NAME

APARM – key/value-based set of C subroutines to facilitate parameter input

DESCRIPTION

aparm is a package of subroutines which facilitate the input of parameters to a C program. The parameters are usually defined in a *program*.**pin** (paramter input) file, but can also be specified on the command line. It is also possible to specify parameters directly from a terminal prompt; in this case, you are prompted for variables as needed. Currently, ten kinds of paramters are supported: char, char *, int, short int, long int, unsigned int, unsigned short, unsigned long, float, and double. This package also writes (unless disabled) a *program*.**log** and (if enabled) a **notebook.log** file.

PIN FILES

The *program*.**pin** file consists of lines of the form *key*=*value.* Optionally, the *key* can be preceeded by a slash '/'. Comments are preceeded by semicolons (';'). It is not conventional style, but it is possible to put several 'key=value' equations on the same line, as long as they are delimited by white space. For strings, one can use quotation marks; these are necessary if the string contains embedded spaces.

Alternatively, the spaces can be "escaped" with the '@' symbol. Thus, both of these are equivalent: **example.string.parm = "Now is the time."** or **example.string.parm = Now@ is@ the@ time.**

SYNOPSIS

To use **aparm** in a **C** program, you need to include the header file:

#include "aparm.h"

The following functions are available for use in any **C** program that employs the **aparm** package:

int aparm_init(int *argc_ptr, char ***argv_ptr)

void aparm_log(char *message);
void aparm_log_enable(void);
void aparm_log_disable(void);
void aparm_log_filename(char *name);
void aparm_notebook_enable(void);
void aparm_notebook_disable(void);
void aparm_notebook_filename(char *name);

*type ntype*_**parm(char *key,** *type* **defaultvalue, char *comment);**
*type ntype*_**parm_nolog(char *key,** *type* **defaultvalue);**

where *type* is one of these datatypes: **int, unsigned int, short, unsigned short, long, unsigned long, char, char *, float, double.**

and *ntype* is an associated name: **int, uint, short, ushort, long, ulong, char, string, float, double.**

EXAMPLE

The best way to describe the functionality of the **aparm** subroutines is with a simple example:

```
#include <stdio.h>
#include "aparm.h"

int
main(int argc, char **argv)
```

```
{
    int i;
    float f;
    double d;
    char *s;
    char c;

    /* note arguments are _pointers_ to argc,argv argument vector */
    i = aparm_init(&argc,&argv);

    /* return value is number of arguments that were processed by aparm,
       the rest are returned for further processing if desired */
    fprintf(stderr,"%d processed arguments.\n",i);
    fprintf(stderr,"%d unprocessed arguments:\n",argc);
    for (i=0; i<argc; ++i) {
      fprintf(stderr,"ARG[%d]= \"%s\"\n",i,argv[i]);
    }

    /* Get an integer parameter, assign the value to i */
    i = int_parm("hours.per.day",24,
            "Hours in the day (default=24)");

    /* Get a floating point parameter */
    f  = float_parm("days.per.year",365.25,
            "Number of days in the average year");

    /* Note that it is possible to use the same parameter keyword
       for different variables, even variables of a different type!

       Doing this however can confuse attempts to read the .log
       file as if it were a .pin file, so we turn off logging
       for this parameter; thus there is no need for a comment
       argument */

    d = double_parm_nolog("hours.per.day",24.0);

    /* the routine string_parm() will allocate the storage necessary
       for to hold the string value */
    s = string_parm("string.parm","hello world",
                "i dont know why you say goodbye i say hello");

    c = char_parm("char.parm",'x',
              "Character parameter (default=x)");

    fprintf(stdout,"i=%d, f=%f, d=%f s=[%s] c=%c\n",i,f,d,s,c);
    fprintf(stdout,"there are %d hours in the day\n",i);

    aparm_log("There aren\'t enough hours in the day\nOr are there?");

    return 0;

}
```

LINKING

The routines are prototyped in a header file called **aparm.h** and the library file is called **libaparm.a**. It can be linked to your program like this:

cc -c -I*/local/include program.c*
cc -o *program* **-L***/local/lib program.o* **-laparm** *[-lm etc...]*

where "/local/include" is the location of the **aparm.h** file, and "/local/lib" is the location of the **libaparm.a** library.

COMMAND LINE

If you program is compiled with the **aparm** package, then it can read options and parameter values from the command line, including:

*/key***=***value*
: over-rides the *key***=***value* statement in the *program***.pin** file.

*/key***=***?*
: forces a terminal prompt for the value of the parameter.

@ *file.pin*
: reads parameter input from the named pin file.

@@
: reads parameter input from the default pin file *program.pin.*

--log*logfilename*
: to specify the name of log file (default is *program***.log**)

--log.disable
: to disable writing to the log file.

--notebook.disable
: to disable writing to the **notebook.log** file. There is also a

--log.enable
: and **--notebook.enable**

INSTALLATION

Although the package in principle can be installed on any system that supports a standard **C** compiler (need not be **ANSI-C**), the installation is most convenient on **UNIX** platforms. In the latter case, the installation is fairly standard.

Assuming the package is available in the form of a file **aparm-***VERSION***.tar.gz** where *VERSION* is a number, such as **2.8,** you will first need to unpack the file into a directory for which you have write privileges. From that directory, you need to configure the **Makefile**'s, make the source, and finally install it.

```
tar zxvf aparm-VERSION.tar.gz
cd aparm-VERSION
./configure
make
make install
```

By default, the **make install** step puts the library into subdirectories of **/usr/local**, but you can change that at the configuration stage; use for example

./configure --prefix=/home/jt

to place **aparm.h** and **libaparm.a** into surdirectories of your home directory.

RELATED PACKAGES

This package was inspired by the **oparm** package written by J. Bloch, and used extensively in the ALEXIS project. But, unlike **oparm,** the **aparm** package:

- **o** does not have any GDF I/O routines. In fact, one of my motivations for writing **aparm** was disentangling the the GDF I/O from the parameter parsing.
- **o** can read parameter values from the command line.
- **o** can handle a wider variety of parameter types.
- **o** has an interface that is simpler and consistent across all parameter types (even strings!)

BUGS

No boolean parameters (flags) are implemented. You have to use integers (or short's or char's).

There is a lot of string manipulation, and for safety this involves memory that is allocated and never given back.

Numeric values are stored both as strings and as direct numerical values. Conversions back and forth between them may lead to loss of accuracy.

There should be a '--clear' option to clear the options defined in the default *program***.pin** file. Or at any rate, there should be some way to tell the program not to read the *program***.pin** file.

AUTHOR

James Theiler, Los Alamos National Laboratory, jt@lanl.gov

NOTICE