PSTUFF
An interim online monitoring package for the PHENIX experiment

Technical note 298

M.L. Purschke

June 3, 1997
version 1.01, Jan 23, 1998

Contents

1 What and Why is Pstuff?

This manual describes the PSTUFF online monitoring package. It was written in April after it became clear that Phenix did not have an adequate package for online monitoring. Using the existing Phenix C++ class libraries internally, the actual package consists of less than 1000 lines of actual Pstuff-specific code. Great care was taken that code running in the Pstuff framework can run essentially unmodified in the future online monitoring as well as in the offline analysis framework.

We could not think of a better name than Pstuff ("Phenix stuff"). But in this way the interim nature of this package is even more apparent.

The reason why it was felt that this package was needed is that the existing analysis packages were geared towards *offline*, not *online* analysis. If you are running online monitoring software rather than offline analysis, your requirements are quite different. Offline, you run a certain number of events (say, a file worth), and when you are done, you store the output/histograms and look at them later.

Online you want to work with the histograms while they are being filled. You want to have the possibility to clear them, fit them, whatever, all this while they are being updated by another process. In other words, at all times you want to have your prompt or a GUI active to give commands.

Pstuff achieves this through the – quite conventional – PAW shared memory, so that histograms are being filled by the Pstuff process while PAW can get at them and work with them.

2 Pstuff operation

The user makes a customized Pstuff executable (which we will call `main_1`, `main_2`, and so on, but it can go by any name). Pstuff provides a framework in which the user can run his or her analysis. The user provides a set of C++ routines which are called at certain points. For example, there is a routine called at the start of the program, where all user-specific initializations can be done, another routine is called for each event, and yet another at the very end before the program exits to give the user a chance to clean up, and so on.

The executable is run in the background and has no connection to a terminal or any other standard input. It receives its command input and gives output exclusively through a combination of a UNIX message queue and a shared memory section (not the PAW shared memory, a different one). In this way, it is very easy to make sophisticated command-line or graphical user interfaces, and Pstuff can accept commands from more than one process or from more than one GUI. All commands are small executables which attach to the message queue, send a command, wait for the acknowledge, and exit. Since those are shell commands, it is very easy to use them in shell scripts for more complex operations.

The actual processing happens when we "start" Pstuff. We need to tell it where to get its event data from (this can be a file, but is most cases this will be an in-memory online data stream out of the Phenix "data distributor"). Then Pstuff reads the events, and for each event calls the proper user routines. The user's analysis routine can analyze the event and fill the histograms, while the user can display the histograms at the same time.

The processing stops when we tell Pstuff to stop (then we can normally resume from the point where we stopped). We can also tell Pstuff how many events to process, and then stop, and have a way to end the processing from our code. The processing will also stop when the event stream ends. Currently the event stream can only end if we get our input from a real file; the data distributor never becomes empty (although we may have to wait until a new event is coming).

3 A quick list of features

Before we go on, we give you a a quick overview of the features which we think are important. We suggest that you actually try the (three) examples in this manual to get an idea how it works.

- Pstuff has a graphical user interface (GUI) which serves as a status display at the same time.
- More than one GUI can be active at the same time, and the GUI operation can be mixed with shell commands.
- A user Makefile template is available. It is set up in a way which makes it portable across platforms.
- More than one Pstuff executable can be run from the same account. The corresponding message queue and shared memory is identified through an environmental variable.
- Pstuff has some utilities which allow to "remote-control" the operation of PAW from Pstuff. This is very useful if you want to look at single-event displays and need to synchronize the PAW display with Pstuff.

Then there are some features which Pstuff does *not* have, of course. This is meant to be a feature in itself, since it keeps the package lightweight and easy (and remember, Pstuff is meant to be retired once the "full" online monitoring package is available). It does not have anything to manipulate histograms other than create, fill, and delete them. We have PAW to do this. Also, there is no provision to save the histograms; this can be done through PAW in a more general way as well. While the user is free to open and close files at will, Pstuff does not know about any special files except the data input stream (which is not a file in most cases anyway).

4 Running the examples

We said in the beginning that Pstuff relies on the existing Phenix C++ class libraries and utilities. In this sense, Pstuff is not a standalone package, but requires that some additional software packages are installed. The most important packages are the Phenix data distributor ("DD"), which provides the stream of online events, and the Event class library, which provides an interface to the raw data. Some Web pages describing the installation of those packages are in preparation. In this manual we assume that those packages are preinstalled on your machine.

In orrder to make it easy to run the examples in this manual, we installed the package on the RCF Pentium Solaris machines (`rpro00.rhicn.bnl.gov` or `rpro01.rhic.bnl.gov`). Execute or add to your `.login` file

```
source ~phoncs/pstuff_setup.com
```

This will define several environmental variables, in particular `PSTUFF_MAIN`, which is the top-level directory for the Pstuff distribution, and will also modify your `PATH` variable to include the Pstuff commands.

In this manual we occasionally refer to the *example* directory. This refers to the directory `$PSTUFF_MAIN/examples`. There you find all the files and examples described in the following chapters, as well as the ready-made executables if you do not want to build them on your own.

5 Some examples

5.1 A very simple example

Now we are ready to actually try it. This first example, to be found in the `example` directory, is a minimal PStuff project – it doesn't do anything remotely useful, but we can run it and see how to build it. Make a file `main_1.C` with the contents

```
#include <p_system.h>
main()
{
  p_system();
}
```

As we said, it won't do anything useful, but it will give us the flavor. All we do in the `main()` routine is calling `p_system()`. This routine sets up everything needed for the operation of Pstuff. Since we do not provide any of the user routines, dummy routines will be taken from a library.

Next, get a Makefile. In the `$PSTUFF_MAIN/examples` directory you will find a Makefile

```
include $(PSTUFF_MAIN)/flags.mk

MAIN = main_1

$(MAIN): $(MAIN).C
	CC $(CFLAGS) -o $(MAIN) $(MAIN).C \
	$(LINKLIBS)
```

The `flags.mk` file contains all the machine-dependencies, so this makefile will work on all supported platforms. In particular the `LINKLIBS` variable can look *very* different for different platforms.
Now we just type

```
make main_1
```

and we have our first Pstuff executable ready to go. Type

```
main_1 &
```

Now `main_1` is waiting for your input (through the message queue).

Before we go on, it's time to see the GUI at work. Type `pcontrol` to start it. You should see the GUI as shown in figure 1.

Now we should get us some events. Apart from an online stream and reading from file, there is a "test stream", which generates events with certain properties. This is meant to be used for testing procedures.
On your window, type

```
ptestopen
```

and you see the GUI change like the one in figure 2.

All event data input streams are called, in C++-lingo, "iterators", so now we have a "Testeventiterator". For the fun of it, type `pclose` now, and see the

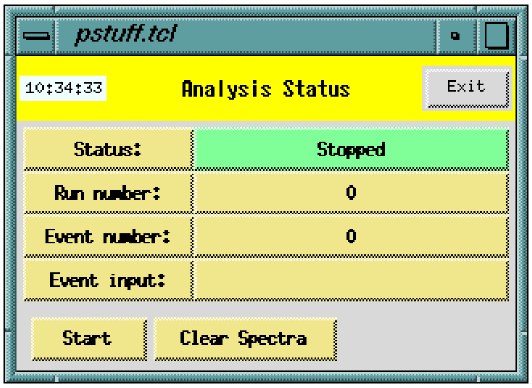

Figure 1: The status display.

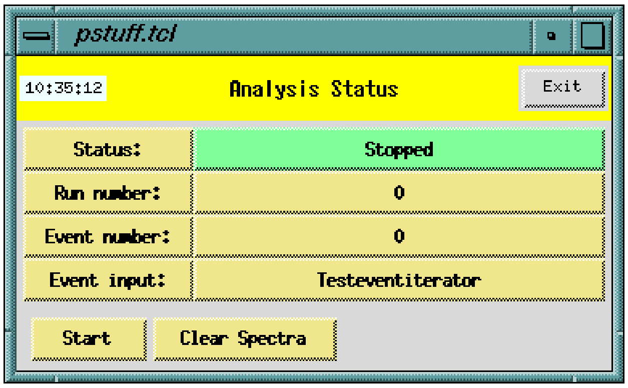

Figure 2: The status display after the ptestopen command.

status revert to the one shown in figure 1. Re-open the Testeventiterator. You can start analyzing events now by clicking the `start` button on the GUI or typing `pstart` at the command line. In either case you see that the status field of the GUI changes to "Running", and the former start button changes to "Stop" (figure 3). You can play start and stop now from the GUI and the command line. The status also tells you about the current run and event number. Since we do not do any analysis, and the events are made up by the testeventiterator very fast, we get a high number of events here.

Since this is basically all which you can do with this Pstuff example, we exit now. Type `pexit`, and watch the GUI change its status line to "not running".

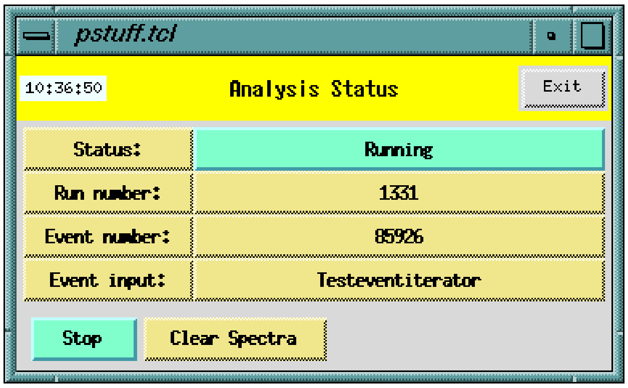

Figure 3: The status display in running mode.

Note: the "Exit" button on the GUI closes the GUI, not PStuff.

5.2 A more complex example

Let's move on now to a real example which actually gets us a few histograms. Before we do that, however, we should take a brief detour to explain a few things.

In the beginning we said that great care has been taken to make the Pstuff code as Pstuff-independent as possible. One means to do this in an object-oriented language such as C++ is the use of *"Factories"*. This is a technique to decouple your code from specific implementations of something. For a real-life example of this, imagine that you want to buy a computer for your office. You could go and get bids yourself, talk to vendors, and so on, and then decide on a particular brand or model. However, this takes you time, and it may happen that your choice is not the best in the end, because there are other criteria for selecting a computer which you are not aware of. Also, your colleague next door might be buying a computer as well, but decides on a model which is not compatible with your choice. That's why your company usually has a *purchase department*. You just tell them that you want a computer, need a specific set of features, and they go and get the bids, talk to vendors, and so on. In addition, they can assure that everybody gets computers which are compatible with each other. By adding this one layer of indirection, you save a lot of time and maybe a lot of problems further down the line. And you do not care what type or brand of computer you get as long as it meets your requirements and it does what you want.

We want our code to have the same "don't care"-attitude towards histograms. We have a purchase department – the software equivalent is called a *factory* – which we ask for *a* histogram. It does not matter which type or brand it is as long as we can use it. In particular, our code is not aware that in this case we happen to get HBOOK histograms. We don't care, since all we want to do is to get them, fill them, clear them, and maybe delete them again. In this way we can easily port our code to other environments where we deal with other histogram packages. There they would be of a different type, but our code will happily work with a different brand of histograms.

This somewhat lengthy introduction was to prepare you for the *Histogram-Factory* which you will see now in the file `main_2.C`:

```
#include <p_system.h>
#include <HistogramFactory.h>
#include <unistd.h>

// global definitions
Histo1d *h1;
Histo1d *h2;
Histo1d *h3;
Histo2d *h4;

HistogramFactory *hf = 0;

//--------------------------------------------------
main()
{
  p_system();
}
//--------------------------------------------------
void p_init()
{
  hf = getHistogramFactory(0);

  h1 = hf->create1dHistogram (100,"Channel 0",64,-50.,50.);
  h2 = hf->create1dHistogram (101,"Channel 1",64,-500.,500.);
  h3 = hf->create1dHistogram (102,"Channel 2",64,-5000.,5000.);
```

```
  h4 = hf->create2dHistogram (200,"Channel 1 vs 0",
         32,-50.,50.,
         32,-500.,500.);
}
//---------------------------------------------------
void p_clear(const int i)
{
  h1->reset();
  h2->reset();
  h3->reset();
  h4->reset();
}
//---------------------------------------------------
int process_event (Event * Evt)
{
  // get subevent 1003
  Subevent *s1003 = Evt->getSubevent(1003);
  if (s1003)         // if it exists, fill the histograms
    {
      h1->fill (s1003->rValue(0));
      h2->fill (s1003->rValue(1));
      h3->fill (s1003->rValue(2));
      h4->fill (s1003->rValue(0) , s1003->rValue(1) );
      delete s1003;  // delete the subevent again
    }
  sleep(1);          // sleep 1 tick to go more slowly
  return 0;
}

void p_exit()
{
  if (hf) delete hf;
}
```

In order to build the `main_2` executable, modify your makefile:

```
include $(PSTUFF_MAIN)/flags.mk

MAIN = main_1

$(MAIN): $(MAIN).C
```

```
CC $(CFLAGS) -o $(MAIN) $(MAIN).C \
$(LINKLIBS)

MAIN2 = main_2

$(MAIN2): $(MAIN2).C
CC $(CFLAGS) -o $(MAIN2) $(MAIN2).C \
$(LINKLIBS)
```

So now you can type `make main_2` and then `main_2 &` to start it.

Let's look at the code step-by-step. In the beginning we have four global variables, `h1`, `h2`, `h3`, and `h4`. They are not in the body of any routine, so they are global to all the routines in this `main_2.C` file. All routines in this file can get at those variables, almost like a `COMMON` block in Fortran The same goes for the "pointer to a HistogramFactory" `hf`, which we make accessible to all routines in this file.

`h1` through `h3` are of type "pointer to a `Histo1d` object", where `Histo1d` refers to a one-dimensional histogram, and `h4` is a "pointer to a two-dimensional histogram". We pick up all the definitions through the `p_system.h` include file.

In the `p_init` routine, we ask Pstuff to point us to its `HistogramFactory` as explained before. Pstuff decides which brand of histograms we will get. Then we ask the factory to please give us 3 one-dimensional and one two-dimensional histograms with the desired properties.

`p_clear` holds the code to clear those histograms. Note that we have to write this code, there is no generic "clear all" routine provided by Pstuff. This routine, by the way, is executed whenever we either give the `pclear` command on the command line or press the "Clear Spectra" button of the GUI.

Now we get to the `process_event` routine. We do not want this example to use up a lot of CPU time, so we sleep for a second towards the end of this routine. We get hold of the current event through the parameter of this routine, which is of type "pointer to an object of class Event". We ask the event to give us the Subevent 1003, which is again an object. We store the pointer to the object in the variable called `s1003`.

A Subevent object can do a lot of things for us. Think of it as something

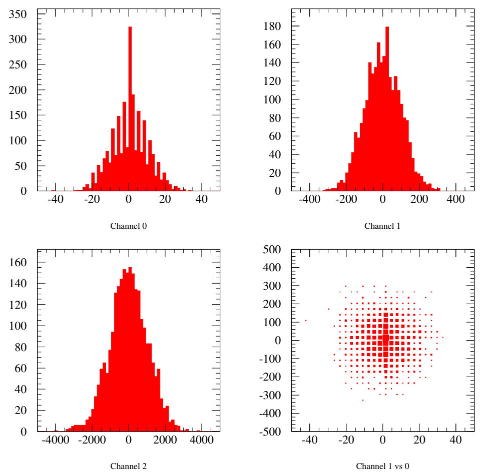

Figure 4: The histograms of example main_2.

that manages the data of a particular readout device in a data acquisition, for example, an ADC. With the `rValue(0)` function we get the value of the 0th channel of this fictitious ADC (we always count from 0) as a `float` (real) number, which we use to fill the histograms.

The `Testeventiterator` generates the events in a way that we can use them for testing software, which is its primary purpose. The Subevents 1001 and 1002 look like an ADC whose channel i contains the value i, so that you can find out if your analysis code gets the channel assignments right. The Subevent 1003 contains 3 values with a gaussian distribution, all with a mean value of 0 but with widths of 10, 100, and 1000, respectively[1]. Since we histogram the values of this subevent 1003, we will see gaussian distributions in our histograms (figure 4).

In the `process_event` routine we first check if we really got a pointer to the Subevent 1003 object. If this Subevent would not exist, we would get a NULL pointer. In this way we safeguard against other events without this subevent (say, if you open a data file instead of the testiterator, and the events in there do not have a subevent 1003). You also see that we delete

[1]The best feature of the internal random generator is its speed, not the quality of the random numbers. Do not attempt to use this distribution for any serious simulation.

the Subevent object when we are done with it. This is very important. You *must* delete all objects which you created in your code. Since this routine is called many times, not deleting an object is a *memory leak*. Over time, the memory would get filled up, and at some point the process crashes. Be very careful to avoid those memory leaks.

In the `p_exit` routine we delete the HistogramFactory which we asked for in the `p_init` routine. This is proper procedure: we are supposed to get rid of all objects which we created, so even if we exit after this, we give the factory a chance to clean up after itself.

6 Overview of the commands and the user routines

6.1 Pstuff commands

You have already seen some of the Pstuff commands so far – `pstart`, `pclear`, `pexit`, and so on. They all start with “`p`” in order to avoid clashes with other shell commands. The complete list of commands is

pcontrol fires up the GUI.

pstart starts the processing of events. This command takes an optional parameter, which specifies the number of events to be processed. If you type “`pstart 100`”, 100 events will be analyzed. The “Start” button on the GUI executes the `pstart` command internally.

pstop stops the processing.

pclear executes Pstuff’s `p_clear` routine, where you should put code to clear your histograms. It is completely up to you what you use this routine for, and it is not reserved for just clearing the histograms. However, if you use it for something else, it might be very confusing. `pclear` accepts an optional integer parameter, which passed as a parameter to the `p_clear` routine. You can use this information to steer the behavior of the `p_clear` routine.

pstatus this utility writes status information to standard output. This output is meant to be parsed by the GUI, but you can read it directly.

ptestopen opens a "Testeventiterator" input data stream.

pfileopen <filename> opens a event data file for processing.

pddopen <dd name> opens a data distributor to read online events from it.

pclose closes the currently open data stream.

pwait waits for the processing to stop, and exits. It exits immediately if Pstuff is in the "stopped" state. See chapter 7.3 for more details.

pexit exits the Pstuff executable.

6.2 Pstuff user routines

This is the list of the user routines you can provide. Some of them we have already seen before. A dummy routine is taken from a library if your code does not provide it.

void p_init() Put all your initializations here,

and book your histograms.

int process_event (Event * Evt) This routine is called for each event read in from the data stream. You should normally return 0 on exit. You can return any other value in order to terminate the event processing abnormally (see chapter 7.2 for an application of this feature).

void p_clear(const int i) put your code to clear your histograms here. You can use the parameter (which is the optional parameter on the `pclear` command line) to control the behavior of this routine. If the parameter is omitted form the `pclear` command line, it defaults to 0.

void p_endstream(const int runnumber) This routine is called when the event loop terminates as a result of the end of the event input stream. This is the only time when this routine is called. It is *not* called when you terminate the loop with the "stop" button or the `pstop` command. Also, it is never called when the event input stream is a data distributor (because this stream never ends). Also, do not confuse this condition with the occurence of a "end-run-event". This is a quite normal event in the data stream which is received by the `process_event` routine, and you can take the appropriate action.

void p_exit() is called just before Pstuff exits after you give a `pexit` command. It gives you a chance to clean up.

6.3 User-callable Pstuff routines

msgqueue *getSyncQueue(const int id) returns a pointer to a message queue object. See chapter 7.2 for an example.

HistogramFactory *getHistogramFactory(const int size) returns a pointer to a HistogramFactory object, which you use to create your histograms. The `size` parameter is used to give the factory a hint how much space is needed for the histograms. Specifying 0 assumes a default value.

7 Advanced techniques

7.1 Running more than one Pstuff executable

For simple setups, you may be able to squeeze all your online monitoring code in only one Pstuff program. Even for modestly complex setups, however, this is no longer practical or desireable. It is better to think of Pstuff programs as tools which you use for a dedicated task. This has the advantage that you can control them individually, and stopping the event loop in one Pstuff process doesn't affect the others. In addition, several users will want to run their monitoring, and they should not interfere with each other.

Since Pstuff receives its input through a message queue and shared memory, there is some potential for interference between different processes and users. In order to distinguish the queues and memory sections of different Pstuff processes, the resources can be tagged by the value of the `PSTUFF_ID` environmental variable. One process may have "`mlp`" set as the value of the variable, another "`monitor`", and they will not interfere. All commands and the GUI will obey the `PSTUFF_ID` scoping.

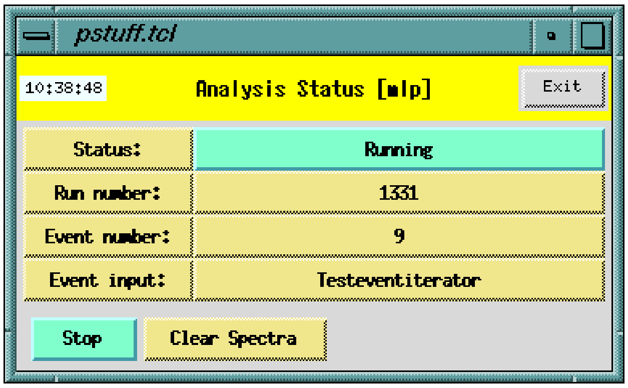

Figure 5: The status display indicates the name given by the PSTUFF_ID environmental variable.

The value of the PSTUFF_ID environmental variable is used to construct a UNIX filename, so you must not put any special characters in there.

You do not need to set the variable at all, which results in an "unnamed" version of Pstuff (the examples before were all unnamed). However, in this case it is easy for another user to unintentionally interfere with your processes if the user forgets to define a PSTUFF_ID value. Get into the habit of defining the variable even if you are running only one version of Pstuff concurrently. If you have the variable defined, the GUI will indicate this in the header (figure 5). In this way it is easier to distinguish the GUIs from different Pstuff processes.

7.2 Synchronizing Paw with Pstuff

One of the major features of Pstuff is that it allows you to look at histograms at any time, while they are being filled, that is, completely asynchronously. Occasionally you want them to be synchronized, however. The most common application where this is needed is a single-event display, where a histogram does not accumulate data from many events but holds a distribution taken from one event at a time. Let's say that in a physics analysis you want to see the distribution of energies of all the photons event-by-event. Each time the histogram is cleared and filled with the distribution from the current event. If you just asynchronously display the histogram in question, you might catch it in a phase when it has just been cleared, or is not filled completely, or you might miss events. So you want to synchronize Paw with the Pstuff event loop.

We have said in the beginning that the Pstuff commands talk to Pstuff through a UNIX message queue. We will use this same technique for inter-process communications now. Each of the queues in use gets an *identifier*, which is made from the value of the `PSTUFF_ID` variable and a number. This ensures that the messaging system obeys the `PSTUFF_ID` scope.

Numbers below 20 are reserved for the Pstuff system, so we pick a number above that, say, 42. We can try this messaging system without Pstuff with two simple commands (which we kept secret up to now), `psyncwait` and `psyncping`. In one window type

```
 psyncwait 42
```

which will just sit there doing nothing. It is waiting for a "ping" from somewhere else and then exits. On another window type

```
psyncping 42
```

which sends this ping on "channel" 42 and causes the other command to exit.

The idea now is that our Pstuff sends pings at certain times, which Paw can wait for. Look at the `main_3` example:

```
#include <p_system.h>
#include <unistd.h>

// global definitions
Histo2d *h4;
msgqueue *m =0;
HistogramFactory *hf =0;

//---------------------------------------------------
main()
{
  p_system();
}
//---------------------------------------------------
```

```
void p_init()
{
  hf =getHistogramFactory(0);
  h4 = hf->create2dHistogram (200,"Channel 1 vs 0",
         32,-50.,50.,
         32,-500.,500.);
  m = getSyncQueue(42);
}
//--------------------------------------------------
void p_clear(const int i)
{
  h4->reset();
}
//--------------------------------------------------
int process_event (Event * Evt)
{
  //clear histogram
  p_clear(0);

  // get subevent 1003
  Subevent *s1003 = Evt->getSubevent(1003);
  if (s1003)         // if it exists, fill the histograms
    {
      h4->fill (s1003->rValue(0) , s1003->rValue(1) );
      delete s1003;  // delete the subevent again
      m->send_ping();
    }
  return -1;
}
void p_exit()
{
  if (m) delete m;
  if (hf) delete hf;
}
```

In order to concentrate on the important things we have removed all histograms except the two-dimensional one. In the `process_event` routine we first clear it and then fill it, in this case with just one entry.

All the synchronization is done with the message queue. We define `m` to be a "pointer to an object of type `msgqueue`". This is just the pointer, the actual object which talks on channel 42 is created in the call

```
m = getSyncQueue(42);
```

in `p_init`. Since we must make sure that this message queue gets deleted when we exit, we again provide the `p_exit` routine, and use it to delete the message queue object in the end (as well as the HistogramFactory).
In the `process_event` routine we send out the ping with the

```
m->send_ping();
```

call.

Now we want to analyze one event at a time, and each time look at the two-dimensionsal histogram. We could achieve this "one event at a time" analysis through repeated "`pstart 1`" commands, but this is not really elegant. In addition, there may be events which we just want to skip for the display, which this method does not allow us to do. Instead we will use the feature that our code can stop the event loop by returning a non-zero value from `process_event`. In our example we send the ping and stop the loop after *each event*, but we could decide to ping-and-stop only for those events which are worth looking at. In addition, we can still use the the "start" button on the GUI this way. So this is a more elegant way of doing this.
Now on the Paw side we write a macro `loop.kumac`:

```
do i=1,100
shell psyncwait 42
hi/plot 200 box
enddo
```

In the loop we wait for the ping, display the histogram, wait for the next ping, and so on. So each time we press the "start" button on the GUI, we get a new histogram displayed in the Paw window. We are in fact remote-controlling Paw from the Pstuff GUI.

7.3 Running Pstuff in batch mode

Pstuff is meant to be an online monitoring tool. But often you will want to use a variant of your monitoring code for a preliminary offline analysis without porting a lot of code to a differnet analysis framework first.

Since all commands are shell commands, making an analysis script is easy. Let's say that we want to analyze the file `run_1223.evt`. The following example, however, does *not* work:

```
#!/bin/sh
#--
main_2 &
sleep 5
pfileopen run_1223.evt
pstart
pexit
#-- end of script
```

The `pstart` command exits immediately, and the `pexit` is executed shortly thereafter, terminating our analysis process shortly after we started it.

The solution for this is the `pwait` command, which just sits there and waits for the processing to stop:

```
#!/bin/sh
#--
main_2 &
sleep 5
pfileopen run_1223.evt
pstart
pwait
pexit
#-- end of script
```

So the `pexit` command is only executed after the processing has stopped. In order to give you an example (`run.sh` in the example directory) which works without a data file, we substitute the file reading by reading 50 events from the Testeventiterator:

```
#! /bin/sh
#--
main_2 &
sleep 5
ptestopen
pstart 50
```

```
pwait
pexit
#-- end of script
```

Try it out, and be sure to bring up the GUI to see what's going on.

8 A few words on memory leaks

In the past months working with the `Pstuff` system, the dominant source of problems with running the system was memory leaks. If you come from a Fortran background, you never had to deal with allocating and managing memory, and it may take some getting used to. We already stressed the most important rule a few times:

"You must delete all objects which you created!"

We might add that objects can only be deleted if they were created in the first place – that's why we protect ourselves from real core dumps by always asking if we succeeded in creating the object:

```
  if (m) delete m;
  if (hf) delete hf;
```

That is, we set the pointer to 0 first, the try to create the object, and later ask if the pointer contains a non-zero value. Not doing this is not a memory leak any more but a bug which will let our program core-dump.

Occasionally, deleting what you created is easier said than done. If you look back at the `process_event` routine of example `main_2`, it read

```
//--------------------------------------------------
int process_event (Event * Evt)
{
  // get subevent 1003
  Subevent *s1003 = Evt->getSubevent(1003);
  if (s1003)         // if it exists, fill the histograms
    {
      h1->fill (s1003->rValue(0));
      h2->fill (s1003->rValue(1));
      h3->fill (s1003->rValue(2));
      h4->fill (s1003->rValue(0) , s1003->rValue(1) );
```

```
      delete s1003;  // delete the subevent again
    }
  sleep(1);          // sleep 1 tick to go more slowly
  return 0;
}
```

No problem here - if (and only if) we actually got an object representing Subevent 1003, we delete it when we are done with it. All proper, leak-free procedure.

Also, most people recognize the problem with the following modification. What we want to achieve here is to also look at subevent 1001 if the channel 0 is above 100:

```
//---------------------------------------------------
int process_event (Event * Evt)
{
  // get subevent 1003
  Subevent *s1003 = Evt->getSubevent(1003);
  if (s1003)         // if it exists, fill the histograms
    {
      h1->fill (s1003->rValue(0));
      h2->fill (s1003->rValue(1));
      h3->fill (s1003->rValue(2));
      h4->fill (s1003->rValue(0) , s1003->rValue(1) );

      // if the value is above 100, also get subevent 1001
      if (s1003->rValue(0) > 100) s1003 = Evt->getSubevent(1001);

      // do something with Subevent 1001...
      //

      delete s1003;  // delete the subevent again
    }
  sleep(1);          // sleep 1 tick to go more slowly
  return 0;
}
```

We hope that the problem is obvious: if the value is above 100, we overwrite the pointer to the Subevent 1003 object and we never have a chance to delete it anymore. If we delete s1003 at the end, we only delete the newly created object representing subevent 1001. This is a classic memory leak: you destroy the pointer to an object and it will continue to exist until the program terminates. Attempting to re-use variables like this is clearly bad programming to begin with, and it invites those little problems.

The next example of a problem may be a little less obvious. Let's say that we want to fill the histogram only if some value of another subevent is 0, say

```
//--------------------------------------------------
int process_event (Event * Evt)
{
  // get subevents 1001 and  1003

  Subevent *s1001 = Evt->getSubevent(1001); //ask for subevent 1001
  Subevent *s1003 = Evt->getSubevent(1003); //ask for subevent 1003

  if (s1003 && s1001)         // if they exist fill the histogram
    {
      // we fill the histogram only if channel 0 of s1001 is 0
      if (s1001->rValue(0) == 0) h1->fill (s1003->rValue(0));

      delete s1001;  // delete the subevent 1001
      delete s1003;  // delete the subevent 1003
    }
  sleep(1);          // sleep 1 tick to go more slowly
  return 0;
}
```

We think that we are doing the right thing: First, get different variables with meaningful names, then look if the objects exist, do something with them, and then delete them when you are done with them.

However, the if-block is entered only if *both* subevent objects were created sucessfully. But what if only one of them was created? For example, if something went wrong in the DAQ and there was never a subevent 1003 in the current event? In that case we create the object for subevent 1001 alright, but the code which would delete it is never executed because the 1003 subevent object was not created, and as we go, we leave the 1001 object behind – a bad memory leak. Bad because you or somebody will probably see the leak in the previous example sooner or later, because it happens in a high number of events. Here, however, the leak will occur only in those events which are corrupt somehow, and chances are that you go a long time before you see enough such events to make the leak apparent. Also, you will probably test your code with some simulated events which are less likely to be corrupt.

Those problems which occur very infrequently are the hardest to recognize and to debug, that's why great care has to be taken to take into account all the possibilities to get an object. The solution here is easy:

```
//---------------------------------------------------
int process_event (Event * Evt)
{
  // get subevents 1001 and  1003

  Subevent *s1001 = Evt->getSubevent(1001); //ask for subevent 1001
  Subevent *s1003 = Evt->getSubevent(1003); //ask for subevent 1003

  if (s1003 && s1001)         // if they exist fill the histogram
    {
      // we fill the histogram only if channel 0 of s1001 is 0
      if (s1001->rValue(0) == 0) h1->fill (s1003->rValue(0));
    }
    if (s1001) delete s1001;  // delete the subevent 1001
    if (s1003) delete s1003;  // delete the subevent 1003

  sleep(1);          // sleep 1 tick to go more slowly
  return 0;
}
```

To summarize the minimum protective measures against memory leaks:

- always initialize global pointers to 0 before assigning objects; then you can do

  ```
  HistogramFactory *hf =0;
   ...
  hf = getHistogramFactory(0);
   ...
  if (hf) delete hf;
  ```

- never re-use variables or pointers for different purposes, declare a new pointer instead. It will help the maintainability of your code
- do not rely on the successful creation of objects as we did in the last example.
- ask for the existence of each object individually, and delete them individually.
- keep the logic of your code simple. Do not be afraid to break up your code into smaller routines.