Introduction

For the purpose of conducting an SQL validation at a vendor site, NIST staff will need to use a familiar editor to change the official test files. These changes may be global changes to all the programs in a given test suite type or modifications to an individual program. In general, these changes will be proposed in advance by the vendor. Each vendor will be provided a copy of the NIST editor software so that they can install and locally test the change file which they propose. The change file will be reviewed and accepted or negotiated by NIST. The file of proposed inputs to the editor also serves as documentation of vendor modifications and will be incorporated into the final validation report. TEd is distributed with the suite for vendor's private use to meet the following goals:

- install NIST maintenance changes to the test suite
- install "implementation defined" specifications allowed by the standard
- patch vendor nonconformities to the SQL standards for more comprehensive test results
- document changes to the test suite in a consistent format for inclusion in a validation summary report

TEd (pseudo-acronym of Test EDitor) is written in C for maximum portability to different hardware and software platforms. During development, system-dependent functions were kept to a minimum, allowing easy porting to most MS-DOS, Unix, and VMS environments.

The Change File

TEd reads an ASCII file of editing commands, called `tedchg` by default. Each command has a file specification, which may include wildcard characters, to restrict the command to operating only on certain file names and/or types. After the input file is read, execution of commands is only attempted if the input file specifications match. This allows a single, human-readable reference to a wide range of changes to *all* programs in the test suite.

The special change file is a sequence of commands that TEd executes. It is a simple ASCII text file containing commands (and user comments) that TEd understands. The `tedchg` file may contain *all* the changes to be applied to any of the programs to be tested. TEd edits one source file at a time and will only apply changes appropriate to the source file.

TEd has the ability to read commands interactively from the user. This allows simple changes to be made without creating a change file. The interactive mode is especially helpful if you are testing a command to see if it works as you expected. See the section "Using TEd" for details about the interactive command input mode.

A sample `tedchg` file might be coded to accomplish the following:

- Globally change the precision of the Embedded COBOL SQLCODE (*.pco) from 9 to 4. This is an implementation-defined option in the standard.

- Globally replace the Embedded C subroutine call to AUTHID with the vendor-required CONNECT statement, as allowed by the instructions for the test suite.

- Delete test number 793 from program dml079 (for all test suite languages), since the feature being tested is not supported by the SQL implementation being tested and the program will not pre-compile as is. It is necessary to remove the test to execute the other tests remaining in the program.

- Globally customize the screen display for each module language COBOL program to identify the vendor and time of day. A DISPLAY statement will be inserted at the beginning of each PROCEDURE DIVISION.

The change file to perform these changes would be as follows:

```
sub *.pco /SQLCODE/
/S9(9)/S9(4)/
rep *.pc /AUTHID(uid)/
    EXEC SQL CONNECT :uid IDENTIFIED BY :uid;
del dml079.* /BEGIN TEST0793/END TEST0793/
ins> *.sco /PROCEDURE DIVISION/
       PRINT-TIME.
            DISPLAY "Vendor XYZ, Op.Sys ABC, Module COBOL Tests"
            DISPLAY "Executed on " DATE " at " TIME.
```

Command File Structure

For each source file read, the file specification of each command in the change file is checked to see if the command should be applied. Commands are tested for possible execution *in the same order as they appear in the change file.*

Comments begin with an exclamation point “!” in column one (i.e., the first character on the line). The rest of a comment line is ignored.

Commands must start at the beginning of a line to be recognized and must appear in lower case.

Command Syntax

Most commands are of the following format:

```
<command> <filespec> <search_spec> [ -c ] [-i] [-e] [-p] [-m]
<text>
```

(arguments enclosed in brackets [ ] are optional)

The <command> argument determines what action will be performed. There are four commands that allow line insertions (**ins>** and **ins<**), line deletions (**del**), line replacements (**rep**), and string substitutions (**sub**). Each command is described in detail in the section “Command Descriptions” below.

File Specification

The <filespec> argument is used to specify a file, or a group of files, the command will be applied to. In our test suite, we have adopted the following file extension naming conventions:

File Extension		Description
`.pad`		Embedded Ada
`.pc`		Embedded C
`.pfo`		Embedded FORTRAN
`.pco`		Embedded COBOL
`.ppa`		Embedded Pascal
`.sad`	`.mad`	Standard Ada and Module Ada
`.sc`	`.mc`	Standard C and Module C
`.sfo`	`.mfo`	Standard FORTRAN and Module FORTRAN
`.sco`	`.mco`	Standard COBOL and Module COBOL
`.spa`	`.mpa`	Standard Pascal and Module Pascal
`.sql`		Interactive SQL statements

The standard program files are derived from the embedded SQL program files by converting the EXEC SQL commands to subroutine calls. The module files are the SQL procedures derived from the embedded SQL commands.

The <filespec> argument is how TEd decides if the command is applicable to the input file's filename. The file specification in the change file is the only place where wildcards (*globbing* or *pattern matching*) is available for files[1]. The following table describes the types of wildcards that are available:

Wildcard	Description
*	matches zero or more occurrences of any characters
?	matches any single character
[ ]	matches single characters specified inside the brackets (ranges permitted by using a hyphen)
{ }	matches comma-separated character *strings* inside the braces

Note that we will typcally prepend file specifications with an asterisk to ensure correct matches even if the current file has a directory prefix.

Although filename pattern matching is **not** case sensitive, in general, *use lower case*. Most operating systems will convert filenames to upper-case when files are opened if necessary (e.g., MS-DOS, VAX VMS).

[1] While some command interpreters will evaluate and expand wildcards to all matching filenames before passing the arguments to the application program (e.g., the Unix shells "sh" and "csh"), while others do not (MS-DOS "command.com", VMS "dcl") and leave searching for matching files to the application program. TEd will not expand wildcards on the non-expanding type of command interpreters for portability reasons.

File Specification Wildcard Examples

File Spec	Description
`*`	matches all filenames
`*.*`	matches all files with a period somewhere in the name (all our files)
`*.pco`	matches all embedded Cobol files
`*.?c`	matches all C files (matches *.pc, *.mc, *.sc)
`*.m*`	matches all module language files (*.mc, *.mco, *.mfo, *.mpa)
`*.sc?`	matches all standard Cobol files (*.sco) but not *.sc
`dml*.*`	matches all files that start with "dml".
`*dml029.pc`	matches file dml029.pc, even if prefixed by a directory path
`dlm00[789].pc`	matches files dml007.pc, dml008.pc, and dml009.pc
`dml02[0-9].pc`	matches all dml02x.pc files (dml020.pc, dml021.pc, dml022.pc, ..., dml029.pc).
`{sd,dm}*.pc`	matches all *.pc files that start with either "sd" or "dm"

Search Specification

The <search_spec> has two forms:

<delim><match_string><delim>

or

<delim><start_string><delim><end_string><delim>

TEd will search for lines that contain either <match_string> or a group (range) of lines surrounded by lines that contain <start_string> and a line that contains <end_string>. Usually, strings are searched for their occurrence in the source text lines. See "Options" below for different methods of specifying string searches. The <start_string> and <end_string> may (or, of course, may not) appear on the same line to denote a range. The <end_string> is always on a line equal to or further down from the <start_string> in the input file. After finding one range, the command (insert, replace, string substitute) is performed on that range, and searching continues for another range after the line containing the <end_string>.

Note: When searching for a range, if the <start_string> is found but the <end_string> is not found, the range is considered to be undefined and the command will not be executed. If this occurs after one or more defined ranges, the command will be executed only for those ranges where the <end_string> was found.

Delimiters (<delim>) may be any *single* character. We generally use the forward-slash character "/" but it may be any character that is unique to the rest of the string(s) with the exception of the hyphen (its use is reserved for command options). Ether the apostrophe (') or the double quotes (") is an aesthetically pleasing alternative.

To match all lines, use an empty <match_string>. That is, two adjoining delimeters. This is useful for the string substitution command, described below.

Options

There are several optional flags that may be applied to a command to modify its behavior. If more than one option is desired, they may be specified without additional hyphens. The order in which options appear is not significant. The following three examples are interpreted exactly the same way: `-m -i -p; -mip; -pim`.

The `-i` option specifies the "ignore case" option on search strings. By default, string matching is case sensitive, so the strings "NIST" and "nist" are different. The `-i` option converts all strings to lower case before comparing them.

The `-e` option specifies that changes are to be "echoed" as they are being performed. By default, changes are not echoed to the terminal as they are performed. In all cases, however, a single line summary of the actions the command performed is always displayed for each command. If TEd is invoked with the verbose option (see "Using TEd") then all commands and their changes are echoed to the screen as they are executed.

The `-p` option changes the concept of the <search_spec> slightly such that string comparisons are performed using pattern matching techniques. The wildcards available are the same as those used for matching filenames with the <file_spec>.[1] Without the `-p` option, the string is searched for occurrence in any part of a line.

If the `-p` option is used, the context of the <search_spec> must be fully described. For example, the following two commands are both valid and operate in exactly the same way:

```
sub * /SQLCODE/                     sub * /*SQLCODE*/ -p
/SQLCODE/SQLCOD/                    /SQLCODE/SQLCOD/
```

Note that the command with the `-p` option requires the '*' character surrounding the searched text. If the asterisks were omitted, then the command would match a line that contained only "SQLCODE" (e.g., "SQLCODE" at the beginning of a line immediately followed by a <newline>). If `-p` was not used, TEd would search for the text "*SQLCODE*", including the asterisks, on any part of a line.

To retain backward compatibility with previous search specifications, TEd will match the following search specification: `sub * /*END EXEC/ -p`. That is, if a string is known to occur at the end of a line, the trailing \n and the associated `-c` option are not required.

If the `-p` option is used, some characters have special meaning and must be *quoted* by preceding them with a backslash "\" if they are to be interpreted as actual text to be searched for, rather than a pattern. The special characters are the open bracket [, open brace {, backslash \, asterisk *, and the question mark ? These special characters cannot be used as delimiters with the pattern matching option (it would get much too confusing).

For example, to search for the text

```
X = 5 * Y[2] \ X
```

a pattern to match it would be

```
*X = 5 \* Y\[2] \\ X*
```

. Notice how the backslash is quoted by itself.

Multiple options can be specified with the use of only one hyphen, then the option characters (in any order). For example,

```
ins<  *.pc /*EXEC*SQL*SELECT*/*;*/ -pie
```

[1] The `-p` option exists to allow more flexible and exact <search_string> searches, but it is an option due to the decreased legibility of the change file if all <search_spec>'s required surrounding with asterisks.

```
    printf("SQLCODE BEFORE SELECT = %ld\n", SQLCODE);
```

selects pattern matching <search_spec> mode, ignore case on the <search_spec>, and echo command and changes.

The `-m` option is to permit access to runtime-specific variables through the use of macros. Macros are preceded by a dollar sign ($) and may occur in any portion of a command (excluding the command name, of course). Thus, macros can occur in the <file_spec>, <match_string>, <start_string>, <end_string>, and/or <text> specifications. The following macros are available:

Macro	Description
`$if`	input filename (complete)
`$ifh`	input filename head (e.g., "dml001" from "dml001.pc")
`$ife`	input filename extension (e.g., "pc" from "dml001.pc")
`$of`	output filename (complete)
`$ofh`	output filename head
`$ofe`	output filename extension
`$ted`	commands filename (complete)
`$ver`	version ID string for TEd executable

Additionally, each macro may be capitalized, resulting in the expansion text of the macro to be capitalized also. E.g., if `$if` (the input filename) was `dml001.pc`, $IF would be `DML001.PC`.

For example,

```
ins> *.pc /*begin*test*/ -mip
  printf("NIST SQL Test Suite version 2.0.  Testing file $of\n");
  printf("Edited from $ifh.$ife from the TEd change file $ted\n");
```

If TEd were given an input filename of `dml001.pc`, an output filename of `dml001a.ccc`, and the commands filename of `dmlchg.ted`, TEd would insert the following two lines after finding the word "begin" followed by the word "test" on any line without regard to case:

```
  printf("NIST SQL Test Suite version 2.0.  Testing file dml001a.ccc\n");
  printf("Edited from dml001.pc from the TEd change file dmlchg.ted\n");
```

In this example, $ifh and $ife are used separately to demonstrate how the original file can be put together again by using the head and extension macros. Normally $if would suffice.

Macros are updated for each source file to ensure expected results when using the `-f` command line option. Refer to "Using TEd" for more information about the `-f` command line option.

Attention must be given when specifying the order in which macros across several commands are to be executed in a change file. If the following sub command were used,

```
sub * // -m
/INFILE/$if/
/INFILE_EXT/$ife/
```

and if the input file (named `dml001.pc`) had included the following two lines,

```
...
/* the input file is INFILE */
/* the input file's extension is INFILE_EXT */
...
```

then both instances of the string INFILE would be expanded with the first substitute replacement, since both lines include the string "INFILE" which is replaced by the macro $if. The generated output would look like,

```
...
/* the input file is dml001.pc */
/* the input file's extension is dml001.pc_EXT */
...
```

obviously not what was intended. To work around this difficulty, either (1) specify macros so their strings don't overlap or (2) list longer macro search strings in the change file first (specify INFILE_EXT *then* INFILE in the commands file).

The `-c` option allows control characters to be embedded in strings throughout a command. This option is often used to either join (concatenate) two lines by deleting a trailing newline or splitting a line into multiple lines by adding newline characters into the middle of an existing line. In each case, after the command is finished, all the lines in memory are reorganized such that each line contains one, and only one, newline at the end of the line. If a newline is deleted, the line will be concatenated with the contents of the next line. Similarly, if newlines are added to a text line, the lines will be split into separate lines. This functionality results in expected behavior for subsequent commands.

Control characters are escaped with the ANSI C programming conventions. For example, newlines are \n, tabs are \t. The supported escape sequences are as follows:

```
\a = audible alert (bell)          \r = carriage return
\b = backspace                     \t = horizontal tab
\f = formfeed                      \v = vertical tab
\n = newline                       \\ = backslash
```

Note how the backslash is quoted by itself.

In addition to the above escapes, any value can be embedded by entering its octal or hexadecimal value. To insert a literal octal value, use the backslash followed by octal digits. For example, \7, \07, and \007 are the same values.

Hexadecimal values are inserted by using the backslash followed by the letter "x" and hexadecimal digits. For example, \x8 and \x08 are the same values.

Note: Care must be taken not to have any other surrounding digits that could be interpreted as part of a value. For example, "\x7feet" is interpreted as the hex value 7fee, not the intended \x7 value.

Note that all values are placed into character strings where each character can hold an 8-bit value. Thus, the largest values are \xff or \177. Trying to exceed these limits will result in unpredictable behaviour, depending on how your compiler handles signed vs unsigned `char`'s.

Here is a simple example:

```
sub * // -c
/this is a very long string/this is not a\nlong string/
```

This command substitutes all occurrences of the string:

```
this is a very long string
```

with

```
this is not a
long string
```

Be careful when making changes to a file containing C source code, as the `-c` option with embedded control characters can get confusing and you must "escape the escape character." For example, let's say you needed to split a long printf statement from:

```
        printf(“Alpha = %d, Beta = %d, Gamma = %f\n”,alpha, beta, gamma);
```

to

```
        printf(“Alpha = %d, Beta = %d, Gamma = %f\n”, alpha,
                beta, gamma);
```

The substitute command could be:

```
sub * /printf(“Alpha/ -c
/Gamma = %f\\n”,alpha, /Gamma = %f\\n”, alpha,\n\t\t/
```

Again, note how the backslash is quoted by itself.

The `-p` option joined with the `-c` option is useful for performing searches that relate to the newline character, \n, which is the last character in each text line after TEd reads the input file. If you substitute the newline character with some other character, TEd "fixes" all the lines in the file. That is, if a line doesn't have a newline at the end after a command executes, it is concatenated (joined) with the next line. This ensures that all lines will have newlines at the end for the next command. For example,

```
sub * /*end-exec\n/ -ipc
/end-exec\n/end-exec  /
```

probably isn't useful (like many of our examples), but illustrative nonetheless. TEd searches for a line that ends with the characters "end-exec" immediately followed by the newline character. When it finds such a line, it replaces "end-exec" with "end-exec " (note the two trailing spaces and lack of newline). When all lines have been searched, TEd fixes the lines such that whatever was on the line following "end-exec" is now on the same line as the "end-exec " line.

Text

<text> refers to the text to be inserted or the replacement text. It may span several lines. If indentation is desired in the output file, it must be done here since the text is copied to the output file verbatim (except when modified by the `-m` macro option). The only restriction on the text is that it must not look like a command (i.e., commands start at the beginning of a line in lower case, comments begin with the "!" character at the beginning of a line).

The substitute command does not use <text> like other commands, but rather the string to be substituted and a replacement string, all surrounded by delimiters:

```
<delim><start_string><delim><end_string><delim>
```

See the substitute command description for additional information.

Command Descriptions

The following commands are permitted by TEd:

Command	Description
`ins>`	Insert lines after a <search_spec>
`ins<`	Insert lines before a <search_spec>
`del`	Delete lines contained in the <search_spec> inclusively
`rep`	Replace lines contained in the <search_spec> inclusively
`sub`	Substitute strings contained in the <search_spec>

As mentioned previously, the general syntax of the commands is:

```
<command>  <filespec>  <search_spec>  [ -c]  [-i]  [-e]  [-p]  [-m]
<text>
```

where the items enclosed in brackets ([ ]) are optional. <search_spec> is either

```
<delim><match_string><delim>
```
or
```
<delim><start_string><delim><end_string><delim>
```

The <match_string> simply looks for a particular string on a line. The <start_string> and <end_string> pair delimit *ranges* of text lines. A command will be applied several times if the <match_string> or the <start_string> + <end_string> pair are found several times.

Note: When searching for a range, if the <start_string> is found but the <end_string> is not found, the range is not defined and the command will not be executed. If this occurs after one or more defined ranges, the command will be executed only for those ranges where the <end_string> was found.

Differences and limitations of these two formats of the <search_spec> will be noted when applicable for each command.

Insert Lines

There are two variations on the insert command to perform insertions before or after the <search_spec> string: `ins<` and `ins>`, respectively. Both commands allow the <search_spec> to be either <match_string> or <start_string> + <end_string> pair. For example,

```
ins>  *.pc /*AUTHID*(*)*/ -p
    printf("Embedded C test of RDBMS13 on April 1, 1990\n");
```

would be executed if the input filename matched "*.pc" (all embedded C programs). The editor would then insert the line "   printf("Embedded C ..." *after* it finds any line that contains the subroutine call to AUTHID with any arguments (notice the liberal use of wildcard characters with the -p option).

```
ins>  *.pc /EXEC SQL SELECT/;/
    printf("SQLCODE = %ld\n", SQLCODE);
```

This command would be executed on all embedded C files for lines containing “EXEC SQL SELECT”. The editor continues searching for a line (starting at the same line) containing a semicolon. If one is found, it inserts the printf text after the line containing the semicolon. Note that the <start_string> (“EXEC SQL SELECT”) and the <end_string> (“;”) may or may not appear on the same line. The text to be inserted will always appear on the line following the <end_string>.

```
ins< *.pfo  /  STOP/
      EXEC SQL END TRANSACTION;
```

This command inserts the line “   EXEC SQL END TRANSACTION;” *before* any “ STOP” statements in all embedded FORTRAN files. Note in this example the replacement text is indented to column seven as required by most FORTRAN compilers.

Delete Lines

The delete (`del`) command allows removal of text lines from the source files. Like the insert-after command, it allows the deletion of lines that match every line that contains a specified string or fall within a range of lines. For example,

```
del  cob002.*co  /01  XYZ5/
```

deletes all lines that contain “01 XYZ5” from any cob002 COBOL input files (cob002.pco, cob002.sco, cob002.mco).

A range of lines can be deleted by giving a <start_string> and <end_string>:

```
del  cob002.*co  /01  XYZ5/design,/
```

This will delete lines between “01 XYZ5” and “design,” ***inclusively***. That is, it will delete the lines containing <start_string> and <end_string> parameters. As usual, if the <end_string> is not found, the command is not executed.

An entire test can be deleted by

```
del dml022.p*  /BEGIN TEST0100/END TEST0100/
```

since (in theory) all test suite programs have comments surrounding the begin and end of a test in the format “BEGIN TESTxxxx” and “END TESTxxxx” where xxxx is the test number.

Note that in our COBOL programs, performed paragraphs belonging to TEST0100 are placed at the end of the program, outside the range of the comments. Therefore, an additional delete command may be required for COBOL, shown below:

```
del dml022.pco /P46./   ./
```

Replace Lines

The replace command (`rep`) allows a range of existing text to be replaced with <text> inclusively. Both <match_string> and <start_string><end_string> forms are valid for the <search_spec>. For example,

```
rep dml082.{pc,pfo,ppa} /AUTHID/
       EXEC SQL DECLARE HU1 SCHEMA FOR FILENAME [HU1]HU1;
       EXEC SQL DECLARE HU2 SCHEMA FOR FILENAME [HU2]HU2;
```

will replace the line which contain the text "AUTHID" with the two DECLARE statements. This replace command will execute on an embedded C, FORTRAN, or Pascal version of the file dml082. It is up to the user, however, to ensure proper indentation of the replacement text for the appropriate language.

Here is an example of a replacement with a <start_string><end_string> search specification:

```
rep cob003.*co /01  XYZ5/design,/
         01  XYZ5.
             02  FILLER  pic x(80) value...
             02  FILLER  pic x(80) value "design,"
```

replaces the lines between "01 XYZ5" and "design," with the three lines of text underneath the **rep** command. Since text is replaced inclusively with the replace command, the <start_string> "01 XYZ5" and the line that contains the <end_string> "design," must appear in the replacement text if needed.

Substitute Strings

The substitute command searches for every occurrence of <search_string> and replaces it with <replacement_string>. It has one basic format:

```
sub  <file_spec>  <search_spec>
      <delim><search_string><delim><replacement_string><delim>
      [<delim><search_string><delim><replacement_string><delim>]
```

<search_spec> may be one of the following forms:

<delim><match_string><delim>

the substitute command is only applied if <match_string> is found on the same line.

<delim><start_string><delim><end_string><delim>

only lines that fall inside the <start_string> and <end_string> of the <search_spec> are checked to see if the substitute command can be applied. If the <replacement_string> is empty, the <search_string> will effectively be deleted (replaced with nothing).

<delim><delim>

all lines in the file are scanned to see if the substitute command can be applied. Note that there are no characters between the two delimiters (unless the -p option is used, in which case an asterisk "*" must appear between the delimiters to provide the same function).

Multiple string substitutions can be specified for the same <file_spec> and <search_spec> by simply adding additional

```
        <delim><search_string><delim><replacement_string><delim>
```

lines after one another. Each search/replace line is interpreted as another command, executed in the same order specified in the change file. Comments, however, cannot appear between multiple search/replace lines. TEd will skip over blanks when searching for the delimiters, so search/replace lines may be indented for clarity.

For example,

```
sub  dml035.* //
  'zzzzz'zzzz'
  'Chris Schanzle''
```

will execute on all dml035 input test files and replaces each string of five z's with a string of four z's. The next command to be executed would have the same <file_spec> and <search_spec>, but the command will now effectively delete all occurrences of "Chris Schanzle" from the contents of any line by replacing the <search_string> with an empty <replacement_string>. Note the <search_spec> for both commands is empty, meaning every line in the input file is searched from top to bottom for the <search_string>.

```
sub  *.*pc  /printf/
     "\n\n"\n"
```

This substitute command will be executed with an embedded C input file. It will search for lines that contain "printf", then check if the line contains the four characters that make up two linefeed characters ("\n\n") and replaces it with two characters that make up one linefeed character ("\n"). This can be used to change double spaced lines to single spaced lines (other languages are similar).

```
sub  dml013.*  /BEGIN TEST0171/END TEST0171/
  /DISTINCT//
```

This substitute command would execute on all DML013 input files. It will delete every occurrence of the string "DISTINCT" occurring between lines that contain "BEGIN TEST0171" and "END TEST0171".

Using TEd

TEd is invoked at the command line with following syntax:

```
      ted [ -options ]  [ input_file1 ]  [ input_file2 ... input_filen]
```

where [-options] is one or more of the following letters and arguments:

`-f filename`

Use the file `filename` to get a list of input, output, and to load additional editing commands. If any `input_file` is given on the command line, the argument is ignored. The "list of files" file has one or more lines of the following format:

```
input_file [output_file] [change_file] [flush]
```

(Optional arguments are shown, but not typed, in brackets.) Each parameter is separated by one or more spaces or tabs. The `input_file` specifies what file is to be edited. The `output_file` specifies the filename to write the file after all commands have been processed (rather than the default action to overwrite the `input_file`). The `change_file` parameter is a file of *additional* commands to be applied to the `input_file` *and for all subsequent input files*. If the final parameter exists, all existing change commands are flushed (deleted) from memory before reading new commands from the `change_file` parameter. Actually, any word will suffice for the final argument, but "flush" provides (slightly more) meaning in the list of files file.

Note that when specifying a `change_file`, the `output_file` must be specified as a place holder, even if it is the same as the first argument (`input_file`).

`-i`

The -i option sets interactive command input mode. Rather than read a file of changes (by reading `tedchg` or by using the `-t` option described below), the user is prompted to enter TEd change commands directly at the keyboard. The commands are parsed (interpreted) the same way as when commands are read from a change file on disk, but with a little more output when a command is "discovered" by TEd.

There are two additional commands that are helpful when using the interactive input mode: `help` and `list`. Help gives assistance on the syntax and/or meaning of a command or its arguments. A help command may be entered where TEd would be looking for a line beginning with a command keyword (e.g., sub, ins>, etc). These commands won't be recognized inside a block of replacement <text> or in the middle of multiple **sub** commands with a common <filespec> and <search_spec>. Remember a comment line always ends a command, so you could enter a line with just a ! character, then use `help` or `list`.

The `list` command will display a list of commands entered thus far in a form that can be saved and used as input into TEd later.

Appendix A contains a sample script of an interactive session with TEd.

`-o output_file`

Use the filename `output_file` as the output filename. The default action is to overwrite the `input_file` after all commands have been successfully processed and if the input file has been changed. This option is ignored (with a warning message) if the -f option is used. If more than one `input_file` is given on the command line, this option is not allowed (an error message will occur and TEd will terminate).

`-t change_file`

Use `change_file` to load the editing commands TEd will apply to input file(s). If the `-t` option is not specified, the file to be read will be `tedchg`. If the `-f` option is also used, commands in the file specified with the `-t` option are loaded first.

`-v`

Turn verbose mode ON (default is OFF). Each command, its parameters, and the lines it affects, are echoed to the screen as they are executed.

TEd uses the pseudo Unix standard `getopt()` technique for parsing command line options and their arguments. All options must precede the first `input_file` argument, options are specified by a hyphen and a single letter (no space between the hyphen and the option letter), and then an argument, possibly preceded by a space. Options that do not require arguments (the -v and -i options in TEd) may be specified before options that do require arguments. For example,

```
% ted -vt alt_change -ffiles
```

and

```
% ted -talt_change -v -f files
```

are both valid and functionally identical. However,

```
% ted -tv alt_change
```

is interpreted as “v” being the argument for the “t” option and alt_change as the first input file.

Note VAX/VMS Users

Because the VMS “run” command will not accept arguments that TEd often requires, a global symbol that points to the executable (usually TED.EXE) must be defined as follows:

```
$ ted :== $ disk:[directory]ted
```

Where *disk* and *directory* are easily discovered by applying the “dir” command where the executable for TEd resides. Now TEd can be invoked by typing its name (ted) and specifying arguments as described below. You may wish to add this line to your LOGIN.COM file.

Examples for Using TEd

```
% ted dml001.pc
```

This is the simplest way to invoke TEd. TEd reads the default commands file `tedchg` in the current directory, reads the contents of `dml001.pc` into memory, executes the commands (as described in "Command Descriptions"), then rewrites `dml001.pc` with the edited version processed in memory.

```
% ted -t mychange -o dml001.pc1 dml001.pc
```

This command tells TEd to read the change file `mychange` instead of the default (`tedchg`), read the contents of `dml001.pc`, execute the commands, then write the edited version to file called `dml001.pc1`. Note that the original file `dml001.pc` is left unchanged.

```
% ted -f files
```

This tells TEd to read lines in the file `files` for an input file, and output file, and a change file. Note that since no change file was specified on the command line, it must be specified on the first line in `files` (shown below as `dmlchg.ted`). After TEd has read a commands file, it can operate on input files. For example, if `files` contained the following,

```
dml001.pc dml001.pc dmlchg.ted
dml002.pc
sdl003.pc sdl003.pc1 sdlchg.ted flush
sdl004.pc sdl004.pc1
sdl005.pc
```

then the commands in the change file `dmlchg.ted` would be loaded, then the text from `dml001.pc`. (If `dmlchg.ted` was not specified on the first line, TEd will notify the user that a commands file could not be loaded and exit.) TEd would then perform the commands, then write the output file `dml001.pc`.

After finding line #2, TEd would "forget" the contents `dml001.pc`, read the contents of `dml002.pc`, perform the same commands from `dmlchg.ted`, then rewrite `dml002.pc` with the edited version.

When line #3 in `files` is read, all existing commands are flushed from memory and new commands are loaded from `sdlchg.ted`. Note that all old commands are lost when a new commands file is loaded when the fourth argument exists. The contents of `sdl003.pc` is read, the commands are performed, and the edited version is written to a file called `sdl003.pc1`.

Line #4 of `files` tells TEd to use the same commands file `sdlchg.ted` on the input file `sdl004.pc` and write the output as `sdl004.pc1`.

Finally, the last line in `files` is read and the contents of `sdl005.pc` is read. The commands from `sdlchg.ted` are performed, and `sdl005.pc` is rewritten with the new edited version.

Implementation Notes and User Feedback

As mentioned in the introduction, TEd was carefully designed and coded in the “C” programming language to allow execution in almost any environment with no source code modification. We use TEd in the following environments:

Computer	Operating System	Version	Compiler(s)
Sun Sparcstation 2	SunOS Unix	4.1.3	cc, gcc 2.4.5
PC Clone (386,486)	MS-DOS	5.0	Turbo C 2.0
VAX 785	VAX VMS	5.5	VAX C

In nearly all instances, TEd allocates memory dynamically as required. This includes storage space for reading the input files and for commands and their parameters. However, input line and search string buffers are limited to 128 bytes; longer lines will be truncated and a warning message will be displayed. You can change this parameter in the source code.

If you have an ancient C compiler that chokes on procedure names and/or identifier names which are longer than eight characters, we can only recommend you see your vendor for an update to your compiler.

You may wish to browse through the code if you are having compilation difficulties. While there are several compilation definitions at the beginning of “ted.c”, there are many other comments in the code which address portability issues. These portions of TEd may be modified as stated in the code. If you have to make any changes to make TEd work in your environment, let us know! Otherwise, there might be surprises when we come to validate you for conformance.

If you have questions on how to apply TEd to the test suite, please contact Joan Sullivan at (301) 975-3258.

If you are having technical difficulties with TEd – e.g., compilation errors, stack/core dumps, crashing, strange error messages, incorrect operation, or (*gasp!*) bugs – or would like to give input on potential enhancements to TEd or this documentation, please feel free to contact the author, Chris Schanzle, at (301) 975-3796 or at the address listed below. You may also send electronic mail on the Internet to “chris@speckle.ncsl.nist.gov”. We welcome your comments and suggestions!

Chris Schanzle
NIST Bldg 225 (Tech) Rm A-266
Gaithersburg, MD 20899

Example Interactive Session

Below is actual captured output of a simple interactive command input session with TEd. Notice the list of valid commands (near the end) displays the commands in a fashion such that they can be captured from the screen and copied directly into a new commands file for later use. The user's input is shown as **bold** type at the Unix shell prompt (%) and TEd's prompt (>). Following the TEd invocation is a 'diff' of the input file (dml001.pc) and the output file (asdf).

```
% ./ted  -io  asdf  dml001.pc
NIST SQL Test EDitor "TEd" vers 5.1 5/17/95
Interactive command input mode set.

Enter commands now.  Type 'help' for help.  Enter "."
on its own line to simulate an end-of-file.

> help

Enter commands just as they would have been entered in a file.
Type "." (w/o the quotes) to simulate an end-of-file.

For additional help, type "help <cmd>" where <cmd> is one of the
following:

    sub        - substitute strings within lines
    rep        - replace lines
    ins        - insert lines before or after a search string
    del        - delete lines
    search     - help with <search_spec> (used in most commands)
    options    - the options available to all commands
    list       - list commands entered thus far

> help  sub

The sub command permits string substitutions with the following syntax:

    sub <filespec> <search_spec> [-options]
    <search_string><replacement_string>
    [ <search_string><replacement_string> ]

The items enclosed in brackets [ ] are optional.  If you have more than one
string to be replaced that meets the same <filespec> and <search_spec>,
simply add a new <search_string><replacement_string> pair underneath.
SEE ALSO:  search

> sub  *  //
> /"HU"/"SCHANZLE"/

This SUB command looks OK...
! From file standard input:4
sub * //
/"HU"/"SCHANZLE"/
> /BEGIN/begin/

This SUB command looks OK...
! From file standard input:5
sub * //
/BEGIN/begin/
> sub  *  /begin  test0003/end  test0003/  -ei
> '15'20'
```

```
This SUB command looks OK...
! From file standard input:7
sub * /begin test0003/end test0003/ -ei
/15/20/
> ins>  *  /*begin*test*/  -ip
>     printf("NIST  SQLVTS  1/2/95:   beginning  new  test!\n");
> .

This command looks OK...
! From file standard input:8
ins> * /*begin*test*/ -ip
    printf("NIST SQLVTS 1/2/95:  beginning new test!\n");

****   L i s t    o f    V a l i d   C o m m a n d s   ****

! From file standard input:4
sub * //
/"HU"/"SCHANZLE"/
!
! From file standard input:5
sub * //
/BEGIN/begin/
!
! From file standard input:7
sub * /begin test0003/end test0003/ -ei
/15/20/
!
! From file standard input:8
ins> * /*begin*test*/ -ip
    printf("NIST SQLVTS 1/2/95:  beginning new test!\n");

****   E n d   o f   C o m m a n d s   ****


Execute these commands?  (y/n) y


Input: dml001.pc    Output: asdf

sub from standard input:4 found 544 lines with the <search_str> and made 1 sub
sub from standard input:5 found 544 lines with the <search_str> and made 9
subs
---------
! From file standard input:7
sub * /begin test0003/end test0003/ -ei
/15/20/
<      for (ii=1;ii<15;ii++)
>      for (ii=1;ii<20;ii++)

sub from standard input:7 searched 54 lines in 1 range and made 1 sub
ins> from standard input:8 inserted blks of 1 line 8 times

% diff  asdf  dml001.pc
31c31
< EXEC SQL begin DECLARE SECTION;
---
> EXEC SQL BEGIN DECLARE SECTION;
51c51
<       strcpy(uid,"SCHANZLE");
---
```

```
>       strcpy(uid,"HU");
69,70c69
< /******************** begin TEST0001 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0001 ********************/
123,124c122
< /******************** begin TEST0002 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0002 ********************/
177,178c175
< /******************** begin TEST0003 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0003 ********************/
201c198
<       for (ii=1;ii<20;ii++)
---
>       for (ii=1;ii<15;ii++)
232,233c229
< /******************** begin TEST0004 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0004 ********************/
292,293c288
< /******************** begin TEST0005 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0005 ********************/
351,352c346
< /******************** begin TEST0158 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0158 ********************/
415,416c409
< /******************** begin TEST0159 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0159 ********************/
483,484c476
< /******************** begin TEST0160 ********************/
<     printf("NIST SQLVTS 1/2/95:  beginning new test!\n");
---
> /******************** BEGIN TEST0160 ********************/
%
```

Contents