NAME

popt − Parse command line options

SYNOPSIS

```
#include <popt.h>

poptContext poptGetContext(const char * name, int argc,
                           const char ** argv,
                           const struct poptOption * options,
                           int  flags);

void poptFreeContext(poptContext con);

void poptResetContext(poptContext con);

int poptGetNextOpt(poptContext con);

const char * poptGetOptArg(poptContext con);

const char * poptGetArg(poptContext con);

const char * poptPeekArg(poptContext con);

const char ** poptGetArgs(poptContext con);

const char *const poptStrerror(const int error);

const char * poptBadOption(poptContext con, int  flags);

int poptReadDefaultConfig(poptContext con, int  flags);

int poptReadConfigFile(poptContext con, char *  fn);

int poptAddAlias(poptContext con, struct poptAlias alias,
                 int  flags);

int poptParseArgvString(char * s, int *  argcPtr,
                        const char *** argvPtr);

int poptDupArgv(int argc, const char ** argv, int * argcPtr,
                const char *** argvPtr);

int poptStuffArgs(poptContext con, const char ** argv);
```

DESCRIPTION

The popt library exists essentially for parsing command-line options. It is found superior in many ways when compared to parsing the argv array by hand or using the getopt functions **getopt()** and **getopt_long()** [see **getopt**(3)]. Some specific advantages of popt are: it does not utilize global variables, thus enabling multiple passes in parsing *argv* ; it can parse an arbitrary array of *argv*-style elements, allowing parsing of command-line-strings from any source; it provides a standard method of option aliasing (to be discussed at length below.); it can exec external option filters; and, finally, it can automatically generate help and usage messages for the application.

Like **getopt_long()**, the popt library supports short and long style options. Recall that a **short option** consists of a - character followed by a single alphanumeric character. A **long option**, common in GNU utilities, consists of two - characters followed by a string made up of letters, numbers and hyphens. Long options are optionally allowed to begin with a single -, primarily to allow command-line

compatibility between popt applications and X toolkit applications. Either type of option may be followed by an argument. A space separates a short option from its arguments; either a space or an = separates a long option from an argument.

The popt library is highly portable and should work on any POSIX platform. The latest version is distributed with rpm and is always available from: ftp://ftp.rpm.org/pub/rpm/dist.

BASIC POPT USAGE

1. THE OPTION TABLE

Applications provide popt with information on their command-line options by means of an "option table," i.e., an array of **struct poptOption** structures:

```
#include <popt.h>

struct poptOption {
   const char * longName; /* may be NULL */
   char shortName;       /* may be '\0' */
   int argInfo;
   void * arg;           /* depends on argInfo */
   int val;              /* 0 means don't return, just update flag */
   char * descrip;       /* description for autohelp -- may be NULL */
   char * argDescrip;    /* argument description for autohelp */
};
```

Each member of the table defines a single option that may be passed to the program. Long and short options are considered a single option that may occur in two different forms. The first two members, *longName* and *shortName*, define the names of the option; the first is a long name, while the latter is a single character.

The *argInfo* member tells popt what type of argument is expected after the argument. If no option is expected, **POPT_ARG_NONE** should be used. The rest of the valid values are shown in the following table:

Value	Description	arg Type
POPT_ARG_NONE	No argument expected	int
POPT_ARG_STRING	No type checking to be performed	char *
POPT_ARG_INT	An integer argument is expected	int
POPT_ARG_LONG	A long integer is expected	long
POPT_ARG_VAL	Integer value taken from `val`	int
POPT_ARG_FLOAT	An float argument is expected	float
POPT_ARG_DOUBLE	A double argument is expected	double

For numeric values, if the *argInfo* value is bitwise or'd with one of **POPT_ARGFLAG_OR**, **POPT_ARGFLAG_AND**, or **POPT_ARGFLAG_XOR**, the value is saved by performing an OR, AND, or XOR. If the *argInfo* value is bitwise or'd with **POPT_ARGFLAG_NOT**, the value will be negated before saving. For the common operations of setting and/or clearing bits, **POPT_BIT_SET** and **POPT_BIT_CLR** have the appropriate flags set to perform bit operations.

If the *argInfo* value is bitwise or'd with **POPT_ARGFLAG_ONEDASH**, the long argument may be given with a single - instead of two. For example, if **--longopt** is an option with **POPT_ARGFLAG_ONEDASH**, is specified, **-longopt** is accepted as well.

The next element, *arg*, allows popt to automatically update program variables when the option is used. If *arg* is **NULL**, it is ignored and popt takes no special action. Otherwise it should point to a variable of the type indicated in the right-most column of the table above.

If the option takes no argument (*argInfo* is **POPT_ARG_NONE**), the variable pointed to by *arg* is set to 1 when the option is used. (Incidentally, it will perhaps not escape the attention of hunt-and-peck typists that the value of **POPT_ARG_NONE** is 0.) If the option does take an argument, the variable that *arg* points to is updated to reflect the value of the argument. Any string is acceptable for **POPT_ARG_STRING** arguments, but **POPT_ARG_INT**, **POPT_ARG_LONG**, **POPT_ARG_FLOAT**, and **POPT_ARG_DOUBLE** are converted to the appropriate type, and an error returned if the conversion fails.

POPT_ARG_VAL causes *arg* to be set to the (integer) value of *val* when the argument is found. This is most often useful for mutually-exclusive arguments in cases where it is not an error for multiple arguments to occur and where you want the last argument specified to win; for example, "rm -i -f". **POPT_ARG_VAL** causes the parsing function not to return a value, since the value of *val* has already been used.

If the *argInfo* value is bitwise or'd with **POPT_ARGFLAG_OPTIONAL**, the argument to the long option may be omitted. If the long option is used without an argument, a default value of zero or NULL will be saved (if the arg pointer is present), otherwise behavior will be identical to a long option with argument.

The next option, *val*, is the value popt's parsing function should return when the option is encountered. If it is 0, the parsing function does not return a value, instead parsing the next command-line argument.

The last two options, *descrip* and *argDescrip* are only required if automatic help messages are desired (automatic usage messages can be generated without them). *descrip* is a text description of the argument and *argdescrip* is a short summary of the type of arguments the option expects, or NULL if the option doesn't require any arguments.

If popt should automatically provide **--usage** and **--help** (**-?**) options, one line in the table should be the macro **POPT_AUTOHELP**. This macro includes another option table (via **POPT_ARG_INCLUDE_TABLE;** see below) in the main one which provides the table entries for these arguments. When **--usage** or **--help** are passed to programs which use popt's automatical help, popt displays the appropriate message on stderr as soon as it finds the option, and exits the program with a return code of 0. If you want to use popt's automatic help generation in a different way, you need to explicitly add the option entries to your programs option table instead of using **POPT_AUTOHELP**.

If the *argInfo* value is bitwise or'd with **POPT_ARGFLAG_DOC_HIDDEN**, the argument will not be shown in help output.

If the *argInfo* value is bitwise or'd with **POPT_ARGFLAG_SHOW_DEFAULT**, the inital value of the arg will be shown in help output.

The final structure in the table should have all the pointer values set to **NULL** and all the arithmetic values set to 0, marking the end of the table. The macro **POPT_TABLEEND** is provided to do that.

There are two types of option table entries which do not specify command line options. When either of these types of entries are used, the *longName* element must be **NULL** and the **shortName** element must be **'\0'**.

The first of these special entry types allows the application to nest another option table in the current one; such nesting may extend quite deeply (the actual depth is limited by the program's stack). Including other option tables allows a library to provide a standard set of command-line options to every program which uses it (this is often done in graphical programming toolkits, for example). To do this, set the *argInfo* field to **POPT_ARG_INCLUDE_TABLE** and the arg field to point to the table which is being included. If automatic help generation is being used, the *descrip* field should contain a overall description of the option table being included.

The other special option table entry type tells popt to call a function (a callback) when any option in that table is found. This is especially usefull when included option tables are being used, as the

program which provides the top-level option table doesn't need to be aware of the other options which are provided by the included table. When a callback is set for a table, the parsing function never returns information on an option in the table. Instead, options information must be retained via the callback or by having popt set a variable through the option's *arg* field. Option callbacks should match the following prototype:

void poptCallbackType(poptContext con,
const struct poptOption * opt,
const char * arg, void * data);

The first parameter is the context which is being parsed (see the next section for information on contexts), *opt* points to the option which triggered this callback, and *arg* is the option's argument. If the option does not take an argument, *arg* is **NULL**. The final parameter, *data* is taken from the *descrip* field of the option table entry which defined the callback. As *descrip* is a pointer, this allows callback functions to be passed an arbitrary set of data (though a typecast will have to be used).

The option table entry which defines a callback has an *argInfo* of **POPT_ARG_CALLBACK**, an *arg* which points to the callback function, and a *descrip* field which specifies an arbitrary pointer to be passed to the callback.

2. CREATING A CONTEXT

popt can interleave the parsing of multiple command-line sets. It allows this by keeping all the state information for a particular set of command-line arguments in a **poptContext** data structure, an opaque type that should not be modified outside the popt library.

New popt contexts are created by **poptGetContext()**:

poptContext poptGetContext(const char * *name***, int** *argc***,**
const char ** *argv***,**
const struct poptOption * *options***,**
int *flags***);**

The first parameter, *name*, is used only for alias handling (discussed later). It should be the name of the application whose options are being parsed, or should be **NULL** if no option aliasing is desired. The next two arguments specify the command-line arguments to parse. These are generally passed to **poptGetContext()** exactly as they were passed to the program's **main()** function. The *options* parameter points to the table of command-line options, which was described in the previous section. The final parameter, *flags*, can take one of three values:

Value	Description
POPT_CONTEXT_NO_EXEC	Ignore exec expansions
POPT_CONTEXT_KEEP_FIRST	Do not ignore argv[0]
POPT_CONTEXT_POSIXMEHARDER	Options cannot follow arguments

A **poptContext** keeps track of which options have already been parsed and which remain, among other things. If a program wishes to restart option processing of a set of arguments, it can reset the **poptContext** by passing the context as the sole argument to **poptResetContext()**.

When argument processing is complete, the process should free the **poptContext** as it contains dynamically allocated components. The **poptFreeContext()** function takes a **poptContext** as its sole argument and frees the resources the context is using.

Here are the prototypes of both **poptResetContext()** and **poptFreeContext()**:

#include <popt.h>
void poptFreeContext(poptContext *con***);**
void poptResetContext(poptContext *con***);**

3. PARSING THE COMMAND LINE

After an application has created a **poptContext**, it may begin parsing arguments. **poptGetNextOpt()** performs the actual argument parsing.

#include <popt.h>
int poptGetNextOpt(poptContext *con***);**

Taking the context as its sole argument, this function parses the next command-line argument found. After finding the next argument in the option table, the function fills in the object pointed to by the option table entry's *arg* pointer if it is not **NULL**. If the val entry for the option is non-0, the function then returns that value. Otherwise, **poptGetNextOpt()** continues on to the next argument.

poptGetNextOpt() returns -1 when the final argument has been parsed, and other negative values when errors occur. This makes it a good idea to keep the *val* elements in the options table greater than 0.

If all of the command-line options are handled through *arg* pointers, command-line parsing is reduced to the following line of code:

```
rc = poptGetNextOpt(poptcon);
```

Many applications require more complex command-line parsing than this, however, and use the following structure:

```
while ((rc = poptGetNextOpt(poptcon)) > 0) {
    switch (rc) {
        /* specific arguments are handled here */
    }
}
```

When returned options are handled, the application needs to know the value of any arguments that were specified after the option. There are two ways to discover them. One is to ask popt to fill in a variable with the value of the option through the option table's *arg* elements. The other is to use **poptGetOptArg()**:

#include <popt.h>
const char * poptGetOptArg(poptContext *con***);**

This function returns the argument given for the final option returned by **poptGetNextOpt()**, or it returns **NULL** if no argument was specified.

4. LEFTOVER ARGUMENTS

Many applications take an arbitrary number of command-line arguments, such as a list of file names. When popt encounters an argument that does not begin with a -, it assumes it is such an argument and adds it to a list of leftover arguments. Three functions allow applications to access such arguments:

const char * poptGetArg(poptContext *con***);**
This function returns the next leftover argument and marks it as processed.

const char * poptPeekArg(poptContext *con***);**
The next leftover argument is returned but not marked as processed. This allows an application to look ahead into the argument list, without modifying the list.

const char ** poptGetArgs(poptContext *con***);**
All the leftover arguments are returned in a manner identical to *argv*. The final element in the returned array points to **NULL**, indicating the end of the arguments.

5. AUTOMATIC HELP MESSAGES

The **popt** library can automatically generate help messages which describe the options a program accepts. There are two types of help messages which can be generated. Usage messages are a short

messages which lists valid options, but does not describe them. Help messages describe each option on one (or more) lines, resulting in a longer, but more useful, message. Whenever automatic help messages are used, the **descrip** and **argDescrip** fields **struct poptOption** members should be filled in for each option.

The **POPT_AUTOHELP** macro makes it easy to add **--usage** and **--help** messages to your program, and is described in part 1 of this man page. If more control is needed over your help messages, the following two functions are available:

#include <popt.h>
void poptPrintHelp(poptContext *con***, FILE *** *f***, int** *flags***);**
void poptPrintUsage(poptContext *con***, FILE *** *f***, int** *flags***);**

poptPrintHelp() displays the standard help message to the stdio file descriptor f, while **poptPrintUsage()** displays the shorter usage message. Both functions currently ignore the **flags** argument; it is there to allow future changes.

ERROR HANDLING

All of the popt functions that can return errors return integers. When an error occurs, a negative error code is returned. The following table summarizes the error codes that occur:

Error	Description
POPT_ERROR_NOARG	Argument missing for an option.
POPT_ERROR_BADOPT	Option's argument couldn't be parsed.
POPT_ERROR_OPTSTOODEEP	Option aliasing nested too deeply.
POPT_ERROR_BADQUOTE	Quotations do not match.
POPT_ERROR_BADNUMBER	Option couldn't be converted to number.
POPT_ERROR_OVERFLOW	A given number was too big or small.

Here is a more detailed discussion of each error:

POPT_ERROR_NOARG
An option that requires an argument was specified on the command line, but no argument was given. This can be returned only by **poptGetNextOpt()**.

POPT_ERROR_BADOPT
An option was specified in *argv* but is not in the option table. This error can be returned only from **poptGetNextOpt()**.

POPT_ERROR_OPTSTOODEEP
A set of option aliases is nested too deeply. Currently, popt follows options only 10 levels to prevent infinite recursion. Only **poptGetNextOpt()** can return this error.

POPT_ERROR_BADQUOTE
A parsed string has a quotation mismatch (such as a single quotation mark). **poptParseArgvString()**, **poptReadConfigFile()**, or **poptReadDefaultConfig()** can return this error.

POPT_ERROR_BADNUMBER
A conversion from a string to a number (int or long) failed due to the string containing nonnumeric characters. This occurs when **poptGetNextOpt()** is processing an argument of type **POPT_ARG_INT**, **POPT_ARG_LONG**, POPT_ARG_FLOAT**, or** POPT_ARG_DOUBLE**.**

POPT_ERROR_OVERFLOW
A string-to-number conversion failed because the number was too large or too small. Like **POPT_ERROR_BADNUMBER**, this error can occur only when **poptGetNextOpt()** is

processing an argument of type **POPT_ARG_INT**, **POPT_ARG_LONG**, POPT_ARG_FLOAT**, or** POPT_ARG_DOUBLE**.**

POPT_ERROR_ERRNO
A system call returned with an error, and *errno* still contains the error from the system call. Both **poptReadConfigFile()** and **poptReadDefaultConfig()** can return this error.

Two functions are available to make it easy for applications to provide good error messages.

const char *const poptStrerror(const int *error***);**
This function takes a popt error code and returns a string describing the error, just as with the standard **strerror()** function.

const char * poptBadOption(poptContext *con***, int** *flags***);**
If an error occurred during **poptGetNextOpt()**, this function returns the option that caused the error. If the *flags* argument is set to **POPT_BADOPTION_NOALIAS**, the outermost option is returned. Otherwise, *flags* should be 0, and the option that is returned may have been specified through an alias.

These two functions make popt error handling trivial for most applications. When an error is detected from most of the functions, an error message is printed along with the error string from **poptStrerror()**. When an error occurs during argument parsing, code similiar to the following displays a useful error message:

```
fprintf(stderr, "%s: %s\n",
    poptBadOption(optCon, POPT_BADOPTION_NOALIAS),
    poptStrerror(rc));
```

OPTION ALIASING

One of the primary benefits of using popt over **getopt()** is the ability to use option aliasing. This lets the user specify options that popt expands into other options when they are specified. If the standard grep program made use of popt, users could add a **--text** option that expanded to **-i -n -E -2** to let them more easily find information in text files.

1. SPECIFYING ALIASES

Aliases are normally specified in two places: */etc/popt* and the **.popt** file in the user's home directory (found through the **HOME** environment variable). Both files have the same format, an arbitrary number of lines formatted like this:

appname **alias** *newoption expansion*

The *appname* is the name of the application, which must be the same as the *name* parameter passed to **poptGetContext()**. This allows each file to specify aliases for multiple programs. The **alias** keyword specifies that an alias is being defined; currently popt configuration files support only aliases, but other abilities may be added in the future. The next option is the option that should be aliased, and it may be either a short or a long option. The rest of the line specifies the expansion for the alias. It is parsed similarly to a shell command, which allows \, ", and ' to be used for quoting. If a backslash is the final character on a line, the next line in the file is assumed to be a logical continuation of the line containing the backslash, just as in shell.

The following entry would add a **--text** option to the grep command, as suggested at the beginning of this section.

grep alias --text -i -n -E -2

2. ENABLING ALIASES

An application must enable alias expansion for a **poptContext** before calling **poptGetNextArg()** for the first time. There are three functions that define aliases for a context:

int poptReadDefaultConfig(poptContext *con***, int** *flags***);**

This function reads aliases from */etc/popt* and the **.popt** file in the user's home directory. Currently, *flags* should be **NULL**, as it is provided only for future expansion.

int poptReadConfigFile(poptContext *con***, char *** *fn***);**

The file specified by *fn* is opened and parsed as a popt configuration file. This allows programs to use program-specific configuration files.

int poptAddAlias(poptContext *con***, struct poptAlias** *alias***,**
int *flags***);**

Occasionally, processes want to specify aliases without having to read them from a configuration file. This function adds a new alias to a context. The *flags* argument should be 0, as it is currently reserved for future expansion. The new alias is specified as a **struct poptAlias**, which is defined as:

```
struct poptAlias {
    const char * longName; /* may be NULL */
    char shortName; /* may be '\0' */
    int argc;
    const char ** argv; /* must be free()able */
};
```

The first two elements, *longName* and *shortName*, specify the option that is aliased. The final two, *argc* and *argv*, define the expansion to use when the aliases option is encountered.

PARSING ARGUMENT STRINGS

Although popt is usually used for parsing arguments already divided into an *argv*-style array, some programs need to parse strings that are formatted identically to command lines. To facilitate this, popt provides a function that parses a string into an array of strings, using rules similiar to normal shell parsing.

#include <popt.h>
int poptParseArgvString(char * *s***, int *** *argcPtr***,**
char * *** *argvPtr***);**
int poptDupArgv(int *argc***, const char ** *argv***, int *** *argcPtr***,**
const char * *** *argvPtr***);**

The string s is parsed into an *argv*-style array. The integer pointed to by the *argcPtr* parameter contains the number of elements parsed, and the final *argvPtr* parameter contains the address of the newly created array. The routine **poptDupArgv()** can be used to make a copy of an existing argument array.

The *argvPtr* created by **poptParseArgvString()** or **poptDupArgv()** is suitable to pass directly to **poptGetContext()**. Both routines return a single dynamically allocated contiguous block of storage and should be **free()**ed when the application is finished with the storage.

HANDLING EXTRA ARGUMENTS

Some applications implement the equivalent of option aliasing but need to do so through special logic. The **poptStuffArgs()** function allows an application to insert new arguments into the current **poptContext**.

#include <popt.h>
int poptStuffArgs(poptContext *con***, const char ** *argv***);**

The passed *argv* must have a **NULL** pointer as its final element. When **poptGetNextOpt()** is next called, the "stuffed" arguments are the first to be parsed. popt returns to the normal arguments once all the stuffed arguments have been exhausted.

EXAMPLE

The following example is a simplified version of the program "robin" which appears in Chapter 15 of the text cited below. Robin has been stripped of everything but its argument-parsing logic, slightly reworked, and renamed "parse." It may prove useful in illustrating at least some of the features of the

extremely rich popt library.

```
#include <popt.h>
#include <stdio.h>

void usage(poptContext optCon, int exitcode, char *error, char *addl) {
   poptPrintUsage(optCon, stderr, 0);
   if (error) fprintf(stderr, "%s: %s0, error, addl);
   exit(exitcode);
}

int main(int argc, char *argv[]) {
  char    c;          /* used for argument parsing */
  int     i = 0;      /* used for tracking options */
  char    *portname;
  int     speed = 0;   /* used in argument parsing to set speed */
  int     raw = 0;     /* raw mode? */
  int     j;
  char    buf[BUFSIZ+1];
  poptContext optCon;   /* context for parsing command-line options */

  struct poptOption optionsTable[] = {
                                        { "bps", 'b', POPT_ARG_INT, &speed, 0,
                                                                "signaling rate in bits-per-second", "BPS
                                        { "crnl", 'c', 0, 0, 'c',
                                                                "expand cr characters to cr/lf sequences"
                                        { "hwflow", 'h', 0, 0, 'h',
                                                                "use hardware (RTS/CTS) flow control"
                                        { "noflow", 'n', 0, 0, 'n',
                                                                "use no flow control" },
                                        { "raw", 'r', 0, &raw, 0,
                                                                "don't perform any character conversions
                                        { "swflow", 's', 0, 0, 's',
                                                                "use software (XON/XOF) flow control"
                                        POPT_AUTOHELP
                                        { NULL, 0, 0, NULL, 0 }
  };

  optCon = poptGetContext(NULL, argc, argv, optionsTable, 0);
  poptSetOtherOptionHelp(optCon, "[OPTIONS]* <port>");

  if (argc < 2) {
                                      poptPrintUsage(optCon, stderr, 0);
                                      exit(1);
  }

  /* Now do options processing, get portname */
  while ((c = poptGetNextOpt(optCon)) >= 0) {
    switch (c) {
      case 'c':
        buf[i++] = 'c';
        break;
      case 'h':
        buf[i++] = 'h';
        break;
      case 's':
        buf[i++] = 's';
        break;
      case 'n':
```

```
            buf[i++] = 'n';
            break;
        }
    }
    portname = poptGetArg(optCon);
    if((portname == NULL) || !(poptPeekArg(optCon) == NULL))
      usage(optCon, 1, "Specify a single port", ".e.g., /dev/cua0");

    if (c < -1) {
      /* an error occurred during option processing */
      fprintf(stderr, "%s: %s\n",
            poptBadOption(optCon, POPT_BADOPTION_NOALIAS),
            poptStrerror(c));
      return 1;
    }

    /* Print out options, portname chosen */
    printf("Options  chosen: ");
    for(j = 0; j < i ; j++)
      printf("-%c ", buf[j]);
    if(raw) printf("-r ");
    if(speed) printf("-b %d ", speed);
    printf("\nPortname chosen: %s\n", portname);

    poptFreeContext(optCon);
    exit(0);
}
```

RPM, a popular Linux package management program, makes heavy use of popt's features. Many of its command-line arguments are implemented through popt aliases, which makes RPM an excellent example of how to take advantage of the popt library. For more information on RPM, see http://www.rpm.org. The popt source code distribution includes test program(s) which use all of the features of the popt libraries in various ways. If a feature isn't working for you, the popt test code is the first place to look.

BUGS

None presently known.

AUTHOR

Erik W. Troan <ewt@redhat.com>

Thanks to Robert Lynch for his extensive work on this man page.

SEE ALSO

getopt(3)

Linux Application Development, by Michael K. Johnson and Erik W. Troan (Addison-Wesley, 1998; ISBN 0-201-30821-5), Chapter 24.

popt.ps is a Postscript version of the above cited book chapter. It can be found in the source archive for popt available at: ftp://ftp.rpm.org/pub/rpm.