Standard Command Line Parsing

by John Orr

One of the new features in release 2.0 is system standard command line parsing. Its presence has two benefits: it standardizes the way in which the user supplies command line arguments, making it much easier on the user, and it also removes some tedious programming (and code size) from every application that uses it.

The *dos.library*'s ReadArgs() routine is the heart of this feature:

```
struct RDArgs *rda = ReadArgs( UBYTE *argtemplate, LONG *argarray, struct RDArgs *myrda );
```

This function stores each argument supplied on the command line in its corresponding entry in the array of LONG words, argarray. The format in which ReadArgs() stores each argument is based on the description of the command line supplied in the first argument, argtemplate. This description is a C-style string containing a name for each argument.

Each argument name in the template should be a full, descriptive name (for example ‘‘Quick’’ not ‘‘Q’’). Each option can be prepended by an abbreviation of the form ‘‘abbrev=’’ (for example ‘‘Q=Quick’’). The argtemplate options must be delimited by commas. Avoid using the names of common commands as keywords, otherwise the user will have to delimit them with quotes.

The ordinal position that an argument appears in the description string determines what its corresponding position in argarray is (the first argument corresponds to the first entry in argarray, the second argument corresponds to the second entry in argarray, ...). There must be an entry in argarray for each argument in the description string so that ReadArgs() has a place to store each argument's value.

Each argument name in the template can be followed by modifiers that tell ReadArgs() the format of the argument. The valid modifiers are:

> /S - Switch. This is considered a boolean variable. If this option is present, ReadArgs() will set the corresponding array entry in argarray (an array of LONGs) to something besides zero. If the option is not present, ReadArgs() will set the entry to 0.
>
> /K - Keyword. This means that ReadArgs() will not fill in the corresponding entry in argarray unless the keyword appears with the parameter. For example, if the template is ‘‘Name/K’’, then unless ‘‘Name=<string>’’ or ‘‘Name <string>’’ appears in the command line, the ‘‘Name’’ entry in argarray will not be altered by ReadArgs().

/N - Number. This means the parameter is considered a decimal integer, and ReadArgs() will convert it to a LONG. If the argument is not valid, ReadArgs() will fail. If the option is present in the command line (and it is valid), ReadArgs() will fill in the corresponding entry with a pointer to the LONG.

/T - Toggle. This is similar to the switch (/S) modifier, but causes the corresponding boolean (in argarray) to toggle. For example, if the array entry corresponding to an argtemplate of ‘‘binary\T’’ is set to something besides `FALSE` and the word ‘‘binary’’ appears by itself on the command line, ReadArgs() will toggle that array entry to `FALSE`.

/A - Always. This modifier tells ReadArgs() that this option is required. ReadArgs() will fail if the keyword does not appear in the command line.

/F - Final (rest of line). If this is specified, the part of the command line that follows this option is taken as the parameter for this option, even if other option keywords appear in it.

/M - Multiple arguments. This means the argument will take any number of strings (or integers as this modifier can be used with the /N modifier), returning them as an array of strings. Any arguments not considered to be part of another option will be added to this option. Only one /M should appear in a template. Example: for a template ‘‘Dir/M,All/S’’ the command-line ‘‘foo bar all qwe’’ will set the boolean ‘‘all’’, and return an array consisting of ‘‘foo’’, ‘‘bar’’, and ‘‘qwe’’. The entry in the array will be a pointer to an array of string pointers, the last of which will be `NULL`.

There is an intentional interaction between /M parameters and /A parameters. If there are unfilled /A parameters after parsing, ReadArgs() will grab strings from the end of a previous /M parameter list to fill the /A’s. This is used for things like Copy (‘‘From/A/M,To/A’’).

If the user does not supply a non-required argument (one without the ‘‘/A’’ modifier) on the command line, ReadArgs() will leave the argument’s argarray entry alone. Before calling ReadArgs(), a program should either set the argarray entries to reasonable default values or clear them, so the application can’t be confused by any garbage values left in the array.

If it is successful, ReadArgs() returns a pointer to a RDArgs structure (from *<dos/rdargs.h>*). ReadArgs() uses this structure internally to control its operation. It is possible to pass ReadArgs() a custom RDArgs structure (myrda in the ReadArgs() prototype above). For most applications myrda will be `NULL`, as most applications do not need to control ReadArgs().

```
struct RDArgs {
    struct    CSource RDA_Source; /* Select input source */
    LONG      RDA_DAList;         /* PRIVATE. */
    UBYTE     *RDA_Buffer;        /* Optional string parsing space. */
    LONG      RDA_BufSiz;         /* Size of RDA_Buffer (0..n) */
    UBYTE     *RDA_ExtHelp;       /* Optional extended help */
    LONG      RDA_Flags;          /* Flags for any required control */
};
```

Any successful call to ReadArgs() (even those that use a custom RDArgs structure) must be complemented with a call to FreeArgs() to free the resources that ReadArgs() allocates:

```
void FreeArgs(struct RDArgs *rda);
```

where rda is the RDArgs structure used by ReadArgs().

An application can use a custom RDArgs structure to provide an alternate command line source, an alternate temporary storage buffer, or an extended help string. The custom RDArgs structure must be allocated with AllocDosObject() and deallocated with FreeDosObject(). See the Autodocs for more details on these functions.

The RDArgs.RDA_Source field is used to supply ReadArgs() with an alternate command line to parse. If this field is non-NULL, ReadArgs() will use it as a pointer to a CSource structure describing the alternate command line. The CSource structure (from *<dos/rdargs.h>*) is as follows:

```
struct CSource {
    UBYTE   *CS_Buffer;
    LONG    CS_Length;
    LONG    CS_CurChr;
};
```

Where CS_Buffer is the command line to parse, CS_Length is the length of CS_Buffer, and CS_CurChr is the position in CS_Buffer from which ReadArgs() should begin its parsing. Normally CS_CurChr is initialized to zero.

ReadArgs() uses the RDArgs structure's RDA_DAList field for internal use. This field *must* be set to `NULL` before ReadArgs() uses this structure.

The RDA_Buffer and RDA_BufSiz fields allow an application to supply a fixed-size buffer in which to store parsed data. This allows the application to pre-allocate a buffer rather than requiring ReadArgs() to allocate buffer space. If either RDA_Buffer or RDA_BufSiz is `NULL`, ReadArgs() assumes the application has not supplied a buffer.

RDA_ExtHelp is a text string which ReadArgs() displays if the user asks for additional help. The user asks for additional help by typing a question mark when ReadArgs() prompts the user for input (which normally happens only after he or she types a question mark as the only argument on the command line).

RDA_Flags is a bit field used to toggle certain options of ReadArgs(). Currently, only one option is implemented, `RDAF_NOPROMPT`. When set, `RDAF_NOPROMPT` prevents ReadArgs() from prompting the user.

The following code, *ReadArgs.c*, uses a custom RDArgs structure to pass a command line to ReadArgs.

```
;/* ReadArgs.c - Execute me to compile me with Lattice 5.10a
LC -b1 -cfistq -v -y -j73 ReadArgs.c
Blink FROM LIB:c.o,ReadArgs.o TO ReadArgs LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/


#include <dos/dos.h>
#include <dos/rdargs.h>
#include <clib/dos_protos.h>
#include <clib/alib_stdio_protos.h>


#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif
```

```
UBYTE *vers = "\0$VER: ReadArgs 1.0";

#define TEMPLATE "S=SourceFiles/A/M,D=DebugLevel/K/N,L=link/S"
#define OPT_SOURCE  0
#define OPT_DEBUG   1
#define OPT_LINK    2
#define OPT_COUNT   3

/* The array of LONGs where ReadArgs() will store the data from
** the command line arguments.  C guarantees that all the array
** entries will be set to zero.
*/
LONG result[OPT_COUNT];

/* My custom RDArgs */
struct RDArgs *myrda;

ULONG StrLen(UBYTE *);

void main(void)
{
    UWORD x;
    UBYTE **sourcefiles;

    /* Need to ask DOS for a RDArgs structure */
    if (myrda = (struct RDArgs *)AllocDosObject(DOS_RDARGS, NULL))
    {
        /* set up my parameters for ReadArgs() */

        /* use the following command line */
        myrda->RDA_Source.CS_Buffer = "file1 file2 file3 D=1 Link file4 file5\n";
        myrda->RDA_Source.CS_Length = (LONG)StrLen(myrda->RDA_Source.CS_Buffer);

        /* parse my command line */
        if (ReadArgs(TEMPLATE, result, myrda))
        {
            /*start printing out the results */

            /* We don’t need to check if there is a value in
            ** result[OPT_SOURCE] because the ReadArgs() template
            ** requires (using the /A modifier) that there be
            ** file names, so ReadArgs() will either fill in a
            ** value or ReadArgs() will fail.
            */
            sourcefiles = (UBYTE **)result[OPT_SOURCE];
            /* VPrintf() is a lot like Printf() except it’s in
            ** ROM, and the arguments are referenced from an
            ** array rather than being extracted from the stack.
            */
            VPrintf("Files specified:\n", NULL);
            for (x=0; sourcefiles[x]; x++)
                VPrintf("\t%s\n", (LONG *)&sourcefiles[x]);

            /* Is there something in the "DebugLevel" option?
            ** If there is, print it.
            */
            if (result[OPT_DEBUG])
                VPrintf("Debugging Level = %ld\n", (LONG *)result[OPT_DEBUG]);

            /* If the link toggle was present, say something about it. */
            if (result[OPT_LINK])
                VPrintf("linking...\n", NULL);
            FreeArgs(myrda);
        }
        FreeDosObject(DOS_RDARGS, myrda);
    }
}


ULONG StrLen(UBYTE *string)
{
    ULONG x = 0L;

    while (string[x++]);
    return(x);
}
```