Persistent Object Manager (POM) Data Logger documentation.

Continuation of the POM documentation POM Docu and POM Gr Docu.

NOTE: The package Gr has to be installed and compiled in order for the plots to be visible. If this document shows "alien" objects (grey rectangles with crosses) you need to install the Gr package, and then re-open the document. You can obtain Gr from the same source that provided you with POM.

```
DEFINITION PomDataLoggerDemo;

  IMPORT Dialog;
  CONST
    minDelay = 0.0;   (* Minimum delay [s]. Increase if program is not responsive.*)
    numDist  = 4;     (* How many DMI interferometers.*)
  VAR
    hSize        : INTEGER;           (* Histogram size (i.e., max num of samples).*)
    resetOnStart : BOOLEAN;           (* Restart time from zero on "Start"?*)

    tsk        : RECORD               (* Control of the periodic background task.  *)
       isRunning     -   : BOOLEAN;   (* Is the task currently running? *)
       update        *   : BOOLEAN;   (* Update the info in GUI on each turn? *)
       elapsed       -   : LONGINT;   (* Time since tsk start [ms], max 28 days.*)
       elapsedSec    -   : REAL;      (* Time since tsk start [s],  max 28 days.*)
       elapsedHours  -   : REAL;      (* Time since tsk start [hr], max 28 days.*)
       nTimes        -   : INTEGER;   (* How many times it ran since started.*)
       delaySec      *   : REAL;      (* Execute how often (seconds), nominal.*)
       actualPeriod  -   : REAL;      (* How often it actually runs in reality.*)
    END;

  PROCEDURE InitHistories;            (* Start new histories, discard previous ones.*)
  PROCEDURE Measure;                  (* A single data collection from all sensors.*)
  PROCEDURE Start;                    (* Start recording.*)
  PROCEDURE Stop;                     (* Stop  recording.*)
  PROCEDURE ResetTimer;               (* Reset the timer to t=0.*)
  PROCEDURE UpdateHistories;          (* Update the tree views in GUI.*)
  PROCEDURE StartGuard (VAR par: Dialog.Par);   (* Attach to Start button in GUI.*)
  PROCEDURE StopGuard  (VAR par: Dialog.Par);   (* Attach to Stop  button in GUI.*)

END PomDataLoggerDemo.
```

Introduction and motivation.

This small demo is a starting point to develop data logging applications. We intend to develop a mission-critical data-logging application at the Laboratory for Laser Energetics, where data from several sensors (temperature, pressure, and so on) will be collected over many days in order to correlate the observed behavior of our optical systems with environmental conditions. The list of monitored variables is open-ended. At the onset of the project we do not yet know which variables are the most critical, nor do we know the exact number of sensors. From our previous experience with the small Tiled Grating Assembly (TGA) we know that recorded histories have provided us with invaluable information and helped us understand many subtle issues related to our mechanics and optics. As we are now starting our work with the large-scale TGA model, the number of recorded variables will grow up quite substantially, from four to a few dozens. We will need to keep our software up to date with all the new additions (and deletions) to our sensor system. Managing the data in a meaningful way will be especially challenging in the situation where every new data collection may be performed with a different configuration of the sensors.

The Persistent Object Manager (POM) has been developed to address a similar problem, albeit in a different field of study (nuclear physics). In essence, POM provides a way to display an arbitrary set of data objects arranged in a tree similar to the familiar Windows Explorer. The data objects can be retrieved from the GUI by double-clicking. Upon a click, a viewer opens that will display the data object graphically. The graphical view can be embedded in a BlackBox document and saved to disk for subsequent analysis. At this time, only the Gr histograms are supported by POM, providing the ability to record integer-valued data (for example, raw ADC data, as well as "counts" from a displacement-measuring interferometer, DMI). This will be a good start for our TGA experiments. Arbitrary other objects can also be managed after a suitable "wrapper" module is developed, as explained in Section 5 of POM-Quick-Start. I plan to develop two such "wrappers", one for the real-valued histories provided by the LibVectors, and the other for two-dimensional scatter-plots provided by LibMatrices. Lacking the Lib wrappers, real-valued data can be recorded after conversion to integer and truncation of the fractional part. (It is a common experimental practice to record the temperature 10.123 degrees as an integer 10123.)

After being displayed on screen, the histories can be embedded in any Windows application such as MS Word. Doing so is not recommended because of the instability of commercial software other than BlackBox. The reader is encouraged to stay within BlackBox for consistently reliable performance. I have accumulated many years of concrete lab experience with BlackBox and I can guarantee it works very reliably as a data-acquisition system.

Even though I did perform data analysis with BlackBox, the POM data logger is meant to collect the data and store it to disk, but not necessarily to analyse it as well. The data-log histories are a particularily simple kind of data, and there are hosts of data analysis software capable of playing with this kind of recordings once they have been saved. It is very easy to write both the Gr histograms and LibVectors to disk in ASCII format, and to read such text files into other data-analysis software. (Please read the disclaimer at the end of this document.)

How does the Data Logger work?

PomDataLoggerDemo is a skeleton application meant to be further developed. Any kind of data can be recorded by calling suitable data acquisition (DAQ) modules. For example, suppose you have a module *MyDMI* that on-demand collects the integer-valued "counts" from a displacement-measuring interferometer (DMI). Suppose the DMI data is obtained by calling a procedure *MyDMI.GetData(axis),* where *axis* is the "axis number" of the DMI, i.e., a particular laser beam. The datum is recorded by adding the following two lines to the body of the procedure named **Measure** that is provided in the source code POM Data Logger Demo:

```
datum := MyDMI.GetData( axis );
dist [axis] . AddSample ( datum );
```

The previous two lines can be abbreviated into a single line:

```
      dist [axis] . AddSample ( MyDMI.GetData( axis ) );
```

In practice I recommend using the two-line convention to improve the clarity of the code. Either way, the datum will be appended to one of the history recordings named *dist* because it records distance. Prior to taking recordings the array of the histories needs to be declared and initialized. The procedure named **InitHists** is provided in the source code POM Data Logger Demo to show how this declaration should be done.

This is about all... All the rest is already handled by the Data Logger. There is a panel with *Start* and *Stop* buttons, as well as another panel where the histories are listed and can be displayed by double-clicking. After being displayed, the histories can be either saved to disk in ASCII format (to be analyzed with other software) or they can be embedded in BlackBox documents and saved to disk. The latter is both preferred and recommeded because BlackBox documents will preserve the time-stamping information provided with the history objects.

The entire history file can be saved to disk in binary form under the name *filename.pom*, where "pom" is the default extension for binary files in POM format. The files can be read back at a later date, and the history objects can be retrieved as discussed above. It is highly recommended to always save the POM file in a dedicated directory at the completion of the experiment.

How are the sensors and instruments managed by the Data Logger?

A short answer: **not at all**. The task of the Data Logger is to record measurements as a function of time. The Data Logger is not meant to also manage the internal settings of the instruments. The latter should be done with knobs and pushbuttons or with vendor-supplied setup software. In case we ourselves build an instrument, or adopt some OEM board, we may also write a GUI panel to setup such an instrument. In all cases the instrument setup *should be kept separate from the Data Logger*. The Data Logger expects to get data from the instruments that are fully setup by some other means.

Explanation of the time measurement.

Time of the measurements should probably be recorded in every data-logging experiment that we will perform. The history labeled *"Time [ms]"* is recorded in the POM folder named *Clock*. The built-in system clock is interrogated prior to each measurement and the number of "clock ticks" is recorded in the time history. While it is not carved in stone that one tick is equal to 1 ms, in all BlackBox installations that I have seen this is indeed the case. The number of ticks elapsed since the measurement has started is therefore recorded in *"Time [ms]"*. The time history may or may not be useful for data analysis, but it should be collected and saved to disk just in case.

In a real lab the individual measurements need be performed quickly enough to use the common timer for all individual sensors. If an individual measurement takes long then the common time is no longer valid for all measurements. In our case the data recordings are likely to be infrequent compared with the typical time constants inherent in our device-under-test (DUT), that are of the order of minutes or hours. Using the common time history is therefore reasonable in case of our DUT, even if the individual measurements are not perfectly aligned in time.

Concerning the time resolution, on my laptop the demo program running "as fast as possible" performs 20 data recordings per second (i.e., 50 ms per loop), what is most likely due to the frequency at which the main BlackBox loop is running. I once heard that the maximum frequency of the loop is limited to 20 Hz under Windows 98, and to 100 Hz under newer versions of Windows. The 50 ms time interval limits the application of this data logger to monitoring long-range drifts (time constant of minutes), but renders it insuitable for vibration analysis (time constants of microseconds).

Explanation of exported constants.

```
minDelay = 0.0;   (* Minimum delay [s]. Increase if program is not responsive.*)
numDist  = 4;     (* How many DMI interferometers.*)
```

Both constants are exported in order to make them prominently visible in the definition of the module. Both should be changed or adjusted for the real application, and new constants may be added as needed. (None of the constants need be exported, including the two above.)

The minimum delay *minDelay* between the measurements helps avoid "suffocating" the program while doing the measurements, what would yield the GUI non-responsive. The value 0.0 means "as soon as possible". Note that you do *not* have to change this constant. It only serves as an emergency lower limit preventing the user from entering a value that would be too short for a particular application. If you are not sure what to do with this constant, leave it alone with the current value 0.0.

The number *numDist* of DMI interferometers (or other such devices) is used in the demo to show how one can declare an array of histories. In a real appliaction there will be other such constants (*numTherm* for thermometers, and so on) that one can use in a similar manner as *numDist* in this demo.

Explanation of exported variables.

```
hSize : INTEGER;     (*Histogram size (i.e., max num of samples).*)
```

The interactor variable linked to the input data field in the Data Logger Panel. The histories will allow for recording at most *hSize* samples.

```
resetOnStart : BOOLEAN;           (* Restart time from zero on "Start"?*)
```

The interactor variable linked to the input data field in the Data Logger Panel. The time history will start from t=0 when this variable is checked. This is usually the preferred option.

```
    tsk : RECORD                       (* Control of the periodic background task.*)
       delaySec      *   : REAL;       (* Execute how often (seconds), nominal.*)
       update        *   : BOOLEAN;    (* Update the info in GUI on each turn? *)
       isRunning     -   : BOOLEAN;    (* Is the task currently running? *)
       elapsed       -   : LONGINT;    (* Time since tsk start [ms], max 28 days.*)
       elapsedSec    -   : REAL;       (* Time since tsk start [s],  max 28 days.*)
       elapsedHours  -   : REAL;       (* Time since tsk start [hr], max 28 days.*)
       nTimes        -   : INTEGER;    (* How many times it ran since started.*)
       actualPeriod  -   : REAL;       (* How often it actually runs in reality.*)
    END;
```

The interactor record linked to the data fields in the Data Logger Panel. The fields *delaySec* and *update* are the only ones writeable by the user. Other data fields are read-only (as indicated by the dash mark) and serve for monitoring.

The *delaySec* is the waiting time between the measurements *not taking into account the time to perform actual measurements*. The latter may vary widely, and there is no way to predict in advance how often the loop can execute. As a rough guidance, it takes minimum of 1.3 seconds to take and analyze the interferogram of our test TGA assembly. The minimum repetition rate is 1.3 + delaySec [seconds] in this case. The *actualPeriod* displayed in the GUI is derived from comparing the number of loop turns versus the system timer, what helps with tuning the *delaySec* to achieve the desired repetition rate.

The field *update* is an optimization to improve the speed by not updating the on-screen statistics, if nobody is watching. I have not seen any noticeable improvement with the demo when the updating was not done. This optimization is therefore left in place as a matter of principle. Its effect may become noticeable if the GUI operations are more elaborate than updating a few statistical variables.

Explanation of non-exported variables (in particular, the histories).

In the present demo the histories are hidden, i.e., private to the module. In actual applications the histories may or may not be exported, i.e., made public. There is no strong argument either way. In the demo I made the histories private to illustrate the design point that they may be hidden and nevertheless accessible to the user, what is an interesting observation from the programming standpoint. (Just to show that the object-oriented programming orthodoxy is not gospel.) Making the histories public is a matter of adding export marks "*" to their definitions and recompiling the code. The declarations in the demo are just examples. More such histories will be declared in actual applications, depending on the needs of a particular experiment.

```
TYPE
  Histogram = GrHistograms.Histogram;
VAR
  time    : Histogram;               (*point in time a sample was taken*)
  temp    : Histogram;               (*temperature*)
  press   : Histogram;               (*pressure*)
  dist    : POINTER TO ARRAY OF Histogram; (*distances to points 0,1,2,...*)
```

The histories are of the type Histogram, declared in the *Gr* package that I released to the public domain in 2001. Initially, *Gr* was used for pulse-height data-acquisition for nuclear spectroscopy. Even though data logging was not initially envisioned, I occasionally used *Gr* for data logging as well. Since data logging is our focal point, I have added a few features to *Gr* to make it more suitable for this kind of applications (e.g., the method *AddSample* is a fresh addition to the *Gr* package). The enhanced *Gr* will be resubmitted to the public web site shortly.

The way the Histograms are created and initialized is fully illustrated in the demo source code in the non-exported procedure *InitHists*. The DataLogger user will need to modify this particular procedure to initialize his/her histograms according to concrete needs. A small technicality: in the demo the *dist* histories are assigned distinct colors from a palette of 8 colors that I had patience to compose by hand from the RGB values. This superficially limits the number of *dist's* to eight. In order to overcome this limit, the user will need to either define more colors (patience permitting), or else assign a predefined color to all the extra histograms. The eight colors are composed from RGB triples at the end of the module, which is the place to modify in case more colors are needed.

Explanation of exported procedures.

```
PROCEDURE Start;                    (* Start recording.*)
PROCEDURE Stop;                     (* Stop  recording.*)
PROCEDURE ResetTimer;               (* Reset the timer to t=0.*)
```

The tasks of the above procedures are self-explanatory.

```
PROCEDURE InitHistories;            (* Start new histories, discard previous ones.*)
```

The procedure *InitHistories* defines the layout of the tree in the POM window. The procedure has to be edited by the user to perform the chore in an application-specific manner. Note that *InitHistories* internally calls *InitHists* that was discussed before.

The histories are both initialized and inserted into the POM tree controls, from where they can be retrieved by double-clicking. The old content of POM is irreversibly lost. This procedure can be potentially destructive to old results. Some degree of protection is provided in module POM through the guards and the flag *PomPOM.mainIsProtected*. Also note that the Data Logger Panel invokes a confirmation dialog. I consider this kind of warning sufficient in the present case.

```
PROCEDURE Measure;                  (* A single data collection from all sensors.*)
```

The procedure *Measure* needs to be edited by the user. It performs the actual measurement (by calling *MyDMI.GetData*, for example) and stuffing the numbers into respective histories. The procedure has a clearly marked section that the user has to change.

```
PROCEDURE UpdateHistories;          (* Update the tree views in GUI.*)
```

The procedure *UpdateHistories* should not be edited by the user. It is a technical procedure to be used in the GUI as follows. Use it as the 2nd procedure call in a button that pops up the POM result panel. For example: the button "Show results" can have the following in the link field (a single string, though here I made it two lines for clarity). The Data Logger Panel already uses it this way.

```
StdCmds.OpenAuxDialog('Pom/Rsrc/DataLoggerResults', 'Data Logger Results');
PomDataLoggerDemo.UpdateHistories
```

Under BlackBox, the *guard procedures* are used in the GUI to selectively enable/disable certain features, i.e., to "grey out" pushbuttons or menu items. The following two guards are used in the Data Logger Panel. The user can employ the following guards in his/her versions of the DataLogger GUI.

```
PROCEDURE StartGuard (VAR par: Dialog.Par);   (* Attach to Start button in GUI.*)
PROCEDURE StopGuard  (VAR par: Dialog.Par);   (* Attach to Stop  button in GUI.*)
```

Explanation of the periodic background task.

The periodic task that operates in the background is implemented in prescribed BlackBox manner and hidden inside the module, i.e., nothing particular to this mechanism is exported. Please study the source code, as the DataLogger provides a particularily simple and thus elucidating example of BlackBox tasking. If you plan to use BlackBox as the data-acquisition environment, this example will be valuable for you.

```
END PomDataLoggerDemo.
```

Explanation of panels and Data Logger menu items.

The Data Logger demo has three panels and two menu items under the POM menu "Pom". All three panels are final and should be sufficient to operate the real, non-demo Data Logger applications.

Data Logger operation panel is invoked by executing Pom->Data Logger. This panel is used to start/stop data logging, set its frequency, and monitor progress.

Data Logger results panel is invoked by pressing the button Show results... from the operations panel. This panel is used to examine and display the histories.

Data Logger confirmation panel is invoked by pressing the button Init Histories... from the operations panel. This panel displays the warning message that the histories are about to be re-initialized.

Note that the Main tree protection has to be removed before its content can be discarded. Use the menu item Pom->Toggle protection to remove the protection from the Main tree. The protection is automatically re-established by InitHistories.

The two menu items in the Pom menu are "start the data logger" and Transfer. The former has already been mentioned. The menu item Transfer allows to overlay several histories on a single plot, what is an indispensable feature for composing lab logs. Here is how to use Pom->Transfer.

1. Collect some history data in memory. You do not need to stop logging to use Transfer, but if the histories are still running while doing Transfer, than overlayed histories may be of non-equal length.

2. Pop an empty Gr viewer, or click onto one of histories, what will pop the GrViewer showing that history.

3. Select one of the histories (other than the one already displayed).

4. Click onto the target Gr viewer (where you want the history to be transfered).

5. Press *Pom* in the main menu, and execute *Transfer*. This menu item will be greyed-out if the target Gr viewer is not "focused". If this is the case, repeat #3, then #4.

6. Copy-Paste the Gr viewer into your lab research report being written under BlackBox. Do not forget to write a description of what is displayed and why it is important to save that plot.

Note: only the part of the Histogram between [beg,end[will be transfered. This is motivated by the DataLogger histories that may contain long portions of unused histogram arrays.

How do the transfered histories look?

How does this all work? well, here are example plots transfered from the Demo Data Logger into this document, what shows how the real lab reports could/should like (i.e., lots of explanations plus a few telling figures). Extremely important: the histories are not just figures! They are live plots containing LIVE NUMERICAL DATA. Thus, the lab report is more than just a document with plots, nice or otherwise.

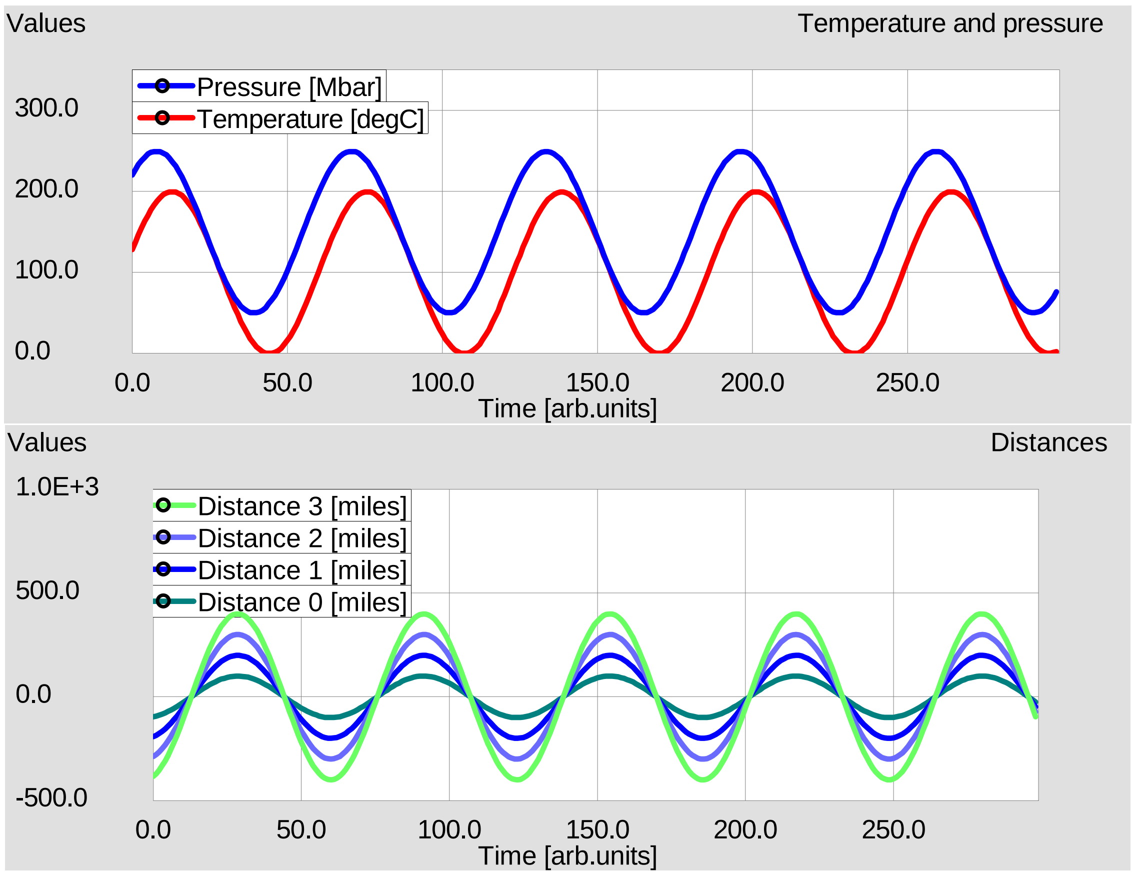

Figure 1. Examples of history plots transfered from the Data Logger Results panel onto two canvas, and embedded into this document. The data in these plots is both live and analyzable.

The lab report with embedded histories contains a full record of the lab work with original analyzable data presented in form of figures. The analyzable data can be immediately extracted from the plots (double-click on plot captions), saved to disk in ASCII format, and analyzed with analysis software of your choice.

DISCLAIMER: Software products other than BlackBox have been used by the author, but the recommendation or lack thereof is a matter of opinion. I only trust BlackBox in my experimental practice. Use your own judgement with any other software.