Parsing the Command Line

Contents

Chapter 1

Parsing the Command Line

1.1 Chapter 5 - Parsing the Command Line

```
                Previous Chapter:                                 Next Chapter:
 4. Buffered IO
---------------------------------------------------------------
```

CHAPTER 5 - PARSING THE COMMAND LINE

1.2 Introduction

INTRODUCTION

When you are using the Shell (CLI) to launch programs you can add arguments after the program name. These arguments will be collected by the program and interpreted. In C you normally use the famous "argc" and "argv" variables.

Previously you had to write your own functions which interpreted the command line. Since every programmer wrote his or her own command line interpretator, or "parser" as they are also called, each program used its own routines to parse the arguments. The user was confronted with hundreds of different methods on how to enter the arguments, and this was very confusing for the user.

With Release 2 some special command line parsing routines were therefore included in the dos library. These routines take care of all the trouble with collecting arguments and interpreting them. As a programmer you will therefore save a lot of time since do not have to write your own parsing routines any more. These new routines will now also give the programs a standardized user-interface.

If you need to collect arguments from the command line you should use theses new routines in the dos library. You will save development time and get programs which follows the new standards. Both you and your users will benefit from it.

Plese note that these routines described in this chapter deals with parsing arguments from the command line (from the Shell, CLI). If you also want your programs to understand arguments from the Workbench you must also include the special routines described in the "Workbench" manual.

1.3 Command Line Information

COMMAND LINE INFORMATION

Not long ago all type of work with the computers was done by typing the commands and instructions. Nowadays the user can work a mouse, a light pen, or even directly talk to the computer (even if this voice control system still needs to be developed quite a bit before it will be practical). However, although there are a lot of new methods of collecting input from the user, the old trusted command line is still very much used. It maybe take some time to learn how to work with it, but the command line gives you a flexibility which can not be matched.

On the Amiga the users work in the "Workbench" enviroment, and can do most things by simply pointing and clicking with the mouse. However, since there are a lot of users, including me, who still now and then want to use the more flexible command line enviroment there exist the "Shell" or the simpler "CLI".

1.4 Programs & Arguments

PROGRAMS & ARGUMENTS

The advantage with the Shell and CLI is that it is very easy to give programs extra information when you start them. You simply write the name of the program and on the same line include one or more "arguments". To delete a file you normally use the program "Delete". The program must of course be told which file(s) to delete, and you give the program this information by adding one or more "arguments" as illustrated below: (See also picture Programs&Arguments.pic)

```
  1.Prog:>
  1.Prog:> Delete df0:ReadMe.doc df0:Unimportant.dat
```

The program is called "Delete", and the arguments are "df0:ReadMe.doc" and "df0:Unimportant.dat".

1.5 Command Line Template

COMMAND LINE TEMPLATE

Some programs can only handle one argument while others can handle several ones. The "Delete" program we discussed could for example handle several file names, while a drawing program maybe only can read one picture file. Some programs expect a "keyword" before the actual argument. When you for example want to format a disk you need to include the keywords "Drive" and "Name":

```
  1.Prog:>
  1.Prog:> Format Drive df0: Name CPrograms
```

Other programs are using "switches" which means that something will be "turned on or off" if a special argumen is given. When you use the Delete command you can for example add the switch "Quiet" which tells the program not to list the files which are deleted:

```
  1.Prog:>
  1.Prog:> Delete df0:System Quiet
```

Some arguments may be always required, you must for example give the Delete program at least one filename. Other arguments are optional like the "Quiet" switch.

As you understand the routines of handling arguments (the "parsing" routines) are very complex. The problem is that this complex enviroment must be made as simple as possible to use. Although every program uses different types of arguemnts and all have different requirements the parsing routines must be standardlized. The good news is that the new functions in the dos library gives us this.

The parsing routines are using what may be called a "Command Line Template". It is a text string in which the possible arguments, their requirements and function are incuded. Every unique argument type use a "Command Template". In the command template you enter the name of the argument and adds any necessary "options" which will be listed later on. These special options tells the parsing routine what type of argument should be expected, and what function it has.

This command line template is used by the parsing routines, but can also be viewed by the user. If you enter the name of the program and adds a question mark as the only argument you will see the complete command line template. To see the command line template used by the Delete program simply type "Delete ?":

```
  1.Prog:>
  1.Prog:> Delete ?
  FILE/M/A,ALL/S,QUIET/S,FORCE/S
```

The Delete program needs at least one argument which is the name of the file that should be deleted. In the "command template" the argument type is called "File". You can call the command templates almost anything you like, but you should notice that the user can look at the command line template by adding a question mark just after the program name (you can turn off this feature if you like but it is not recommended), and the name should somehow give the user a hint of what is expected.

Since the "File" argument was needed or else the program would fail a special "/A" option is added after the template name. This option tells the parsing routines that this argument is "Always Required". If the user does not enter an argument which is always required the parsing routine will fail. Since the user may enter several file names this command template also have the option "/M" set. The "/M" option tells the parsing routines that this is a "Multiple Argument", and one or more arguments of this type should be expected. The complete command template for the "File" argument of the "Delete" program is:

```
  "FILE/M/A"
```

(The template is called "File", one or more arguments of this type should be expected, and at least one argument of this type must be entered.)

The file name(s) was not the only type of argument the Delete program understands. You can also add the switch "Quiet". This argument type is, as previously explained, a switch which will be turned on if the argument is included, and be off if it is not included. To tell the parsing routines that the command template is a switch you add the "/S" option ("Switch"). The name of this command template is not surprisingly "QUIET", and the whole command template is therefore:

```
  "QUIET/S"
```

There are two more types of arguments that can be added,
"Force" and "All", and they are also switches so I will not
explain them further.

Each command template must be separated with a comma, and no
spaces are allowed. (However, when the user should run the
program and enters the argument he/she should separate the
arguments with spaces and not use commas).

The parsing routines are not case sensetive which means that
you can write "qUiEt" as well as "QUIET".

The complete "Command Line Template" for the Delete program
will therefore look like this: (Please notice the destinction
between "Command Template" which is the definitions of one
argument type, and the "Command Line Template" which is the
name of all the "Command Templates".)

```
  "FILE/M/A,ALL/S,QUIET/S,FORCE/S"
```

This types of command line template can be used with the parse
functions. The advantage with this "string" form is that it can
by used by the parsing routines, but still be displayed for the
user and he/she will then know what must be typed and which
features are available.

1.6 Template Options

TEMPLATE OPTIONS

Here is the complete list of all avilable options you may add
to the command template. You may use several options for each
command template. Simply add them together after the name.
(To add both the "/A" and "/M" option simply add the string
"/A/M" at the end of the command template name.)

```
  Option  Description
  -------------------------------------------------------------
      /A  Always required argument. This argument must always
          be included in the command line, or else the parse
          function will fail.

      /F  Final argument of the line. All text after this
          argument will be considered as a part of the argument
          nomatter what is written.

      /K  Keyword is required. The user must include the name
          of the command template before he/she type the actual
          argument. The user may add an equal sign between the
          "keyword" (the name of this command template) and the
          actual argument.

          Example. If your program has the following command
          template, "NAME/K", the user must write:
```

```
        NAME <the actual argument>   or
        NAME=<the actual argument>

  /M  Multiple arguments may be entered. The user may write
      more than one argument of this type. If this option
      is not used the user may only enter one argument of
      this type. If the option is on the user may enter one
      or more arguments. (The user may of course enter no
      argument at all of this type as long as the "/A"
      option is not included as well in the command
      template.)

      The advantage with this option is that all arguments
      that could not fit in anywhere else will be added to
      this command template. For example: the Delete
      program has the following command line template:

        "FILE/M/A,ALL/S,QUIET/S,FORCE/S"

      As you can see, the user may enter as many file names
      as he/she wants since the "/M" option is set. (At
      least one file name must be given since the "/A"
      option is set.) To delete the the files f1, f2 and f3
      and do it quietly you may write:

        Delete f1 f2 f3 Quiet

      You can also write:

        Delete f1 Quiet f2 f3

      Since the arguments f2 and f3 did not fit any other
      template (were not "All", "Quiet" or "Force") they
      will be added to the "FILE" template.

      Note! Only one "/M" option may be used on a command
      line template. (It would otherwise be impossible for
      the parse functions to know to which commad template
      the extra arguments should go to if there were
      several "/M" options.)

  /N  Number argument. If the user enters this argument it
      must be an integer value (a value with no period).
      The value may be negative.

  /S  Switch argument. This option should be used if you
      want the command template to act like a switch. If
      his argument is entered it will be turned on, else
      it will be turned off.

  /T  Toggle argument. This option should be used if you
      want the command template to act like a switch that
      is toggled on and of. If this argument is enterd the
      switch will be toggled (if it was off it will be
      turned on, and if it was on it will be turned off).
```

1.7 Abbreviations

```
ABBREVIATIONS

You should always use complete words as a name for a command
template since it will be easier for the user (and for you as
a programmer also) to understand and remember what that
argument type is about. However, to save time you can allow
the user to enter abreviations. If you have the command
template name "Rate" you might want allow the abreviation
"R" as well. To do this you simply add "R=" at the begining
of the name: "R=Rate".
```

1.8 Reserved Names

```
RESERVED NAMES

There are some command template names that are "reserved". You
may only use these names if your command templates mean the
same thing as the reserved names. This will give the user a
more standardized interface. ("Commodore Amiga Technical
Information")

  Name       Purpose
  -------------------------------------------------------------
  FILE         The argument should be the name of the file that
               the program should use. (One file.)

  FILES/M      The arguments should be the name of the files
               that the program should use. (One or more files,
               note the "/M" multiple arguments option.)

  NOGUI/S      This argument should be used as a switch to turn
               off the graphics interface (GUI = "Graphics User
               Interface"). This can for example be handy if
               the user want to save memory, and does not want
               fancy gadgets with a lot of nice but memory
               hungry graphics. As little graphics as possible
               should be used if your program receives this
               argument. (Mouse operations ar usually
               considered to be a part of the GUI, but if you
               want to turn of the mouse completely or simply
               remove fancy graphics is up to you.)

  PORTNAME/K   This argument should be the name of the ARexx
               port that will be used. Note that the keyword
               "PORTNAME" must preceded the name of the
               ARexx port (the "/K" keyword option is used.)

  PUBSCREEN/K  This argument should be the name of the public
               screen this application should use. Note that
               the keyword "PUBSCREEN" must preceded the name
               of the public screen.
  SETTINGS/K   This argument should be the name of the
               preferences file that will be loaded when the
               program is started. Note that the keyword
               "SETTINGS" must preceded the file name.

  STARTUP/K    This argument should be the name of the ARexx
               program that will be executed when this program
               is started. Note that the keyword "STARTUP" must
               preceded the name of the ARexx file.
```

1.9 Some Examples

SOME EXAMPLES

For a summary of the different parts of the command line template see picture CommandLineTemplate.pic .

I will now show some examples on different types of command templates. I will also show some command lines that the user may and may not use. (I assume that the program name is "MyProg".)

Template: "FILE"

```
  Command line                    Comments
  --------------------------------------------------------------
  MyProg                          OK! The file name was optional
                                  since the "/A" option was not
                                  used.

  MyProg Bird.snd                 OK! One file name may be
                                  entered.

  MyProg Bird.snd Sea.snd         WRONG! Only one file name may
                                  be entered since the "/M"
                                  option was not used.
```

Template: "FILES/M"

```
  Command line                    Comments
  --------------------------------------------------------------
  MyProg                          OK! The file name was optional
                                  since the "/A" option was not
                                  used.

  MyProg Bird.snd                 OK! One file name may be
                                  entered even if the "/M"
                                  option is used.

  MyProg Bird.snd Sea.snd         OK! Several file names may
                                  be entered since the "/M"
                                  option was used.
```

```
Template: "FILE/A"

  Command line                    Comments
  --------------------------------------------------------------
  MyProg                          WRONG! A file name is required
                                  since the "/A" option was
                                  used.

  MyProg Bird.snd                 OK! One file name should be
                                  entered.

  MyProg Bird.snd Sea.snd         WRONG! Only one file name may
                                  be entered since the "/M"
                                  option was not used.


Template: "VOLUME/N"

  Command line                    Comments
  --------------------------------------------------------------
  MyProg                          OK! The volume number was
                                  optional since the "/A" option
                                  was not used.

  MyProg 30                       OK! The volume may be set as
                                  long as it is an integer
                                  value (without any period).

  MyProg 30.25                    WRONG! Only integer values may
                                  be entered!

  MyProg Bird.snd                 WRONG! It must be an integer
                                  number since the "/N" option
                                  was used.


Template: "VOLUME/N/K"

  Command line                    Comments
  --------------------------------------------------------------
  MyProg 30                       WRONG! The number must be
                                  preceded by the keyword
                                  "VOLUME" since the "/K" option
                                  was used.

  MyProg Volume 30                OK! The keyword, as well as
                                  all other arguments, are not
                                  case sensetive.

  MyProg Volume=30                OK! You may use an equal sign
                                  between the keyword and the
                                  actual argument instead of a
                                  space.


Template: "F=FILTER/S"
```

```
  Command line                    Comments
  -------------------------------------------------------------
  MyProg Filter                   OK! The switch "FILTER" will
                                  be turned on.

  MyProg F                        OK! The switch "FILTER" may be
                                  abbreviated to "F" since we
                                  added the "F=" string.
                                  ["<Abbreviation>="]


Template: "FILES/A/M,V=VOLUME/N/K,F=FILTER/S"

  Command line                    Comments
  -------------------------------------------------------------
  MyProg Bird.snd Volume=30       OK!

  MyProg Volume=30 Bird.snd       OK! The order of the arguments
                                  is irrelevant.

  MyProg Volume=30 Filter         WRONG! A file name is required
                                  since the "/A" option was
                                  used.

  MyProg Bird.snd Sea.snd V=30 F  OK! Several file names may be
                                  used since the "/M" option was
                                  used, and the "VOLUME" and
                                  "FILTER" words may be
                                  abbreviated to "V" and "F".
```

1.10 Parse the Command Line

```
                PARSE THE COMMAND LINE

When you want to parse the command line (collect arguments from
the command line you simply have to follow these six steps:


                 1. Check the dos library version.

                 2. Design the command line template.

                 3. Prepare the argument array.

                 4. Collect the arguments.

                 5. Examine the result and act accordingly.

                 6. Finally you should clean up after you.
```

1.11 Check Dos Library Version

CHECK DOS LIBRARY VERSION

The parsing functions were introduced with Release 2. The first version of the new dos library (V36) was rather buggy, and the parsing routines did not work correctly. Before you may use any of the parsing functions you must therefore make sure that the user really has dos library version 37 or higher.

How to check the current dos library version was explained in the first chapter "AmigaDOS". However, here is a short example:

```
  /* Declare an external global library */
  /* pointer to the Dos library:        */
  extern struct DosLibrary *DOSBase;

  - - -

  /* We need dos library version 37 or higher: */
  if( DOSBase->dl_lib.lib_Version < 37 )
  {
    /* Too old dos library! */
    printf( "This program needs Dos Library V37 or higher!\n" );

    /* Exit with an error code: */
    exit( 20 );
  }
```

1.12 Design the Command Line Template

DESIGN THE COMMAND LINE TEMPLATE

The parsing functions are using a command line template, which is simply a string where you include all your requirements as previously explained.

When you set the command template names you should give them complete names which are easy to understand and can not be mixed up. If you want to can give the user an option of abbreviate long names.

Any arguments that are necessary for your program to work must have the "/A" option set, and if your program supports multiple arguments of one type you must set the "/M" option for that command template. Please remember that only one "\M" option can be used in the command template.

It is important that you design the command line template carefully, and make sure that the parsing functions really can handle it. If you use several command templates in the command line template you must be careful so that the parsing routine can separate the arguments correctly:

* There is no danger with "switches" (option "/S" anb /T") since the parsing routines will automatically know which

argument belongs to which command template. For example: the command line template "MONO/S,FILTER/S" is easy to handle. If the user enters the argument "Mono" it can only be the "MONO" switch.

* When you are using the other types of arguments you must design the command line template so the different arguments can be separated, and there is no risk of mixing them up. To do this you normally need to use the "keyword" option (option "/K").

  For example: We have a small program that loads a sound file and a picture file. A command line template like "SOUND,PICTURE" should however not be used! It is imposible to know which argument is the name of the sound file and which is the name of the picture. The user maybe writes:

  ```
  MyProg Bird.snd Fun.pic
  ```

  However, the user might equally well write:

  ```
  MyProg Fun.pic Bird.snd
  ```

  As you see, the user has changed the order of the arguments, and it would be impossible for the parsing functions to know that the first argument "Fun.pic" actually is the picture file, and that the second argument "Bird.snd" is the sound file. (The order of the arguments on the command line should not matter, so do not expect the first argument to be the sound file and the second one the picture file.)

  What you should do is to add a keyword for at least one of the command templates. Possible examples:

"SOUND/K,PICTURE"	The user must now enter the keyword "SOUND" before the actual file name. There will then not be any uncertainty of which argument is which.
"SOUND/K,PICTURE/K"	Same as above except that the user now must enter the keyword "PICTURE" just before the actual picture name.
"SOUND/K,PICTURE/K"	The user must now enter the keywords "SOUND" in front of the sound file, and "PICTURE" in front of the picture name.

Here is a typical example on how to decalre a command line templat:

```
/* Here is our command line template: */
#define MY_TEMPLATE "SoundFile/A,V=Volume/K/N,F=Filter/S"
```

1.13 Prepare the Argument Array

PREPARE THE ARGUMENT ARRAY

After the command line has been parsed are collected data
stored in an array of LONG variables. One entry (LONG variable)
is needed for every command template.

```
  ****************************************************
  *                                                  *
  *  Remember! The array must contain at least as many  *
  *  entries (LONG variables) as there are command      *
  *  templates in your command line template!           *
  *                                                  *
  ****************************************************
```

It is important that this "argument" array has been cleared,
all variables set to 0, before it is used. There is only one
exception, and that is if you want to use the "toggle" ("/T")
option, which will toggle the corresponding variable in the
array. More about this later.

Here is an example: (Our command line template has three
command templates, and consequently we need an array with at
least three entries.)

```
  /* Three command templates are used: */
  #define NUMBER_TEMPLATES 3

  /* The results of the parsing will be stored here: */
  LONG arg_array[ NUMBER_TEMPLATES ];

  - - -

  /* We will now clear the array: */
  for( loop = 0; loop < NUMBER_TEMPLATES; loop++ )
    arg_array[ loop ] = 0;
```

1.14 Collect the Arguments

COLLECT THE ARGUMENTS

Once you have designed the command template and allocated
(declared) an array of LONG variables you can parse the command
line. To do this you simply call the ReadArgs()

The ReadArgs() function needs a pointer to the command line
emplate that should be used and a pointer to the array of of
LONG variables.

If the command line could successfully be parsed a pointer is
returned to a RDArgs structure that has been allocated. If the
command line could not be parsed NULL is returned. (The RDArgs
structure will be explained later on in this chapter.)

1.15 Examine the Arguments

```
EXAMINE THE ARGUMENTS

Once you have successfully parsed the command line you may
examine the "argument" array. The result of the first
command template will be placed in position 0 of the array,
the result of the second command template will be placed in
position 1, and so on...

What type of values that will be stored in the array depends on
what type of command templates are used (which options that
where set): (In all examples we assume that the argument type
we work with has the position "POSITION" in the argument array.)

1. The default is that the array will contain a pointer to a
   string where a copy of the corresponding argument is placed.
   If no argument of this type was entered the pointer will be
   NULL.

   Remember to check that the pointer really points to a string
   before you try to examine the string! If the "/A" option was
   used together with this template we know that the pointer
   must point to a string since this argument was required, and
   the  ReadArgs()
   was not included. However, it it best to always check the
   pointer that it does not point to NULL before you use it!

   Here is an example on how to print a collected string
   argument:

     /* Print single string argument: */
     if( arg_array[ POSITION ] )
       printf( "File name: %s\n", arg_array[ POSITION ] );


2. If the "/M" (multiple arguments) option was used there may
   be zero or more arguments that fitted this template. The
   LONG value in the array  will therefore contain a pointer to
   an array of strings, where the last string pointer is set to
   NULL. (Note that it is a pointer to an array of string
   pointers.)

   Here is an example on how to print all of the collected
   strings used with a multiple command template:

     /* Simple loop variable: */
     int loop;

     /* Store the pointer to the array of string pointers here: */
     UBYTE **string_array;

     - - -

     /* Before we can use the pointer we must check that  */
     /* we realy have a pointer to other strings, and not */
     /* just NULL.                                        */
     if( arg_array[ POSITION ] )
     {
       /* Store the pointer to the array of stirng pointers: */
       /* (Double pointers are a bit tricky...)              */
       string_array = (UBYTE **) arg_array[ POSITION ];

       /* What we have to do now is to examine all strings  */
       /* with help of a simple while loop. The last string */
       /* in the array has been set to NULL so we know were */
       /* the list ends.                                    */

       /* Start with the first string: */
       loop = 0;

       /* Print all file names: */
       while( string_array[ loop ] )
       {
         /* Print the file name: */
         printf( "File name: %s\n", string_array[ loop ] );

         /* Increase the counter: */
         loop++;
       }
       /* All file names have now been printed! */
     }
     else
       printf( "Not a single file name...\n" );
```

3. If the "/N" (numeric argument) option was used the LONG
   variable in the array will contain a pointer to the value
   that was entered, or be NULL if no value was entered.

   Here is an example on how to print a collected value:

   ```
     /* A pointer to the volume value: */
     LONG *value;

     - - -

     /* Check that the user really has entered a number: */
     if( arg_array[ POSITION ] )
     {
       /* Get a pointer to the value: */
       value = (LONG *) arg_array[ POSITION ];

       /* Print the value: */
       printf( "The value is: %ld\n", *value );
     }
     else
       printf( "No value was entered!\n" );
   ```

4. Finally there exist the switches "/S" (or "flags" as they
   are sometimes called). If the switch was set the variable in
   the array will be non-zero. If the switch was not set the

```
   value will be 0.

   If you used the "/T" switch the value in the array will
   toggle if the argument was set. If the value in the array
   was 0 it will be toggled to a non-zero value, and if the
   value was non-zero it will be togled to 0. The argument
   array should normally always be set to zero before you
   collect the arguments. The variable used for a toggle
   switch may however be set to ~0 if you want it to have a
   non-zero value before you collect the arguments. (The toggle
   option is normally only used when you parse your own command
   strings. The command line is only entered once and a toggle
   option would not be of much use here.)

   Here is an example to check if a switch (flag) was set or
   not:

     /* Was the switch set? */
     if( arg_array[ POSITION ] )
        printf( "The switch (flag) was set!\n" );
     else
       printf( "The switch (flag) was not set!\n" );
```

1.16 Clean Up

```
CLEAN UP

When you successfully call  ReadArgs()
allocate some memory where the collected arguments will be
stored. It is this memory the pointers in the argument array
points to. Once you have examined the collected arguments and
do not need them any more you should free the data. You do this
by simply calling the  FreeArgs()

If you did not supply the ReadArgs() function with your own
RDArgs structure the function will have created one for you.
This structure will then also be deallocated when you call
FreeArgs().

If you have created your own RDArgs structure you have to
deallocate it yourself with the help of  FreeDosObject()
you must still call FreeArgs() to deallocate the data used for
the arguments.
```

1.17 Advanced Parsing Routines

```
                ADVANCED PARSING ROUTINES

As you saw most of the parsing job was done automatically for
you. It collected the command line, parsed it automatically
(you only had to tell it which arguments to look for), and
finally the parsing routine inserted all the sorted arguments
```

into your array. Handy and easy to use.

However, you might want to have more control over the parsing routine, add some extra help or parse your own strings rather than the default command line etc... This is also possible, and will be explained in the following sections.

1.18 The RDArgs Structure

THE "RDArgs" STRUCTURE

When you use the advanced parsing routines you need to work with a structure called
RDArgs
.

To allocate a RDArgs structure you must use the special
AllocDosObject()
function is that any future changes in the size of the RDArgs strucute will not affect the compability of your program. For example, if the strucute is extended the AllocDosObject() will automatically allocate more memory. The function will also initialize some fields of the structure so it can be used directly.

If you have successfully created an RDArgs structure with help of AllocDosObject() you also have to deallocate it yourself when you do not need the structure any more. Anything which has been allocated with AllocDosObject() must be deallocated with help of the FreeDosObject()

Here is an example on how to create and later on free a RDArgs structure:

```
  /* Declare a pointer to our RDArgs structure: */
  struct RDArgs *my_rdargs;

  - - -

  /* Allocate a RDArgs structure: */
  my_rdargs = (struct RDArgs *)
    AllocDosObject( DOS_RDARGS, NULL );

  /* Did we get the RDArgs structure? */
  if( !my_rdargs )
  {
    /* Problems! Inform the user: */
    printf( "Could not create the RDArgs structure!\n" );

    /* Quit with an error code: */
    exit( 20 );
  }

  - - -

  /* Here you may use the RDArgs structure as will */
  /* be explained in the following sections.       */

  - - -

  /* Deallocate the RDArgs structure: */
  FreeDosObject( DOS_RDARGS, my_rdargs );
```

1.19 CSource Structure

CSource Structure

The CSource structure looks like this: (defined in header file "dos/rdargs.h")

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

CS_Buffer: Pointer to some memory where a string may be stored.

CS_Length: The lenght of the string.

CS_CurChr: The current "position" in the string. This field is normally used by the parsing routines to keep track of where they are for the moment in the string they are examening. Normally this field should be set to zero.

1.20 RDArgs Structure

RDArgs Structure

The RDArgs structure looks like this: (defined in header file "dos/rdargs.h")

```
  struct RDArgs
  {

                  struct CSource
```

```
                RDA_Source;
    LONG RDA_DAList;
    UBYTE *RDA_Buffer;
    LONG RDA_BufSiz;
    UBYTE *RDA_ExtHelp;
    LONG RDA_Flags;
  };

RDA_Source:  The first object is a CSource structure. It is
             in this structure you can set your own command
             string which will be parsed instead of the the
             defult command line. (If the fields in this
             structure is set to NULL the defautl command
             line will be parsed.)

RDA_DAList:  Private area, used by AmigaDOS only. Set to NULL.

RDA_Buffer:  Normally when you use ReadArgs() some memory will
             be automatically allocated to store the arguments
             in. (It is this memory which you then have to free
             with help of FreeArgs().) If you want ReadArgs()
             to take care of all memory handeling you should
             set this field to NULL and the next one
             ("RDA_BufSiz") to zero.

             However, sometimes you might want to allocate
             the memory yourself before you call the ReadArgs()
             function. What you have to do is to allocate some
             mempory and give this field a pointer to that
             memory. You must then also tell this structure
             how much memory you have allocated, and this you
             do in the next field ("RDA_BufSiz").

             Note! If you have allocated the memory yourself
             for this field you should not call FreeArgs()
             but instead deallocate the memory yourself.

RDA_BufSiz:  Normally set to 0, but if you have allocated the
             memory yourself as described above you must set
             the size (in bytes) here.

RDA_ExtHelp: If you want you can give this field a pointer to
             a string which will be used as an "extra" help
             line. If the user types a question mark he/she
             will see the command line template. If the user
             now types another question mark he/she will see
             this string.

RDA_Flags:   In this field you can set special flags which will
             in different ways affect the parsing routines. For
             the mement you can only use the "RDAF_NOPROMPT"
             flag. It will turn off the command line template
             so the user can not see it if he/she types a
             question mark (?).
```

1.21 Turn Off the Command Template

TURN OFF THE COMMAND TEMPLATE

It is possible to turn off the command line template so the user can not see it when he/she types a question mark. This should not be done in most cases since the user may need this help. However, you maybe want to print your own help messages instead. (If you turn off this "autohelp" you will be able to scan for question marks and if you find one you can print some instructions for example.)

To turn off the command line template so the user can not see it you need to set the flag "RDAF_NOPROMPT" in the "RDA_Flags" field of the RDArgs structure before you call ReadArgs().

Here is a (very) simple example:

```
  /* Turn of the comman line template help: */
  my_rdargs->RDA_Flags = RDAF_NOPROMPT;

  /* Call ReadArgs() etc... */
```

1.22 Add Extra Help Line

ADD EXTRA HELP LINE

If you want you can add an extra help line which will be displayed if the user first types a question mark to see the "command line template help" and then types another question mark. The advantage with this extra help line is that you can give the user some more understandable help. The command line template is not the easiest thing to understand in the beginning.

To set an extra help line simple give the "RDA_ExtHelp" field of the RDArgs structure a pointer to a text string which contains the text you want to display. (This must of course be done before you call ReadArgs().)

Here is a simple example:

```
  /* Set an extra help line: */
  my_rdargs->RDA_ExtHelp = (UBYTE *)
    "You must enter a name of a sound file!";
```

See Example 4 for more information:

   Read!   Run!   Edit!

1.23 Parse Your Own Command Strings

```
                PARSE YOUR OWN COMMAND STRINGS

The parsing functions are normally examining the command line
to look for arguments to act on. However, you can equally well
use the routines to parse normal strings. This can be useful
if you have collected information from the user from a string
gadget for example, and now want to examine that string.

What you have to do is to prepare the CSource structure in
the RDArgs structure with the command line you want to parse:

  1. Give the "CS_Buffer" field a pointer to the string that
     you want to examine (parse).

  2. Set the lenght of the string in the "CS_Length" filed.

  3. Set the current character position to zero in the
     "CS_CurChr" filed.

Here is an example:

  /* Here is our own command line we want to parse: */
  UBYTE *my_command_line = "Bird.snd Volume=35 Filter\n";

  - - -
  /* Allocate a
                 RDArgs
                 structure etc... */
  - - -

  /* 1. Give the RDArgs structure our own command line: */
  my_rdargs->RDA_Source.CS_Buffer = my_command_line;

  /* 2. Set the length of the command line: */
  my_rdargs->RDA_Source.CS_Length = strlen( my_command_line );

  /* 3. Set the current character position to zero: */
  my_rdargs->RDA_Source.CS_CurChr = 0;

  - - -
  /* Parse the command line, examine the */
  /* results, and so on...               */


See Example 5 for more information:

  Example 5:  Read!    Run!    Edit!
```

1.24 Examples

```
                EXAMPLES
```

```
These examples should normally be executed from a "CLI" or
"Shell" window. When you run them with the help of AmigaGuide
you can not add any arguments, and since that is the main topic
of this chapter I would recommend you to  open a Shell  window and
try to run the examples from there instead.

Example 1:  Read!    Run!    Edit!
  This example demonstrates how to parse the command line.
  Since this is the first example it is relative simple.
  The program expects one argument. If no argument is given
  will the parse function fail (uses the "/A" - "Always
  required" option), and if more than one argument is give
  it will also fail (the "/M" - "Multiple argument" option
  is not set).

Example 2:  Read!    Run!    Edit!
  This example demonstrates how to parse the command line
  with several arguments. This example handles two types of
  command templates. First it can collect one or more
  words which will be used as file names. This demonstrates
  the "/M" (Multiple argument) option. Secondly the example
  accepts a special argument used as a switch. This
  demonstrates the "/S" ("Switch") option. The special
  argument is "Filter", but can also be abbreviated as "F".

Example 3:  Read!    Run!    Edit!
  This example demonstrates how to parse a command line with
  both a string and an optional value argument which require
  a keyword. (Demonstrates the "/N" and "/K" options.)

Example 4:  Read!    Run!    Edit!
  This example demonstrates how you can create a
               RDArgs
                structure yourself and prepare it before you parse the
  command line with the help of the  ReadArgs()
  we can prepare the RDArgs structure we can include extra help
  (will be displayed if the user types "?" to display the
  command line template and then types "?" again.), decide if
  the user should be able to see the command line template,
  etc...

Example 5:  Read!    Run!    Edit!
  This example demonstrates how you can create your own strings
  (command lines) which you then can parse with help of the
   ReadArgs()
  last example, but this time we initialize the "RDA_Source"
  field with our own command line. When we later call
  ReadArgs() it will notice that it already have a string to
  parse, and it will therefore use that string and not read one
  from the default input handler.

Example 6:  Read!    Run!    Edit!
  This is a simple example on how to use the  ReadItem()
  function. It will simply collect all command line arguments
  (items) and print them each on one line together with some
  extra information. (If the item was inside quotation marks or
  not, if it is an equal sign, if there was an error etc...)
```

```
Example 7:  Read!    Run!    Edit!
  This example demonstrates how to use the  FindArg()
  We simply use it to scan the command line template and return
  the position of each command template.
```