P.D.S.

PAM: Parameter Acquisition Module

Description

PAM is a set of procedures which may be called by programs to acquire parameters. For the background to the approach, see the note "Getting Command Parameters into Programs" (appended). The module is available for IMP programs on Vax/VMS and the APMs.

On ECSVAX/VMS the relevant specifications for inclusion in programs are available in:

IMP_INCLUDE:PAM.INC

and the implementation module for linking is available in:

IMP_INCLUDE:PAM.OBJ

On the APM system, the relevant specifications for inclusion in programs are contained in the general-purpose file:

I:UTIL.INC

and the implementation modules are pre-installed.

As well as the procedures described in the following sections, the include files specify:

- the string function CLI PARAM which is the standard means of obtaining from the environment the parameter string from the command line invoking the program;
- the record map PAM which is used to determine PAM_GROUPSEP and PAM_KEYFLAG -- the user's choice of separator characters;
- the constant definitions for the values of FLAG attributes described below -- these all have the prefix PAM to avoid possible identifier clashes, so that, for example, the attribute cited below as INFILE is actually defined as PAM_INFILE.

Parameter types

The definition procedures distinguish four general types:

- integer
- enumeration
- string -- covering also input and output files
- sets of Booleans -- a specialised requirement

Mode of use

In order to acquire its parameter values, a program requires to execute the following steps:

- call the appropriate definition procedure for each parameter in turn
- call the parameter processing procedure

Example of Use

Suppose that a program has two input files, one output file, two numeric parameters and a three-way diagnostic option. The relevant declarations in the program might be.

```
ownstring (255)   MAIN                  design to be plotted,
                  PREDEF="PLOTPREDEF"   pre-definition file,
                  RESULTS="STANDOUT"    analysis of design,
owninteger        MAX=8                 max number of levels,
                  ITER=5                iterations,
ownbyte           DIAG=0                diagnostic option
                                        0:none, 1:brief: 2:full
```

The parameters for this program could be acquired by the following sequence of calls:

```
define param("MAIN -- design to be plotted", main, infile+nodefault)
define param("PREdef", predef, infile)
define param("RESults", results, outfile)
define int param("MAX -- max number of levels", max, newgroup)
define int param("ITERations", iter, 0)
define enum param("NONE,BRIEF,FULL -- diagnostics", diag, 0)
process parameters(cliparam)
```

Parameter definition

There are three main procedures for defining parameters:

routine DEFINE PARAM (string(255) text, string(*)name variable, integer flags)
-- to define string parameters

routine DEFINE INT PARAM (string(255) text, integername variable, integer flags)
-- to define integer parameters

routine DEFINE ENUM PARAM(string(255) text, bytename variable, integer flags)
-- to define enumeration parameters

The significance of the three arguments to the DEFINE procedures is as follows:

TEXT: defines the keyword or keywords for the parameter, and optionally provides an expanded explanation of its significance. Keywords should be sequences of letters only and the convention is adopted that acceptable initial abbreviations are given in upper-case, optional trailing parts in lower-case. The expanded explanation should be separated from the keyword(s) by at least one space.
In the case of an enumeration, there are a number of keywords separated by commas (NB no spaces). The keywords given correspond to the values 0,1,... in the order of presentation.

VARIABLE: specifies the program variable corresponding to the parameter. This should always be an own variable.
For string parameters, it must be a string variable of any appropriate length;
For integer parameters it must be a full integer;
For enumerations it must be a byte integer.

FLAGS: specifies the set of additional attributes for the parameter. The attributes which may be specified are:

- NODEFAULT: indicates that there is no default value for this particular parameter -- causes interactive acquisition if no value is provided in the parameter string;
- MAJOR: indicates that this is an important parameter, for use in determining which parameters should be made visible to the casual enquirer
- NEWGROUP: indicates that if the value for this parameter is presented positionally within a parameter string, it must be preceded by a group separator character (typically used to segregate output file-names from input file-names);
- INFILE: for a string parameter, indicates that the value is an input file-name to be opened at the time of parameter acquisition;
  input stream numbers are used in sequence starting from 1;
  any failure to open the file causes an alternative name to be requested interactively.
- OUTFILE: for a string parameter, indicates that the value is an output file-name to be opened at the time of parameter acquisition;
  output stream numbers are used in sequence starting from 1;
  the first defined OUTFILE parameter implies NEWGROUP;
  any failure to open the file causes an alternative name to be requested interactively.
- KEEPCASE: by default, letters occurring in string parameters are standardised to upper-case; KEEPCASE disables this translation.

Sets of Booleans

There is an additional definition procedure for the specialised case of sets of binary-valued options which are required to be held internally as bits within a single word. This is:

routine DEFINE BOOLEAN PARAMS (string(255) text, integername variable, integer flags)

In this case, the text component should start with a sequence of comma-separated names, similar to an enumeration. However, each of these names is treated as an individual binary enumeration, using the prefix 'NO' to derive the negated case. For example, the set LIST,CHECK is treated as two enumerations (NOLIST,LIST) and (NOCHECK,CHECK). For this case VARIABLE should be a full integer, and bits are assigned in sequence starting from the more significant end (null names may be used to pad).

Acquiring parameter values

After defining the types of the parameters in sequence, a single procedure call is made to the procedure PROCESS PARAMETERS. This procedure takes as argument a single string which it processes to yield values for assignment to the relevant parameters. Typically this string will be derived from the standard function CLI PARAM which makes available to a program the parameter string from the command line invoking it. PROCESS PARAMETERS should be called immediately after the last definition has been set up and it can be used once only for any set of definitions.

Errors

When a localised error is discovered in a parameter string, the user is given the opportunity to correct the mistake interactively. Where this is not feasible or if the program is not being run interactively, a report is made and the program stops.

User appearance

The conventions applied to the processing of the parameter string are as follows. Parameter values for string or numeric parameters may be presented positionally or by keyword selection. Enumerated type parameters may be presented by keyword selection only. Keyword selection is indicated by the appearance of a keyword flag character immediately followed by a letter; the character '-' (minus) is always honoured as a keyword flag character, as is the user's chosen alternative if there is one. The former should be used to achieve independence of user choice. In the case of string or numeric parameters, the keyword must be followed by an equals sign and a value of the appropriate type (eg -OUTFILE=TEMP2 or -CASES=20); in the case of enumerations, the keyword itself indicates the value (eg -FULL).

Positional parameters within any group are separated by commas; any parameter defined with the attribute NEWGROUP requires that the value should instead be preceded by a group separator character, as chosen by the user.

If the parameter string consists of, or is terminated by, one or two query symbols ('?'), this is interpreted as a request to enter interactive mode, displaying to the user all (two queries) or the major (one query) parameters and their default values (or values assigned thus far) and permitting additional values to be specified.

The facility for the user to specify an individual choice for certain separator symbols is exercised on ECSVAX/VMS by providing an alternative definition for the symbol PAM_INFO. This should be a two-character definition: group separator followed by keyword flag. For example, to select space as the group separator and '!' as the keyword flag the definition would be:

DEFINE PAM_INFO " !

Note the use of the double-quote immediately before the two charcters. On the APM, the command PAMSET is used and expects as data the group separator followed by the keyword flag. Space is not a valid choice for keyword flag, nor dash (minus) for group separator, and there are a number of other characters which would not be sensible for either, depending on the syntax of file-names on the particular system.

Limitations

Some of the details of this facility are provisional and suggestions for improvement are welcome. The existing implementation of interactive enquiry about, and acquisition of, parameters is limited, and will surely be extended and varied; this should not, however, perturb the way in which programs interface to the module.

The module seeks to cover the most common requirements in terms of types and defaults, and to leave the way open for other cases to be handled by the program itself. Thus it is fully recognised that the INFILE/OUTFILE attributes will not deal with cases where it is inappropriate to open files at the outset, or where names have to be derived by more complex procedures, or where the mode of access is non-standard. It would be desirable to cover a few more cases automatically, such as default extensions, and suggestions in this area would be particularly welcome.
Note that, for consistency of user appearance, the first parameter corresponding to an output file-name should be given the attribute NEWGROUP, even if it is not designated OUTFILE.

There are also quite a few detailed points about 'syntax' which will require to be refined on a pragmatic basis.

The module does not include provision for acquiring values of type real. There is no problem in principle in extending it to do so, but it seems best to be conservative about what is supported at such a basic level.

At present on the APM implementation there is no checking on the validity of string lengths. Also at present the module does not distinguish interactive execution from non-interactive.

Getting Command Parameters into Programs

Axioms

1. The definition of what the parameters of a program are should be independent of the specification of how they are acquired.

2.1 The proper place for definition of what the parameters are is within the program.

2.2 The proper place for specification of how the parameters are to be acquired is not in the program.

3. There should be a variety of different ways for furnishing values for parameters, provided by different user interfaces and reflecting different user preferences.

4. The methods for specifying parameter values provided by a system should be applicable to all programming languages supported on the system.

5. As well as providing ways of communicating parameter values from user to program, there needs also to be communication to the user of information about the parameters of a program (or, more precisely, a command): their names, significance and default values.

6.1 The form of parameter definition within programs should be supportable on a variety of systems.

6.2 The form of parameter definition within programs should be consistent with the possibility of eventual compiler support.

Consequences of the axioms

Axioms 1 to 3 are incompatible with the philosophy of such command languages as the VAX/VMS DCL, in which the definition of what the parameters for a command are is inextricably bound up with the form and order of presentation of these parameters in a conventional command language.

It is clearly advantageous if the requirement imposed by Axiom 5 can be handled within the same mechanism as used for parameter acquisition, though this may be difficult to achieve.

Axioms 4 and 6.1 are at odds with an approach which pre-supposes that programs are 'called' with typed parameters as for procedure calls, attractive though this would be in many ways.

Axiom 6.2 stems from 2.1. The justification for 2.1 is that the definition of the parameters to a program, their types and possible ranges, is as much a part of the definition of what the program does as the instructions it contains. If this is so, we should expect it eventually to be reflected in the language definition, which makes it sensible to design forms of definition, and accept principles of usage, which are consistent with this assumption.

Status of parameters

One obvious implication of the considerations just mentioned is that parameters may be expected to be individual global variables within programs. So we postulate that there should be a way of designating as parameters a subset of the global variables declared in a program, just as another such group might be designated as volatile variables.

One natural thought is to regard parameters as nothing more than externals -- data to be supplied from outside the program itself. There are both logical and practical difficulties to identifying parameters with externals. External variables, though not (necessarily) volatile, may

have their values altered by arbitrary external procedure calls, whereas the concept of a parameter is that of a variable which may be initialised to an externally provided value, but is not otherwise capable of being altered from outside the program. In any case, external data does not provide useful flexibilty in systems which do not support dynamic external linkage.

Accordingly, parameter-hood must be regarded as a distinct status. Since existing languages do not in general recognise the concept, it is necessary to introduce it, either by language extension or by a technique of informal language extension using stylised comments. This applies to other attributes of parameters as well.

Attributes of parameters

The members of any group of variables are necessarily presented in a certain order; it is open to make use of this to provide a positional significance, as well as keyword identification, in a command language.

As variables, parameters would be typed. Axiom 4, however, imposes a restriction on the over-enthusiastic use of esoteric type mechanisms. At least the following three cases seem indispensable:

integer (preferably with range specification)

enumerated type (individually named cases)

string

Integers

For this case, the user interface must provide a convenient means for the user to supply arbitrary integer values, and the acquisition mechanism should check that these values are within the relevant range. In a conventional command language, the form of presentation would typically take a form such as:

```
MAX=1000
```

The pre-requisites for support within source language programs are: variables of type integer; range specification; initialisation option to provide default value.

Enumerations

At first sight, Boolean variables would appear to be the appropriate way to handle binary choices, with the typical form of presentation for a Boolean option LIST, say, being:

```
LIST or NOLIST
```

However, enumerated types provide a more flexible means of handling both binary and multiple choices. Given the declaration of a parameter V of an enumerated type with constant identifiers THIS, THAT and TOTHER, one possible form of specification, following the pattern of the integer case, would be:

```
V=THIS   or   V=THAT   or   V=TOTHER
```

However, if there is only one parameter of that type, the constant identifiers themselves should suffice:

```
THIS   or   THAT   or   TOTHER
```

It is suggested that user convenience is probably best served by accepting the constraint that all enumerated-type parameters should be of different types, so that the second form is the only one that needs to be used.

It is also suggested that Boolean options should simply be treated as special cases of enumerations, as, for example, (NOLIST,LIST) or (NOPLOT,PLOT).

Strings

If string parameters were only used to convey pieces of text for use as, say, headings, they would be as straightforward as the first two types considered. They raise distinct problems by virtue of their use to specify the files or other streams to be operated on by a program.

The root of the problem that this consideration creates is the issue of file-names versus files. In the case of an integer parameter, the user types a sequence of digits, and we expect the program to receive an object of integer type which is the appropriate internal representation of that sequence. It would remove a number of problems if the same treatment could be applied to files, so that there would be parameters of various file types which would be initialised to appropriate files by an externally applied process of referencing the name supplied by the user.

The main problem with this approach is that it leads to over early binding, which eliminates the possibility of applying various systematic operations or substitutions to file-names within the program. To achieve such effects, we are obliged to operate with file-names rather than files, with the consequence of importing into programs objects which are highly system dependent, and complicating a number of aspects of parameter acquisition and defaulting.

However, it should be possible to handle at least the most straightforward cases of input and output streams automatically.

Defaults

As noted earlier, the use of an initialisation statement in the program may be quite a reasonable way of providing a default value for a simple parameter. However, as well as any long-stop defaults built into the program there is also a need to provide a way of defining commands which call that program with alternative defaults. Such layered defaulting can be supported reasonably well in a command language such as that on VMS in which a new command can be defined as the application of a partial parameter string to an existing command. With this approach, it is important that the ultimate parameter string is interpreted sequentially, so that user supplied values over-ride the defaults. It is less easy to see how this requirement can be integrated with an environment providing interactive parameter acquisition, though it may not be impossible to do so.

More attractive for some purposes is to have the possibility of specifying the default value as some external or environmental variable.

A separate difficulty attaches to the use of string parameters for file-names. Including explicit default values for these in a program immediately compromises its portability across systems. The use of an external or environmental variable would avoid this difficulty, but if the system supports symbolic name substitution at the point of file reference, it is probably best handled by having a symbolic name as the default.

Implementation approaches

At the low level, the implementation will have to be in terms of procedure calls out of the program to a parameter acquisition module. It would be preferable to be able to access the parameter definitions of a program in a static fashion from outside, so that this information would be in the same category as linkage information, but this is ruled out by Axioms 4 and 6.1.

Three means of deriving the low level implementation are relevant:

Compilation: where the language and implementation support all the required apparatus

Translation: to convert a program with part of the information conveyed by language features and part by stylised comments to a form suitable for execution

Programming: direct use of the low level facilities (only as a last resort).

Low level interface

It would be attractive to present all the information about parameters in one go, but this would put an excessive strain on the type mechanisms of most languages, so that an approach involving one call per parameter is indicated. The parameter acquisition module on the other side of the interface could simply implement these sequentially, but full flexibility (of enquiry as well as acquisition) requires that they should be cumulated until all are available. (This may involve some jiggery-pokery behind the scenes, which is one reason why automatic generation of the call sequences is desirable).

Accordingly the interface would provide:

a limited set of procedures for defining parameters of different types, one parameter defined by each call and the order of the calls determining the positional significance of the parameters;

one procedure with a parameter string as argument to process the parameter string in the light of the previously executed definitions and/or to interact with the user to inform about and/or acquire parameters.