Using the AmigaDOS Pattern Matching Functions

by Ewout Walraven

One of the additions made to *dos.library* for release 2.0 is a series of functions to do standard pattern matching. Using a set of standard of string matching tokens, any application can use these functions to test if a particular string matches a pattern. The Amiga OS uses these functions for processing file name strings for its new directory scanning functions.

These functions can be used in every circumstance where you would like to enable the user to enter a pattern to indicate more than one target string. Using these functions not only makes it unnecessary to implement your own pattern matching routines, but by using the familiar DOS pattern tokens in your application, it is easier for the user to learn how to use your application.

Patterns

An AmigaDOS pattern matching string is a combination of alphanumeric characters and a series of special token characters. These token characters are part of the ASCII character set and they denote such things as string matching wildcards, string repetitions, and string negation. Pattern matching strings can use parentheses to delimit pattern matching substrings.

? The question mark matches any single character. For example, the pattern matching string ‘‘A?B’’ matches any string that is three letters long, that starts with an ‘‘A’’ and end swith a ‘‘B’’.

The number sign matches strings containing one or more repetitions of the expression that immediately follows the # in the pattern matching string. For example, the pattern matching string ‘‘#A’’ matches any string that consists of one or more of the ‘‘A’’ character. The pattern matching string ‘‘#?’’ matches any non-NULL string. The # can apply to entire substrings delimited by parentheses. For example, the pattern string ‘‘#(AB)’’ matches any string consisting of one or more repetitions of the substring ‘‘AB’’ (AB, ABAB, ABABAB...).

% Matches the NULL string.

| This is the OR symbol. This matches strings that contain the expressions on either side of the OR sign. The expressions and the OR symbol need to be enclosed in parentheses. For example, the pattern matching string ‘‘(A| B)’’ matches the string ‘‘A’’ or the string ‘‘B’’. The pattern matching string A(B| %| C) matches the strings ‘‘AB’’, ‘‘A’’, and ‘‘AC’’.

~ The tilde negates the expression that follows it. All strings that do not match the expression that follows the tilde will match the expression with the tilde. For example, the pattern matching string ‘‘~(#?.info)’’ matches any string that does not match the string ‘‘#?.info’’ (does not end with the substring ‘‘.info’’).

* The star is provided as an synonym to "#?". This is an option which can be turned on. Note that the star can not be used by itself on all non-FileSystem devices, like a logical device name assigned to a directory on a file system. For example:

```
Assign A: dh0:tmp
cd a:
list *
```

will produce an error. The *SetStar.c* example at the end of this article is a small stand alone utility to turn this option on and off.

[] All characters within brackets indicate a character class. Any character in the character class qualifies. Within a character class, a character range can be indicated by specified the start and stop character, separated with a minus sign. Note that character classes are case sensitive. If character classes are to be used in a case insensitive form, they should be translated to uppercase. Here are some example:

[ACF]#?matches strings starting with ‘A’, ‘C’, or ‘F’
[A-D]#?matches strings starting with ‘A’, ‘B’, ‘C’, or ‘D’
[~ACF]#? matches strings *not* starting with ‘A’, ‘C’, or ‘F’

’ The quote character neutralizes the special meaning of a special character. Here are some examples:

’#’? matches only the literal string ‘‘#?’’
’?(A| B| C| %)’# matches the literal strings ‘‘?#’’, ‘‘?A#’’, ‘‘?B#’’, ‘‘?C#’’
’’ matches ’

() Parentheses group special characters. The expression within the parentheses is a subpattern.

Parsing

When you want to use a string as an AmigaDOS wildcard pattern, the system must first parse it. The system builds a token string which the system uses to match strings. There are two functions in dos.library to parse pattern matching strings:

```
LONG ParsePattern(UBYTE *SourcePattern, UBYTE *MyDestination, LONG DestLength);
LONG ParsePatternNoCase(UBYTE *SourcePattern, UBYTE *MyDestination, LONG DestLength);
```

The ParsePattern() function creates a case sensitive token string, whereas the ParsePatternNoCase() functions creates a case insensitive token string. Both functions require a pointer to a destination buffer (MyDestination in the above prototype) to place the tokenized string in. Since every character in the pattern can be expanded to two tokens (3 in one case), this buffer should be at twice as as large as the original pattern plus 2 bytes. As a general rule, allocating a buffer three times the size of the pattern will hold all patterns. The third argument, DestLength, indicates the size of the destination buffer provided. These functions will returns one of three values:

-1 if there is an error (if the buffer is too small to hold all the tokens or the source string contains an invalid pattern),

1 if the wildcard pattern was parsed successfully and the pattern contains one of the special token characters.

0 if the wildcard pattern was parsed successfully and pattern contained only alphanumeric characters (no special token characters)

Matching

Once a pattern is parsed, it can be compared to a string using either the case sensitive MatchPattern() or the case insensitive MatchPatternNoCase() functions. These functions have the following synopsis:

```
BOOL MatchPattern(UBYTE *mytokenpattern, UBYTE *mystring);
BOOL MatchPatternNoCase (UBYTE *mytokenpattern, UBYTE *mystring);
```

These functions will compare the wildcard pattern string, mytokenpattern (created by ParsePattern/NoCase()), to mystring. If mystring matches the pattern in mytokenpattern, these routines return `TRUE`, otherwise they return `FALSE`.

Because these routines are recursive, they can use a lot of stack space, although they will not use more than 1500 bytes of stack space. Make sure the stack space is at least 1500 bytes before calling these routines.

In V36, these routines did not perform any stack checking. This was added in V37. If either of these functions detect a stack overflow, they will return 0 and IoErr() will return `ERROR_TOO_MANY_LEVELS`. If IoErr() returns 0, there was simply no match. The *Pattern.c* example at the end of this article shows how to use the parse and match functions and allows you to test and experiment with patterns.