりです．

kernel 2.6

Fedora-Core 6
　　ドライバをコンパイルする前に，使用しているカーネルに対応する kernel-devel パッケージをインストールしてください．
Fedora-Core 5
　　ドライバをコンパイルする前に，使用しているカーネルに対応する kernel-devel パッケージをインストールしてください．
Fedora-Core 4
　　そのままコンパイルできます．
Fedora-Core 3
　　ドライバの Makefile 中の SYSTEM を FC3 に設定してください．
Fedora-Core 2
　　ドライバの Makefile 中の SYSTEM を FC2 に設定してください．

kernel 2.4

Fedora-Core 1
　　ドライバをコンパイルする前に，kernel-source パッケージをインストールしてください．
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください．
Red Had Linux 8.0
　　ドライバをコンパイルする前に，kernel-source パッケージをインストールしてください．
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください．
　　ドライバの Makefile 中の SYSTEM を REDHAT に設定してください．
Vine Linux 2.5
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/include に設定してください．

Red Had Linux 7.1J
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux-2.4/include に設定してください.

kernel 2.2

Red Had Linux 7.0J
　　ドライバをコンパイルする前に, kernel-source パッケージをインストールしてください.
　　ドライバの Makefile 中の KERNEL_INCLUDE_DIR を /usr/src/linux/include に設定してください.
Red Had Linux 6.2 / 6.2J
　　そのままコンパイルできます.
Kondara MNU/Linux 2000
　　そのままコンパイルできます.
Vine Linux 2.1.5
　　そのままコンパイルできます.
Vine Linux 2.0
　　そのままコンパイルできます.
Debian GNU/Linux (woody)
　　そのままコンパイルできます.
　　もしかしたら, kernel-headers パッケージをインストールする必要があるかもしれません.
Debian GNU/Linux (potato)
　　そのままコンパイルできます.
　　もしかしたら, kernel-headers パッケージをインストールする必要があるかもしれません.

ダウンロードとフィードバック

- ダウンロード
  - Version 1.1.1
  - Version 1.1.0
- フィードバック
  - 関連情報掲示板
  - メールによる問い合わせ

質問などは, なるべく掲示板のほうにおねがいします(匿名でかまいません). 同じ内容の質問が多くなっていますので, 御協力をおねがいします.

また, トラブルに関する質問の際は, 使用している Linux ディストリビューションの種類・バージョンやデバイス構成・型番, エラーメッセージや dmesg コマンドの出力など, なるべく詳細な情報を記述してください.

ハードウェア設定

SBS Model-617/618/620 の工場出荷時の設定は以下のようになっています.

- アダプタは VME のシステムコントローラの機能を持たない.
- VME の割り込みを PCI に伝達しない.
- アダプタの I/O レンジは 2000 - 201F (hex)

つまり, 他にコントローラがあって, 割り込みの取得を行なわないなら, そのまま使えるということです.

アダプタを VME のコントローラとして使うためには, SYS ブロックにある以下のジャンパを設定する必要があります.

- Jumper 4: [接続]: VME のバックプレーンに SYSCLK を供給する.
- Jumper 5: [接続]: VME のバックプレーンの SYSRESET をコントロールする.
- Jumper 6: [接続]: タイムアウト(48usec)が発生したときに BUSERR を発行する.

アダプタをコントローラとして使う場合は, アダプタをバスアービタとして機能させるために, BGI-BGO ブロックにある以下のジャンパも設定しなければなりません.

- Jumper ARB: [接続]: アダプタをアービタにする.
- Jumper P/R: [選択]: ラウンドロビンアービトレーションを行なう場合に接続する. 開放の場合はプライオリティアービトレーションとなる. 特に理由がなければ開放でいいと思う(工場設定値).

vmedrv で VME の割り込みを取得したい場合，VME の割り込みを PCI に伝達させるため，T-INT ブロックにある以下のジャンパを接続します．

- Jumper 1-7: [接続]: 7 つのジャンパがそれぞれ VME の IRQ 1-7 に対応する．接続した IRQ の割り込みが PCI に伝達される．特に理由がなければ全て接続していいと思う．

アダプタをコントローラとして使う場合，アダプタはクレートのいちばん左のスロットで使わなければなりません． また，一部の高級なクレートを除いて，モジュールのささっていない全てのスロットに対し，バックプレーンのジャンパ (BG[0-3]-IN/BG[0-3]-OUT, IACK-IN/IACK-OUT) を接続しなければならないことに注意してください． 逆に，割り込みを発行するモジュールの裏の IACK-IN/IACK-OUT のジャンパをはずすことを忘れないように特に注意してください．

インストール方法

1. ドライバのインストールの前に，VME アダプタを PC に接続し，VME の電源を入れておいてください．(Linux kernel 2.6 用の vmedrv では，あらかじめ電源を入れておく必要はなくなりました)
2. パッケージを展開し，ドライバソースのあるディレクトリに移動します．

```
% gunzip vmedrv-1.0.0.tar.gz
% tar xvf vmedrv-1.0.0.tar
% cd vmedrv/Linux2.6_Bit3_617
```

バージョン番号の部分は使用している vmedrv のバージョン，Linux のバージョンに置き換えてください．Model 616/618/620 でも 617 と同じコードを使用します．

3. 2.4.x 以前のバージョンの Linux カーネルを使用している場合は，環境に合わせて，Makefile を編集する必要があります．Makefile の先頭付近にある以下の変数を設定してください．開発者がテストした環境での設定が上記の動作環境のリストに書いてありますので，参考にしてください．

KERNEL_INCLUDE_DIR
「カーネルヘッダファイル」のあるディレクトリです．動作中のカーネルがコンパイルされたときに使用されたヘッダファイルと厳密に同じものでなければなりません．
一部のディストリビューションでは，kernel-headers パッケージに入っているカーネルヘッダが実際にカーネルのコンパイルに使用されたものと違う場合があります．このような場合は kernel-source パッケージをインストールし，その中のヘッダファイルを指定してください(カーネルソースのコンパイルは必要ありません)．
自分でカーネルを再構築した場合は，その際に使用したソースのヘッダファイルのディレクトリを指定してください．

USE_MODVERSIONS
動作中のカーネルが MODVERSIONS を有効にしてコンパイルされたものなら 1 を，そうでない場合は 0 を指定してください．
最近のほとんどのディストリビューションでは，MODVERSIONS が使われているようです(″1″ を指定する)．
自分でカーネルの再構築を行なった場合は，そのときの設定に従ってください．
現在のカーネルが MODVERSIONS を使っているかは，/proc/ksyms を cat してみて，表示されたカーネル関数にバージョン情報が付加されているかを見れば分かります．

```
% cat /proc/ksyms | grep kmalloc
c012e880 kmalloc_R93d4cfe6              <-- MODVERSION が使われている
```

```
% cat /proc/ksyms | grep kmalloc
c012e880 kmalloc                        <-- MODVERSION が使われていない
```

SYSTEM (Linux 2.4 版のみ)
RedHat Linux を使用している場合は SYSTEM=REDHAT，そうでない場合は OTHERS を指定してください．これは，RedHat が独自に行なったカーネルの変更に対応するためのものです．特に RedHat 9 で必要です．Fedora-Core では必要ありません．
RedHat を使用している場合でも，カーネルソースを RedHat 以外から持って来て，それをコンパイルして使用している場合は OTHERS となります．

4. ドライバをコンパイルします．

```
% make
```

5. root になって，ドライバをシステムに登録します．dmesg で成功したかを確認できます．

```
% su
# make install
# dmesg
```

```
        いろいろなメッセージ
vmedrv: Bit3 617 Bus Adapter was detected at ioport 0xdcc0 on irq 10.
  I/O Mapped Node at 0xdcc0.
  Memory Mapped Node at 0xfe010000.
  Mapping Register at 0xfe000000.
  Remote Memory at 0xf8000000.
vmedrv: successfully installed at 0xdcc0 on irq 10 (major = 127).
#
```

6. Linux をリブートしたときは，もう一度ドライバを登録しなければなりません．

```
# make install
```

リブート時に，自動でドライバを組み込む方法については，KiNOKO-DAQ Technical Tips を参照してください．Linux kernel 2.4 以前用の vmrdrv では，ドライバインストール時に VME クレートの電源が入っている必要があることに注意してください．

7. システムからドライバを取り除きたい場合，以下のようにします．

```
# make uninstall
# dmesg
        いろいろなメッセージ
vmedrv: removed.
#
```

作者が動作確認をしたバージョンよりも新しいバージョンの Linux を使用した場合，コンパイルやインストールに失敗するかもしれません．その場合は，可能な限り対処しますので，上記のフィードバックのリンクより開発者に連絡をください．

動作テスト

vmedrv には，いくつかのテストプログラムが用意されています．これをコンパイルして，動作テストをおこないます．ただし，使用しているモジュールにより，若干の書き換えが必要です．

```
% cd ..
% 少し書き換え
% make
```

以下は，用意されているテストプログラムの一覧です．引数なしで実行すると簡単な使用方法を表示します．

vmeread.c / vmewrite.c	Programmed I/O のテストプログラム
vmemap.c	メモリマップトアクセスのテストプログラム
vmedmaread.c / vmedmawrite.c	DMA 転送のテストプログラム
vmeint.c / vmeintwait.c / vmeintcheck.c	割り込みのテストプログラム
vmepoll.c	ポーリング (poll()/select() システムコール) のテストプログラム

vmeread.c, vmewrite.c, vmemap.c, vmedmaread.c, vmedmawrite.c の実行には，VMEのメモリボード(またはそのようにアクセスできるモジュール)が必要です．また，これらのプログラムのディフォルトの転送モードはアドレス 32bit，データ 32bit です．転送モードを変更する場合には，プログラムの先頭付近にある以下の部分を変更して，再コンパイルしてください．

```
#define DEV_FILE "/dev/vmedrv32d32"
```

vmeint.c, vmeintwait.c, vmeintcheck.c, vmepoll.c を実行するには，VME割り込みを発行できるモジュールが必要です．また，それらのモジュールのレジスタを適切に設定して割り込みを発生させるコードも書かなければなりません．テストプログラム中では，"test-interrupter-XXX.c" を #include し，その中で定義されている以下の関数を使用しています．テストプログラムでは林栄精器の RPV-130 を使用していますが，それ以外のモジュールを使用する場合はこれらの関数を置き換えてコンパイルしなおす必要があります．

```
test_interrupter_enable()
test_interrupter_disable()
test_interrupter_clear()
```

使用方法

vmedrv は，汎用データ収集システム KiNOKO から利用すると便利です． 詳細は，KiNOKO ホームページ を参照してください．

vmedrv は，KiNOKO から利用すると便利ですが，ドライバに直接アクセスして使うこともできます．上記の動作確認に用いたプログラムは，ドライバに直接アクセスする方法を示すサンプルとなっています（サンプルとして読みやすいように書いたつもりです）．

配布パッケージに含まれている vmeslib は，KEK vmelib と類似のインターフェースを実装したものです．vmelib を使っていた場合には，比較的容易に移行できます．ライブラリファイルとテストプログラムは配布パッケージの中の vmeslib にあります．

従来の vmelib と完全互換なライブラリも用意しましたが，DMA や割り込み処理での複数のデバイスのオープンに問題があるため，新しいライブラリの使用を推奨しています(新しいライブラリも KEK との共同開発です)．従来の vmelib は，配布パッケージ中の KEK-vmelib にあります．

以下は，ドライバに直接アクセスして使用するためのインターフェースです．

ドライバのオープン

open () システムコールにより行ないます．

```
int fd = open("/dev/vmedrv32d32", O_RDWR);
```

vmedrv をインストールすると，以下のエントリが /devに作成されます． これらは全て同じものですが，デフォルトの転送モードが違います． 転送モードは，オープンした後でも，ioctl ()システムコールによりいつでも変更できます．

エントリファイル名	デフォルト転送モード
/dev/vmedrv	なし
/dev/vmedrv16d16	アドレス 16bit，データ 16bit，PIO 転送
/dev/vmedrv16d32	アドレス 16bit，データ 32bit，PIO 転送
/dev/vmedrv24d16	アドレス 24bit，データ 16bit，PIO 転送
/dev/vmedrv24d32	アドレス 24bit，データ 32bit，PIO 転送
/dev/vmedrv32d16	アドレス 32bit，データ 16bit，PIO 転送
/dev/vmedrv32d32	アドレス 32bit，データ 32bit，PIO 転送
/dev/vmedrv24d16dma	アドレス 24bit，データ 16bit，DMA 転送
/dev/vmedrv24d32dma	アドレス 24bit，データ 32bit，DMA 転送
/dev/vmedrv32d16dma	アドレス 32bit，データ 16bit，DMA 転送
/dev/vmedrv32d32dma	アドレス 32bit，データ 32bit，DMA 転送
/dev/vmedrv24d16nbdma	アドレス 24bit，データ 16bit，Non-block DMA 転送
/dev/vmedrv24d32nbdma	アドレス 24bit，データ 32bit，Non-block DMA 転送
/dev/vmedrv32d16nbdma	アドレス 32bit，データ 16bit，Non-block DMA 転送
/dev/vmedrv32d32nbdma	アドレス 32bit，データ 32bit，Non-block DMA 転送

DMA 転送モードでも，メモリマップトアクセスは，PIO 転送になります．

戻り値は，ドライバのファイルディスクリプタです． オープンに失敗した場合は，負の値が返され，大域変数 errno が設定されます． 詳しくは，マニュアルページ open(2) を参照してください．

シーク

lseek () システムコールにより行ないます．

```
lseek(fd, offset, SEEK_SET);
```

ここで，offset は VME 上のアドレスです．第 3 引数(whence) に指定できるのは，SEEK_SETおよびSEEK_CURのみです． 戻り値は，シーク後の新しいオフセットです． シークに失敗した場合は，負の値が返され，大域変数 errno が設定されます． 詳しくは，マニュアルページ lseek(2) を参照してください．

PIO 読み出し

read()システムコールにより行ないます.

```
int size = read(fd, buffer, max_size);
```

戻り値は, 読み出しに成功したサイズです. 読み出しに失敗した場合は, 負の値が返され, 大域変数 errno が設定されます.
詳しくは, マニュアルページ read(2) を参照してください.

PIO 書き込み

write()システムコールにより行ないます.

```
int size = write(fd, buffer, size);
```

戻り値は, 書き込みに成功したサイズです. 書き込みに失敗した場合は, 負の値が返され, 大域変数 errno が設定されます.
詳しくは, マニュアルページ write(2) を参照してください.

メモリマップ

mmap()システムコールにより行ないます.

```
void* start = mmap(0, length, prot, flags, fd, offset);
```

ここで, offset は VME のアドレス, length は マッピングを行う領域のサイズです. offset は, MMU のページサイズの整数倍でなければなりません. 半端なアドレスからマップしたい場合は, 先頭アドレスをページサイズで切り捨てて, 返されたアドレスに切り捨てたぶんを足して使用してください. この処理を行うプログラムの例を以下に示します.

```
#include <asm/page.h>

map_offset = vme_address & PAGE_MASK;    /* 中途半端な部分を切り捨てる */
page_offset = vme_address - map_offset;      /* 切り捨てたサイズ */
map_length = length + page_offset;           /* 切り捨てた分余分にマップする */

start = mmap(0, map_length, PROT_WRITE, MAP_SHARED, fd, map_offset);
if (start == MAP_FAILED) {
    perror("ERROR: mmap()");
    exit(EXIT_FAILURE);
}
mapped_address = start + page_offset;        /* 切り捨てた分を足す */
```

386 以降の x86 シリーズ CPU では, MMU ページサイズは 4kB になっています. それ以外の CPU については, $(KERNEL_INCLUDE_DIR)/asm/page.h 中の PAGE_SIZEの定義(たぶんマクロ)を参照してください. vmedrv には, PAGE_SIZE を画面に出力するサンプルプログラム pagesize.c が含まれています.

戻り値は, マッピングされた領域の先頭アドレスです. マッピングに失敗した場合は, 負の値が返され, 大域変数 errno が設定されます. 詳しくは, マニュアルページ mmap(2) を参照してください.

割り込みの設定・取得

ioctl()システムコールを使って IRQ とベクタをシグナル番号とともに登録することにより, VME の割り込みをシグナルに変換できます.

```
struct vmedrv_interrupt_property_t interrupt_property;
interrupt_property.irq = IRQ;
interrupt_property.vector = VECTOR;
interrupt_property.signal_id = SIGNAL_ID;

/* 割り込みの登録 */
ioctl(fd, VMEDRV_IOC_REGISTER_INTERRUPT, &interrupt_property);

/* 割り込み許可 */
ioctl(fd, VMEDRV_IOC_ENABLE_INTERRUPT);

/* ここで割り込み処理を行なう */
/* VME 割り込みで, 登録してある SignalID のシグナルが発生する */
```

```
/* 割り込み禁止 */
ioctl(fd, VMEDRV_IOC_DISNABLE_INTERRUPT);

/* 割り込みの削除 */
ioctl(fd, VMEDRV_IOC_UNREGISTER_INTERRUPT, &interrupt_property);
```

vmedrv が使用する割り込みベクタは 16 bit です. 8 bit のベクタ返すモジュールに対しては, ioctl() で指定するベクタの上位 8bit を 1 で埋めてください(例 0xf0 -> 0xfff0).

割り込みの許可により, その時点で登録してある全ての割り込みに対して, ハンドリングが開始されます. 登録されている割り込みが発生した場合, 割り込みプロパティで指定した シグナルがプロセスに投げられます.

割り込み待ち

VME 割り込み登録の際に指定するシグナル番号を 0 にしておくと, ioctl() で割り込み待ちを行うことができます. この際のコマンドには VMEDRV_IOC_WAIT_FOR_INTERRUPT を指定します.

```
struct vmedrv_interrupt_property_t interrupt_property;
interrupt_property.irq = IRQ;
interrupt_property.vector = VECTOR;
interrupt_property.signal_id = 0;

/* 割り込みの登録 */
ioctl(fd, VMEDRV_IOC_REGISTER_INTERRUPT, &interrupt_property);

/* 割り込み許可 */
ioctl(fd, VMEDRV_IOC_ENABLE_INTERRUPT);

/* 割り込み待ち */
interrupt_property.timeout = TIMEOUT_SEC;
result = ioctl(fd, VMEDRV_IOC_WAIT_FOR_INTERRUPT, &interrupt_property);
if (result > 0) {
    /* VME 割り込みを取得 */
}
else if (errno == ETIMEDOUT) {
    /* タイムアウト */
}
else if (errno == EINTR) {
    /* シグナル等他のものに割り込まれた */
}
else {
    /* エラー */
}

/* 割り込み禁止 */
ioctl(fd, VMEDRV_IOC_DISNABLE_INTERRUPT);

/* 割り込みの削除 */
ioctl(fd, VMEDRV_IOC_UNREGISTER_INTERRUPT, &interrupt_property);
```

vmedrv が使用する割り込みベクタは 16 bit です. 8 bit のベクタ返すモジュールに対しては, ioctl() で指定するベクタの上位 8bit を 1 で埋めてください(例 0xf0 -> 0xfff0).

割り込みのチェックとクリア

割り込み待ちを行う設定(シグナル番号を 0 にする)で, ioctl() に VMEDRV_IOC_CHECK_INTERRUPT または VMEDRV_IOC_CLEAR_INTERRUPT を指定することにより, VME 割り込みが来ているかのチェックと, 割り込みのクリアが行えます.

```
/* ここまでで, ioctl(fd, VMEDRV_IOC_WAIT_FOR_INTERRUPT) が使えるようにしておく */

int result = ioctl(fd, VMEDRV_IOC_CHECK_INTERRUPT, &interrupt_property);
if (result == 0) {
   /* 割り込みがなかった */
}
```

```
else if (result > 0) {
   /* 割り込みがあった */
   /* 必要な処理を行う */

   /* 割り込みのクリア */
   ioctl(fd, VMEDRV_IOC_CLEAR_INTERRUPT, &interrupt_property);
   test_interrupter_clear();
}
else /*(result < 0)*/ {
    /* エラー */
}
```

多重割り込みの処理

VME 割り込みを行うモジュールごとに vmedrv を open() しておけば, poll() または select() システムコールにより, 複数の割り込みソースからの割り込み待ちを同時に行えます. キーボードやネットワークからの入力待ちを同時に行うこともできます. VME 割り込みの登録は, ioctl()による割り込み待ちのときと同じ様に, シグナル番号に 0 を指定しておきます.

```
/* ここまでで, ioctl(fd, VMEDRV_IOC_WAIT_FOR_INTERRUPT) が使えるようにしておく */
struct pollfd fd_set[fd_set_size];
fd_set[0].fd = fd_my_module_1; fd_set[0].event = POLLIN;
fd_set[1].fd = fd_my_module_2; fd_set[1].event = POLLIN;
fd_set[2]... /* その他の入出力デバイスを登録 */

int status = poll(fd_set, fd_set_size, timeout_sec);
if (status < 0) {
    if (errno == EINTR) {
        /* poll() 中にシグナル割り込みが発生した */
    }
    else {
        perror("ERROR: poll()");
    }
}
else if (fd_set[0].revent & POLLIN) {
   /* my_module_1 が割り込みを出している */
}
else ...
```

VME 割り込みが発生すると, vmedrv は POLLIN イベントを発生させます. エラーが発生すると, poll() は -1 を返し, 大域変数 errno を設定します. 詳しくは, マニュアルページ poll(2) を参照してください.

DMA 読み出し

ioctl() によって転送法を DMA に設定し, あとは PIO と同様に read()システムコールにより行ないます.

```
/* 転送法を DMA にする. デフォルトが DMA のエントリをオープンした場合には必要ない. */
int transfer_method = VMEDRV_DMA;
ioctl(fd, VMEDRV_IOC_SET_TRANSFER_METHOD, &transfer_method);

int read_size = read(fd, buffer, max_size);
```

戻り値は, 読み出しに成功したサイズです.

DMA 転送中は, VME へのアクセスはしないようにしてください. また, 全ての VME 割り込み処理は DMA 転送が終了するまで延期されます.

Non-block DMA を行う場合は, transfer_method に VMEDRV_NBDMA を指定してください. デフォルトが Non-Block DMA のエントリをオープンした場合には, これは必要ありません. なお, Non-Block DMA では, 転送速度は DMA の半分程度かそれ以下になります(手持ちのメモリカードと PC での測定で, DMA が 15MB/sec, Non-Block DMA が 9MB/sec. ただし, これはメモリカードの性能に制限されている可能性もある).

読み出しに失敗した場合は, 負の値が返され, 大域変数 errno が設定されます. 詳しくは, マニュアルページ read(2) を参照してください.

DMA 書き込み

ioctl() によって転送法を DMA に設定し，あとは PIO と同様に write()システムコールにより行ないます．

```
/* 転送法を DMA にする．デフォルトが DMA のエントリをオープンした場合には必要ない．*/
int transfer_method = VMEDRV_DMA;
ioctl(fd, VMEDRV_IOC_SET_TRANSFER_METHOD, &transfer_method);

int written_size = write(fd, buffer, size);
```

戻り値は，書き込みに成功したサイズです．

DMA 転送中は，VME へのアクセスはしないようにしてください．また，全ての VME 割り込み処理は DMA 転送が終了するまで延期されます．

Non-block DMA を行う場合は，transfer_method に VMEDRV_NBDMA を指定してください．デフォルトが Non-Block DMA のエントリをオープンした場合には，これは必要ありません．なお，Non-Block DMA では，転送速度は DMA の半分程度かそれ以下になります．

書き込みに失敗した場合は，負の値が返され，大域変数 errno が設定されます．詳しくは，マニュアルページ write(2) を参照してください．

VME アクセスのエラーチェック

ioctl() に VMEDRV_IOC_PROBE を渡すことにより，VME 読み出しアクセスのエラーチェックを行うことができます．特に，指定したアドレスにモジュールが存在しているか，正しく動作しているかなどを調べるのに有用です．

```
vmedrv_word_access_t word_access;
word_access.address = address;
if (ioctl(fd, VMEDRV_IOC_PROBE, &word_access) == -1) {
    /* エラー：データが読み出せない */
}
else {
    /* 成功：読み出した値は word_access.data に記録されている */
    /* もしかしたらバイトオーダが違うかもしれない */
}
```

ioctl(fd, VMEDRV_IOC_PROBE, &word_access) は，読み出しに成功した場合 word_access.data に読み出した値を格納し，0 を返します．ただし，読み出した値はバイトオーダが違うかもしれないので，直接使用しないでください．失敗した場合は -1 を返し，大域変数 errno を設定します．

ドライバのクローズ

close()システムコールにより行ないます．mmap()してある場合は，close() の前に munmap() を呼んで下さい．詳しくは，マニュアルページ munmap(2), close(2) を参照してください．

```
munmap(start, map_length);
close(fd);
```

ここで，munmap() に渡す address 引数は，mmap() で返された値，map_length 引数は，mmap()で実際に渡された値です．ページ境界の処理をした場合には，混乱しないように注意してください．

既知の不具合

- DMA 転送中に VME アクセスを行なうと，たぶん大変なことがおこる．
- 割り込みサイクルで，IRQ が来た後にベクタを取得できないと，永久に固まる（VME クレートのバックプレーンのジャンパが正しくない場合など）．VME の電源を入れ直せば，とりあえず PC は回復する．
- DMA 転送中にプロセスにシグナルを投げると固まることがある．