Named External Parameter Sets in the CL

and related revisions

Doug Tody

October 1986

1. Introduction

The CL has recently been modified or extended in a number of areas to support named, shareable parameter sets. The purpose of this memo is to document the changes and the new facilities. All revisions are upwards compatible with previous versions of the CL.

2. Parameter Set Extensions

The major modification to the CL was the addition of support for named external parameter sets, breaking the one-to-one relationship between tasks and what were formerly called parameter files. This was done to make it possible to better structure the parameter sets of tasks with very large numbers of parameters, and to make it possible for tasks which use the same piece of code to share a common parameter set.

It should be emphasized that most tasks are not expected to need the new pset facilities, and the use of a single main task parameter set should prove the simplest and best approach for most tasks. Tasks which would otherwise have very large parameter sets, subsets of which would appear in the parameter sets of other tasks in the same package, are most likely to benefit from the use of the new pset facilities.

2.1. Conceptual Model

2.1.1. Pset Tasks

A *pset-task* is an object whose function is to maintain a global set of parameters for access by other tasks or by the user. Pset tasks may be either visible or hidden, but normally they serve an important role in describing some data object or controlling some algorithm, hence they should be a visible and well documented part of the user interface to a package. One of the main reasons for implementing a named pset as a type of task is this desire to have them be a visible part of the user interface. They appear in the package menu, have on-line manual pages, can be run as tasks (*eparam* is called up to edit the pset), and their contents (the pset parameters) can be accessed using the facilities already provided for accessing the parameters of an ordinary executable task or package.

A named pset is very similar to a global (external) data structure in C. Psets and tasks are two different but equally viable classes of objects from which packages are constructed. In a sense it is not correct to think of a pset as a type of task, rather, psets and tasks are instances of some more abstract type of object, and a package is a collection of objects of various types which operate upon a particular kind of data.

Examples of psets are the control parameters for a sky fitting or centering algorithm, the control parameters for a graphics plotting utility (e.g., an axis drawing and labelling routine), or a set of parameters describing the characteristics of a particular type of data. The conceptual basis of a pset should be obvious to the user, else the use of a pset is probably not justified.

By concentrating the information pertaining to a particular object in one place, a pset can significantly reduce the complexity of a large package without loss of flexibility or control. The information hiding capability of a pset can reduce the coupling between the modules of a large package or task, much as the use of an opaque external data structure (descriptor or handle) can

reduce the coupling between the procedures in a large program.

2.1.2. Pset Parameters

A pset parameter serves as a pointer to an external parameter set. A pset parameter is a string valued parameter of datatype *pset*. The string value of the parameter is the filename or taskname of the pset to be used when the task is run. If the value of the pset parameter is the null string, the parameter set of a pset-task with the same name as the pset parameter will be used. All pset parameters in a package that refer to the same type of pset should have the same name, and a pset-task of the same name should be defined in the package to define the contents of the parameter set and to supply default values (if appropriate) for the parameters.

Whereas a pset-task is a named, statically allocated instance of some data structure, much like a C external data structure, a pset parameter is like a C structure pointer. Imagine a C function which takes a pointer to an external structure as an argument. If the value of the pointer is null the function references a particular default global structure, otherwise the referenced structure is used. Within the program the external structure is always referred to via the pointer, regardless of the name or storage class of the particular external data structure being referenced. This is very similar to the way psets are used in CL programs.

2.1.3. Relationship to other Data Structuring Facilities

The main parameter set of a CL task is very similar to the arguments and local variables of a conventional compiled procedure. The pset construct adds a basic data structuring capability to the CL. It is not surprising that, just as data structuring becomes increasingly important as the size of a compiled program or package increases, we should start to experience the need for it in our large CL tasks and packages. Part of the need can and should be met by non-CL data structures, e.g., image structures and various kinds of database facilities, but the CL pset mechanism offers unique facilities for queries and command line parameter assignments which are not provided by these other, more lower level facilities. In general, the pset mechanism should be used for high level interactive control, and the lower level data structures should be used to access and store large quantities of data.

2.2. Implementation of Psets

A pset-task is declared with the *task* or *redefine* statements in much the same way that a CL script task is declared, except that the filename extension of the referenced file is `.par` rather than `.cl`. For example,

```
task skypars = "mypkg$skypars.par"
```

adds the new pset-task *skypars* to the current package. The parameter set of a pset-task is initially read from the named file in the package directory and is updated in the the users `UPARM` directory, just as for a conventional task. For those who prefer to have their parameter declarations parsed, the actual declarations file may be a `.cl` file, e.g., `skypars.cl`, but the extension *.par* must still be used in the *task* statement. A pset-task may be called like any other task, with or without arguments on the command line; the function of a pset-task is to call up *eparam* to edit and update the parameter set.

A pset parameter is declared like any other parameter except that the datatype is specified as *pset*. For example, either

```
skypars,pset,h,,,,"sky fitting parameters"
```

or

```
pset skypars { prompt = "sky fitting parameters" }
```

would suffice to declare the pset parameter *skypars* in the parameter set for some task.

Binding of a pset parameter to a particular instance of the referenced parameter set occurs when a task is *executed.* This means that the pset parameter must be set to point to the pset to be used either before the task is called, or on the command line when the task is called. The pset parameter should not be changed to point to a different pset while a task is executing.

When a task is invoked the main task parameter set and all external psets are loaded into memory and the names and values of all the non-query parameters therein are passed via IPC to the CLIO cache in the subprocess. The query mechanism works exactly the same for pset parameters as for the main task parameters. For example, if a parameter does not have a valid value or the effective mode is something other than hidden, the parameter is not cached in the subprocess during task startup, and a query will result if the parameter is referenced at runtime. The values of pset parameters may be set with *clput* calls and the parameter set will be updated if the task exits normally, just as for the main task parameter set.

The runtime context of a task which has a parameter set has always included three different psets, the main task pset, the pset of the package in which the task is defined, and the global CL pset; runtime parameter references are resolved by searching the three psets in the order *task-pkg-cl.* This mechanism has been extended to include any psets referenced in the main task pset. For example, if the task references the two psets A and B, the search path for a parameter is *task-A-B-pkg-cl.* If the package itself references psets they will be loaded when the package is loaded, since a package is a type of task, however the use of psets in the parameter set of a package is not recommended (cacheing frequently used psets at package load time may be desirable, however).

Since the psets are included in the search path for a runtime parameter reference, the pset name is optional when specifying a parameter. This has two major implications: first, the parameters to a task may be organized into psets without the applications code having to know about the existence of the psets, and second, the user may override the values of pset parameters on the command line when a task is invoked, without having to specify the pset name., i.e., *param=value* rather than *pset.param=value.* The full syntax is however supported for both command line and runtime parameter references, and should be used in applications code to eliminate the possibility of name collisions, to improve error detection, and to speed searches.

In the current implementation of the CL, psets may not reference other psets. There is nothing in the conceptual model of a pset which excludes this, but it would complicate the implementation considerably and the use of such a feature would probably be undesirable in any case.

2.2.1. Example

A brief example may help to clarify the discussion presented in the last section. Assume we have a task *center* with the following parameters (a real task would have lots more):

```
image,s,a,,,,"image name"
objlist,*imcur,a,,,,"list of initial object centers"
cenpars,pset,h,,,,"centering algorithm parameters"
skypars,pset,h,,,,"sky fitting algorithm parameters"
```

Assume further that the *skypars* parameter set contains a parameter *ksigma* defining the k-sigma cutoff level for some iterative rejection feature of the sky-fitting algorithm.

The task would normally be called as follows when working interactively:

```
cl> center pix
```

This would run the task, taking cursor input interactively from the image display, using the default centering and sky fitting parameter sets. To use a different set of centering algorithm parameters, stored in the file `cen1.par` in the current directory:

```
cl> center pix cen=cen1.par
```

To use the default parameter sets, but override the value of the *ksigma* parameter in the *skypars*

parameter set:

```
cl> center pix sky.ksigma=3.5
```

Assuming that the *cenpars* parameter set does not also contain a *ksigma* parameter, this could be shortened to:

```
cl> center pix ksigma=3.5
```

The default *cenpars* and *skypars* parameter sets would be stored in the files `cenpars.par` and `skypars.par` (or in equivalent `.cl` files) in the package source directory.

2.3. CLIO Pset Extensions

The CLIO package has been modified to support the new CL pset facilities. The major modification was the addition of a subpackage of get/put routines to be used to access parameters in named psets. These are very simple routines, provided to avoid having programs explicitly reference the pset name in every *clget* or *clput* call, and to eliminate the need for string concatenation to construct the *pset.param* parameter name if the pset name is parameterized. The new routines are summarized below.

pp = `clopset` (psetname)		open pset
`clcpset` (pp)		close pset
pval = `clgpset`[`bcsilrdx`] (pp, param)		get parameter
`clppset`[`bcsilrdx`] (pp, param, pval)		put parameter
`clgpset` (pp, param, sval, maxch)		get string parameter
`clppset` (pp, param, sval)		put string parameter

The CLIO pset routines are functionally equivalent to the corresponding *clget* and *clput* routines except that ‘‘*psetname.*’’ is prefixed to the given parameter name to form the full parameter name before the parameter value is retrieved or updated. The string *psetname* is the name of the pset parameter (pset pointer) in the main parameter set of the calling task.

Continuing with the example presented in §2.2.1, we would open the pset and fetch the *ksigma* parameter as follows:

```
pp = clopset ("skypars")
ksigma = clgpsetr (pp, "ksigma")
```

A conventional `ksigma = clgetr ("ksigma")` statement could also have been used, in which case the CL would search for the named parameter using the search path described in §2.2, referencing the first parameter found. Use of the CLIO pset routines will be preferable in most applications since they eliminate the possibility of the wrong parameter being referenced.

3. Eparam and Lparam Extensions

The *eparam* and *lparam* tasks have been modified to work upon either the main parameter sets of tasks (as before), or upon named parameter files. The presence or absence of a `.par` filename extension is used to determine whether an operand is a taskname or a filename. For example,

```
cl> eparam skypars.par
```

will edit the parameter *file* `skypars.par` in the current directory, whereas

```
cl> eparam skypars
```

will edit the parameter set for the pset-task *skypars*. Lastly, since *spypars* is a pset-task, we could just type

```
cl> skypars
```

to edit or review the contents of the pset.

The parameter file `skypars.par` in the above example would probably be created using the new colon-command extensions to eparam. The original eparam supported only single keystroke editing commands. The new colon commands are used to enter command lines of arbitrary length to be processed by eparam.

A colon command is entered by typing the colon character (‘`:`’) while the cursor is positioned to the starting column of any value field of the parameter set being edited. The colon character is not recognized as a special character beyond column one, e.g., when entering the string value of a parameter. When colon command mode is entered, the colon character will be echoed at the start of the bottom line on the screen, and the cursor will move to the character following the colon, waiting for the command to be entered. The command is read in raw mode, but the usual delete, <ctrl/c>, <ctrl/u>, etc. sequences are recognized.

The following eparam colon commands are currently supported. All commands are carefully error checked before being executed to avoid having eparam abort with a stack trace. An illegal operation causes colon command entry mode to be exited, leaving an error message on the command entry line. All commands which cause editing of the current pset to terminate may include the `!` character to avoid updating the current pset before reading in the new one or exiting eparam. The default is to update the current pset. In all cases, *pset* may be either the name of a task or the name of a parameter file. Parameter files are always indicated by a `.par` extension, even though the actual file may be a `.cl` file: only `.par` files will be written, although either type of file may be read.

:e[!] [*pset*]
: Edit a new pset. If *pset* is omitted and the cursor was positioned to a pset parameter when the colon command was entered then eparam descends into the referenced pset; when editing of the sub-pset is complete eparam returns to editing the higher level pset at the point at which the ‘`:e`’ command was entered. If a pset is named the editor context is switched to the new pset, updating the current pset first unless the ‘`:e!`’ command was given.

:q[!]
: Exit eparam for the current pset; equivalent to a <ctrl/z>. The variant ‘`:q!`’ causes eparam to be exited without updating the current pset. Entering this command when editing a sub-pset causes an exit to the higher level pset. To abort eparam entirely without updating anything, <ctrl/c> should be used.

:r[!] [*pset*]
: Read in a new pset. If the command is ‘`:r`’, an error message is printed. If the command is ‘`:r!`’ the pset currently being edited is reread, cancelling any modifications made since the last update. If a pset is specified the contents of the named pset are merged into the current pset, i.e., the named pset is loaded into the current pset, overwriting the contents of the current pset. The command ‘`:r pfile.par`’ is commonly used to load a pset formerly saved in a user file with ‘`:w pfile.par`’ into the UPARM version of the parameter set for a task.

:w[!] [*pset*]
: Write or update a pset. If *pset* is omitted the pset currently being edited is updated on disk. If *pset* is given it should normally be the name of a parameter file to be written. If the file exists an error message will be printed unless the command ‘`:w! pfile.par`’ is given to force the file to be overwritten.

:g[**o**][!]

Run the task. Eparam exits, updating the pset and running the task whose pset was being edited. This is implemented by pushing a command back into the input stream of the task which called eparam, hence if eparam was called in a script or with other commands on the same line, execution may be delayed until these other commands have been edited. The feature works as expected when used interactively. Since the run command is pushed back into the command input stream it will appear in the history record and in any log files.

To get out of colon command mode without doing anything, simply type delete until the colon prompt is deleted and the cursor returns to the parameter it was positioned to when colon command entry mode was entered.

3.1. New Task Dparam

A new builtin task *dparam* (dump-parmeters) has been added to the *language* package. This task is similar to *lparam* except that the output is formatted in a way which is more convenient for use in scripts or as input a program, whereas *lparam* output is intended more for the user. The *dparam* task dumps the parameters for the named psets, one parameter per line, in the format *task.param* = *value*. Nothing is abbreviated or truncated. The value string will be absent if the value of the parameter is undefined. Array valued parameters are not supported.

4. Pfile Access Semantics

The code which reads and writes parameter files had to be almost completely rewritten to accomodate the above features. In the process several other improvements were made.

- The old "Warning: pfile X is out of date" message will no longer be seen. If the package pfile is newer than the user copy of the pfile, the package pfile will be read and the parameter values from the old user pfile will be merged into the new pfile. This preserves the learned parameter values but allows the master pfile to change, e.g., new fields can be added, old fields deleted, or the names of parameters may be changed. If an old parameter cannot be found in the new pfile the old value is simply discarded.
- Zero length `UPARM` pfiles are now detected, causing the package pfile to silently be read.
- A bug was fixed which would sometimes cause *unlearn* to fail.
- The low level CL procedure which reads a pfile will look for a `.par` file followed by a `.cl` file with the same root name, allowing either type of pfile to be used virtually anywhere.

5. Showtype Option in Package Menus

A new boolean parameter `showtype` has been added to the set of global CL parameters. If this option is enabled the special tasks will be flagged in package menus by appending a character to the end of the taskname. Currently, two types of special tasks are recognized thusly: package script tasks are marked with a trailing '`.`', and pset-tasks are marked with a trailing '`@`'. The default value of this switch is currently `no` for compatibility with previous versions of the CL.

In order for the *showtype* option to work properly for package script tasks, the CL must be informed that the script will define a new package when executed. This was done by extending the syntax recognized by the *task* statement to permit a `.pkg` extension on the taskname. For example, `task plot.pkg = "plot$plot.cl"` defines the new package-task *plot*. The `.pkg` is optional and it is harmless if it is omitted, but the *showtype* option will not work properly without it.