

# The MINOS Midad Mini-Framework

B. Viren, *BNL*

May 20, 2004

## Abstract

The Minos Interactive Data Analysis and Display (Midad) mini-framework provides a way for users to display MINOS data and Monte Carlo in the context of the larger Offline Software framework. There exists pre-written displays as well as hooks at various levels for users to create and add their own displays. This documents the use and internals of this framework.

## 1 Introduction

This documents the use and extension of the Minos Interactive Data Analysis and Display (Midad) mini-framework. It assumes the reader is familiar with using and programming in the MINOS Offline Software Framework, proper, but an attempt is made to not rely on obscure details.

The Midad mini-framework provides the user a method to display MINOS data. A user can interact with the framework at various levels from pointing and clicking at a pre-existing GUI to writing a full library. This document is structured so that each subsequent section documents what one needs to know in order to interact at progressively lower levels of the framework.

## 2 First steps: adding Midad to a Job script

At a minimum, any user must know enough to add Midad initialization and setup to a CINT script. Midad works with, but outside of, JobControl. All communication from a Job script is done through the `Mint` (Midad Interface) object which is given a `JobC` object. This puts Midad at the same level as you are yourself with respect to JobControl.

### 2.1 Basic Midad initialization.

To create this `Mint` object one would have code similar to the following:

```
JobC* jc = new JobC;
Mint* mint = new Mint(*jc);
// more needed, including Job Path definition...
```

after this, one needs to use the JobC object to set up any path or to do any other JobControl related configuration. As the above stands, there will be no display. To create an instance of a display which comes with Midad you could add the following lines:

```
PageDisplay* pd = mint->SpawnDisplay();
pd->AddPage("Multi");
```

This will create an instance of a display and add a “page” named “Multi” to it. Details of what a Page is and how the name is used are given below.

## 2.2 Data prerequisites

Any Midad display will get at the data after a Job Path has run (except when implementing a display from a JobCModule, see below). This means that if you want to see a particular data object it must exist in the data stream. This may seem obvious, but it must be stressed that a display should not do any reconstruction but rather represent what is there already.

One particular thing worth noting is the difference between a CandDigit and a CandStrip. A “digit” (digitization) is a common MINOS term with a strict definition. Many other experiments would call it a “hit” but because of two ended readout and multiplexing it does not directly correspond to an event depositing energy in a particular strip. However humans like to visualize an event as a series of energy deposits (hits) in various strips. To do this, digits need to be demultiplexed and have the two ended readout associated with a strip object. In the offline framework, this is done by running Algorithms which demultiplex and which produce CandStrips. This can be done by adding these methods to your job path:

```
jc->Path.Create("somepath",
                //...
                "DigitListModule::Get "
                "DigitListModule::Reco "
                "DeMuxDigitListModule::Reco "
                "StripSRLListModule::Reco "
                //...
                );
```

Additionally, if a display is to display tracks or showers, at a minimum one needs to run Algorithms that produce CandTracks and CandShowers.

## 2.3 A note about CINT/C++ scoping.

ROOT’s C++ interpreter (CINT) is very finicky about how objects are created. One has to worry about usual C++ scoping, the desire to access any objects outside of a script (ie, at the CINT prompt) as well as having any necessary libraries loaded *before* any of their objects are encountered. Below is a fully working but minimal Job script written in a style the author prefers.

## 2.4 A working, but minimal Job script example

```
// example_midad.C

// forward declare
class JobC;
class Mint;

// Global pointers so we can access them from
// the interactive CINT prompt after the script is run
JobC* jc = 0;
Mint* mint = 0;

// forward declare some helper functions.
void load_libs(void);
void do_it(void);

// this is the main function and must match the name of the file
void example_midad()
{
    load_libs();
    do_it();
}

// Load in any needed libs. This must be done in a separate
// scope from where objects from these libs are used.
void load_libs(void)
{
    const char* lib[] = {
        "libBubbleSpeak.so",
        "libCandStripSR.so",
        "libCandSliceSR.so",
        "libCandTrackSR.so",
        "libCandClusterSR.so",
        "libMidadMultiPage.so",
        "libMidadUserDisplay.so",
        0
    };
    int l;
    for (l=0; lib[l]; ++l) {
        gSystem->Load(lib[l]);
    }
}

// Set initialize Midad and setup the Job. Assumes input file
// has CandDigits and CandTracks
```

```

void do_it(void)
{
    // Create JobC, Mint
    jc = new JobC;
    mint = new Mint(*jc);

    // Create an instance of a display and add a "Multi" page.
    PageDisplay* pd = mint->SpawnDisplay();
    pd->AddPage("Multi");

    // One can also create a page which is in a separate window:
    // pd->SpawnSinglePage("Multi",800,600);

    // Create a path with just a UserDisplay::Ana method
    // which will create a second display
    jc->Path.Create("default", "UserDisplay::Ana");

    jc->Input.Set("Streams=DaqSnarl,Cand");

    jc->Path("default").Run(1);
}
// End example_midad.C

```

This will create a display with the “Multi” page in a tab due to the `AddPage()` call. Commented out just below that is a `SpawnDisplay()` which would accomplish the same thing but with the page in its own window instead of sitting in a tabbed page of the main display. In both cases, all connections between page and display are the same.

As a side effect, the addition of `UserDisplay::Ana` to the job path will create a separate window holding the provided example of a display created in the context of a `JobCModule`.

After loading a few things from the menu you can expect to see something like Figure 1.

The last call to `Run(1)` isn’t strictly needed, but is handy as `Midad` won’t implicitly advance to the first record.

### 3 Overview of Midad design

Before going further it is useful to understand the basic design and individual elements of the `Midad` framework.

#### 3.1 Relationship between Midad and JobControl

As mentioned above, `Midad` relates to `JobControl` at the same “level” as you do when you write `Job` scripts. `Midad` holds on to the `JobC` object given to the `Mint`

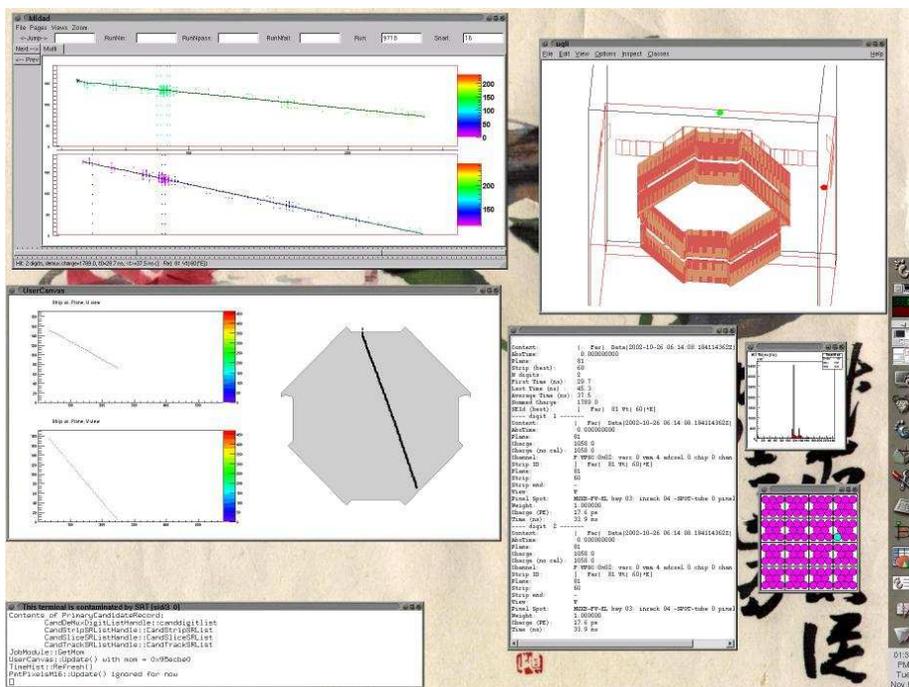


Figure 1: Screen shot of various graphical displays that come with the Midas mini-framework.

object's constructor. It then calls on this object for things such as navigating the data stream (next, previous record) or running Job Paths. Although Midad can be an active controller of the JobC object, it is a passive observer of any changes (such as new data being read in to memory). In this way either Midad or you can navigate the data stream and Midad will respond identically.

Midad should also be considered outside the context of a Job Path. This means that in most cases, while a path is running, Midad is completely dormant and should not interfere. Once the path is finished, JobControl will notify Midad that a Path has run and new data is available for display. One case where this doesn't hold true is if you have linked in any Job Modules which talk to Midad directly (eg, the provided UserDisplayModule example). However, in these cases the only communication which is done is with a single TCanvas and not any part of Midad proper.

## 3.2 Major elements

There are a few major elements of the Midad mini-framework that one should be aware of before attempting to add on custom extensions. This section first describes the layout of the source code and then details some important classes.

The code is layed out as a set of mini-packages, each to a directory. There are strict non-cyclic dependencies between the mini-packages which are managed by the order of the SUBDIRS variable in the top level GNUmakefile. The currently existing mini-packages are listed from least to most dependent:

### 3.2.1 Util mini-package

The **Util** mini-package contains a small but important set of classes which have nothing to do with event displays or even the offline software in general. The most important are briefly described.

**Undoable** provides a base class for objects whos states can be changed and changed back. This also includes a **UndoHistory** and related code implementing an undo/redo stack.

**Range** is a templated class encapsulating a min and a max value. It provides a signal to notify when either change. It is used extensively throughout Midad.

**RangeControl** is also templated and is used to marry a **Range** with an **UndoHistory**.

**NamedFactory** along with base class **NamedProxy** provide a mechanism to register a proxy class at link time. Later this proxy can be looked up for the purpose of performing something on behalf of another class (usually creating an instance). In Midad, this mechanism is used to register objects which are dynamically linked in and provide some graphical element.

### 3.2.2 Gui mini-package

The **Gui** mini-package provides wrappers for ROOT's TG GUI classes (windows, frames, buttons, sliders, etc). This was done to simplify the interface, handle memory management and provide a robust signal/slot mechanism.

### 3.2.3 Base mini-package

The **Base** mini-package contains the main part of Midad. It has classes which provide the mini-framework as well as those which are generically useful to many types of displays.

**Mint** provides the interface between the Midad framework and the rest of the world (ie, your Job script). In particular it gives methods to spawn displays, add "pages" (described next) are forward commands to the **JobC** object. For compiled code it gives access to elements of the data such as digits, tracks, time and charge ranges.

**PageDisplay** is one (currently the only) type of big-D Display. In the future there may be a need for other types, in which case parts of this will be abstracted into a base class. This display provides a menubar, buttonbar, status bar and a central "notebook" with pages, one for each type of graphical data representation.

**PageABC** is the base class for any thing that will go into a **PageDisplay**. It has various virtual methods, some optional, which allows one to hook in essentially any ROOT based graphics. Through these methods the inheriting class is notified of new data, zoom or printing requests, etc. A central frame is presented to the inheriting class to fill.

**CanvasPage** is a **PageABC** which simple fills the central frame with a **TCanvas** that can be used by subsequent sub classes for canvas based displays.

**UserCanvas** is a **CanvasPage** and allows users to access the canvas either from interpreted CINT code or from Job Modules (details below).

**various** there are various other classes which are useful but best explored in the source.

### 3.2.4 MultiPage mini-package

The **MultiPage** mini-package provides a full example implementation of a "page" that can be added to a **PageDisplay**. It inherits from **UserCanvas** and draws a U vs. Z and V vs. Z view of the data. It creates its own menu in order to provide a control on configuration and fulfills various duties of a page such as being printable and responding to zooms. It is a very good place to look for inspiration on creating your own page. Details on how to write your own page are given below.

### 3.2.5 UserDisplay mini-package

The `UserDisplay` mini-package provides a `JobCModule` which tries to be the Midad version of the `Demo/UserAnalysis` example module. It shows how you can integrate your own display into Midad from a `JobCModule`. This is probably the easiest way to incorporate a display into midad, but doesn't allow for as much integration as writing a full `Page`. Details on how to do this are given below.

## 4 Writing a display in the context of a JobC-Module

To create a display in the context of a `JobCModule` one makes typically makes use of the `UserCanvas` page. You can also create and register a full blown page from a `JobCModule`, but it is much easier to simply do this from a job script.

As stated above the `UserDisplayModule` from Midad's `UserDisplay` mini-package provides a fairly full featured example. This section will walk through the highlights of that class.

There are no particular requirements on a `JobCModule` which provides a display other than those demanded by all `JobCModules`. In this example we implement the `Ana()` method as well as some private helper methods:

```
class UserDisplayModule : public JobCModule
{
public:
    // ... ctor/dtor
    virtual void BeginJob();
    JobCResult Ana(const MomNavigator *mom);
private:
    void BuildDisplay(const MomNavigator *mom);
    void UpdateDisplay(const MomNavigator *mom);
    void AddTrack(const CandTrackHandle* cth);
    // ... private data
};
```

The responsibilities of each of these methods are described. You are of course free to structure things differently. This should be taken as just one way to do it.

**BeginJob()** is called once. In here `BuildDisplay()` is called in which a connection with Midad is made via the global `gMint` pointer (if non-zero), a `TCanvas` requested and filled.

**Ana()** This gets called once every time the path is run. In here, previously existing graphical elements (if the `BuildDisplay()` succeeded) are modified to fit the current event. Finally, one beats on `ROOT` to convince it that, yes, the canvas has been modified.

## 5 Writing a display in the context of a Job script

To create a display from a Job script one uses a `UserCanvas` like in the above case. However, here, one gets back a pointer to a class called `CanvasSignals` in order to respond to, eg, changes in the data. This code shows the process:

```
// Create JobC and Mint objects like normal
jc = new JobC;
mint = new Mint(*jc);

// Add ‘Multi’ display which comes with Midad
PageDisplay* pd = mint->SpawnDisplay();
pd->AddPage("Multi");

// Add a ‘UserCanvas’ and save the resulting CanvasSignals
CanvasSignals* cs = pd->SpawnSinglePage("UserCanvas",200,200);
```

The `CanvasSignals` provides a set of Rt signals to which you can connect interpreted methods. For example, the main signal one would most likely connect to is `Update(const MomNavigator*)`. Assume you have some function called `redraw()` that redraws the latest event into the canvas, you would connect like:

```
void redraw(const MomNavigator*)
{
    ...
}
...
cs->Connect("Update(const MomNavigator*)",0,0,"redraw(const MomNavigator*)");
```

Where `cs` is the pointer to the `CanvasSignals` one got back from creating the `UserCanvas`. Note that in compiled code it is preferable to use the `libsigc++` signals provided by this class instead of the Rt ones.

The `CanvasSignals` object also gives access to the `TCanvas` via the `CanvasSignals::GetCanvas()` method. These two features allow essentially arbitrary displays to be created entirely in interpreted code. However, since much of Midad is hidden from interpreted code due to CINT’s inability to handle templates well, some Midad features will not be accessible.

A full working example is provided with Midad in the `macros/test_user_canvas.C` script.

## 6 Writing a full blown Page

To gain full access to useful Midad features one can write a concrete `Page` class. This is preferable for a few reasons. It gives you full access to the amenities of C++ and Midad that comes with not using interpreted code and it also lets others use your work in a way that fits in with the Midad mini-framework better.

To write your own Page you start by subclassing another Page, either the base class `PageABC` or some pre-existing one (such as `CanvasPage` which provides a `TCanvas` for drawing and takes care of printing for you).

As mentioned above, the example page `MultiPage` which comes with Midad in a mini-package of the same name is a good place to look for how to write your own page.

The primary methods one needs to provide are found in the `Base/PageABC.h` abstract base class header file. The main method that need implementation is

```
virtual TObject* Init(Mint* mint, PageDisplay* pd, GuiBox& box) = 0;
```

It is through this method that you get access to the `Mint` object (from where all access to the data comes from) as well as the parent display and the `GuiBox` in which you can draw more `Gui` elements.

Additionally, you can implement these two methods

```
virtual void Clear() {}  
virtual void Update() {}
```

to be notified just before new data is read in and just after, respectively.

Besides the other optional methods, you should have code like the following in your implementation file:

```
#include <Midad/Base/PageProxy.h>  
...  
static PageProxy<MyPageType> gsMyPageProxy("MyPageName");  
...
```

This registers your page, in this example of type `MyPageType` and named `"MyPageName"`, with the `NamedFactory` for pages. This allows Midad to create an instance of your page by name (remember `PageDisplay::AddPage()` above?).

## 6.1 Writing a user display based on a `CanvasPage`

As mentioned above one of the available entries into developing your own display is to subclass `CanvasDisplay` which provides a `TCanvas` in which to draw.

Besides the complex example of `MultiPage` there is the simpler one called “`CheezyPage`” can be used as a starting point for writing your own page. It embeds the venerable `CheezyDisplay` into the Midad framework. All the actual drawing is done by the `CheezyDisplay` object.

This class also provides an example of how you can make use of the `GUI` elements Midad provides in order to allow for configuration and other interaction between code and human.