Contents

A preface to STedi

The past years have seen an explosive development of powerful computers and workstations. The availability of these computers for the use at home has posed completely new demands on their software. In addition the computer has become less than previously a limitation on the possible programs that can be created. The way is open for programs that are increasingly able to take over small chores which traditionally are handled by the user, and leave the user free to concentrate on the larger task at hand. Programs can also become more and more user friendly, efficient to operate and easy to learn. Thus with a task like editing, the computer can help with many of the tasks of internal organization and checking, leaving the user free to concentrate on his work as a whole. Speed is also important so that minimum time is spent waiting for the computer to perform the various operations, and the time gained can be put to productive work.

A program that is both truly user-friendly and powerful should not only be relatively easy to learn and to remember, but it should be designed in such a way that a minimum of the user's actions are necessary for performing any given task. However even this latter criterion should not be used blindly. There are some instances when an extra key stroke or two are called for to guard against accidental loss of work. After all, what is really behind all requirements of user-friendliness and power is how productive a program will be for the user in the long run. Ultimately it is not only the time spent learning the program or the time spent performing individual tasks alone that counts; what really matters is the overall time required to start a project and get all the way to the end of it.

So there are certain protections that ought to be built into a program so that the user will not be liable to make serious and irreversible mistakes which may cause him to lose large amounts of work in the process. Thus for an editor, the program should not allow the user to quit and leave unsaved files behind, without at least asking the user whether these files are meant to be abandoned. If the program allows such a mistake, it would not be maximizing the user's time at all but may cost the user a lot of extra work in spite of how fast individual tasks are performed.

The advent of the mouse was truly a breakthrough toward user-friendliness in programs. Experience shows that a program can be much easier to learn if mouse operations are available as an alternative to key strokes. Often the mouse

is first used for simple tasks and then as the user becomes more familiar with the features of the program, the mouse gradually fades into the background as more efficient keyboard commands take over. For some users, the mouse may remain the preferred method for performing a specific task. It remains a very useful and powerful tool in the program environment.

It was with these thoughts in mind that STEDI was created. On the one hand, careful attention has been paid to speed in every detail of the program so that individual actions can be performed with maximum efficiency. On the other hand, thoughtful consideration was put into the design of the keyboard usage and the user interface to minimize mistakes that cannot be easily reversed. Flexibility was also a major consideration. There are many options open to the user to determine the default actions of commands; extensive use of the mouse is made as an alternative to all major commands, and STEDI's keyboard actions can be completely reprogrammed by the user as desired. In addition a powerful macro language enables the user to design his own composite commands. These features allow the user to create an environment tailored to his individual needs and tastes.

Many people work nowadays on more than one computer. This means that they may be confronted with different systems and different software on each computer. In the worst case they have to work with completely different editors for their program development. This can be very annoying. STEDI has been programmed to allow it to run on different computers. It will run best on the rather small computers with their flexible screen and keyboard control, but it is also possible to use it with a standard terminal on mainframe computers. It depends on the flexibility of the terminal and the operating system how well the superior workstation and micro computer environment can be approached.

Because of these considerations, STEDI makes as little use as possible of native operating systems without suffering an unbearable loss of efficiency. This means for instance that on Atari ST systems the GEM environment has been abandoned. The exception to this rule is the support of X-windows which is expected to become a standard of the 1990's. Rather than the usual windows, menus and dialog boxes, STEDI makes use of a command line interface that gives the user the capability of entering very powerful commands with only a few key strokes; this leaves most of the screen available for displaying text. When available the mouse is implemented in such a way that, at the beginning, it serves a helpful role in learning all the phases of the program; as learning progresses, faster and more powerful habits can be developed. The mouse can truly recede into the background if desired, although for some users it will remain the quickest and simplest way of performing certain tasks.

1.1 A General Orientation To STedi

STEDI makes use of keyboard commands, the mouse and a command line which replaces the normal menu bar of the native window system. The keyboard commands take care of most of the basic editing operations, such as the insertion and deletion of text and movements within the file being edited, while the command line is used for executing more powerful commands. A number of other operations such as cutting and pasting are programmed into the function keys and hence can also be performed without using command line commands. The mouse is available as an alternative to many of these commands.

1.1.1 The Keyboard Commands

With this organization, the editor maximizes the use of direct keyboard commands for executing common functions and makes these commands as intuitive as possible. Where possible, commands are given mnemonic key assignments and the key layout has been carefully chosen to minimize the possibility of making an irreversible mistake by accidentally hitting the wrong key. In addition, care has been taken to make as much use of the keyboard layout as possible so that learning how to use the editor would be speedy and require little effort. In this regard function keys have been utilized to perform some of the most frequently used commands, and whenever possible specific keys such as <insert>, <help> and <undo> have been given their intuitive meanings.

1.1.2 The Mouse

Extensive use is made of the mouse as a tool for such tasks as moving the cursor around the screen, scrolling and even providing an alternative method for entering most of the important commands which are available by other means. If the system supports it a file selector box is utilized as a convenient way of searching through directories (folders) for input and output files when the full path name is not known or when one prefers not to have to type in the full path name of a file.

1.1.3 The Command-Line

The normal menu bar that is provided by many systems has been replaced by a command line which appears at the bottom of the screen and which serves both as a source of information about the current status of the editor settings as well as a means for entering commands either with key strokes or by use of the mouse. Whenever the <escape> key is pressed, the cursor moves to the

command line where the commands are then entered. For example, this is the method for searching and replacing text strings. The command line also serves as the medium of communication from the editor to the user - all warnings, error messages and other helpful messages indicating what the editor is doing at any given time are printed there. Another function of the command line is to serve as a status line to remind the user of the status of the various settings of the option flags.

Yet a third function of the command-line is to serve as the menu bar for the mouse. Whenever a mouse button is clicked, various letters and numbers appear on the command-line. Subsequently, by moving the mouse cursor to one of these letters or numbers and then clicking a mouse button again, a command will be executed. In this way, a powerful mouse-oriented interface is implemented as an alternative to some of the normal key and command-line commands.

1.1.4 Advanced Features

STEDI has a number of advanced features which make it one of the more powerful editors in existence today, including those which run on mainframes. A rich number of options exist for creating one's own personal environment while using the editor. These are stored if desired in a file which can be read at startup to recreate the environment each time the editor is used. STEDI also contains a very flexible buffer system with up to ten different buffers available for editing purposes, and a number of commands which are not normally found in other editors but which can be invaluable to programmers. In addition, there are ten 'learn buffers' which can remember up to 100 key strokes each for replaying, and the keyboard can be completely reconfigured so that a key can perform multiple actions if desired. Macro's can be created as text files and executed either from memory or from file. The 'folds' feature allows the user to organize his file according to segments which can selectively be displayed or hidden for editing purposes. The language for its regular expressions that can be used to define search patterns is very extensive.

1.1.5 The Manual

The first chapter of the manual, entitled 'Getting Acquainted', is an overview of the editor and provides a tour through the basics of using the program. It serves both as a tutorial and as an introduction to some of the advanced features of the editor. Next come chapters, arranged topically, which treat the various features of the editor in detail. A list of error messages is included with

explanations. Finally there is an index.

It is not necessary to read the entire manual prior to using STEDI because many of the editor's commands do exactly what one might expect them to do. For some users, a cursory glance through the chapters on various commands may suffice to get started. Alternatively, a reading of the first chapter of the manual should be sufficient to get an overview of the basic features of the editor. However, for greater versatility and more exhaustive use of the advanced features that set this editor apart from others, a thorough reading of the manual is recommended.

1.1.6 A note on key notation

Before ending this section, a few words are in order about the notation used in the manual to refer to the keys of the keyboard. All standard keys are referred to by the symbols which appear on them. For example, in referring to the T key or the $ key, the use of the Shift key is implied in the latter but not in the former. A number of keys may be labeled with words such as 'Help' and 'Return'. These keys will be referred to by enclosing the labels (or obvious short forms) in <>, such as <Help>, <Return>, <Esc>, etc. Among these keys are also some which can be used in combination with others to create a greater range of possible key commands. Such keys are the <Control> key, the <Alternate> key and the <Shift> key. When these keys are used in combination with others, the notation Ctrl-A, Alt-B, and Shift-C, etc. will generally be used. The notation sh-F1 may sometimes be used as an abbreviation of Shift-F1. The arrow keys will generally be spelled out, for example Shift-left-arrow or Shift-down-arrow, etc. Lastly, it should be mentioned that the keys of the numerical key pad to the right of the keyboard are not used at all except when they are equivalent to the normal keyboard keys, or when they are used in key redefinitions; also, the <Clr/Home> key is mostly used in a way related to the <Home> designation rather than in any capacity related to the word 'clear'. Hence this key will be referred to as <Home> instead.

1.2 Version 3.0

The first two versions of STEDI were programmed in assembler language and ran only on the Atari-ST computers. For version 3.0 STEDI has been reprogrammed in the language C. On the Atari-ST computers this gave a small slowdown of some functions and in addition more space in memory is needed. The advantages were also very great. It is now possible to use STEDI on various types of computers and many more features have been added. Some features have been

abandoned. STEDI doesn't exercise control over the display fonts any more. It will use the font that is available when it is started. If an external program is run that changes the font STEDI will adapt to the newly installed font whenever possible. In addition the so called mouse-help has been removed. It took very much space and not many people used it.

Some of the new features of version 3.0 are:

- A complete control language with variables and flow control. Scripts written in this language are referred to as macro's.
- A very complete language to describe patterns during a search operation. This language contains variables that can be used for the transfer of matched characters to the replacement string.
- The screen can be split into two windows. This split can be either horizontal or vertical. On systems that use terminals the vertical split may not be available because terminals rarely possess commands for scrolling a limited number of columns.
- The key redefinitions can be programmed with mnemonics. The scheme of key codes has been redefined to make it system independent. This makes it possible to move key redefinition files from one system to another with at most a few minor changes.
- A sort command has been added.
- The search command has some new options.
- The key redefinitions can also be used for the mouse menu. It is now possible to redefine the characters in the menu bar and compose ones own menu with the corresponding actions.

Basic operations

The fundamental operations of the editor concern cursor control, entering text and deleting text. This chapter covers these basic topics in detail. Before embarking on this tour of the basics of the STEDI program however, you must understand some aspects of how STEDI deals with text in terms of the screen representation. This topic will first be covered, followed by cursor control, entering text and deleting text in that order.

2.1 The screen

The screen is but a limited window onto the contents of a buffer. Unless a file is rather small, only a part of it can be made visible on the screen. The number of characters that STEDI can display on a single line may depend on the computer on which STEDI is running. Regular terminal displays (as with most PC, AT, PS/2 and Atari ST systems) can show 80 characters per line in their standard text fonts. Many screens will allow only 25 lines on the screen. If at all possible STEDI will adapt to the local restrictions. One line is reserved to allow the user to type in commands, obtain messages from STEDI and to see what the settings of some important parameters are. This leaves all other lines to be used to display text.

Lines that need more characters for their display than can be shown on a screen line can only be shown partially. This doesn't mean that the rest of the lines doesn't exist: it is simply not visible on the screen. Each line may actually contain up to 255 characters. If you attempt to add characters past this 255 character limit however, they simply will not be added. As for the length of an allowed file, STEDI can handle up to 99999 lines flawlessly, provided that your computer has enough memory for all those lines. Beyond this limit nothing is discarded, but the line number can no longer be displayed properly since only 5 characters are allocated for the display of this number. On the other hand, the internal line counter in STEDI can deal with numbers up to 2^{31}.

Besides the lines which actually belong to the file, STEDI knows the concept of **'virtual lines'**. These are lines beyond the end of the file that don't exist in reality (no characters are 'in' them and therefore they do not take up space in memory) but nevertheless the cursor can be moved to these lines. Likewise the cursor may be moved anywhere on the screen regardless of where the text appears. Thus not only can it be moved to virtual lines, but it can also be

moved past the last character in any line to the virtual territory to the right of a line. The movement of the cursor is not restricted by the text in any way when it is moved around. This behaviour is completely different from what is found in some editors which have an annoying 'dancing cursor' effect when the cursor is moved from line to line.

Aside from the case when a file is rather short and more virtual lines are necessary (for example, the 'Hello World' program in Chapter 1), the number of virtual lines cannot exceed 5. This means that when you scroll down past the bottom of a file, the end of the file will scroll above the bottom of the screen until five blank (virtual) lines appear. Remember: these lines don't exist in so far as the file is concerned, so they are not written when the file is written to disk.

In addition to the text cursor there is also a cursor associated with the mouse, called the **mouse cursor** or **mouse pointer**. It appears whenever there has been some mouse activity. Whenever this mouse cursor appears, the text cursor may stop blinking (this is systems dependent) so that it doesn't draw attention away from the mouse cursor. The rule is that the text cursor blinks when it was involved in the last action taken and this renders the mouse cursor invisible, while after any movement of the mouse, the mouse cursor becomes visible and the text cursor stops blinking.

2.2 Moving the cursor

2.2.1 Keyboard commands

The text cursor, or just cursor for short, can be moved around the screen with the arrow keys. As long as the cursor doesn't traverse any of the screen boundaries within which text can appear, the moving of the cursor is rather straightforward. If the cursor is at the edge of the screen on any side, and the move induced by the arrow key results in a move of the cursor to a position currently not displayed on the screen, a scroll will be induced. This means that STEDI will re-adjust its screen representation to show a different part of the file.

For **vertical scrolling**, the screen scrolls only one line up or down, giving the impression of motion through a file. This vertical scrolling can be continued until either the first line of the file has been reached, or the fifth virtual line after the last line of the file is displayed. The **horizontal scroll** moves the screen (actually it moves the screen representation of course) by a number of columns to the left or to the right. This means that the screen shows a range of columns which for instance can be columns 1 to 80, 21 to 100, 41 to 120,

etc. Once the display moves to a certain range of columns this range will be kept until a user command forces STEDI to move to a different range. The number of columns that will be scrolled when a scroll is needed can be set with a command line command. The command

```
set hstep = number
```

given from the command line (see p. 26 and later in this chapter) in which number is less than or equal to the number of characters that can be displayed in a single line, sets this stepsize.

The **arrow keys**, in conjunction with the **Shift keys** , can be used to move through a file in larger steps. **Shift-up-arrow** and **Shift-down-arrow**, also called Shift-up and Shift-down, scroll the screen by its number of lines minus 4 if the screen has at least 20 lines. For windows that aren't that high the 4 is correspondingly less. This action can be taken independently of which position the cursor occupies on the screen. Similarly **Ctrl-left-arrow** and **Ctrl-right-arrow** can induce a horizontal scroll independent of the cursor position. On computers with a **PageUp** and a **PageDown** key these keys take on their natural meaning and replace therefore the Shift-up-arrow and Shift-down-arrow combinations. These combinations may remain active though.

The lack of uniformity between the commands for horizontal and vertical motion has to do with the fact that another type of horizontal scrolling is used more frequently. **Shift-left** and **Shift-right** are assigned to these commands. The more often used commands which induce large horizontal movements of the cursor cause the cursor to go to either the beginning or the end of the current line. The end is defined as the position just after the last character in a line. Thus Shift-left moves the cursor to the beginning of the current line and Shift-right moves the cursor to the end. Either command may cause horizontal scrolls when necessary.

There are some extra cursor operations that are performed with the arrow keys. **Ctrl-up** or **shift-PageUp** scrolls the screen down by one line while leaving the cursor in a fixed position. This has the effect of scrolling the cursor up one line relative to the screen and is similar to what happens when the cursor is at the top of the screen and the up-arrow key is pressed. Similarly **Ctrl-down** or **shift-PageDn** induces a scrolling of the screen up by one line while the cursor remains fixed. These operations can be very useful at times.

Ctrl-Shift-left and **Ctrl-Shift-right** move the cursor in a text bound fashion. They move the cursor from character to character, treating a tab mark with its induced spaces as a single character and moving to the previous or the next line when there are no more characters in the current line. Some people prefer this way of moving the cursor. If one would like to change the

keyboard such that this cursor movement is the one connected to the normal arrow keys, one should consult the chapter on key redefinitions.

Finally there are some special positions in a file which need to be moved to rather frequently. These are the beginning of the file and the end of the file. When the **Home key** is pressed, the cursor moves to the beginning of the file. This key can also be used if for some reason the screen display becomes distorted or the line numbers are miscounting. In that role it is a kind of clear or editor reset key that re-evaluates the layout of the file being edited. The end of the file can be reached with the **End** key if there is one and otherwise with **Shift-Home**. This places the cursor after the last character in the file. For moving the cursor to the first position on the current screen (top line leftmost corner), one may use **Ctrl-Home** while **Ctrl-Shift-Home** will move the cursor to the left most position in the bottom line of the current screen. On computers with an 'End' key these codes are **shift-Home** and **shift-End** respectively.

2.2.2 Mouse commands

The cursor can also be moved around the screen with the mouse. When the mouse is moved, the mouse pointer becomes visible. Then if it is positioned at any desired point on the screen and the left mouse button is clicked, the text cursor will move there.

The equivalent of PageUp (Shift-up) and PageDn (Shift-down) can also be performed with the mouse. This is done by pressing the right mouse button while the mouse pointer is in the top line or the bottom line of the screen. The equivalent of Ctrl-left and Ctrl-right can be performed by clicking the right mouse button while the mouse pointer is in the leftmost or rightmost column of the screen.

2.2.3 Command line commands

The above operations concern local moves through a file. Absolute moves can be made to any line when its number is known. This is done via the command line. To this end, one presses the key marked <**Esc**> after which the cursor leaves the text area and reappears in the line at the bottom of the screen to wait for a command line command. The command to move to a given line can be given by just typing the line number followed by a return. The cursor will then reappear in the text window and the screen will be moved to the proper part of the file. This command would be used very frequently during the development of a program when a compiler tells the number of the line in

which errors were found.

It is also possible to select the column to which the cursor is to be moved. The syntax of this statement is rather similar and actually together with the above command, it is part of a single more general command. With the cursor in the command line (via pressing the Escape key) type the line number desired, followed by a comma, then the column number desired and then type Return. This will result in the cursor moving to the designated line and column in the file corresponding to the numbers given. If the first number is omitted (i.e. just a comma followed by a number) the cursor moves to the designated column inside the current line. If the column number is omitted (the comma is then irrelevant and therefore not necessary) the column number will not be changed when the cursor is moved to the designated line. Finally this command has an option for very advanced programmers who like occasionally to go to a given character in the file which is known by its number in the file. For such a command one should type two comma's, followed by the number of the character.

There is still a class of commands that are specially designed for people who would like more flexibility. The cursor can be moved from the current line forward or back a designated number of lines. To move the cursor forward a given number of lines, go to the command line and type a plus sign followed by a number (+`#`) and <Return>, where `#` is the number of lines forward the cursor is to be moved. Similarly, to move backward a designated number of lines, type a minus sign and then a number (-`#`).

The first five spaces in the command line (the bottom line on the screen - sometimes called the message bar) indicate the **line number** of the cursor at any given time. If you would like more information about the location of the cursor, the command **'='** in the command line will give a message in the command line that also tells the column number of the cursor. In addition, this command gives the following information. First, we have the character number the cursor would be on if the file would be written in its current form with the current settings (this is a function of whether the tabs would be expanded into blanks and how the end-of-line marks should be represented - see the chapter on File input and output). Next is the number of bytes that the whole file would occupy if written with the current settings. Then the line number of the cursor on the screen is given, followed by the column number of the cursor on the screen. The final number tells how many lines there are in the current buffer. An example is:

```
1 bytes: 13612/22567 screen:21,1 lines 383
```

while the line number is 231. It indicates that there are 383 lines in the file

and that the cursor is in column 1 (on the screen row 21, column 1) and that the byte position is 13612 out of 22567 bytes in total.

2.3 Entering text

Entering text is always done at the position of the cursor. Any newly-typed character is added to the text at the position occupied by the cursor, after which the cursor is moved one column to the right. If this motion moves the cursor to a part of the file presently outside of the screen display, the screen is scrolled horizontally.

The edit mode determines what happens to the character that was at the position of the character just added. There are two of these modes: the **insert mode** and the **overstrike mode**. In the overstrike mode, the old character is simply replaced by the new character, so the length of the file isn't altered, unless there was no character under the cursor to begin with. In the insert mode, the character under the cursor and all characters to the right of it are moved one position to the right and the new character gets 'inserted' between the other characters. Which mode is currently selected can be seen from the status characters.

The first status character (appearing on the right side of the command line - sometimes also called the status bar) is either an `I` indicating that the insert mode is selected or an `O` for the overstrike mode. The insert mode can be selected with the **Alt-I** key combination while the overstrike mode can be selected with the **Alt-O** key combination. Another way to change from one mode to the other is to click either mouse button while the mouse cursor is over either the `I` or the `O` in the status bar.

There are some keys which cause some special effects when inserting text. These are the <Tab> key and the <Return> key. The tab is used to move the cursor to the next tabulated position on the screen. A tab is represented in the buffers as a single character and its interpretation depends on what tabulator position the user has selected. This is all explained in the chapter about tabs.

Since normally the presence of a tab in the file cannot be distinguished visually from the presence of several spaces, a special command is available for that purpose. If you press **Alt-T**, all blanks will be replaced on the screen by small open circles, and tabs are indicated by small closed circles. In addition the character with the internal representation 00 (which is rarely used in text files) is represented by a fat dot. On some systems the character with the internal representation 255 (hex FF) is represented by a colon. In this way you can always find out exactly what characters are in the file you are working on.

Pressing Alt-T again reverts the screen representation back to normal. Further information about this and related commands can be found in the chapter on tabs. On some systems the above characters that are used to display the tabs etc. may not be available. In that case other characters are used. The user can find out quickly which characters these are by experimenting.

The **<Return> key** terminates the current line at the position of the cursor, opens a new line after the current line but before any lines that follow it, and moves all characters that were after the cursor (including the one at the same position as the cursor) to the beginning of the new line. If you want to simply insert a new line, you may press the **Insert key**. This key creates a new empty line just above the current line and moves the cursor to the beginning of this new line. Likewise, a new line can be created just below the current line with the **Shift-Insert key** combination. Both these commands are independent of which column the cursor was at previously.

The return or end-of-line character is not seen by STEDI as a character. Internally each line is a separate entity, and each entity has an end automatically. This allows for considerable flexibility when writing the file with several options for the interpretation of what character should be included to indicate the end of a line.

When a file is read in, all end-of-line characters are stripped off, and each line of the file is assigned a separate line in STEDI's buffer. This is done in such a way that STEDI can read either UNIX conventions or the more widely used sequence of carriage return + linefeed for indicating the end of a line. The latter convention is used by most programs on PC's, AT's, PS/2 systems and Atari's. This way of handling an end-of-line in the editor buffers is the reason that when a return is typed at the end of the last line of a file, a new line isn't created yet. It is only created after a character has been entered in the new line or when a second return is pressed. If one needs to have a file with no end-of-line characters at the ends of the displayed lines, one should edit in the **raw** or **binary mode**. For more information one should consult the chapter on reading, writing and printing.

The cursor can be positioned anywhere on the screen, so there might be a question about what would happen if the cursor is at a position in virtual territory and a character is typed in. The rule is that any character typed in virtual territory remains exactly where it is in relation to the rest of the text on the screen, and STEDI takes care of adding the appropriate number of spaces or new lines so that the new character becomes part of a contiguous body of text. Thus if the cursor is moved to the right past any text in a given line, and a character is inserted, enough blanks will also be inserted to make this new character the end of the line. (This can be tested with the Alt-T command

on.) If the cursor is moved past the end of the file into virtual lines, and a character is added, enough lines will be created to include the new character in a line of the text. This makes the adding of new characters into the text very intuitive, and much easier than if the cursor would be restricted to the text.

There is another way of entering text into a buffer that is very convenient for macro's and stream files. The command

```
"string"
```

given from the command line (or a macro) will enter the characters in 'string' in the text. There are only very few characters that can give problems if one would like to insert them this way. One is the double quote itself. Another character is the linefeed. Finally the dollar sign can give problems as it is used to indicate a variable (see p. 111). All these characters can be used if they are preceded by an escape character which is either the character <escape> or the backslash (for linefeeds only if the file system doesn't use the backslash as a directory separator). In addition a linefeed can be indicated by the combination `\n`.

2.4 Deleting text

The **Delete** and the **Backspace keys** are used for the simplest delete operations. They work fully naturally: The character under the cursor can be deleted by pressing the **Delete** key. This is a so called forward delete. All characters to the right of the cursor are moved one position to the left and the cursor stays at the same place on the screen. Repeated use of this operation deletes more and more characters that used to be to the right of the cursor. A backward delete is executed by pressing the key marked **Backspace** . This deletes the character that is to the left of the cursor. Afterwards the cursor moves one position to the left and all characters that were on the cursor or to the right of it are moved one position to the left. Repeated use of this key deletes more and more characters that were to the left of the cursor.

When either one of these commands is used and there are no more characters in the current line for it to delete, the end-of-line of either the current line (for the forward delete) or the previous line (for the backward delete) will be removed and two lines will be joined together. Using the backspace inside the range of a tab (the extra space on the screen created by the presence of a tab) or in virtual territory to the right of a line (at least one column past the end of the line) results only in moving the cursor one position to the left. No other action is taken.

If the backspace is used when the cursor is just to the right of a tab character, the tab character is deleted and the cursor may move several columns depending on the tab settings (also the induced spaces are removed). A tab character is also deleted when the delete key is pressed and the cursor is inside the range of the tab and again the cursor and text may jump several columns. When the delete command is given and the cursor is in virtual territory beyond the end of a line, blank characters will be generated to fill the space between the end of the line and the cursor and then the end-of-line is removed. That is, the next line is joined to the current line at the position of the cursor.

The above backspace actions hold when STEDI is in the insert mode. In the overstrike mode, the action of the backspace is to overwrite the previous character with a blank. Only at the beginning of a line will it resort to its normal action of deleting the end-of-line and joining the current line with the previous line.

An entire line can be deleted either with the **shift-<delete>** (on the PC) or with the **Ctrl-<delete>** (on the Atari-ST) combination. **Ctrl-D** deletes all characters to the right of the cursor (including the character under it). This is called 'Delete to end of line'.

All the above deletions, with the exception of the 'backspace' in the overstrike mode can be undone with the undo key, as long as no other actions have been taken. Sometimes more than one delete can be undone. In that case either consecutive lines were deleted after each other, or character deletes (or delete to end of line) were made in consecutive lines. The details are explained in the chapter on 'undo'.

2.5 Exchange operations

There are two operations that neither add nor delete text. They only exchange the position of two objects. The first command is **Ctrl-T**. Its effect is to repair the most common typing error, being the wrong order of two adjacent characters. The cursor should be in the position immediately after the two characters that should be exchanged. This is the position in which the cursor is, just after the mistake has been made. This is the transpose command.

The other exchange operation exchanges the line with the cursor in it with the line under it. This can affect even whole folds. as a closed fold is seen as a single line (see the chapter on folds p. 64).

The status bar and options

In this chapter we will examine the status bar and the editor options displayed on it. In addition we will go on to examine other options.

3.1 The status bar

Besides the messages that appear in the center of the status bar from time to time, and the letters which comprise the mouse menu, the status bar has a number of fields which record various settings of options of the editor. On the left side is the line number. On the extreme right is the buffer name. Then just to the left of the buffer name are nine character fields which indicate settings of the editor. These fields will be covered in order from left to right. They are called status characters because each one indicates the status of a particular setting of STEDI. A summary of these fields is as follows.

I/O	Insert or Overwrite mode
A/U/R/P	MS-DOS or Atari/ Unix/ Raw/ Printer output mode
0 or ∞/≫/≪/>/<	Search Direction mode
N/S	Case Sensitivity
Y/y	Yank buffer in use
B/b/!! or †	Backup options
1/2/.../0	Current buffer
∘	'Dirty Bit' telling whether a file has been changed

3.1.1 Insert and Overwrite mode

The first letter on the left of the string is either an 'I' or an 'O' indicating that the editor is in 'insert' or 'overwrite' mode respectively. These options are exercised with the **Alt-I** or the **Alt-O** command. Alternatively the mouse can be used to toggle between these two modes. This is done by clicking a mouse button while the mouse pointer is pointing at the first field.

If the insert mode is set, characters will be inserted into the text at the place of the cursor whenever keys are pressed and characters that are already there are simply moved aside. In overstrike mode the effect of typing characters into

the text will be to overwrite characters that are already there. See the chapter on basic operations p. 17.

3.1.2 The Write Mode

STEDI is able to save files in several write modes. The indicator for these options exhibits the value 'A' for the regular MS-DOS mode used both by MS-DOS and the Atari-ST, 'U' for Unix mode, 'R' for Raw mode or 'P' for Printer mode in the second status character of the status bar.

The following table shows the actions performed on the lines of the text when the file is saved under the various write options:

MS-DOS and Atari mode (A)
: a carriage return and a linefeed represent the end of each line when the file is written.

Unix mode (U)
: only a linefeed represent the end of each line when the file is written.

Raw mode (R)
: no linefeed or carriage return is included at the end of the lines when the file is written. If a file is read in in this mode carriage returns and linefeeds will not be interpreted as end-of-line characters. This gives a limited possibility for editing binary files directly.

Printer mode (P)
: a carriage return and a linefeed represent the end of each line when the file is written. In addition all tabs are expanded into the appropriate number of blank spaces. (This mode can be used to prepare a file for printing when the printing cannot be done directly from the editor or for use with a compiler that cannot interpret tabs properly.)

These options can be toggled with the mouse, or set with the commands **Alt-A**, **Alt-U**, **Alt-R** and **Alt-P** respectively.

3.1.3 The Search Direction mode

The directions for search and replace operations (p. 70) can be set to:

- forward mode ($>$ or $\gg$)
- reverse mode ($<$ or $\ll$)

- circular mode (either 0 or ∞)

The forward and reverse modes apply to single and multiple replaces respectively in a search and replace command (see p. 71 and p. 73). The third character indicates which of these options is set.

The **Alt-D** command and the **Alt-E** command toggle circularly in opposite directions among the various printer options. The mouse can also be used to toggle among these settings, the left mouse button being equivalent to Alt-E and the right mouse button to Alt-D.

The circular mode is represented either by a zero or an ∞ sign, depending on whether the local font has an infinity sign.

3.1.4 Case Sensitivity

The fourth status character ('S' or 'N') indicates whether search and replace operations will be case-sensitive (S) or not (N) (p. 70).

To set this command you may use the **Alt-S** or **Alt-N** keyboard commands to set S or N respectively, or the two can be toggled between using the mouse.

3.1.5 Yank buffer

The fifth status character tells which of the two yank buffers of the editor is in use (see section on 'Buffers' p. 59). 'Y' stands for the YANK buffer (buffer 9) and 'y' stands for the yank buffer (buffer 0). These two settings can be toggled between using the **Alt-Y** command or the mouse.

3.1.6 Caps Lock

The sixth position among the status characters is a position for the caps lock option. Whenever the caps lock option is on an upper case 'C' occurs in this field. Otherwise this field remains blank. On some computers this option doesn't serve a purpose as one can see on the keyboard whether the caps lock is active. There are however many keyboards that don't show the status of the caps lock key.

3.1.7 Backup Mode

STEDI allows three modes for creating backup files. These are the normal backup mode 'B', the invisible backup mode 'b', and the no backup mode (either ! or !! or †depending on the character fonts). A 'V' can also occur in this field indicating that the buffer is in 'view only' mode. The seventh status

character indicates which of these settings is currently chosen. The backup options have the following effects.

B
: Whenever a file is saved to disk, the previous version will be kept, with the file extension '.bak' now appended to its name. If the system allows only a single file extension the original extension is removed first.

b
: The backup will be made invisible or hidden. This can be useful for those using a hard disk and have very full directories. The same naming convention applies as with the 'B' option.

! or !! or †
: No backup at all is made. Many users start with using this option and a fair percentage switches to another option after a while. When trying things out with a difficult source file it may be nice to still have an older version after execution of the new version turns out to be a disaster.

Care should be exercised with the use of the last option, though it may be necessary at times if no room for a backup is available. The backup flag can be toggled with **Alt-B**.

A 'V' can also occur in this space of the status bar. This indicates that the file is 'View Rights Only' In this mode, a file can be edited but the editor will not allow the altered program to be saved. This option can be set as follows:

Alt-V
: Sets the 'View Only' option.

Ctrl-V
: Gets you out of the 'View Only' mode.

No mouse command is available for toggling among these options.

3.1.8 The buffer number

STEDI has 10 buffers for editing purposes. The eighth character in the status bar tells which buffer you are currently working in. (see the chapter about buffers p. 59.)

To change buffers, press **Alt-#** where # is a number from 1 to 0 (0 standing for 10). The last two, buffers 9 and 0, are reserved for yanking (cutting),

copying and pasting. Nevertheless, they are treated like the first eight buffers for all other purposes.

To change buffers with the mouse:

1. Press either button of the mouse (to make the mouse menu bar appear)
2. Point the mouse pointer at one of the numbers (1 to 0) in the center of the menu bar
3. Press the left mouse button.

The editor will now be switched to the buffer you indicated. The default is usually Buffer 1.

3.1.9 'Dirty Bit'

The last indicator before the name of the file tells you whether or not the file in the current buffer has been changed since an editing session began. If the file is unchanged, this field will be blank. If any change has been made, a small empty circle (○) will appear. In addition, if the editor is waiting for direct input of hexadecimal characters (see the chapter on 'Hex Mode' p. 144) then the circle becomes filled (●). For completeness the settings of the dirty bit can also be toggled. This is done with the command line commands

```
set dirty = on
set dirty = off
```

If should be noted that turning the dirty bit off is rather dangerous. If you try to leave the editor after turning it off the editor will not notice that the buffer has been changed and leave quietly.

Note: In addition to the attributes of the editor which are indicated in the status bar, there are a number of other optional settings available. (See for example, tabs (p. 94), auto-indent (p. 103) and word wrap. (p. 101)) Many of these, including all of the options mentioned above except the caps lock and yank buffer options, can be set differently for different buffers. They can also be stored away in a default file for later editing. For information on this subject, see the chapter on 'defaults', p. 51.

The command line

A number of the commands can be entered by typing them in on a command line. These are usually commands for which more than a minimal amount of information is needed to specify the command fully. In systems that are equipped with windows the standard solution to this problem is to select a command, perhaps through a menu using the mouse, and then to enter further information through dialog boxes. It was judged however that such an interface, however productive it is in learning for those users who are not yet very well acquainted with STEDI, it would only be a hindrance for serious work. After some expertise in editing is gained, dialog boxes are rather counterproductive as one is forced to switch continuously from the mouse to the keyboard and back. Moreover dialog boxes obscure part of the text and this can be rather annoying during some operations like the replacement of a complicated string.

The command line allows the user to type in the commands that would normally be handled with a dialog box but in a part of the screen that "isn't in the way". The line at the bottom of the screen is called (among other things) the command line. One can enter it by pressing the <Escape> key. The cursor is then positioned at the ninth column of this line and the character at the seventh position serves as a prompt. The user may then type his command and end it by pressing the <Return> key. After this STEDI will execute the command, possibly give a message, and return the cursor to the text window. Clearly this procedure is very versatile, and extremely fast for those who have some expertise in typing.

If the escape key is pressed and there has been a command issued from the command line previously, this old command will be displayed in the command line, with the cursor over the first character of this command. One can now either modify the old command or type a new command. The default typing mode in the command line is the insert mode, so whenever a new character is typed everything to the right of the cursor is moved one position to the right. When a carriage return is typed, all characters to the right of the cursor are discarded and the command executed is formed by the characters to the left of the cursor. Normal editing can be performed in the command line. It isn't possible to place a mark in the command line or to execute a search operation in it but it is possible to paste parts of a line into the command line. The commands in the command line are also entered into a history mechanism which has a fixed number of entries. The default is 8 entries. One can scroll

through these entries with the up and down arrows. The buffer is 'circular', i.e. after going up 9 steps one comes back to the most recent command (9 back = 1 back). There is one exception to the storing of commands in the history mechanism. If the write command is given with the <shift>-F8 key a very special line is used to ensure that the user can always write his file. This line isn't entered in the history because it should always be available. The number of entries in the history mechanism can be changed with the command

```
set maxhist = number
```

in which 'number' is the new size of the history buffer. The size of the buffer is stored in the default file when it is written with the 'DW' command (see p. 52).

It is possible to change the typing mode in the command line to the overstrike mode with the Alt-O key combination. For this the cursor has to be in the command line. Switching back to the insert mode is done with the Alt-I key combination. Pressing an incorrect key during the typing of a command can possibly generate a message which overwrites the partially typed command. The partially typed command can be restored by pressing the <Home> key or by simply continuing to type the command into the command line.

When the command to be issued is longer than the available space on the screen the command line will be scrolled horizontally. The same will occur when the cursor is moved to the edge of the active space in the command line. The number of columns over which the command is scrolled can be controled with the built in variable 'mstep'. This variable can be set with the command

```
set mstep = 'number'
```

in which 'number' is any resonable step size.

The variety of commands issued from the command line is rather large. An attempt has been made to keep them ordered somewhat by having the commands divided into groups that all start with a single character. This character is chosen so that it corresponds as much as possible to the name of the family. An example is 'p' for printing. STEDI is not case sensitive with respect to the keywords and the options of the commands.

Before any command is executed STEDI will scan the contents to see whether any variables should be substituted first. These variables all start with a `$` sign so if the user needs a `$` in his command he may escape it by preceding it with either a backslash (\) or an <escape> character. If the characters after the `$` cannot lead to confusion (no defined variable) then there is no need to escape the `$`. See the chapter on variables (p. 111).

Finally if the **Ctrl-R** commands is pressed, whatever command line command that was last issued is repeated. This can be a rather useful command, especially during repetitive search operations.

4.1 The regular commands

The commands that can be issued from the command line are (# stands always either for a single digit or for a multi digit decimal number):

A, A+, A-, A#, A#+
Auto-indent commands. See the chapter on word-oriented commands p. 103.

DD, DR, DW, DD pathname
Default file actions. See the chapter on defaults p. 51.

F, FH, FV, F+#, F-#, FH#, FV#
Split window commands. See the chapter on screen control p. 135.

I#
Take command input from buffer #. See the chapter on stream editing p. 128

K0, K filename
Change key redefinitions (p. 148).

L# (# is any digit)
Start learning in buffer # (p. 108).

MC name, MD name, MV, MV name
Create a macro, delete a macro, view macros. See p. 119.

O, O#1,#2, O:#1,#2, OF#<char>
Sort operations. They are explained in the chapter about sorting, p. 138.

P, PF, PP, PP, PP#, PP#,#, P=
Print commands. See the chapter on reading, writing and printing, p. 47.

Q
Quit (= leave the editor). See the chapter on running STEDI p. 170.

R name, R name ¡ number, Rf
Read commands. See the chapter on Reading, Writing and Printing p. 36.

S, SQ
Save, Save+quit, See the chapters on reading, writing, printing and on running STEDI p. 170.

T#, T#∗#, TG#, TG#∗#
Tab settings. See the chapter on tabs p. 94.

TE, TT, TAB
Special tab commands. See the chapter on tabs p. 94.

U, U+, U-
The U or U+ command forces a screen update. U- command switches off the updating of the screen during the current replay of either a learn buffer, a key redefinition, a macro, or a stream script. If during such a replay the U+ command is encountered the updating of the screen will be resumed. See the chapter on screen control p. 135.

W name, W>name, WF, WF>
Write or append commands. See the chapter on Reading, Writing and Printing p. 36.

WW, WW-, WW#, WWF
Word wrap. See the chapter on word-oriented commands p. 101.

X macroname arguments
Execute a macro. See the chapter on macro's p. 120.

!command, !+command, !-command
See the chapter on executing external programs p. 131.

 # ,#,# ,,# ,,,# ,+# ,-#
Various move to line or column commands. See the chapter on basic operations p. 16.

[, [a, [#, ['name
Fold opening commands. See the chapter on folds p. 64.

],]a,]#,]'name
Fold closing commands. See the chapter on folds p. 64.

=
Find position of the cursor. See the chapter on basic operations p. 16.

/string, /string/options
Search command. See the chapter on searching and replacing p. 70.

/string1/=/string2/options
Replace command. See the chapter on searching and replacing p. 72.

//string, //string/options
Search command with the use of pattern matching. See the chapter on regular expressions p. 78.

//string1/=//string2/options
Replace command with the use of pattern matching. See the chapter on regular expressions p. 78.

=/string/options
A special type of replace commands. See the chapter on searching and replacing p. 75.

?drive
Find free diskspace. See the chapter on reading, writing, printing.

<#, >#
Commands to place a tag (<) or to go to a tag (>). See the section on tags p. 57

'name
Find buffer 'name'. See the chapter on buffers p. 60.

"string"
Put 'string' in the text at the current position of the cursor. See p. 19

∼
When ∼ is followed by a single character this command has the same effect as the Alt-char key combination.

∧
When the ∧ is followed by a character the effect is the same as when the combination of this character and the control key has been pressed.

The ∼ and ∧ commands have been provided for use in macro's and stream scripts. These commands use the 'raw' bindings of Alt, Ctrl and function keys, so they are insensitive to key redefinitions.

4.2 Special commands

In addition to the above commands there is a list of commands of which the full name has to be spelled out. These commands are:

After
Same as shift-insert. Adds a line after the current line (p. 18).

Alt-<char>
Same as the combination of the alternate key and the given character.

Backspace
Same effect as pressing the <backspace> key in the text buffer (p. 19).

Bcopy
Copies the rectangular block between the mark and the cursor to the current yank buffer. Same as shift-F4 (p. 61).

Bcut
Cuts the rectangular block between the mark and the cursor to the current yank buffer. Same as shift-F3 (p. 61).

Bpaste
Pastes the contents of the current yank buffer in block mode into the text at the position of the cursor. Same as shift-F5 (p. 61).

Clear
Empties the current buffer. Same as shift-F9 (p. 60).

Copy
Copies the region between the mark and the cursor to the current yank buffer. Same as F4 (p. 61).

Ctrl-<char>
Same as the combination of the control key and the given character.

Ctrl-down
Moves the cursor to the next line, but tries to leave it at the same position on the screen. The net effect is that the screen scrolls up (p. 14).

Ctrl-home
Moves the cursor to the top left corner of the current window. See also p. 15.

Ctrl-left
Scrolls the screen horizontally. The cursor ends a number of columns to the left of its original position but keeps (if possible) its position on the screen (p. 14).

Ctrl-right
Scrolls the screen horizontally. The cursor ends a number of columns to the right of its original position but keeps (if possible) its position on the screen (p. 14).

Ctrl-up
Moves the cursor to the previous line, but tries to leave it at the same position on the screen. The net effect is that the screen scrolls down (p. 14).

Cut
Cuts the region between the mark and the cursor to the current yank buffer. Same as F3 (p. 61).

Delete
Same effect as pressing the <delete> key in the text buffer (p. 19).

Deleteline
Same as Ctrl-delete. Deletes the current line in the text buffer (p. 20).

Delmark
Deletes the mark in the current text buffer. Same as shift-F1 if the computer has a help key (p. 56).

Down
Moves the cursor down in the text buffer (p. 13).

End
Same as shift-home or <End>. Moves the cursor to the end of the current text buffer (p. 15).

Exchange
Exchanges the position of the mark and the cursor in the current text buffer. Same as F2 (p. 56).

First pattern
Initiates a search for files of which the names match the given pattern. See file searches p. 126 and the chapter on macro's p. 114.

Garbage
Forces a complete garbage collection. See the section on garbage collections (p. 142).

Gotomark
Moves the cursor to the position of the mark in the current text buffer. Same as shift-F2 (if there is an undo key) (p. 56).

Home
Same as pressing the <home> key in the current text buffer. Moves the cursor to the home position and reevaluates the entire buffer (p. 15).

Insert
Same as pressing the insert key. Puts a new line before the current line in the text buffer (p. 18).

Left
Moves the cursor one column to the left (p. 13).

Mark
Places a mark at the position of the cursor in the text. Same as F1 (p. 56).

Message "string"
Puts the given string in the message line (p. 142).

Next
Moves the cursor to the next character in the text buffer. Same as shift-ctrl-right (p. 14).

Paste
Pastes the contents of the current yank buffer into the text at the position of the cursor. Same as F5 (p. 61).

Pause number
Makes STEDI wait for the required number of deciseconds (tenth of seconds). See the corresponding section p. 143.

Previous
Moves the cursor to the previous character in the text buffer. Same as shift-ctrl-left (p. 14).

Quit
Quits the current edit session. Same as shift-F10 (p. 171).

Right
Moves the cursor one column to the right in the text buffer (p. 13).

Save
Saves the contents of the current buffer. Same as F9 (p. 42).

SaveQuit
Saves the contents of the current buffer. Next the Quit command is executed. Same as F10 (p. 171).

Set var = expression
The command with which to set some internal variables and variables that can be used for instance by the macro processor. See the chapters on variables p. 111.

Sh-Ctrl-home
Moves the cursor to the bottom left corner of the current window. See also p. 15.

Sh-home
Same as shift-home or end. Moves the cursor to the end of the current text buffer. This can be confusing on systems that have an 'end' key. Those systems use shift-home to place the cursor at the top of the screen. It is safer to use the 'end' command (p. 15).

Sh-insert
Same as shift-insert. Adds a line after the current line (p. 14).

Sh-left
: Same as shift-left. Moves the cursor to the first column in the current line (p. 14).

Sh-right
: Same as shift-right. Moves the cursor to the last column in the current line (p. 14).

Show var
: Show the contents of an internal variable. See the chapter on variables p. 112. In this command there shouldn't be a `$` before the name of the variable.

Up
: Moves the cursor one line up in the current buffer (p. 13).

Reading, Writing and Printing

Having covered the basics of moving around in a file and the various possibilities for inserting and deleting text, in this chapter some file systems will be discussed in some detail. Then we will go on to explain the various possibilities for reading files into STEDI, writing them out to disk to save your work, and sending a file directly to a printer. The sections of the chapter will cover these four topics in the order just mentioned.

5.1 MS-DOS/Atari file names

The MS-DOS and Atari name conventions differ slightly from the more widely used UNIX conventions. This difference may sometimes lead to confusion. The first element in the full path name of a file is the drive indicator. This indicator is a letter in the range A to P inclusive followed by a colon. Whether the letter used is upper or lower case is inconsequential even though the Atari desktop seems to make a distinction in showing the first partition on a hard disk as drive C and a ROM cartridge inserted in the cartridge port as drive c. Some restrictions in the use of drive designations are that drives A: and B: are usually reserved for the floppy disk drives and if there is a hard disk, it must include a partition labeled C:.

After the drive indicator, the full path name of a file includes a possible list of folder (or directory) names each separated by a backslash (\) and then the name of the file. In order to specify a file at any given time, certain elements of the full path name may be omitted if circumstances permit, as will be evident from the following description.

If a path name includes the name of a drive, the operating system assumes that the file is on that drive. If such a name is absent, the file is assumed to be on the currently active drive. There is always one drive that is designated as the 'currently active drive'.

For each drive there is a 'current directory'. If the drive has never been opened, the current directory is the root directory, indicated by a single backslash (\) after the drive indicator. Thus the root directory in drive A: is indicated by A:\ while if the backslash is omitted, A: stands for the current directory in drive A:. These two are not necessarily the same.

Then follow the names of subsequent directories, each separated by a backslash followed by the file name. A file name consists of two fields separated by

a period (.) and totaling up to 12 characters in length. The first field which is the proper name of the file can be up to eight characters long and the second, called the 'file extension' can be up to three characters. The file extension may also be omitted entirely along with the period.

The characters of a file name may include many of the non-alphanumeric characters but there are some restrictions. The character with the hexadecimal code E5 is used by the file system as the first character of a name to indicate that this file has been erased. It would not be very wise to make a file of which the first character has this code. An asterisk (*) can be used as a wildcard for one or more arbitrary characters. But when the operating system encounters an asterisk, it does not look beyond the period of the extension, so for a completely general search one has to use the string *.* in place of the more common single * in UNIX. The question mark is used as a wildcard for a single character. The names of directories (folders) obey the same rules as those for file names with the exception that wildcarding is not allowed. Internally all file and path names are converted to upper case. It is however possible to use lower case names in addressing these files as the file system is essentially case insensitive.

The directory that was the current directory when the editor was started up is taken over as the current directory for the entire session of the editor. The one exception is when one starts a child process (with the `!` command, see p. 131) that changes the current drive. From the above it is evident that if you wish to designate a file in the current directory, the path name can be omitted and only the file name need be specified. For files in a subdirectory of the current one, only the subdirectory need be specified followed by a backslash and the file name. Finally, as in UNIX, a special name exists for the parent directory of the current directory (the directory in which the current directory is). This directory can be represented by two periods (..).

Examples of path names :

```
B:\folder1\file1.001
```

Drive B has in its root directory the folder 'FOLDER1' and it contains the file FILE1.001

```
A:FOLDER2\file2.xyz
```

The current directory on drive A contains the folder FOLDER2 and the latter contains FILE2.XYZ. If `A:\FOLDER3` was the folder opened last on drive A this path name would correspond with `A:\FOLDER3\FOLDER2\file2.xyz`

Similar to the difference above is the difference between the following:

```
D:\file3.a
```

This signifies an 'Absolute' path name.

```
D:file3.a
```

This file is to be found in the current directory on drive D:.

Finally some path names excluding the disk indicator:

```
file2.x
```

This is file called file2.x to be found in the current directory.

```
down1\file1
```

This is file called file1 in a directory called down1 which is contained in the current directory.

```
..\file0
```

This is a file called file0 in the parent of the current directory.

Although for the most part STEDI follows the local file and path name conventions, the editor has a somewhat more flexible approach to path names. The most notable difference is that in place of the usual backslash (\) of MS-DOS and GEMDOS, a slash (/) may also be used at any point. When STEDI encounters a slash, it is converted to a backslash before passing it on to the operating system. Afterwards the backslash is changed back to a slash when communication with the user is called for. Should you desire, this allows you to use the more widely known UNIX conventions for path names as well as those of MS-DOS or the Atari. Although STEDI tries to be agreeable and will use slashes or backslashes when you do, if you insist, you may use both in the same path name and cause some rather unorthodox path names to be displayed. This will not affect the proper execution of input/output commands.

5.2 Reading a file

Reading a file can be done in one of five different ways:

The F8 command
: The F8 command, when executed with the cursor in the text field, gives a prompt in the command line requesting a file to be read in. One may

type the name of the file that is desired to be read (including path name) and follow it by a carriage return. This results in the file being read into the editor at the current position of the cursor. Whenever the command is used, the file name specified is saved so that upon the next use of the read (F8) or write (shift-F8) command, the name will be displayed automatically after the prompt. You can use the entire name again, or part of it, by means of the normal editing procedures for the command line (see command line editing p. 26). Entering a null string or just blanks will cause no action.

The mouse menu R command

This read command is executed by clicking a mouse button on the R in the mouse menu. It is of course only supported on systems with a mouse. This command may be slightly different from the F8 command. If a file selector is supported it will not prompt for a file name in the command line but will cause the file selector to be displayed. Using the rules of the file selector the user may select a directory/folder and a file name, after which a click on the OK box will cause the file to be read. If no file selector is supported this command is identical to the F8 command.

Reading a file at startup

At startup time the editor may read one or more files. If the editor has been started without a command tail—as would be the case from a desktop—and a file selector is supprorted by the loacl system the editor starts by displaying the file selector. What follows is exactly the same as if the mouse menu bar command 'R' were used. If there is no file selector, the editor will simply wait for you action, showing an empty buffer.

Startup with a command tail

If the editor is started up from a shell program like in MS-DOS or in any UNIX system, it is possible to provide it with a command tail. This tail is scanned for options (p. 168) and for the names of files that should be read. Up to eight files can be read this way. Any wildcard characters that are given to the editor are passed to the file system and if more than one file matches the pattern the various files are read in different buffers. The number of buffers is of course still limited to 8.

Several options are available when reading files this way. They should precede the name of the file as a separate parameter. They are:

-r Read the file in the 'raw' mode. This is used for binary files in which carriage returns and linefeeds are left uninterpreted.

-v Put the buffer in which the file is read in the 'view only' mode to avoid accidental writing.

-# After reading the file is positioned at the line indicated by the number #.

-i The following name is interpreted as the name of a macro. This macro will be executed after startup. Any parameters after the name of the macro are passed on as arguments to the macro. After the execution of the macro is finished buffer 1 is saved and the execution of STEDI is halted. This is the stream editing mode, in a way comparable to *sed* in UNIX systems.

-x Same as option -i.

+# The number given indicates the number of bytes that should be skipped in the file before reading starts. This allows the user to edit a part of a very big file that won't fit in memory in its entirety.

With the R command

This command knows three varieties: In its simplest form the R is followed by either a blank space or a quote. The name that follows is then interpreted as the name of the file to be read. This command is fully equivalent to the F8 command.

The other form concerns the reading of a part of a file. The command 'R name < number' reads from the file 'name' starting at byte 'number'+1 Again there may also be a quote between the R and 'name'. This mode is very useful when processing very large files. Normally only the first part of such a file can be read and the message "Not enough memory. Buffer made View-Only." would appear, notifying the user of the inaccessibility of the tail part of the file. By reading from different positions in the file one can edit the file in several steps.

Finally the command 'RF' is functionally equivalent to clicking on the R in the mouse menu. It can only be used if a mouse and a file selector are available on your system. The file selector box will appear and the user is expected to make his selection.

When a file is read into a buffer that doesn't have a name yet, the buffer will inherit the name of this file. Its path name will also be remembered for later writing. If the save command is issued (see below) the editor will try to overwrite the old file (usually after making a backup). More details follow in the next section.

When a file is read, all occurrences of a carriage return or a linefeed will be seen as the end of a line. If a carriage return is encountered, the editor checks

whether a linefeed follows after it. If this is the case, the linefeed is skipped. If a linefeed is encountered, the editor checks whether a carriage return comes after it. If this is the case, the carriage return is skipped. The result is that <cr>, <lf>, <cr><lf> and <lf><cr> are all interpreted as a single end of line. This avoids problems with the various conventions that exist. In addition there is the raw mode in which neither <cr> nor <lf> are interpreted. They are put in the text like all the other characters. For more information about this mode, one should read the information about it in the next section.

Lines in STEDI should never contain more than 255 characters. Thus in order not to lose any part of a file being read in, any line that contains more than 255 characters is split up into one or more lines during the reading in process. The user will be notified that this may be about to happen and will be asked for his permission. If this permission is denied the reading will be stopped at the offending position.

If there is not enough memory to read in a complete file that has been specified, STEDI will read as much as will fit in the memory and then issue an error message. The only limitation on the size of a file being read in is the amount of memory available. Hence in marginal cases a file may still be edited if you are able to gain more memory by removing some utilities or making a ram disk a little smaller. If you are faced with such a big file that it cannot be edited in one piece, one may start with editing the part that could be read, write it to a file with a different name (!), clear the buffers and then read in a part of the file after skipping a number of bytes. When writing parts of a file they can be pasted together as can be seen in the > option of the write command.

5.3 Writing out a file

STEDI has several commands for saving the results of an editing session to a file. These commands have counterparts respectively using the function keys, the command line commands and the mouse. Moreover there is a number of options to be considered which affect the saving of the file. The commands and the associated options will be discussed below. All the commands are operational only when the cursor is in the text field.

The function keys associated with output commands are as follows.

shift-F8

Write file. Pressing the shift-F8 key combination yields a prompt on the message line. One can type in the name of a file followed by a carriage return. This results in saving the file in the current buffer under the

name typed in. This write command cannot be saved in the history buffers of the command line. Pressing shift-F8 invokes the use of a very special buffer that is always available, even when all other commands in the editor start complaining about a lack of memory. Thus the user can always write his results to file.

F9

Save file. The contents of a buffer can be given a name that is known by the editor if the file being edited was read into the buffer by name originally, or if a write command has been issued which assigned the file a name. (This is the name displayed on the right hand side of the message bar - including the path name associated with it which can be displayed with the Alt-L command.) If the current buffer already has a name for its contents, the F9 command causes the contents of the buffer to be written to a file of that name. If no name exists yet - the last 12 characters of the message bar are empty - the editor will report that it cannot save a buffer without a name.

F10

Save and quit. The first stage of this command is the same as the F9 command, that is, it saves the contents of the current buffer. This is followed by an exit command which is equivalent to shift-F10 (see the chapter on Running STEDI p. 171).

With the command line, the following commands are available:

W name or W'name

This write statement is equivalent to the shift-F8 command except for that now the regular procedures for the use of commands in the command line are followed. This means that this command will be entered in the command history. If the requested file exists already the user is asked to confirm that he wants to overwrite this file. This is to protect the user against loss of files, as it happens occasionally that one uses the write command instead of the read command.

W name > or W'name >

This is a write statement that appends the output to the named file, rather than creating a new file. Because the original file still exists as the head part of the new file no backup is made.

WF
: This induces the file selector (if a mouse exists in the system) to allow the user to select a name for the file to be written.

S or Save
: Save file. This is the same as F9.

Q or Quit
: Quit. This command is the same as the shift-F10 command (see the chapter on Running STEDI p. 171).

SQ or SaveQuit
: Save and Quit. This command is the same as F10.

The corresponding commands with the mouse are:

W
: Write file. If the system has no file selector this command is identical to the shift-F8 command. The user will be prompted in the command line for a file name. If there is a mouse and the local system provides a file selector the file selector will be displayed. Using the rules of this file selector, the user can select a directory/folder and a file name, after which a click on the OK box will cause the the file to be written under the specified name. The rules governing this command are similar to those for the shift-F8 command. If the requested file exists already the user is asked to confirm that he wants to overwrite this file. This is to protect the user against loss of files, as it happens occasionally that one uses the write command instead of the read command.

S
: Save file. Clicking on the S in the mouse menu is equivalent to pressing the F9 key. It saves the current buffer.

When a write command is issued, there are several messages that could indicate that something prevented the writing of the file. There could be several possible causes: a lack of disk space, a name was selected that belonged to a directory, the file that existed by that name could not be renamed to a .bak file (backup version) or the backup version that existed already could not be removed. These last two causes could have to do with a lack of rights to do anything with these files.

5.3.1 The write mode

The settings that are relevant for writing out a file are in two categories. The first governs the conventions used to write out the file, and the second governs the way a previous file of the same name is dealt with in terms of backups. Finally there is a view-only setting which doesn't allow the writing of a file at all. The various settings are explained below.

The current write mode is indicated by a letter A,U,R or P in the second position of the status characters in the status bar:

A - MS-DOS and Atari mode.
This mode can be selected with **Alt-A** or by toggling the second status character with a mouse button. In this mode each line of output is terminated by both a carriage return and a linefeed. This is the most widely used convention and also the convention officially used by MS-DOS and Atari.

U - Unix mode.
This mode can be selected with **Alt-U** or by toggling the second status character with a mouse button. In this mode each line is terminated by only a linefeed. This shortens the length of the file by one character per line which can be useful. All Unix systems use this convention and most C programs love it. If you have a system that is used to the A-mode many programs might make a mess of your file though.

R - Raw mode.
This mode can be selected with **Alt-R** or by toggling the second status character with a mouse button. In this case the lines are written without any linefeed or carriage return termination characters at all. Hence the lines will be run together and unrecognizable thereafter as separate lines. When reading in a file in this mode, the linefeeds and carriage returns are not interpreted but simply treated as other characters. Hence this mode can be used for reading in a binary file for limited editing and the file can be saved without the introduction of any extraneous characters (see also the chapter on hex code p. 144). Since writing in this mode could cause loss of information which may be time-consuming to recover, whenever an attempt is made to write in this mode, STEDI asks for confirmation before proceeding.

P - Printer mode.
This mode can be selected with **Alt-P** or by toggling the second status

character with a mouse button. Not only is each line terminated by both a carriage return and a linefeed, but the tabs are also expanded into the appropriate number of blank spaces. This is useful for sending a file with special tab settings to a printer on a different system. Its most frequent use is however to avoid problems with compilers that don't like tabs. Note however that expanding the tabs can make a file substantially longer.

The writing mode is one of the variables that is kept in the default file. It is also one of the variables that can be set for each buffer independently.

5.3.2 The backup mode.

Usually it is safest to have STEDI keep the original of a file as a backup version when writing a buffer to file. This may not always be possible due to diskspace limitations, so several options are offered. The indicator showing which option is set is the seventh character of the status bar and it is either a 'B', a 'b', or either a !, a !! or a †:

B - Normal backup.
If this option is selected, the original version of the file is first moved to a file with the same name but now with the extension .bak before the writing takes place. This .bak file is a normal visible file. This is the default mode.

b - Hidden backup.
If this option is selected, the original version of the file is first moved to a file with the same name but with the extension .bak as in the previous case, before the writing takes place. The difference is that this .bak file is a hidden file so it will not be seen in normal listings of the contents of the disk. The advantage of this setting is apparent when running the editor from a shell program in which the invisible files can be seen or hidden at will, so that the screen needn't be unnecessarily cluttered with files. In some environments however, the invisible files are usually really hidden and can only be removed or copied when the action is taken on the whole directory which contains them.

! or !! or † - No backup.
This is a rather dangerous mode, as the old version of the file is removed before the new version is written. This can result in loss of data if anything goes wrong (like badly malfunctioning floppy disks). The advantage

of this mode comes when there is not enough diskspace for both the file and its backup together. Caution is recommended when using this mode.

The backup mode can only be toggled from the keyboard with the **Alt-B** key combination. It is also stored in the default file. All that has been said above concerning backups is invalid when the 'View-Only' mode has been activated:

5.3.3 The View-Only mode

When experimenting with a file, or when reading a documentation file, it is best not to be able to write to that file. Most people use the F10 key (save and exit) when leaving the editor. With this habit, quick accidents can happen without warning. The View-Only mode, which is indicated by a 'V' as the seventh status character, is included to help prevent against such loss of data. In this mode the buffer cannot be written to a file. If a write is attempted, an error message is given. This mode can be selected with **Alt-V**. It can be 'de-selected' - in case you change your mind - with **Ctrl-V**. In that case, the backup mode that is in the default buffer is selected as the new backup mode. For more information about the default buffer, see the chapter on STEDI's default settings p. 51.

5.3.4 Messages concerning file output

If the writing of a file is successful, the editor will return with a message indicating the number of lines written.

If the .bak version of a file is not a proper file (e.g. a read only file), no backup can be made and the editor returns one of the error messages : "Cannot move old file to backup.", "Cannot remove old backup." or "Cannot make a proper backup.". The write command will not be executed in this case.

If anything goes wrong during the writing of a file, the writing will be aborted with the message "Error while writing" or "Error while writing. Disk full?". This may indicate one of several things:

- There is no more space on the disk.
- There was a change of diskette during the writing (very dangerous).
- The drive is not connected properly.
- The disk is not formatted properly or contains bad sectors.
- ...?

If in doubt as to whether enough space on a disk exists, one can find out the amount of free disk space with the ? command (see the section on 'Free disk space' p. 49) and the length of the file to be written can be found with the = command. If there is indeed not enough disk space, one could use another diskette with enough space or (on mainframes) get your systems administrator to give you a larger file quota. If there are no more formatted diskettes, one should remove some files on a formatted diskette or format a new one (easy with a shell that can be used from the editor like MS-DOS, one of the UNIX shells or the GPSHELL (on Atari) - see Execute an external command p. 131). Another alternative (if you are not working with a shell) is to overwrite an unimportant existing file by writing an empty buffer to that file, and selecting the option 'No backup'. This effectively erases the file, replacing it with the new and empty one. It is anyway a wise policy to always keep at least one empty formatted disk on hand for such emergencies.

To do the writing quickly the editor needs a buffer of about 9 Kbytes. If this memory is not available the subsequent action will depend on the system. On some systems memory it is borrowed from the screen. A message will indicate that this is the case and a band of noise will appear on the screen. After the writing is finished, the screen will be restored. If such an emergency operation cannot be performed writing cannot take place and the user should try to remove some of the contents of the buffers before trying again. If there is a particularly lengthy block of key redefinitions (9 Kbytes is very much here!) one might try to remove the key redefinitions instead with the K0 command. This command can be looked up in the chapter on key redefinitions p. 165.

5.4 Printing

If the system allows the use of a printer, STEDI offers the possibility of sending the file in the current buffer directly to the printer in one of two modes. The **P** command line command prints the current buffer just as it appears on the screen in the editor, and the **PF** command sends the file to the printer with limited formatting capabilities. The **PP** command sets the length of a page for the latter. These commands are explained below. In addition a port selection can be made if more than one printer port exists. This is done with the 'P=<port>' command.

P

Print the contents of the buffer. These contents are sent to the printer just as they appear on the screen. Thus folds that are closed are printed as closed folds and tabs are expanded to the number of spaces they currently

represent according to the editor's settings. Otherwise all characters are sent as they are. The end of a line is indicated by a carriage return + line-feed which will be interpreted correctly by most printers. Otherwise all characters are sent directly to the printer uninterpreted. Hence if the file contains non-standard ASCII characters, these may have consequences for the printer settings. (You may also put such characters in the text deliberately for printer control - see the chapter on 'Hex mode' p. 144.)

PF

As with the P command, this command causes the contents of the buffer to be printed, but with the added feature that a limited number of lines per page are printed with a page number at the bottom of each page and an optional left margin offset. The length of a page and the left margin offset can be set with the PP command. Note that the length of the page should include one empty line and a line for the page number. After each page number has been sent to the printer, the editor sends a formfeed to skip to the next page.

PPn1,n2

This command sets the number of lines per page (n1) and optionally it sets a left margin offset (n2). This latter number causes the printer to create a left margin by printing that number of spaces in each line before starting to print the contents of the buffer. These two numbers are stored in the default buffer so when the DW command is issued they are written to the default file for future use. The number of lines per page set by the PP command must be two more than the number of lines of text actually desired to allow for a blank line before the page number and then a line for the page number itself. As all printers are not alike, some experimenting may be required in order to find the best setting for a given printer. The default number of lines per page is 60 and the default left margin offset is set to zero. If the first number is less than five, STEDI sets the number of lines to five. If both numbers are omitted entirely, the PP command causes a message to be printed in the message bar telling the current number of lines per printer page and the current left margin offset.

P=

Some computers have more than one printer port. For instance the Atari ST has two printer ports: a serial port with the name 'AUX:' and a parallel port with the name 'PRN:'. The PRN: port is the default port but some printers prefer the other port. This can be selected with the

command P=PRN: while the command 'P=' will tell the user which port has been selected. The name of the port will be stored in the default file when the DW command is executed. On most computers this command won't have any effect.

Examples:

PP66
sets the number of lines per printer page to 66

PP60,5
sets the number of lines per printer page to 60, with a left margin of 5 spaces.

PP
queries for the current number of printer pages and the right margin offset.

5.4.1 Messages concerning printing

A message 'Printer not available' indicates that either there is no printing capability, or the printer is not connected or the printer has not been turned on.

If the printer is turned off during printing, or when it runs out of paper one may get the message "Printer not ready. Continue ? (Y/N)". This will enable you to add paper or turn the printer back on again before telling STEDI to continue. The answer n or N will abort the printing.

Note that there is no print command that can be accessed with the mouse.

5.5 Free disk space

In some instances, you may want to see how much space exists on a disk before writing a program to it. The ? command exists for this purpose. This command is entered from the command line. The syntax is:

```
?drive-specification
```

The 'drive-specification' is a single letter between A and P indicating one of the allowed drives or partitions. The letter may be upper case or lower case. If the indicated drive is properly connected to the computer, STEDI will respond with the number of bytes that are still available for writing on this drive. On systems

where partitions may be indicated with names of more than one character one should of course specify the entire name. On UNIX systems this command is not available. One should instead use the appropriate ! command (like '!quota -v' or '!df drive' in the cshell).
Example:

```
?a or ?A
```

This command gives the number of free bytes on the diskette in drive A.

The ? command is useful when it is not clear whether a file will fit on a diskette. In that case one can find the size of the file to be written with the = command and compare this with the space on the diskette.

If the answer to a request for the amount of free diskspace is the message "Drive not connected." there is no record in the systems variables that the requested drive exists.

Default settings

The editor contains a number of internal variables such as the screen color, the backup mode, the write mode, the search mode, tab settings, and so forth, as well as possibly such things as contents of learn buffers, macro's and keyboard redefinitions. In order to save all these settings for future use, STEDI allows them to be written away to a file which can be read in at some later time. The file, called STEDI.DFT is also read automatically at startup if STEDI can find it.

Actually the situation with the default settings is not quite so simple as explained above because of STEDI's ten text buffers. Certain settings are global, that is, they affect all buffers (for example, the color, learn buffers, and key redefinitions), but others can be set individually for each of the ten buffers. For example, the backup mode can be set differently for each file in one of the ten text buffers. These latter settings will be called local.

STEDI maintains a central defaults buffer which is filled at startup with the contents of STEDI.DFT or set by built in defaults if this file is not available. All global and local settings are kept in this buffer. Then each time a text buffer is opened, a copy of the local settings which are in the central buffer is copied to the text buffer to begin the editing in that buffer. After this, the user is free to change the settings as desired and the settings for each individual buffer will be remembered until the buffer is cleared, or until the end of the editing session. If a buffer is cleared, the settings will revert back to those in the central buffer.

Naturally for greatest flexibility, STEDI requires a number of commands for input and output of these settings and for moving them around from one buffer to another if desired. These commands are the subject of this chapter. Below, the commands will be explained in a logical order, building up from the more basic operations. However it should be realized that the first commands mentioned are not necessarily the most used. In fact, one may get by, using only one or two of the commands explained in this chapter. However they are all included for completeness. All these commands are issued from the command line.

The first commands are for reading and writing from a file to the central buffer and vice versa. These commands are:

DI

This command reads the settings from the file called 'STEDI.DFT' and

places the contents in the central defaults buffer. This command should be remembered as 'Defaults Import'.

DE

This command is used to write the contents of the central defaults buffer to a file called 'STEDI.DFT'. If the file exists, it is overwritten. This command should be remembered as 'Defaults Export'.

In order to copy the defaults which are stored in the central buffer into the default settings of one of the ten text buffers, the following commands are available.

DL

This command loads the settings for local variables from the central defaults buffer into the current buffer. This should be remembered as 'Defaults Load'.

DS

This command copies the local settings of the current buffer into the central defaults buffer. Consequently the settings for local variables present there are replaced. This should be remembered as 'Defaults Save'.

Of course if only the above commands were available, loading the settings of local variables into individual buffers could become quite laborious. Much more often one wants a combination of two of the above commands. Single commands accomplishing this are therefore available. These commands are:

DR

This command is a combination of the commands DI and DL above. This command thus reads in a default file called 'STEDI.DFT' and fills the central defaults buffer with it. Then the settings for local variables are copied to the current buffer. It can be remembered as 'Defaults Read'.

DW

This command is by far the most useful of all the commands described so far in this chapter. It is a combination of the DS and the DE commands. Thus it copies the local settings of the current buffer into the central defaults buffer, and then writes the entire contents of the central buffer out to a file called 'STEDI.DFT'. This command is generally used when you have a set of current settings, both global and local, and you want to write them out to disk for a later editing session. The command can be remembered as 'Defaults Write'.

It is useful to be able to specify a default directory or folder (other than than the current one which is searched automatically) from which STEDI can read the default file STEDI.DFT. The DD (Default Directory) command serves this purpose. The default directory is also used for writing out the defaults. The associated commands are:

DD pathname

This command sets a default path name to define a directory in which STEDI will look for the default file STEDI.DFT, other than the current directory. This directory, called the 'default directory', is used whenever STEDI.DFT is read or written, i.e. with the commands DI, DE, DR and DW. In the case of read operations, the editor first looks for the default file in the default directory, and then looks in the current directory. If the file is not found after this, an error message is issued. For write operations, STEDI will try to write the default file in the currently set defaults directory first. If this directory is not found, the file will be written in the current directory. Finally the default directory can also be set at startup (see below).

DD

When the DD command is given without an argument, the name of the currently set default directory is reported on the message bar.

All commands mentioned so far in this chapter are case insensitive. Thus for the DW command, you may equally well type dw, dW or Dw.

During startup, you may pass a parameter to STEDI to specify a default directory or a default file for reading the startup defaults. This is done by giving as a first parameter the characters -d or -D followed by one or more blanks and then a name. The name is interpreted as a path name and this will be the default directory. At the same time this directory will be searched when the help file is needed, or when a macro should be loaded from disk. After this come the file names to be edited separated by one or more blanks. For example suppose the startup defaults are in a directory called 'BIN' on disk D and the file BIG.BUG in the current directory is desired to be edited. Then if starting from a shell, you would issue the command

```
stedi -d d:\bin big.bug
```

The other way to specify the default directory during startup (and also the preferred one) is by setting the environment variable STEDIDFT. If this variable can be located during startup its contents are interpreted as the name of the default directory. A trailing directory separator isn't needed.

The order in which directories are searched for the default file is different at startup than during the DW command. At startup (p. 167) first the current directory is searched, independent of a -d parameter or an environment parameter. If this isn't successful the path given by the -d option is tried. If the -d option hasn't been used the environment is tried.

Finally we close this chapter with a list of those settings which are stored in the default file.

The global variables which affect all text buffers are:

- The version number of the default file.
- The global screen color (see p. 135).
- The local screen color (see p. 135).
- The offset for printing files (see p. 47).
- The number of lines per page for the printer (see p. 47).
- The complete learn buffers (see p. 108).
- The delay factor when scrolling the screen (pagedelay) (see p. 135).
- The key redefinitions
- The loaded macro's
- The choice of the printer port (see p. 47).
- The wait flag for waiting after a call to shell (waitflag) (see p. 132) .
- The flag that determines whether line numbers will be displayed in the message line (numbers) (see p. 135).
- The mouse menu bar (see p. 89).
- The versionnumber of the menu bar (allows the user to define his own menus and submenus) (p. 92).
- The value of the variable 'menuspaced' which determines whether only every other position in the menu bar is sensitive to mouse clicks (see p. 92).
- The flag for writing folds with information about whether they are open or closed at the time of writing. This flag determines also whether at reading folds are closed (autoclose).
- (MS-DOS and Atari only): A flag (BIOS) that can force the screen output to be written by means of the BIOS routines (see p. 135).
- The table with the information about which characters can belong to words (see p. 97).
- The maximum number of entries in the command line history (maxhist).

The local variables which can be changed locally for various text buffers are:

- The Insert/Overstrike mode (I/O)
- The output mode (A/U/R/P)

- The search replace direction (</≪/0 or ∞/≫/>)
- The case sensitive/non-sensitive mode (N/S)
- The tab settings
- The word wrap settings
- The auto-indent mode
- The backup/view-only mode (B/b/†/V)

The mark

A mark can be placed in the text of a file being edited for later use. The mark can be used to remember a place in the text or it may be used in its role of defining a 'marked range' (or 'range of action'). A marked range is defined as the range of text between the cursor and the mark. This range is used for cutting and copying operations and optionally with certain other commands such as search and replace. Each buffer has its own mark.

There are several commands involving the mark directly:

F1 - Put mark.
This command places the mark at the current position of the cursor. If the mark was at a different location before this command is issued, it is removed from that location and all memory of it will be lost.

shift-F1 - Remove the mark.
(Not available on systems without a help key. There shift F1 invokes the help facility). This is a rarely used command that exists mainly for security reasons. As some commands cannot be executed without the existence of a mark removing the mark may avoid inadvertent use of these commands.

F2 - Exchange the position of the cursor and the mark.
The region between the cursor and the mark is called the marked range or range of action. Some commands have options that will let them operate on this range of action only. The order of the cursor and the mark in the file is unimportant. Some editors will highlight such a range of action. In STEDI however the mark is also used as a means to remember a position and it would be very annoying to highlight a quasi random piece of text. The exchange of the positions of the cursor and the mark allows for a quick inspection of the exact boundaries of the marked range.

shift-F2 - Go to mark.
(Not available on systems without an undo key. There shift F2 invokes the undo facility). This command moves the cursor to the position of the mark. With this command the user can go back quickly to a previously defined position in the current buffer.

Three of these commands can also be given with the **mouse**. The placement of a mark can be accomplished by clicking on the **M** in the mouse menu (equivalent to F1). Clicking either button on the **G** in the mouse menu makes the cursor go to the position of the mark (equivalent to shift-F2) while clicking on the **E** causes the exchange of the cursor and the mark (equivalent to F2).

Here is a list of the commands involving the mark in its role of defining the marked range.

- Cut or Yank
- Copy
- Case conversions and
- the R or 'range' option in the following commands
  - Search
  - Replace
  - Tab
  - Detab
  - Trim
  - Sort

If no mark has been set, these commands will abort with an error message.

Several commands may erase the mark, for the reason that during the execution of the command, the position of the mark may become dubious. A good example is the replacement of a string that contains the mark in it. Also a cut operation will erase the mark. Technically this is not necessary, but it happens frequently that after the cut has been made, the copy key is pressed inadvertently, rather than the paste key. This would result in the loss of the contents of the paste buffer. Without a mark, nothing will happen.

The paste operation places the cursor at the beginning of the pasted region and puts the mark at its end, so a paste operation can be undone by pressing the cut key, as long as the cursor hasn't been moved yet.

7.1 Tags

The mark is used very often as a means to go back to a previously stored position. For this use it is often a restriction that there is only a single mark per buffer. Therefore 10 tags have been introduced. Tags are somewhat loosely organized marks that are not bound to a specific buffer. When a tag is placed it is bound to the current buffer and the current column in the current line. It is then possible to go back to that position from any buffer with either the corresponding Ctrl-Function key (shift-ctrl-# key on the Atari-ST) combination

or the ># command in the command line (# stands for a single digit of the normal keyboard). The tag is placed with the <# command.

Once a tag is placed it will survive anything the editor may do internally (like garbage collections), but when the user changes the line to which the tag is bound this may make it unbound and pointing to a fictitious address. If the user wants to go to this tag afterwards there are two possibilities: Either the tag points now to a different line in the same buffer as the original one (this could be caused by garbage collections moving another line in place) in which case this is considered to be the tag position, or the above is not the case and the message 'tag not found' will be displayed. This is the price to be paid for a significantly smoother performance than when the tags would be arranged like the marks. In practice the above 'erroneous jumping' occurs very rarely, although this may depend on the habits of the user.

When a tag is hidden inside a closed fold STEDI will also not be able to find the tag and go to its position so it will also report 'tag not found'.

Buffers

STEDI is equipped with **8 buffers** for normal editing. Under ordinary circumstances this should be more than enough. In addition there are **two special buffers** for cut and paste operations. They are called the **yank buffers** and are used for intermediate storage of pieces of text that have been cut or copied from the text of one of the other buffers. The yank buffers can also be used as normal buffers when necessary. They are then labeled YANK buffer (buffer-9) and yank buffer (buffer-0) respectively. There are however some caveats. During a cut or copy operation, the yank buffer that will be used is cleared first, so anything that might be in it before the operation is lost. Which yank buffer will be used for a cut or copy operation can be seen in the status line. The fifth status character is either a **Y** or a **y** indicating the YANK or the yank buffer respectively. The use of the **Alt-Y** key combination toggles between these two yank buffers to select the one to be used for the next cut or paste operation.

The buffer that is currently displayed on the screen is called the current or active buffer. The number of the current buffer is displayed in the status bar in the eighth position among the status characters. There are three different ways to change from one buffer to another.

8.1 Switching between buffers

First one can press the **Alternate key and one of the number keys** on the main key pad. This selects the buffer with the corresponding number as the current buffer. When you switch from one buffer to another, if the new buffer has previously been used, it will be displayed on the screen exactly as it was left when last viewed. That is to say that the line numbers displayed will be exactly the same and the position of the cursor will be in the same position as before. The change buffer command can also be executed via the command line as is the case with all key combinations involving the alternate key. In that case one should enter the command line by pressing <Esc> and type the command Alt-# in which the # represents a single digit and a <return>. The digit indicates then which buffer is selected. This command is particularly useful for working with macro's and stream editing. It can also be typed as 'alt-4' if one likes to go to buffer 4.

The second way to switch between buffers is with the mouse. Pointing the

mouse cursor to a number in the mouse menu and clicking the left mouse button is completely identical to pressing the Alternate and number key combination with the corresponding number. With a split screen it suffices to click any mouse button in the window of ones choice to activate the window and make it into the top window.

Finally a third way to switch between buffers is by the name of the buffer via the 'quote' command of the command line. To give this command, first press <Esc> to enter the command line. Then type a single quote followed by some of the first few characters of the name of the desired buffer. The editor will search forward from the current buffer for a buffer whose name begins with the same characters entered after the quote. The search is circularly forward and when no match occurs, no change of buffer is made. Depending on the setting of the fourth status character (S/N for sensitive or nonsensitive) this buffer search is either case sensitive or case nonsensitive. It is not necessary to type a closing quote after the string in this search command.

In addition to the above ways there are two other commands that can result in a flipping between buffers. The first involves tags. A tag is a mark that is buffer independent. If a tag has been placed in one buffer and the user goes to another buffer the action to find the placed tag will result in a tranfer back to the original buffer. For more about tags one should consult the section on tags p. 57. The other way involves the use of Alt-F when the screen is split in two regions. The Alt-F flips the cursor between the two different buffers that are displayed. Also clicking the mouse in one of the two 'windows' will move the cursor there. There is more about this in the chapter on screen control p. 135.

The **name** of the file being edited in a buffer usually determines the name of the buffer. When the editor is entered with a command tail that contains one or more file names the corresponding buffers in which these files are placed inherit the names of these files. Even if these files don't exist yet, the buffers will be named accordingly and STEDI will assume that they are new files to be created. If no files are specified or some buffers are not yet filled at startup then they are not yet given a name. Whenever a buffer has no name, the first read operation to this buffer or the first write operation from this buffer will determine its name. When a buffer is cleared, its name is also removed.

The current buffer can be cleared with the shift-F9 key. If the work in the buffer has been altered and the **'view only'** flag is not active, there will be a warning that the buffer is not saved yet and the clear operation will have to be confirmed. The ninth status character in the status bar indicates whether or not a file has been altered. If this character is a blank, then no changes have been made and if it is a little circle, the file has been changed since last saved.

If the **'view only'** flag is set, the seventh status character in the status bar

will be a 'V'. A buffer will be given this status when the first file that is read into it is a file that is marked read-only. The user can also switch to a 'view only' status at any moment with the **Alt-V** key command. To switch from this status to the normal mode so that the contents of the buffer can be written to disk, one may use the **Ctrl-V** command.

Each buffer has a number of settings assigned to it. These settings may therefore be different for each buffer. When one of these settings is changed in one buffer, this doesn't affect the others. When a buffer is cleared, its settings are lost and new settings are taken from the default buffer (see the chapter on default settings). The buffer dependent settings are :

- Insert/Overstrike mode as indicated by the first status character.
- The write mode as indicated by the second status character.
- The search and replace modes as indicated by the third and fourth status characters.
- The 'View only' and the backup modes as indicated by the seventh status character.
- The word wrap mode.
- The auto indent mode.
- The tab settings.

Cutting and Pasting

The editor is equipped with two buffers that can be used to move pieces of text. These buffers are called the yank buffers. They are the buffers 9 and 10 of the file buffers. Buffer 9 is called the 'YANK buffer' and buffer 10 (or 0) is called the 'yank buffer'. The cut (also called yank) operation moves a piece of text from the currently active file buffer to the current yank buffer. This text is removed completely from the current file buffer. Which yank buffer is current can be seen in the status line. The fifth status character is either a 'Y' or a 'y' for YANK and yank respectively. If it is necessary to make the other yank buffer the current one in use, one can toggle between the two with **Alt-Y** or by clicking the mouse on this status character. The first 8 buffers can have either yank buffer as their active yank buffer. As the yank buffers themselves can also be edited, one of these cannot be its own current yank buffer.

The copy operation copies a piece of text from the currently active file buffer to the current yank buffer. The paste operation copies the text in the current paste buffer to the position of the cursor in the currently active file buffer.

The first action taken by the editor when a cut or copy operation is given is to clear the yank buffer that is going to be used. Therefore it is rather dangerous to use a yank buffer for ordinary editing. Next the 'range of action' is determined. This is the text between the mark and the cursor. The order of these is not important. Finally, for the copy operation the text in the range of action is then copied into the yank buffer. For the cut operation, the text in the range of action is also removed from the current buffer.

The paste command leaves the contents of the yank buffer unchanged so that it can be used again.

There are also some special cut, copy and paste commands for columns of text. The **copy columns** (or 'block copy') command copies the text in the range of action, but only that which falls in the columns between those of the mark and the cursor (order unimportant) If this range involves empty spaces, they are filled with blanks. The **cut columns** (or 'block cut') command also removes those columns from the current buffer. The **paste columns** (or 'block paste') command copies the contents of each line of the yank buffer into successive lines of the text of the current buffer, starting in each line in the column of the cursor. These operations can be convenient for moving tables or parts of a program for which you wish to maintain a certain column relation between lines. They can also be very handy to change the indentation of a

range of text.

The cut, copy and paste commands can be performed with the function keys as follows.

F3	Cut (yank) the current range.
F4	Copy the current range.
F5	Paste to the cursor position.
sh-F3	Cut columns (block cut) from current range.
sh-F4	Copy columns (block copy) from current range.
sh-F5	Paste columns (block paste) to cursor position.

These commands can also be performed with the mouse:

- Click with the left button on the Y of the mouse menu: Cut (yank) the current range.
- Click with the left button on the C of the mouse menu: Copy the current range.
- Click with the left button on the P of the mouse menu: Paste to the cursor position.
- Click with the right button on the Y of the mouse menu: Cut columns (or block cut) from the current range.
- Click with the right button on the C of the mouse menu: Copy columns (or block copy) from the current range.
- Click with the right button on the P of the mouse menu: Paste columns (or block paste) to the cursor position.

In addition all these commands can also be given from the command line. This feature is mainly useful for writing readable macro's. The command is given as a word in the command line or in the macro and the magic words are 'cut', 'copy', 'paste', 'bcut', 'bcopy' and 'bpaste'. The last three are for the cut, copy and paste in the block mode.

To see (or edit) the text in the yank buffers, one switches to these buffers with the Alt-9 or Alt-0 key combinations. This is explained further in the chapter on buffers.

Folds

STEDI is equipped with a powerful feature which helps in the organization of a program or text file by allowing the programmer to segment a file into a number of smaller units. Then the smaller units can be selectively viewed or suppressed as desired for editing purposes. This feature is called '**folds**'.

Briefly, a fold is created by entering two extra lines in the text which define the beginning and the end of the segment desired to be placed 'in the fold'. Then the fold can be 'closed' which means that the whole segment of text between these two special lines together with the two lines is replaced on the screen by a single line of text which is used to represent the entire segment. Subsequently, such commands as search and replace will not affect the text hidden in a closed fold. Thus, for example, a programmer could put various subroutines in different folds and then open only the one desired to be edited at any given time. A number of commands exist for opening and closing folds to provide quite a flexibility in using this feature. A full description of the folds feature follows.

10.1 Fold line syntax

The beginning line of a fold must have the following syntax: the first three characters are arbitrary and are followed by a 'number sign' (#) as the 4-th character and an opening square bracket (`[`) as the fifth. After this comes the label, which may consist of any characters with the exception of carriage returns, linefeeds and colons. A colon (:) is used to terminate the label field and must be present! After the colon, arbitrary characters are allowed as in normal text. The ending line should look exactly the same as the beginning line in the range between the number sign (#) and the colon (:) that terminates the name field except that the fifth character (`[`) should be replaced by a closing square bracket(`]`). The characters outside this field (the first three or those following the colon) need not be the same. When a fold has been closed the line that represents it is a copy of the opening fold line (or beginning fold line) but it has a second 'number sign' where the opening fold line has its square bracket. The closed fold line may not be changed under any condition. If you don't like it you have to open the fold first and change at least the opening fold line.

With this syntax, the first three characters can be used to set off the line as commentary for a compiler, and the characters after the colon can be used

as commentary for the programmer. Note also that tabs count as single characters so that when used in the first three characters of the line, a measure of indentation can be achieved.

Folds may be nested to any depth. The only restriction lies in a potential stack overflow, because some operations for nested folds work via a recursion. Typically at least 50 levels should be possible. Beyond that the reaction of STEDI may depend on the computer on which it runs. If the stack is not protected agains overflow a crash may result. In practice the user will rarely go beyond 4 or 5 levels.

10.2 Opening and closing folds

10.2.1 Function key commands

A fold is closed with the 'close current fold' command which closes the deepest fold (nesting) in which the cursor currently resides. This command is given with the F6 key. The whole fold then becomes represented by a single line which is created upon issuing of the 'close fold' command. This line looks like the first line of the fold with one exception: the opening square bracket (`[`) is replaced by a second number sign (#). Internally this line is quite different from normal lines, as it has to keep track of where the lines are that are hidden inside the fold. Therefore this line may not be changed! The only thing one is allowed to do with such a line besides opening the fold again is to cut and paste text which contains the whole line. In this way, whole blocks of text can be moved around quite easily as closed folds.

A fold is opened by putting the cursor in the fold line (the line representing the whole fold) and issuing the 'open fold' command F7. In opened state all lines of a fold are are normal lines. Hence there is no restriction on the alteration of the beginning and ending fold lines when the fold is open, so care must be taken to maintain the proper syntax for the folding mechanism to work properly.

To close all folds of a file you may issue the command Shift-F6 and to open all folds the command Shift-F7 is to be used.

10.2.2 Command line commands

All fold commands can also be entered from the command line. They start with a closing square bracket (`]`) for close fold commands, or an opening square bracket (`[`) for open fold commands. Just a `]` is the same as F6, and a `[` is the same as F7. To open and close all folds, the commands are:

]a
: Closes all currently opened folds. This command is the same as shift-F6.

[a
: Opens all closed folds. This command is the same as shift-F7.

In addition, folds can be opened and closed from the command line by name or line number. These commands have the following syntax:

]'name'
: This command closes the first fold found with the name or label which is given between the quotes. The search for the fold is always started at the beginning of the file. This gives usually the best interpretation of nested folds with the same name. Note that all characters between `#[` and : are relevant so if a fold line contains the string `#[ NAME :`, this fold must be closed with the command `]'` NAME ' in which the blanks are relevant. Note also that there are no escapes in this search string, so it is very unwise to use quotes in the name of a fold. Case sensitivity in the search for this fold is determined by the same setting as for normal searches (See the S/N status character).

['name'
: This command opens the fold with the given name. No recursive searches are done so STEDI won't look inside closed folds.

]#
: `#` stands for a number. This command moves the cursor to the line indicated by the number given and closes its current fold.

[#
: `#` stands for a number. This command moves the cursor to the line indicated by the number given. On its way all folds it has to enter to find this line are opened.

10.2.3 Mouse commands

The mouse can also be used for opening and closing folds. If the right mouse button is clicked while the mouse pointer is pointing anywhere in the text field of a fold (with the exception of it pointing to the name of a closed fold), then the fold is closed. This is the same as the F6 command. If the mouse is pointing to the name of a closed fold, that fold is opened. This is the same as the F7

command. However there is another restriction on these actions. As clicking the mouse on the edge of the screen is generally used for scrolling the screen, the scrolling will override the opening and closing of folds.

10.3 Miscellaneous

When a file is written there are two possible options concerning the folds. It can be written in such a way that at the next reading of the file STEDI has some memory about which folds were closed, or this information can be omitted. The feature of this 'memory' is called the autoclose feature. It is toggled with the command 'set autoclose = on' or 'set autoclose = off'. The value of autoclose can be stored in the default file with the DW command (p. 52). The information of whether a fold was closed at the time of writing is stored as a trailing blank in the closing fold line, so no compiler should have any problems with it. Printing a file (P command) sends the file in the same representation as displayed on the screen: what you see is what you get.

Examples:

```
/* #[ multtwo:  Multiply an integer by two. */

int
multtwo(n)
int n;
{
n = n * 2;
return(n);
}

/* #[ multtwo:    */
```

This example puts a C language subroutine called 'multtwo' in a fold with the same name. Note that the freedom in the formatting of the beginning and ending fold lines is used to make the fold line commentary to the C compiler. If the fold were to be closed, it would look as follows:

```
/* ## multtwo:  Multiply an integer by two. */
```

The next example is for text files:

```
---#[ Fold1:
```

```
Chapter 1   All about folds

--+#[ Fold1.1:   Introduction

Folds are wonderful.
They may revolutionize your life.

   #] Fold1.1:
--+#[ Fold1.2:   Commands

Folds with the command line, etc.

   #] Fold1.2:
--+#[ Fold1.3:   Mouse

Folds work with the mouse, too! And etc.

   #] Fold1.3:

   #] Fold1:
```

In this example which simulates how a manual writer might organize his sections, there is one outer fold which contains three inner folds. The label of the fold has been used to provide outline numbering. If the command shift-F6 (close all) is issued, the above would look like:

```
---## Fold1:
```

Now if Fold1 is opened, the inner folds will remain closed and appear as follows:

```
---#[ Fold1:

Chapter 1   All about folds

--+## Fold1.1:   Introduction
--+## Fold1.2:   Commands
--+## Fold1.3:   Mouse

   #] Fold1:
```

For a further and more extensive example, the source code of the program 'keycomp.ttp' has been included. This program, which is responsible for compiling key redefinition files so that they can be read by the editor, is written in the C language. The program is fully commented and makes use of folds for organization. This program can be found in the directory called 'SRC' on your original disk.

Search and Replace

11.1 The search command

One of the central features of any editor is its search facility. Searching should be fast and versatile. Therefore much effort has been put into providing STEDI with a set of very fast search routines which utilize the rather modern Boyer and Moore search algorithm (see R. S. Boyer and J. S. Moore, A fast string searching algorithm, *Comm. ACM, 20,* 10 (Oct.-1977), 762-772). As a consequence, searching for a string of 6 characters is done at a rate of more than 400 Kbytes per second on a Motorola 68000 at 8 Mhz or a 80286 at 10 Mhz. This allows for moving through even very large files at an extremely fast pace.

Both the search command and the search and replace are issued from the command line. A slash (/) is the starting character for either of these commands.

The basic syntax for a search command is:

```
/string/
```

where 'string' is the string of characters desired to be searched for. The cursor will be positioned at the first character of the first occurrence of 'string' that is encountered. If no (further) occurrences are found, this will be announced in the message line. The trailing slash is optional. Experience shows that it is often forgotten and as searching isn't destructive (it doesn't alter the text) there is no reason to be very strict about syntax for this command. Note that the search operation doesn't look inside closed folds.

The search command can also take various optional parameters. If no options are specified, a default setting is taken. The full syntax of the search command is:

```
/string/options
```

where 'options' is a string of characters specifying the options desired. If a character in this string corresponds to one of the option characters listed below, a flag for that option is set. If two options contradict each other, the last one given over-rides the first. The possible options are:

N or n

The search is case **N**on-sensitive. This means that the case of the characters in the search string is not considered. This holds also for the

characters used in the national character sets, such as characters with accents, if both the upper case and the lower case versions of such a character are present in the standard character fonts for your computer.

S or s
The search will be case **S**ensitive. N and S are mutually exclusive: only the last one given counts. These options can be used to override the default set with the **Alt-N** or **Alt-S** commands.

R or r
The search will only be done in the **R**ange between the mark and the cursor. This enables the user to search (and replace) strings in a part of the file only. A match can only occur if the string searched for is fully included in this range between the mark and the cursor.

B or b
The search will only be done in the **B**lock of rows and columns between the mark and the cursor. With this option, searches (and replacements) can be performed in specified columns. A match can only occur if the string to be found is fully included inside the block defined by the mark and the cursor.

W or w
The string to be searched for is interpreted as a word. This means that a match can only occur if the string is encountered where it is both immediately preceded by and followed by a character that does not belong to words. Characters belonging to words are a-z, A-Z, 0-9, _ and the special alphabetic characters that may be present in the local default font that are used for the national character sets.

0
The search will be circularly forward. This means that the search will be started in the forward direction. If no match is encountered between the position where the cursor started and the end of the file, the search is continued starting at the beginning of the file. When the original position of the cursor is encountered, a message 'No match found.' is given. The original position of the cursor never counts as a match in searching (it does in search and replace though !).

$>$ or $\gg$
The search will be forward. If no match is found between the original

position of the cursor and the end of the file the message 'No match found.' will be given.

< or ≪
The search will be backward. If no match is found between the original position of the cursor and the beginning of the file the message 'No match found.' will be given.

. (a period)
If the search is is executed from a learn buffer a macro or by means of the I command (p. 128) and the search is unsuccessful no further execution will be attempted.

[**col1,col2**]
The search takes place only inside the specified range of columns (inclusive). If either number is omitted (the comma is relevant) it is set at its minimal or maximal value respectively.

Another useful search command, which is executed from the keyboard, is the 'matching brackets' search. If the cursor is placed on any type of parenthesis or bracket and **Alt-=** is pressed, the matching bracket will be found and the cursor moved there. This holds for normal parentheses ((or)), and both square ([or]) and curly ({ or }) brackets. If no match is found an error message is printed out on the message bar. This command can be extremely useful for finding unmatched sets of parentheses in a program.

11.2 The search and replace command

The syntax for a search and replace command is:

```
/string1/=/string2/options
```

This command causes STEDI to search for one or more occurrences of string1 and to replace each occurrence by string2 according to the settings of the options. As with the search command, the optional parameters are a string of characters of which the last character overrides previous characters in the case of conflicts. All options of the search command can be used with this command also. In addition, there are several more options for the replace operation:

V or v

This is the **V**eto option, allowing the user to veto a replacement. If this option is chosen, STEDI will make a replacement only after a confirmation. If string1 is found, STEDI will position the cursor at the first character of the string and ask '`Replace ? (G/Y/N/Q)` '. If the answer '`Y`' or '`y`' is given, the replacement will be made. If the answer '`N`' or '`n`' is given, the replacement will not be made. In either case STEDI will continue to search for the next match (unless > or < is one of the options). If the answer is '`Q`' or '`q`' the search and replace operation will be aborted and no more replacements will be made. The answer '`G`' or '`g`' indicates that the editor can *go* on now and make all further replacements.

>

With this option, searching is forward, but after the first match has occurred and the replacement has been made (unless vetoed via the V option) the command is terminated. This is called 'forward search, single replace.'

≫

With this option the searching is also forward, but replacements will be made until the end of the file is reached (unless vetoed via the V option). This option is called the 'forward search, multiple replace' option.

<

With this option, searching is backward, but after the first match has occurred and the replacement has been made (unless vetoed via the V option) the command is terminated. This option is called the 'backward search, single replace' option.

≪

With this option the searching is also backward, but replacements will be made until the begin of the file is reached (unless vetoed via the V option). This is called the 'backward search, multiple replace' option.

0

This option is a multiple replace option also, called the 'circular search, multiple replace option'. Using this option, all occurrences of string1 in the file will be replaced by string2 unless the veto option is used.

If an option is not specified its default setting is used. This default setting is recorded among the status characters which are found on the right side of

the status line. If an option is not represented among the status characters its default is off. These defaults are as follows.

The third status character can be toggled with **Alt-D** , **Alt-E** or either mouse button. The possible settings are:

$<$	Backward search, single replace.
$\ll$	Backward search, multiple replace.
0 or ∞	Circularly forward search, multiple replace.
$\gg$	Forward search, multiple replace.
$>$	Forward search, single replace.

The fourth status character has the following possibilities:

N Case non-sensitive. This option is selected with Alt-N or by toggling the status character with either mouse button.

S Case sensitive. This option is selected with Alt-S or by toggling the status character with either mouse button.

The settings of these status characters can be stored in the default file for future edit sessions.

11.3 Related commands

There exist a few other commands that fall in the category of search and replace. They are:

Ctrl-A
: This command causes STEDI to determine the word the cursor is currently on and then to search for the next occurrence of this word according to the default options. As the command is not issued from the command line, it can greatly speed up the finding of all the various occurrences of a particular word.

Ctrl-B
: If the target word of a previous Ctrl-A search has been modified by the user, it is impossible to continue the search for the old string using Ctrl-A. The Ctrl-B command remedies this situation. It will search for the same word that the last Ctrl-A operation searched for. Its equivalent with the mouse is obtained by pressing the right button on the '**r**' in the mouse menu.

Left mouse button + right mouse button
: This is a mouse command that performs the same action as the Ctrl-A command. It is issued by placing the mouse cursor on the word to be searched for and then depressing the left mouse button. This moves the cursor to this word. Then, without releasing the left mouse button, one presses the right button. Upon pressing the right button, the cursor will move to the next occurrence of the same word. The order in which the buttons are released is not important as the release of a button isn't considered to be an event by STEDI.

The right mouse button and the 'r' in the mouse menu bar
: When pointing the mouse at the lower case 'r' in the mouse menu bar and clicking the right mouse button, the action is equivalent to the Alt-B command. The last word searched for with the Ctrl-A command or the previous mouse command is searched for again.

=/string/options
: This command is issued from the command line. It makes STEDI determine the 'current word' as with the Ctrl-A command, but rather than just searching for this word STEDI will generate the command:

```
/current word/=/string/W+options
```

: This command is particularly useful as a very large percentage of replace operations involve whole words and the target words are already part of the text, so there is no need to type them in again.

Ctrl-U
: This command changes the case of all characters in a marked range (the range between the mark and the cursor) to upper case.

Ctrl-L
: This command changes the case of all characters in a marked range (the range between the mark and the cursor) to lower case.

Ctrl-F
: This command changes the case of the character under the cursor. The F stands for 'Flip case'.

11.4 Special characters

The above conventions still have one great defect: one cannot search for a character like '/' as it is part of the syntax. For this purpose, an 'escape sequence' has been defined. This escape sequence is initiated by the escape (<Esc>) key which enters an escape character into the command line. After the escape character is entered, the next character typed will be put into the command line no matter what it may be (even if it is a backspace). When the command is interpreted (parsed), such a sequence of escape followed by any character is treated separately if the character that follows has a special meaning to STEDI . If it doesn't then the ASCII code of this second character will replace the two characters and STEDI goes on reading the next character in the command string. In this way, one can search for characters such as backspaces in binary texts. The relevant escape sequences that are initiated via an escape character are:

<Esc>/
: Insert a slash (/) in the search or replacement string.

<Esc><Esc>
: Put an escape character in the search or replacement string.

<Esc><Return>
: Put the ASCII code for a carriage return in the string. This search will have no effect unless there are such characters in the text. This can be the case in a binary file. If the end of line should be matched one should study the chapter on regular expressions which describes searching with patterns, rather than fixed strings.

In most of the above 'escape' sequences one could also use the character \ instead of the <escape> key. This makes the typing of a backspace or a return harder, but the typing of the / into the text conforms more to the UNIX standards this way. So searching for the string '10/13' can be done in one othe following ways:

```
/10<escape>/13
/10<escape>/13/
/10<escape>/13/options
/10\/13
/10\/13/
/10\/13/options
```

There is also another way to put characters like a backspace into the command line via the hex mode (entered with **Ctrl-H** p. 144) which allows for the insertion of any hex ASCII character. There is a subtle difference between the codes entered with the escape sequence and those entered using the Hex mode. With Ctrl-H, any character can be entered from the keyboard into a string, including a return or a slash (/). However in this case, the characters are inserted before the command is interpreted. Thus any slash entered with Ctrl-H will not differ from a normal slash entered from the keyboard and thus will be seen as belonging to the syntax of the search and replace statement. Likewise, a return will be inserted as a 'normal' character that has no special status. This is well suited for binary files but is rarely useful for normal text files.

Note that the search/replace command cannot be initiated with the mouse. Very often a single search operation will be repeated several times. In that case, one can use the mouse by clicking the left button on the '**r**' in the mouse menu. This has the same effect as using Ctrl-R to repeat the previous command.

11.4.1 Special search commands

Sometimes one needs to do a search that involves a pattern, rather than a fixed string. We call a string a pattern if it describes (in a special language) a whole class of strings that could result in a successful search. The language for such patterns is given in the next chapter on regular expressions. It may also be needed sometimes to search for strings that contain an end of line. If such a string is taken out it would result in putting together two lines. Also this kind of searches should be dealt with via the language of regular expressions. Finally the regular expressions should also be used when the **replacement** string contains an end of line, unless the user likes to see this end of line inserted in the text as a funny character (sometimes needed in a binary file).

Regular expressions

At times the user may wish to search for a pattern rather than for a fixed string. A pattern is a description of all strings that should be acceptable during the search. An example of a pattern would be "all strings that start with an A and end with a B and don't contain any blanks". It is of course necessary to have a language for the specification of patterns. The language that is used follows the definitions in the book by Aho, Sethi and Ullman ("Compilers, principles, techniques and tools, Addison Wesley 1986, page 148) rather closely. This means that people who are familiar with UNIX will have to note only a few differences (mainly extensions) over what they are used to. In addition the current implementation has fewer restrictions and extensions have been made to facilitate the matching or replacement of linefeeds.

Of course the greater generality of using complete patterns rather than a fixed string makes a search operation much slower. Therefore the user should select the use of patterns specifically by starting the search or search and replace operation with `//` rather than with a single `/`. In the single slash mode the searching is performed with the Boyer and Moore algorithm, while in the double slash mode searching uses a complicated pattern matching "engine". The language which defines the patterns is defined by Aho et al. and is referred to as **regular expressions**. It is possible to define patterns that take so much time during the searching, that the user may decide to discontinue the operation. In several implementations of STEDI this can be done by pressing the key combination that indicates a break. On the PC family and the Atari ST this would be done by pressing both shift keys simultaneously. In a UNIX version this would be accomplished by pressing Ctrl-C.

In addition to the speed advantage the use of the single `/` offers also the advantage of simplicity. There are very few special characters, so the searching for strings containing characters that have a special meaning in the language of the regular expressions doesn't need special thought.

12.1 Single objects

A pattern consists of *single objects*. Single objects can be one of the following objects:

- A single character that has no special meaning.

- An entire string enclosed by double quotes as in "string".
- A group of characters enclosed by straight braces.
- The contents of a pair of regular parentheses.
- A linefeed.

Parentheses can be nested as in

```
//a("bc"(de)f)/
```

Here the single objects d and e are taken together to form a single object on a lower level. The string "bc" is a single object that should match the characters b followed by c. The objects "bc" (de) and f are then combined to form one single element on the lowest level. The whole pattern consists of this object with an a to the left of it. All operators act on single objects.

12.1.1 Groups

Groups are special objects that can match a class of characters. A group is indicated by a pair of braces `[` and `]`, with the characters that belong to the class between the braces. So the pattern

```
//[13579]/
```

will match any of the single characters 1, 3, 5, 7, or 9. This isn't very practical when many characters are involved, so there is a way to indicate a range of characters:

```
//[3-7f-p]/
```

The above group contains the characters 3 to 7 and f to p. If the first character in the group is the character ∧ the group contains all characters except for the characters that are mentioned. So

```
//[^3-7f-p]/
```

matches anything except for the digits 3 to 7 and the characters f to p. The above leaves one problem: How are the characters `[`, `]` or `-` included in a group? This can be done by putting them in a position in which they 'cannot' occur, or which would make the whole group meaningless as in:

```
//[]-[]/
//[^]^-[]/
//[-z]/
```

The first group contains exactly the three special characters, the second group contains all characters except for the characters `]`, `-`, `^` and `[`. The third group contains the two characters - and z. To facilitate the use of the special non-ASCII characters that occur in the native character fonts on some computers there are some special ranges of characters: Whereas a-z means all lower case regular characters a-ä means all lower case characters, including the accented ones. The ä may be replaced by any other accented character in the extended character set. Similarly A-Ä means all uppercase characters including the ones in the extended set. The range ä-ö (or any other two lower case extended characters) gives all extended lower case characters, Ä-Ö gives all extended upper case characters and ä-Ä gives all characters in the extended character set.

Finally there are some shortcuts for groups that are used frequently. These are:

character	group
#	[0-9]
&	[a-äA-Ä] (all alphabetic characters)
∼	[0-9a-äA-Ä] (all alphanumerics)
!	any 'word' character
!∧	any character not in words

Special groups

The word characters are explained on page 97. These shortcuts are an extension over the regular UNIX definitions.

12.2 Repetitors

The first type of operators are the repetitors. Such a repetitor acts only on the single object directly to the left of it. Repetitors are:

repetition element	effect
+	take object one or more times
	take object zero or more times
?	take object zero or one times
{*m*,*n*}	take object at least m, at most n times
{,*n*}	take object at most n times
{*m*,}	take object at least m times
{*m*}	take object exactly m times

The repetitors

Some examples are:

```
//ab*c/
//a(b*c)*c{4}/
//a[bc]*c{5}/
//"abc"+/
```

The first pattern will match to an a, followed by zero or more characters b, after which there should be a c. The second pattern is more complicated. The first character should be an a. Then we want zero or more times the object `(b*c)`. This object would match any number of b's followed by a single c. The effect is that `(b*c)*` will match any string that contains only the characters b and c, with the side condition that the last character must be the character c. Finally there should be 4 more c's. The third pattern shows how this can be done simpler with the use of a group. The fourth pattern will match one or more occurrences of the string abc. This means that the `+` operates not only on the c. It is equivalent to `(abc)+`, but the searching with the string is much simpler and faster.

Repetitors are always given the maximal value that they can take. This means that the left most repetitor in a pattern tries to match as many characters as possible. Then the next repetitor tries to match a maximal substring. The effect is usually a maximal match.

When no upper limit is mentioned the editor substitutes a maximum of 255. In practice the limits of the pattern matcher may be reached earlier as should be clear from the following pattern:

```
//(.*\n)+/
```

which should match an entire file, whatever its length (the `\n` indicates a linefeed as explained below). In practise the editor will display the message

```
Expression too complicated during matching
```

after about 256 characters in the match. After it sees that the match is longer it cannot continue because its internal buffers are full. This restriction may be lifted in the future.

12.3 Or and If

In addition to the repetition operators there are an 'or' and an 'if' operator. These are given by

```
//a|"bc"/
//a%b/
```

The symbol | indicates the 'or'. It indicates that either the character a or the string bc will cause a match. The % sign is a kind of 'if'. The patterns matches to an a if it is followed by a b, but the b isn't consumed yet. The difference with `//ab/` would become very clear in a replace statement as the last pattern would also take out the b.

12.4 Additional special characters

There are some special characters to indicate a position in a line. These are the characters ∧ for the first position in a line and $ for the end of a line. So

```
//^a/
```

looks for lines that begin with the character a.

```
//a$/
```

looks for lines that end with the character a. This use of the character $ cannot interfere with the use of the dollar sign to indicate variables, so that the command line processor may substitute them (p. 111).

The end-of-line character is indicated either by the two characters `\n` or by a linefeed character inside the pattern. The presence of linefeeds inside the patterns isn't allowed in most regular expression programs, but it can be very handy:

```
//\n{2,}/=/\n/
```

This would remove all empty lines from a file. A word of caution is in order here. Substitutions of the type

```
//\n/=//
```

would remove all linefeeds from a file. This would have the effect of making one giant line. Lines are however limited to 255 characters, so the replacements that would make longer lines are skipped.

When a character is needed that has a special meaning it should be 'escaped'. This can be done either by putting a backslash character in front of it as in `\$` to look for a dollar sign, or to put the character <escape> in front of it. The use of either of these escape characters switches the interpretation of the character off (with the exception of the n with is used for the linefeed).

12.5 Replacements

Regular expressions can also be used for making replacements. The search part is the pattern as described above, and the replacement string can be specified in the same way as this is done for the regular search and replace command p. 72. All options that can be used there apply also for the search and replace with regular expressions. In addition the replacement string may now also contain linefeeds (indicated by `\n` or <escape><return>).

There is one restriction with respect to the options available for the regular search and replace. Only the forward search modes are available for the regular expressions. There exist no good definitions of a backward mode. One could either try to work back trough the file, matching from the back, or stepping back through the file, matching the pattern from the left to the right. The first method is rather against intuition, while the second method may not yield the 'longest' match. Neither is satisfactory.

12.5.1 Substitution variables

Sometimes it is necessary to tranfer some information about the match to the replacement string. This is done with variables. There are 10 variables, indicated by the 10 digits. Such a variable is used by specifying the character @ followed by the corresponding digit. In the pattern the variable is filled with the contents of what the single object to the left of it matched to. In the replacement string the contents of the variable are substituted. Example:

```
//<[^>]*@1>/=/@1/
```

In each line objects of the type "<return>" would be replaced by the plain string "return". The pattern says: First a character `<`, then any number of characters, unless it is the character `>`. This sequence of characters is put together into the variable 1. Then the pattern needs a `>`. If such a match is found the whole thing is taken out and replaced by the contents of the variable 1. The following is a little fancier:

```
//!+@1!^+@2!+@3/=/@3@2@1/
```

The object !+ is a sequence of characters that can belong to a word, in other words: a word. We put it in variable 1. Then a sequence of non word characters should be put in variable two. The word after it goes into variable 3. The replacement string has then the two words exchanged. This single replacement exchanges all pairs of words in a file!

Multiple occurrences of the same variable in the pattern force the pattern matcher to have these match identical objects:

```
//[0-9]@1[0-9]@1/=/:@1:/
```

Here all pairs of identical digits are replaced by a single digit enclosed by semicolons. It can be even wilder:

```
//(&@1){2,}/=/@1/
```

Here all strings of at least two the same alphabetic characters are replaced by a single occurrence of that character.

There is a special variable that exists in the replacement string only. The character & in the replacement string signifies the whole match of the pattern. So

```
//./=/&=/r
```

puts an equals sign after each character in the current range between the cursor and the mark.

12.6 Overview

The full syntax of the regular expressions is given in the following table.

character	what it matches with
.	any character
∧	first position in line
$	last position in line
[*xyz*]	one of the given characters
\'char'	don't interprete 'char'
<esc>'char'	don't interprete 'char'
\n	linefeed
<ret>	linefeed
"string"	match the string as one object
#	any digit
&	any alphabetic character
∼	any alphanumeric character
!	any word character
!∧	any character not in words
/	end of search string
( )	consider contents as one object
*a**	take object *a* zero or more times
a+	take object *a* one or more times
a?	take object *a* zero or one times
a{*m*,*n*}	take *a* at least m, at most n times
a|*b*	either *a* or *b*
a%*b*	*a* if followed by *b*
a@'digit'	put *a* in variable number 'digit'

The special characters
a,*b* are generic objects

12.7 Efficiency

The use of a number of repetitors in one pattern can make the search rather slow. The pattern

```
//.*.*.*abc/
```

needs for each position in a line that doesn't contain the string abc a search time that is proportional to the third power of the number of characters in the line.

Such a situation could be improved by making the pattern matcher smarter. In the above example `.*.*.*` could be replaced by `.*`. That is functionally the same. One could also start looking for the string 'abc' first (this doesn't work always because patterns can contain linefeeds, unlike the patterns in most regular expression matchers). All this intelligence would add much code, and even then users will invent patterns that will take much time. So it is left to the user to keep his patterns simple. A general rule is that if the first character in a pattern is a fixed character the search will be much quicker. When a search takes much more time than expected the search can be interrupted by pressing the 'interrupt' key combination. This is either the combination of both shift keys (MS-DOS, Atari-ST) or the Ctrl-C combination (a UNIX solution).

The above effect will occur mainly when repetitors are used that interfere with each other. Two repetitors of the type `.*.*` leave an ambiguity of some type during the match. So when the pattern

```
//E.*.*2/
```

is confronted with the line

```
E = m * c^3
```

the first `.*` will be made maximal at first (10 characters). The second `.*` contains then 0 characters and then there is no more character for the 2. So now the first repetitor goes down to 9. The second takes 1 and there is no room. Then the second becomes 0 again and the 2 is compared with the 3. No match! The the sizes (8,2),(8,1),(8,0),(7,3) etc. are tried. In total 66 combinations are tried!

On the other hand the pattern

```
//ab*cd*ef*g/
```

has hardly any problems with the line

```
abbbbcddddddeffg
```

even though there are three repetitors. Here the repetitors don't interfere and the match is found in one attempt.

Patterns that start with a large degree of freedom will be rather slow. The pattern

```
//.zzzz/
```

will be significantly slower than

```
//zzzz/
```

The first pattern will always score a partial match at the position of the period. Then the whole pattern matching engine is started to find that there is no full match. In the second case the first character is a fixed character and looking for it can be done fast. In most files there should be very few hits and the pattern matchine engine is rarely needed.

Using the Mouse

The mouse can be used as an alternative for executing many of STEDI's commands. These mouse actions can be divided into several classes:

- moving the cursor and scrolling the screen
- opening and closing folds
- toggling internal settings
- selecting commands from the mouse menu.

These categories will be dealt with below.

13.1 Moving the text cursor

13.1.1 Positioning the cursor

Positioning the cursor is done by moving the mouse cursor to the desired spot on the screen and clicking the left mouse button. The editor is not sensitive to double clicking so its response is instantaneous, thus avoiding the often annoying delay while the system is waiting to see whether there is a second click. After any action of the editor not involving the mouse, the mouse cursor is generally rendered invisible so as not to obscure any text. As soon as the mouse is moved however, the mouse cursor appears in the form that depends on the computer. On some systems it will be just a cursor like object, while on more graphically oriented systems there will be a little arrow. Clicking the left button also performs another function besides positioning the cursor: this action causes the mouse menu, a collection of various letters and numbers, to appear in the message line. If the mouse is pointing at the message line when clicked, and not somewhere in the text field, no repositioning of the cursor occurs.

13.1.2 Scrolling with the mouse

It is possible to **scroll** the screen over one page by moving the arrow to either the top line or the bottom line of the text field, line 1 and line 24 respectively for normal character fonts and no split screen, and then clicking the right mouse button. If the right button is clicked in the left most or the right most column the screen is scrolled horizontally over the distance currently set by the variable hstep (see p. 116).

13.2 Fold manipulations

The normal function of the right mouse button is to open and close **folds**. When the arrow of the mouse is pointing to text within matching fold lines, a click of the right mouse button closes the fold defined by these lines. This action is similar to the effect of the F6 key on a fold when the cursor is within the fold. There are a few exceptions to this rule. First, when the arrow is on top of the name field of a closed fold, clicking the right button will open the fold instead. Second, when the mouse is pointing to the top or bottom line of the screen, a click of the right mouse button scrolls the text as is explained above. Third, when the mouse is in the first or the right most column a click with the right button will scroll the screen horizontally as explained above.

13.3 Toggling settings

When the mouse cursor points at one of the first five **status characters** in the status bar, it is possible to toggle the corresponding setting by clicking either mouse button. In contrast to this, the backup mode cannot be toggled with the mouse. The ability to change this mode too easily could be a substantial danger for one's files. In particular, the mode without a backup should be used with caution because in that case the old version of a file is deleted before the new one is written. Carelessness with this mode can result in loss of data.

13.4 The mouse menu

To make the **mouse menu** visible, click either button of the mouse. Then the mouse menu will appear in the message bar. If you do not want the click to result in moving the cursor or in opening/closing a fold as well, you should first move the mouse pointer into the space of the message bar before clicking. Selecting commands with the mouse is not possible unless the menu bar is showing. The menu bar appears as follows.

```
1© M G E  Y C P  1 2 3 4 5 6 7 8 9 0  r  R W S Q IA∞NYB1○TEST.C
```

On systems that have no infinity sign in their standard font the infinity has been replaced by a zero. As is obvious from the diagram, the menu contains several groups of commands. A command can be selected by moving the mouse cursor over the corresponding character and then clicking one of the buttons. Some positions in the menu are sensitive to whether the left or the right button

is clicked. To help avoid clicking on the wrong character, the blank spaces between the characters of the menu are insensitive to the clicks. We will see later how this can be changed. The various characters stand for the following actions:

M

Put a mark at the current cursor position. This command is the same as the F1 key.

G

Go to mark. This command moves the cursor to the position of the mark, and is the same as the command shift-F2 on systems that have an undo key. Otherwise there is no single key for this instruction.

E

Exchange the position of the cursor and the mark. Same as F2. As the mark is also used for moves through the file, it is not convenient to introduce highlighting of the region between the cursor and the mark. If one has forgotten where the mark has been placed, the use of this E-command comes in handy.

Y

Yank or cut. Issued with the left mouse button, this command cuts the region between the mark and the text cursor from the text and places it in the currently selected yank buffer. This is the same command as can be given with the F3 key. Issued with the right mouse button, it cuts the block range between the mark and the cursor as with the shift-F3 command.

C

Copy the text between the mark and the cursor to the current yank buffer (left mouse button). This is identical to the use of the F4 key. The right mouse button will cause a copy operation of the block between the mark and the cursor as with the shift-F4 command.

P

Issued with the left button, this command pastes the contents of the currently selected yank buffer into the text of the displayed file at the position of the cursor. This is the same as with the F5 key. The right mouse button will cause a block paste operation as with the shift-F5 command.

The ten numbers

They have different meanings depending on whether the left mouse button or the right mouse button is clicked. When the left mouse button is clicked on one of them, the buffer corresponding to that number is displayed. This is analogous to the combination of the alternate key and one of the number keys of the main key pad. When the right mouse button is used, the learn sequence corresponding to that number is replayed.

r

Repeat. Clicking the left button on this letter causes the last command that was issued from the command line to be repeated. This is the same command as Ctrl-R and is rather useful for repeated string searches. Clicking the right button causes a repeat search of the word that was last searched for with a Ctrl-A command. This latter is equivalent to the Ctrl-B command.

R

Read a file. This command is for reading a file from disk into the currently displayed buffer. The file will be inserted at the current cursor position. To select which file will be read, a file selector is displayed if the system offers one.

W

Write the contents of the current buffer to a disk file. This command also invokes the file selector if the system offers one. Note that the R and W commands have analogous but slightly different commands among those of the keyboard unless there is no file selector.

S

Save the contents of the currently displayed buffer to disk. This command tries to save the file currently being edited to disk under its own name. If the buffer has no name yet, no writing can be done and an error message is issued. This is the same as the F9 key command.

Q

Quit. If there is no unsaved work in the editor, the edit session will be terminated. Otherwise there will be a warning telling which buffers still have not been saved since being altered and a query whether to quit anyway and disregard the warning. This question can be answered with either a y (=yes) or a n (=no). This command is the same as the shift-F10 command.

There are some more commands that can be issued from the command line with the mouse. Regardless of whether the menu is visible or not, clicking either button on the command line prompt (in some versions a copyright mark (©), on MS-DOS a smiling face) in the seventh column enters the command mode as if <Esc> had been pressed from the keyboard. Clicking in the field that displays the **line number** causes the display of the current position of the text cursor to be displayed in terms of lines and columns, in terms of bytes and its position on the screen. This is the same as the = command as entered from the command line p. 16. Clicking either button in the field that displays the **name of the buffer** will cause the full path name of the file being edited to be shown in the message field. This is identical to the Alt-L command.

Finally, there is a command which can be issued with the mouse that is equivalent to the Ctrl-A command (search for the next copy of the word under the cursor). If you point the mouse cursor at a word, and then press the left mouse button, the cursor will move there. While holding this button down, if the right button is pressed the cursor will jump to the next occurrence of this word. The equivalent to the Ctrl-B command (jump to next occurrence of last word specified with the Ctrl-A command) is explained above. This is the clicking of the right mouse button on the lower case 'r' of the menu bar.

More details about the commands that can be issued with the mouse can be found in their corresponding chapters. See, for instance, the chapters on folds, the mark, cutting and pasting, the file buffers, the command line, reading, writing, printing and the status bar.

13.5 Changing the mouse menu

It is possible to restructure the mouse menu entirely to ones taste and write this new menu in the default file (p. 52) so that the new layout can be installed automatically at startup. To this purpose the mouse menu is divided in positions. Each menu position corresponds to two character positions in the menu bar: the position with the character in the default menu and the position to the right of it. Some positions have been left empty in the default menu.

The first position in the menu is the position of the M. The Y has position 5 (position 4 isn't used in the default menu). the last position in the default menu is the Q and it has number 26. When the screen is wider there will be more positions. The maximum number of positions is 50.

The character representation in the menu bar can be changed with the set command. The command

```
set {4} = "string"
```

sets the fourth item in the menu bar to be the given string. Only the first two characters of the string are relevant as each position occupies two character locations. It isn't critical whether the string is enclosed by the double quotes, unless the string contains a leading blank. To change the 19-th position by the string 'Ha' one would issue the command

```
set {19} = "Ha"
```

As this can also be done from macro's one can prepare a new menu rather easily (and reinstall it without pain if the default file gets lost). The menu thus defined is stored in the default file when this file is written with the DW command (p. 52).

If one likes to make a menu with full words rather than single characters it would be annoying to have dead zones in the middle of these words. Therefore the dead zones can be switched off with the command

```
set menuspaced = off
```

After this command all positions in the menu bar are sensitive to mouse clicks. The dead zones can also be turned on again with the command

```
set menuspaced = on
```

Also the value of the variable menuspaced is stored in the default file when it is written with the DW command.

In addition to the above there is a special variable 'mversion' which can be set to a numerical value. It is for use inside macro's that are made to change the mouse menu. This way the macro can keep track of what the current menu looks like. The user should design his own numbering system. The default value is zero and also this variable can be stored in the defaults file with the DW command.

There is of course no point in changing the characters in the mouse menu if the corresponding actions cannot be changed. The action corresponding to a click inside the mouse menu can be redefined by a key redefinition. Internally STEDI translates all mouse events into pseudo key codes. These codes are then treated in the same way as the real keyboard events i.e., they can be redefined. How this is done is explained in the chapter on key redefinitions p. 164.

In addition to the above method to change the mouse menu there is a quick and very dirty way to change the menu. One could read the default file into one of the buffers, locate the menu string in it and change it there. Note however that this requires great care. The length of the file may not be changed and everything has to be done in the raw mode. For more about such an operation one should consult the section on Binary editing, p. 146. Be warned though that this isn't the 'official way'.

Tabs

STEDI has a number of commands for controlling tabs and blanks in a file. These commands are summarized in this chapter.

All tab commands are initiated by the character T as the first character in the command line. This family of commands allows for a rather flexible manipulation of white space (blanks and tabs). It is possible to expand tabs into blanks (detabbing), to replace blanks by tabs (tabbing) wherever possible, to remove trailing blanks and tabs (trimming) and to define the positions of the tab stops.

14.1 Defining tab stops

The tab stops are defined with the command:

```
t [ColumnNumber] [Number*ColumnIncrement] ....
```

If a column number is specified, a tab stop is placed at that position. In the second argument an entry n*i generates n tab stops each i columns to the right of the previously defined stop. There may be any number of the above parameters in any order as long as the column numbers of the tab stops that are generated are in ascending order. A tab stop that has a column number that is to the left of its predecessor is ignored. The buffers allow for 100 tab positions. Any additional tab positions will be ignored. This would be very rare as there are only 255 character positions on one line. Any tab stop beyond column 255 is irrelevant and will be ignored. It is also impossible to put a tab stop in column 0. If a tab stop is put at position 0, all tab stops after it are ignored.

Examples:

```
t 100*8
```

This is the default tab setting on most computers. Actually many computers and/or compilers can become very confused if you use any other tab setting. Note that only $255/8 = 33$ tabs are relevant. We could of course also have used 33*8 to get the same effect but the 100 or any other big number avoids having to do arithmetic. This mode is often used in assembler programs.

```
t 100*4
```

Often used in C.

```
t 7 10*4 73
```

A good setting for Fortran. Alas many Fortran compilers get confused by it and move parts of the code beyond column 72 if there are too many tabs. The UNIX fortran compilers have no problem with it.

Warning: There are many compilers and assemblers that don't know what to do with tabs. Some assemblers will not even allow tabs. Other compilers may interpret them as being at the fixed positions 8,16,24,32,... so that the next character is taken at the positions 9,17,25,33,..., even if this is very unnatural for the language involved. Some experimenting should show the tab sensitivity of the compilers involved. If a compiler/assembler will not accept any tabs, they can be removed during the writing of the text to file with the use of the printer mode for the write command (p. 44).

The tab positions defined in the above way hold only for the current buffer. If you like to set the tabs for all buffers simultaneously you should put the character g after the t (indicates global). The tab positions can be stored in the default file with the DW command. This way they may be used in a future editor session.

14.2 Tabbing, expanding, trimming

The other three commands are rather similar in nature, so they are treated together.

tab[r][f]
: This command replaces as many blanks as possible by tabs without changing the screen appearance of the file. For program files that use deep indentations, this may save 30 to 40 percent in file space when the file is written. This operation is called 'tabbing'.

te[r][f]
: This command replaces all tabs by as many spaces as needed without changing the screen appearance of the file. The name of this command is 'tab expand' or 'detabbing'.

tt[r][f]

This command removes all unnecessary blanks. Blanks are considered superfluous if they occur inside the range of a tab or at the end of a line. Also tabs at the end of a line are removed. This operation is called 'trimming'.

These command operate on the whole file, unless the optional parameter r is specified. If this parameter is used, the range of action is the set of whole lines from the mark to the cursor, including both the line with the mark and the line with the cursor. The order of the mark and the cursor is unimportant. If the parameter f isn't specified the first three characters of potential opening and closing fold lines should not be treated by the tabbing, detabbing or trimming routines. When the f is specified also those positions will be treated. Thereby these lines may loose their folding properties.

The expansion of the tabs in a file may make the file much longer. Occasionally this will exhaust the memory that is available. The editor will then attempt a garbage collection to make more space available. In case this is not sufficient the editor will give the message that there is not enough memory and leave its job unfinished. If there are files in other buffers or in the undo buffer one could remove those and continue the expansion of the tabs. If this is either not possible, or does not free enough memory the expansion can still be executed by writing the contents of the buffer to a file, using the printer mode (p. 44), although in that case the file will no longer fit in the editor.

If you are in doubt as to where tabs and blanks are in your file, and where no characters are at all, you may use the **Alt-T** command. This command toggles between a special mode in which all characters on the screen have a unique representation and one in which all 'white characters' look alike. In particular, in the special mode blanks appear as small hollow circles in a superscripted position, tabs appear as small filled circles also in a superscripted position, and places at which no character at all resides remain blank. The other character that is seen as a 'white character' in the normal representation is an ASCII null character. In the special representation, this character is given the appearance of a small filled circle in a subscripted position. In some computer fonts the character indicated by the hexadecimal code FF (255 in decimal) is also represented as a blank. In the Alt-T mode this character is represented as a colon.

Word-oriented commands

15.1 Words

STEDI has a rudimentary knowledge of words allowing a number of word-oriented commands to be built in. For most purposes of the editor, a word is defined in one of two ways. A word is defined as either a string of alphanumeric characters delimited by a non-alphanumeric character or a single non-alphanumeric character of a certain class.

In order to make these definitions precise, let us separate the various characters into three classes:

1. The first class, the 'alphanumeric characters', consists of characters from the set a-z, A-Z, 0-9, _ and the special alphabetic characters in the character font that can be used in the various national character sets.

2. The second class consists of some 'in between' characters which are mainly punctuation characters. These are: `!` `"` `#` `$` `%` `&` `'` `(` `)` `*` `+` `,` `-` `.` `/` `@` `[` `\` `]` `^` `‘` `{` `|` `}` `~` and Δ. Sometimes they are seen as a word by themselves, regardless of the characters next to them.

3. The third class of characters is considered to be 'whitespace' and cannot be part of a word. This class includes all control characters, blanks, tabs, carriage returns, linefeeds and all characters with an ASCII code greater than 7F that are not included in the first class.

It is possible to change the class of a character with the command

```
set <char> = on/off/single
```

The value on puts the character in class 1, the value single puts it in class 2 and the value off puts it in class 3. Example:

```
set <$> = on
set <_> = single
```

This setting would be appropriate for Fortran. When the default file is written with the DW command (p. 52) the 'word settings' are also stored in it.

In STEDI a word is defined to be either a string of class 1 characters delimited by characters from either class 2 or class 3 or a single character from class 2. Below, we will refer to the first of these definitions as a type 1 word and the second as a type 2 word.

15.2 Commands related to words

Given these definitions, there are a number of word-oriented commands. They are described below:

Command line commands:

/string/w
: This command searches for 'string' occurring as a word. This word option (w) for a search or a search and replace command signifies that there is only a match with the search string if the characters left and right of the match don't belong to the words in the search string. The search string may contain any characters so, for example, a search for the string 'one plus two ' as a word is legal, even with the trailing blank.

`/string1/=/string2/w`
: This command replaces the word string1 by string2 wherever string1 occurs as a word.

`=/string2/`
: This command replaces the 'current word' or the word on which the cursor is by string2. If the current word occurs elsewhere, it will also be replaced, as with the normal replace command. This command only works for type 1 words.

Key commands:

Ctrl-W
: 'Move to next word'. This command moves the cursor to the first character of the next word. If there are no more words in the current line, the search is continued in the next line.

Ctrl-Q
: 'Move to previous word'. This command moves the cursor to the last character of the previous word. If there are no more words in the current line, the search continues at the end of the previous line.

Ctrl-X
'Delete word forward'. This command deletes all characters between the current position the cursor and the beginning of the next word, leaving the cursor on the first character of the next word. Hence if you want to delete a full word, use this command with the cursor on the beginning of the word. If the cursor is not on a word, this command deletes the character the cursor is on plus all white space till the next word. The delete stops at a linefeed unless the linefeed is the first character to be deleted.

Ctrl-Z or Ctrl-Y
'Delete word backward'. This command deletes all characters between the current position of the cursor and the end of the previous word. Normally the cursor will then be just after the word previous to the one deleted. If the cursor is not on a word, the character left of the cursor is deleted and all white space left of it to the previous word. The delete stops at a linefeed unless the linefeed is the first character to be deleted.

Ctrl-A
'Jump to the next occurrence of the same word'. With this command the cursor is moved to the next occurrence of the word under the cursor. It can be issued with the mouse by pointing at a word, and holding the left button down while clicking the right button. A fuller explanation of this command is given below.

Ctrl-B
This command repeats the last Ctrl-A command. It will also be explained more fully below.

15.3 Word searches

Because the Ctrl-A and related commands have proven so useful for most programmers currently using STEDI, we give a full description of these commands here. They act also on the 'current word'. This is the word the cursor is in at the moment of the command. If one would like to inspect what this word is according to the editor one can type the command line command:

```
show word
```

The variable 'word' can also be used for more complicated operations (e.g. in complicated macro's).

15.3.1 Find current word

To find the next occurrence of the word on which the cursor currently lies, you can use the Ctrl-A command. This command first determines the cursor position, then finds the word boundaries to the left and the right of the cursor, and then copies the word under the cursor into a buffer. Afterwards, this buffer is used to generate a search command for the next occurrence of the word on which the cursor was.

One use of this command is to find where in a program a certain variable or label is defined or used or whether a name has been used already. Once the user is adapted (or addicted) to it, the Ctrl-A command can become one of the main ways to move through a file. This way motion is based on correlation rather than on distance.

This command can also be issued with the mouse. To do this, move the mouse cursor on top of the word to search for, then click the left button without releasing it. Then while the left button is still depressed, click the right button. The use of this command with the mouse is however somewhat less useful than the Ctrl-A form, because a match may be found at any arbitrary position on the screen. This makes continued searches somewhat more involved.

15.3.2 Repeat current word search

The Control-B command gets the word that is in the Ctrl-A buffer and uses it for searching for the next occurrence of that word in the text. This is particularly useful when the word of the previous Ctrl-A search is not in view, either because the cursor was moved around, or the word found was altered or the user switched to another buffer. The mouse equivalent of this command is to click the right button of the **r** (=repeat) in the mouse menu.

15.3.3 Replace current word

The replace current word command is a command line command with the following syntax:

```
=/string/options
```

As with the Ctrl-A command, the current word is determined and copied to a buffer. It is then used to generate a new statement that looks like:

```
/CurrentWord/=/string/w+options
```

All settings and options that are relevant for a normal search and replace operation are available for this command. The word option is automatically implied.

15.4 Word wrapping

For word wrapping purposes, the definition of a word is slightly different from that given above. For this purpose, blanks, tabs, linefeeds and carriage returns are considered 'whitespace'. Then a word is considered to be any string delimited by but not containing any 'whitespace' characters. These are referred to as words of type 3. The Fortran word wrap will only look for such a 'whitespace' separator in the columns 63-72 as Fortran is a language that doesn't require whitespace in its statements.

15.4.1 The word wrap command

The word wrap option is invoked with the WW command on the command line (a single W is reserved for the write command). This command causes the cursor to jump back to the left side of the screen when the word being typed goes beyond a prescribed column. This option is well known in the context of word processors, but can also be very useful for programming and especially for typing commentary or manuals that will be processed further with powerful formatting systems like TEX.

The syntax is as follows:

```
WW#
```

where # is the column number at which you wish the word wrap to occur. So for example, WW78 sets the word wrap column at column 78. If the number is omitted, the editor reports the current word wrap mode. The command

```
WW-
```

turns the word wrapping off.

The algorithm for wrapping is rather simple: whenever a character is entered from the keyboard, a check is performed to see whether it comes to the right of the 'wrapping column'. If so, a search begins to find a blank or tab to the left of it (and also to the left of the 'wrapping column'). If such a blank or tab is found a <Return> is inserted after it. This causes the new line to begin

(usually) with a non whitespace character, while the trailing blank or tab in the old line indicates a so called soft linefeed.

For Fortran programming, a special word wrap command, the WWF command, is provided which installs a special wrapping formula for Fortran programs. In Fortran a continuation line needs a nonblank nonzero character in column 6 and columns after column 72 are not recognized as part of a program statement. Thus the formula for Fortran word wrap is to set the word wrap at column 72. When this column is reached, the cursor moves to column 6 of the next line and inserts a character ('+') there to indicate that it is a continuation line, before it copies the word being wrapped to that line. This indentation is accomplished by means of the auto-indent mode (explained below) and hence any preset values for that command will be erased when the Fortran word-wrap command is issued.

The word-wrap mode belongs to the current buffer. Each buffer may have its own mode. The setting of a buffer can be moved to the default string with the DS command and to the default file with the DW command (See the chapter on defaults p. 51).

Summary of word-wrap commands:

WW#	Normal word-wrap after column.
WWF	Fortran formula wrap.
WW-	Word wrap off.
WW	Report word wrap status.

Rewrapping paragraphs can be done in one of two ways. These two ways have different definitions of a paragraph, and in the end a different effect. The commands are:

15.4.2 The rewrap commands

Whenever a word-wrap option is active (with the exception of the Fortran mode of course), the command which is executed by pressing **Alt-W** will try to determine the first line of the current paragraph and then work its way to the end of the paragraph, rewrapping the paragraph as if it had just been typed in. It recognizes lines belonging to the same paragraph by the trailing blanks that are left at the end of each line when the word wrap is invoked. Therefore the first line of the paragraph is recognized as the line following the first line without a trailing blank that the editor encounters when working its way backwards from its start position. The last line of the paragraph is the first line that the editor runs into without a trailing blank when it is rewrapping. These rules are superceded by two other rules:

- A closed fold line never belongs to a paragraph.
- A line with only blanks and/or tabs doesn't belong to a paragraph.

Note that the trim command removes trailing blanks, therefore destroying paragraph information. Redefinition of paragraphs can be done however by adding a blank to all lines over a given range. After marking the range of the paragraph as a marked range, this can be done with the following command line command (see regular expressions p. 82):

```
//$/=/ /r
```

Then by definition the range becomes a paragraph (assuming it has no blank lines). It is also possible to make much more sophisticated restorations of paragraphs by using macro's (p. 119). In conjunction with the Alt-W command it may be useful to recall the **Alt-T** toggle command which allows blanks to be displayed as little circles (see the chapter on tabs p. 96). With this command the presence of the necessary blanks can be checked.

The second rewrap command is entirely different. It is executed with the Alt-Q combination. Its definition of a paragraph is a range of lines that is enclosed either by fold lines or lines that contain only white space characters (blanks and/or tabs). STEDI will determine the current paragraph, eliminate all unnecessary blanks, rewrap the paragraph (obeying the auto-indent rules) and then insert extra blanks to make the right edge of the text look nice. There is a limit to the number of blanks inserted. This second rewrap is nice if the text typed has to be printed directly on a line printer. Note however that its definition of a paragraph is different from the definition for the other type of rewrap and that the 'soft linefeeds' get destroyed.

15.5 The auto-indent mode

The main purpose of the auto-indent mode is to allow automatic generation of indentation while programming. It can also be used for writing texts to govern the position of the left margin on the screen.

The auto-indent mode comes in three types. The first type is invoked with the **A+ command** and is the 'normal' autoindent of most editors. When a new line is opened, the leading tabs and spaces are copied from the line in front of it. This mode is very useful for programming in many languages.

Some languages work with statement labels in the first few columns which makes the normal autoindent feature not so useful. Therefore a second type of auto-indent is also available which is activated with the **A#+ command** in which # stands for a number less than 100. In this mode the cursor goes

to column # +1 in the new line and copies tabs and blanks only from that column on. All characters of the previous line that are in front of this position are ignored. The net effect is that statement labels are skipped in the auto-indent up to a given fixed position.

Finally the A# command is used for typing text files with a fixed left margin. Every time a new line is opened, the cursor is placed in column # +1 and no further indentation is attempted.

The auto-indent mode is turned off with the **A- command**. The mode can be inspected with the **A** command.

Summary of auto-indent commands:

A+	Normal auto-indent.
A#+	Auto-indent to at least column #.
A#	Fixed indentation to column #.
A-	Auto-indent off.
A	Show auto-indent mode.

The auto-indent mode belongs to the current buffer. Each buffer may have its own mode. The setting of a buffer can be moved to the default string with the DS command and then to the default file with the DW command. See also the chapter on default settings p. 51.

The undo feature

Much effort has been made to provide for the ability to recover from mistakes that have been made inadvertently, while using STEDI. For example, it is possible that you may delete some lines and then realize that you would like to have them again. For this purpose, STEDI generally monitors deletions by writing them away to a special buffer so that they will be available in case they are needed. Thus in many cases, if no other action has been taken after a deletion the part of text deleted is recoverable. This chapter is a summary of STEDI's undo features.

The <**Undo**> (on keyboards that have no undo key this is generally shift-F2) key provides the possibility to undo deletions which are made with the basic delete operations for deleting characters, words, lines, or when a buffer is cleared. To undo cut and paste operations, paste and cut operations themselves are used. These will be explained in what follows.

No undo capability is provided for search and replace operations because STEDI would have to save almost the whole file in order to be able to undo such complicated changes. If you would like to make substantial changes with the search and replace command, you may like to save a backup copy of the file first before proceeding. Alternatively you can use the veto option in the search and replace command to guard against errors which are difficult to undo.

16.1 The Undo key

16.1.1 Characters and words

When deleting characters and words in a line, or deleting to the end of the line, STEDI stores the original contents of the line in the undo buffer. Any deletions of this type that are made all on the same line can be restored with the undo key. The delete commands that are relevant are the two character delete commands <**Backspace**> and <**Delete**>, the delete word commands **Ctrl-X** and **Ctrl-Z**/**Ctrl-Y**, and the delete to end of line command **Ctrl-D**.

In addition, in some instances, contiguous characters that are deleted sequentially using these commands can be restored with the undo key, regardless of whether they are on the same line or not. Thus if you hold down the <Delete> key and delete several lines, they all will be restored. In general the above deletes are remembered as long as they are in lines that are inside lines

that are already in the undo buffer or in lines that are adjacent to lines in the undo buffer.

In the case of these deletions, after restoring the deleted characters they can be deleted again by a second pressing of the <Undo> key. In that sense, the undo key acts as a toggle between the original lines and those from which some characters were deleted.

16.1.2 Lines

When successive lines are deleted with the **Ctrl-<Delete>** command, they can all be recovered with the <Undo> key (shift-F2 on some systems). The line delete command acts separately from those above, so consecutive lines deleted with a combination of this command and the word and character delete commands above cannot all be restored.

16.1.3 Buffers

The <Undo> key (or shift-F2 on some systems) can also be used to restore a buffer which was cleared using the **Shift-F9** key. Whenever a buffer is cleared, a copy of the file cleared is placed in the undo buffer. Since this buffer takes up space in some of the computer's memory, if you are short on memory, you may want to hit shift-F9 twice when you are clearing a buffer. The first time will move the file into the undo buffer, and the second time will copy the contents of the (now empty) buffer into the undo buffer, thus effectively clearing it and freeing up the memory allocated to it.

Note: For all the above operations, there is only one undo buffer for all files. Thus if you make a deletion in one file and then move to another to do some editing, the information to undo the delete in the first file will be lost. This is a compromise to the fact that there are still so many computers in which memory is a hard item to get. In the future this restriction may be lifted and a more versatile undo will be implemented.

16.2 Cutting and pasting

Often when a cut (yank) or a paste is made, there may be desire to reverse the operation. For this purpose the <Undo> key (or shift-F2 on some systems) is not used. Whenever a paste operation is performed, after the operation is completed STEDI places the mark and the cursor in the appropriate positions so that an immediate cut operation will cut out the part of text that was just pasted in. This holds for both paste (**F5**) and block paste (**shift-F5**) which

are undone by cut (**F3**) and block cut (**shift-F3**) respectively. In the case of a cut operation (**F3**), an immediate paste (**F5**) will restore the text just cut out. The only one that is not completely straightforward is a block cut (**shift-F3**). In this case, in order to restore the text cut out, the cursor must be placed in the first line from which the text was cut, at the position of the cut. Of course, if a mistake is made, the text can be immediately cut out again and the cursor re-positioned.

The learn buffers

There are 10 learn buffers that allow the user to combine several key strokes into a single command. Each buffer may contain up to 100 key strokes. The contents of the buffers can be stored in the default file so that they may be used in later edit sessions. The 10 buffers are labeled 1 to 0, the zero representing learn buffer 10.

17.1 Filling a learn buffer

To begin putting key strokes into a learn buffer, enter the command line command `L#` where `#` is one of the 10 digits from 0 to 9. After this command is issued, you may begin entering commands as normal. These commands take effect in their usual manner; at the same time, they are recorded in the learn buffer corresponding to the number given. While a buffer is learning, the © symbol on the command line is replaced by the number of the buffer learning. If more than one buffer is learning at the same time, the number of the lowest buffer will be displayed. (For purposes of ordering, the zero stands for ten.)

Terminating the learning process is rather system dependent. On the IBM-PC like computers one should press the combination of the Alternate key and the function key with the number of the buffer that was learning. On the Atari-ST one should press the <Control> key and the number key on the main key pad corresponding to the buffer which was learning. If there is a mouse one may also click the right mouse button on the corresponding number in the mouse menu. If more than one buffer is learning at the same time, only the buffer with the lowest number can be terminated. A message will be given, indicating the number of characters in the buffer. If a buffer overflows it will also be reported and the user will have to terminate the learning process before the editor stops complaining.

17.2 Replaying a learn buffer

Replaying a learn buffer is done either with the Alt-Function key (MS-DOS) or with the control-digit combination (Atari-ST) or by clicking the right mouse button on one of the digits in the mouse menu. Replaying a sequence in one

of the buffers can therefore only be done after its learning process has been terminated.

One may replay a learn sequence of one of the lower buffers while learning in a higher buffer. The inverse is not allowed. This prevents loops and other undesirable effects. The buffers can not tell whether e.g. Alt-F4/Ctrl-4 means the replay of buffer 4 or 'stop learning in buffer 4'. Therefore it is not possible to first stop the learning in the higher buffer when two buffers are learning simultaneously.

If for some reason one wants to stop the replay of a learn sequence before it has ended, this can be done by pressing both shift keys simultaneously. Once stopped, it is not possible to continue at the position where the replay was halted.

In cases when a response is required from the programmer to confirm or veto an action, the response cannot come from the learn buffer. Examples of this are when the veto flag is set during a search and replace operation, or when a buffer that has not been saved is cleared. If such commands are included in a learn sequence, the editor will pause and wait for a response from the keyboard before continuing the learn sequence.

The contents of the learn buffers can be stored in the default file with the DW command.

The best way to see the power of these learn buffers is by means of some examples. The first one is used to teach buffer 1 to declare the line that contains the cursor to be a comment line in the C language. This is done as follows. With the exception that ASCII characters to be typed are grouped together, each key stroke is spelled out in detail for clarity. Commentary is given in parentheses.

- <Escape>
- l1
- <Return> (Now you have entered the learn mode)
- F1 (place a mark)
- shift-left arrow (go to column 1)
- /* (this starts commentary)
- shift-right arrow (go to the end of the line)
- */ (this ends commentary)
- F2 (return to the old position)
- <Alt>-F1 or <Control>-1 (terminate the learning in buffer 1)

Executing a sequence in buffer 3 five times is done with :

- <Escape>

- l4
- <Return> (begin learning in buffer 4)
- Ctrl-3
- Ctrl-3
- Ctrl-3
- Ctrl-3
- Ctrl-3
- Ctrl-4 (stop learning in buffer 4)

The keys that are entered in the learn buffers are the keys after they come from the key redefinitions (p. 148). If key redefinitions are used one may notice that the learn buffers can fill up rather quickly. Much used sequences can therefore better be programmed as a key redefinition or a macro. The learn buffers are mainly for little things that come up during a particular edit session and that have to be done several times.

There is one restriction to the actions you can undertake from a learn buffer: It isn't allowed to read a default file from a learned sequence. The reason is rather simple: the contents of the learn buffers would be overwritten by the the contents of the default file, because the defaults file contains also the learn sequences. This could lead to effects that are so interesting that they are forbidden.

Variables

To support a programming language for macro definitions (p. 119) STEDI is equipped with the possibility to define and use variables. The syntax that is connected to the use of these variables resembles the syntax of the UNIX c-shell csh very much. When variables are used their name is preceded by a dollar ($) sign. The name of the variable should consist of alphanumeric characters of which the first should be alphabetic. In addition the underscore may be used at any position (also the first). There may be no more than 10 characters (the dollar sign doesn't count). If the use of the variable makes it necessary that the contents of the variable are immediately followed by an alphanumeric character the name may be enclosed in curly brackets: `${name}`. The use of names is case sensitive as this doesn't interfere with the file system. There is a number of reserved names with a special meaning. These are given later in this chapter.

When a variable is defined or a value is assigned to it its occurrence at the left hand side of the statement shouldn't be preceded by the dollar sign. The syntax of such a statement is:

```
set variable = expression
```

If the variable exists already its old 'value' is replaced by its new 'value'. If the variable didn't exist a new entry in the list of variables is made. If this is originated from inside a macro, the variable is removed from the list again when the macro is terminated. There are two options for the set command. The global option makes that for an already existing value the editor looks also among the variables of the parents (and these can be changed) This is done with

```
set -G variable = expression
```

The local mode makes that the editor will not look among the variables of the parents and won't change these. If there is no variable yet by this name among the variables of the macro a new one will be made, even if this means that there is now more than one variable with that name. If a child process looks for a variable with this name it will run into the closest variable. The local mode (the default mode) is forced with the command:

```
set -L variable = expression
```

The L and the G are case insensitive. The effect of this is that a macro has full control over the variables of its parents, but all variables of its children are hidden from it. The syntax of the expression at the right hand side is explained in the chapter on macros (p. 121). If just a simple string is needed it can be provided, enclosed by double quotes:

```
set quotation = "Eureka!"
```

If a dollar sign is needed inside the string this can be done by 'escaping' it with a backslash character. The other character that should be escaped this way is the double quote. In addition linefeeds can be put in the string by an escape character (or a backslash for systems that don't use the backslash in the file system) at the end of the line. In this last case a linefeed is put in the string and the next statement is seen as a continuation. The matching double quote should then be on this next line. In the command line a linefeed or carriage return can always be inserted after typing an escape character, or with the Ctrl-H command (p. 144)

The 'value' of a variable is always a character string. For some purposes STEDI will try to interpret this string as a number. If this turns out to be impossible an error message may be the result. Some operations can give different results, depending on whether the variables involved can be interpreted numerically (p. 121.

The contents of a variable can be inspected with the show command. This command has the syntax:

```
show variable
```

In this command the dollar sign should not be used in front of the variable. You can try this out with the statements:

```
show date
```

and

```
show $date
```

In the second case the date is substituted before the show command is executed. This means that there is a rather funny name that doesn't obey the rules for names, so an error message will be given.

There are names that have a special meaning. Most of these give the user access to internal information so that he may use it inside macro's. Others are meant to control some settings of STEDI. In addition the variables in the environment can be inspected.
The reserved names concerning the internal information are:

buffer
The number of the current buffer.

byte
The number of bytes in the file before the cursor position.

char
The current character. An empty string when the cursor is in virtual space.

column
The number of the column of the cursor in the current buffer.

cwd
The name of the current working directory. This is usually the name of the directory from which the editor was started.

date
The date.

direction
Either '<', '<<', '0', '>>' or '>' for the search/replace direction modes.

filename
The name of the file in the current buffer. Only its local name is considered.

fold
The name of the current fold if the current fold would be closed.

fullname
The full name of the file in the current buffer. This name includes the path name.

insertmode
Either an 'I' or an 'O' for the insert or overstrike modes.

isfold
This variable indicates whether the cursor is in a closed fold line.

ismark
The value ON indicates that there is a mark. If there is no mark the value is OFF.

key
The use of this variable causes STEDI to wait for a character from the keyboard. The character is presented as a string of 8 hexadecimal digits in the same notation as the key that is entered in the text after a ctrl-K.

lastmess
The last message that was displayed in the message line. This could be used for analysis to make the editor jump to an error message of the macro processor.

line
The number of the current line in the current buffer.

linechars
The number of characters in the current line.

maxcol
The number of the column if the cursor were to be moved to the end of the current line.

nextatt
The file attribute of the file that was obtained after $nextfile has returned the value true. See also p. 141.

nextdate
Gives the date and time of the last file that was found with $nextfile. The format is yyyy/mm/dd-hh:mm:ss to allow for lexicographic sorting of dates. See also p. 141.

nextfile
When this variable is read it returns true (the digit 1) if in the file search that was initiated with a 'first' command a new file was found. Each use of $nextfile tries to find a new file. If no new file is found the value false (a digit 0) is returned. The information about the file can be obtained with the variables nextatt, nextdate, nextname and nextsize. See also p. 141.

nextfname
Gives the name of the last file that was found with $nextfile. If the pattern given the the 'first' statement contained any path information it is included in the name. See also p. 141.

nextname
Gives the name of the last file that was found with $nextfile. There is no path information in this name. See also p. 141.

nextsize
Gives the size of the last file that was found with $nextfile. See also p. 141.

numchar
The number of characters left of the cursor.

range
This variable contains the characters of the current line that are between the column of the mark and the column of the cursor. It doesn't matter whether the mark is in the same line as the cursor. Only its column position is relevant.

returncode
The string that was set in the last return statement that the editor encountered. If a return statement doesn't mention a return code or when there is no return code the string is empty.

screenline
The number of the line on the screen in which the cursor is.

searchmode
Either an 'S' or an 'N' for the case sensitive or case non sensitive search mode.

shell
Indicates whether there is a command shell present that can accept commands via the ! command. On the PC-like computers this means that the environment variable COMSPEC has been set.

totlines
The total number of lines in the current buffer.

word
The current word.

wordwrap
The size of the word wrap. This will be an empty string when the word wrap is either off or in the fortran mode.

writemode
: The character that indicates in which mode STEDI would write the contents of the current buffer it it would be written (A, P, R or U).

wword
: The current word in the wordwrap sense. This means a word that is enclosed by the white space characters blank, tab or ASCII zero, rather than the more sophisticated definition for the variable 'word'.

yankbuf
: Indicates the current yank buffer with either 'Y' or 'y'.

The above variables should not be used in the left side of a 'set' command. If the user tries to set such a variable a new variable with this name is made and the old meaning of it is lost until the variable is removed again. One can experiment with the above variables by using the show command as in

```
show cwd
```

to see what the current directory is.

The variables that are meant to control some of the settings of STEDI are:

autoclose
: Value is "on" or "off". determines whether the autoclose facility is used when files are read or written.

backup
: The character of the current backup mode (b, B, V or a ! indicating that no backup is made).

bios
: (PC and Atari-ST) The value "on" forces screen output to be written via the BIOS routines of the operating system. The value "off" causes STEDI to use its own routines which are much faster.

color
: Indicating whether the colors of the text screen and the message line should be exchanged. Values are "on" or "off". See also the Alt-C command (p. 135).

dirty
: Value is "on" or "off", depending on whether the dirty flag is on or off.

The dirty flag indicates whether the file has been modified. It is displayed as one of the status characters.

hstep
The stepsize for horizontal scrolling in the text buffers.

maxhist
The maximum number of lines in the command line history.

menuspaced
Determines whether only every odd position in the menu bar is sensitive to clicks.

mstep
The stepsize for scrolling in the command/message line.

mversion
Variable to be used for keeping track of which mouse menu is currently visible.

numbers
Sets the updating of the line number in the status line "on" or "off". This is mainly for terminal connections. For those the regular updates of the line number can mean a significant slowdown.

pagedelay
This variable has a numeric value which is the number of milliseconds that STEDI will take at least for drawing one screen when scrolling with shift-up or shift-down (page-up or page-down).

waitflag
Sets the wait flag on or off. This flags determines whether after the execution of an external program or a call to shell STEDI will wait for a key to be pressed. This waiting avoids disturbing the screen before the user has read it. See also the chapter on executing an external command p. 132.

<char>
The character can be any single character. The value assigned is "on", "off" or "single". When "on" it forces the indicated character to be interpreted as belonging to words. If the value is "off" it will never be

seen as part of a word and the value "single" makes the character into a single character word. See also the chapter on Word-oriented commands p. 97.

These variables can be set and their settings can be stored in the default file (with the exception of 'dirty').

If a variable is used which has not been defined by the user, and whose name isn't one of the reserved names STEDI will inspect the environment (p. 168). If there is an environment variable of which the name matches the name of the searched for variable in a case insensitive way the return value will be the contents of the environment variable. This way one can for instance test for the setting of the environment variable STEDIMAC etc.

The number of variables that can be used in STEDI is limited only by the size of the available memory. The same mechanism that is used to store the lines in the text is also used to store the variables. This has an advantage and a disadvantage: The advantage is that no fixed size buffers for names have to be allocated at startup (so that space isn't lost). The disadvantage is that it is impossible to do a binary or hashed search for a name, so that when there are very many variables searching for a name may become slow (everything is relative).

Macro's

The possibility to execute little 'editor programs' can make an editor into a really powerful tool. In STEDI there are already many built in commands that the user would have to define in terms of macro's if he would be working with another editor. Yet the implementation of more and more commands can make an editor so large that there is no memory left to use it. Therefore also STEDI is equipped with a complete macro processor that can execute user defined procedures that may use parameters, variables and control flow. Those macro's may either be loaded in memory or reside on disk. There are several possibilities that are attempted in locating a macro when the user decides to invoke one.

The macro's that are located fastest are the ones that are stored inside the default file. These macro's are loaded at the startup of STEDI only. If another default file is read later neither the key redefinitions nor the macro's are replaced by those of the new default file. Macro's can also be loaded during a session. The command

```
mc name
```

creates a macro with the given name. The contents of the macro are a copy of the current buffer. When the default file is written also these loaded macro's are put into it. To make this scheme complete there is also the

```
md name
```

command that deletes the given macro from the buffer with the loaded macro's. It is not allowed to make more than one macro with the same name. The old macro has to be deleted first. In addition to the above two commands there is a command that allows the user to see which macro's are currently loaded. The command

```
mv
```

makes a list of all available macro's. This list is put in the current text buffer (it has to go somewhere). If this would upset the current buffer too much one could first go to one of the yank buffers (buffer 9 or buffer 0, see p. 59) and then give the mv (= macro view) command. The command

```
mv name
```

will list the contents of the macro with the given name into the current buffer, provided the macro can be located.

If the user asks for a macro to be executed and the macro cannot be located inside the above buffers STEDI will look inside the current directory for a file with the given name and an extension '.mac'. If such a file is located it will be loaded and executed. After its execution it is removed from memory again. A file system which possesses a good caching mechanism will still give a rather fast performance, even when one macro is calling another macro from disk many times. Finally, if such a file cannot be located in the current directory it is also looked for in the macro directory. This directory is indicated either by the path name after the -d option at startup (p. 167) or by the contents of the 'STEDIMAC' environment variable (p. 168).

On those computers which have file systems that allow only capitals in the names of their files (MS-DOS, Atari-ST) STEDI will do the search for macro's in a case insensitive way. On those computers that allow case sensitivity in their file names this restriction doesn't exist. In principle only alphanumeric characters should be used for the names of the macro's although this may differ from one system to the other.

The statement with which to execute macro's is the X statement in the command line :

```
X name arguments separated by blanks
```

will execute the macro indicated by 'name' with four arguments. The first argument is the string "arguments", the second one is the string "separated" etc. These arguments can be used as variables inside the macro. These variables are referred to by a dollar sign followed by a single digit. The digit may be enclosed inside curly brackets. The arguments are referred to by their order of occurrence. The first argument is `$1` etc. There can be no more than nine arguments this way. Arguments that are used but that were not provided in the call to the macro are taken as empty strings. The special object `$0` is the name of the macro itself.

If, during the execution of a macro, an error condition is encountered, its execution is halted and the appropriate error message is printed. In addition the number of the line in which this error occurred is appended to the message. This number is between parentheses. If it happens that the offending line is in a macro that was called from another macro the parent macro is also stopped, so also its line number will be appended. This gives a full tracing of how the execution got to the point that gave the offending condition. It could be that the user would like to intercept an error condition, so that he may clean up a partially finished operation. If a macro contains the label statement (see

below)

```
label onerror
```

the editor will go to the first statement after this statement, rather than returning immediately. Such a transfer of control cannot take place when the error is against the syntax on one of the flow control statements given later in this chapter. The label will be used only once in a given macro. If a second error occurs the macro will be exited.

The command to enter text from a macro into the current buffer is the 'double quote' command (p. 19). A command that consists of a string enclosed by double quotes, will put the string in the text at the position of the cursor.

Any line that starts with either the character # or the character * is considered to contain commentary and is skipped. When a macro is loaded and it contains a closed fold the fold is loaded in an opened form. This means that the statements inside a closed fold are also executed.

19.1 Operators

The syntax of the expressions is rather peculiar. All variables in STEDI are in principle string variables. This means that they are stored as character strings and only when the need is there they may be interpreted as numbers or logical variables. The same variable may be interpreted differently, depending on the operations that are applied to it. Note that the value false corresponds either to the digit zero, or to a string with only blanks and/or tabs or to an empty string. All other strings result in the value true if their logical value is asked for. The operations are in the order in which they take precedence:

!

This is a unary operator. It precedes a logical object and makes it into its logical complement. A logical object is the character '1' for **true** or the character '0' for **false**. Any other string is first converted according to the rule that an empty string or strings that contain only blanks, tabs and/or zeroes represent the logical value false. All other strings represent true.

strlen

A unary operator giving the length of the string following it. It can be very handy when a string has to be processed character by character. For an example see the section on binary editing p. 146.

toupper
: Converts the object after it to upper case, according to the built in tables. If the native display fonts contain alphabetic characters belonging to national character sets they may be changed too.

tolower
: Converts the object after it to lower case, according to the built in tables. If the native display fonts contain alphabetic characters belonging to national character sets they may be changed too.

*
: A numerical multiplication. The objects to the left and the right of this operator must be readable as a number or an error message will be printed and the execution will be halted.

/
: A numerical division for numerical objects only. This command has the same precedence as $*$. The evaluation is strictly from left to right.

%
: The remainder after the division of the object left of the `%` sign by the object to the right of it.

+
: For numerical objects this is the regular addition. If either one of the objects cannot be interpreted as a number it is a string concatenation. It can also be interpreted as a unary plus sign.

−
: A regular subtraction for numerical objects only. It can also be interpreted as a unary minus sign.

:>
: This operator needs as its left argument a string and as its right argument a number. It shortens the string from the left by the given number of characters. So only the right most characters are left. It is referred to as 'string take right'.

<:
: Again the first argument is a string and the second a number. Now the characters are taken away from the right. It is referred to as 'string take left'.

$=\wedge$

The two arguments are interpreted as strings. The first string is scanned from the left to see whether it contains the second string. If so all the characters to the left of this occurrence and the occurrence itself are removed. So only the characters to the right of this match are left. It is called 'string strip left'.

$\wedge^{=}$

Same as the operation before but now the matching is done from the right and only the leftmost characters are left. If there is no match the resulting string is identical to the first string. It is called 'string strip right'.

>>

The left argument is a string and the right argument should be a number. The leaves the rightmost indicated number of characters of the string. It is referred to as 'string make right'.

<<

The left argument is a string and the right argument should be a number. The leaves the leftmost indicated number of characters of the string. It is referred to as 'string make left'.

~

This is a string concatenation. The result of this operation is a string take consists of the combination of the contents of its left and right arguments.

//

This is another notation for string concatenation.

!=

This is a logical operator. Its arguments are interpreted as strings and the result is either true when the objects are unequal and false when they are equal in a lexicographic sense.

==

The logical equal operator: result is true is the objects are equal as strings (in lexicographic sense).

>

If both the arguments are numeric the comparison is a numeric comparison. Otherwise the arguments are compared in a lexicographic sense. If

the first is greater than the second the result is true.

<

If both the arguments are numeric the comparison is a numeric comparison. Otherwise the arguments are compared in a lexicographic sense. If the first is less than the second the result is true.

>=

If both the arguments are numeric the comparison is a numeric comparison. Otherwise the arguments are compared in a lexicographic sense. If the first is greater than or equal to the second the result is true.

<=

If both the arguments are numeric the comparison is a numeric comparison. Otherwise the arguments are compared in a lexicographic sense. If the first is less than or equal to the second the result is true.

> , < , >= , <=

Same as >, <, >= and <= but now the compare is forced to be lexicographic.

&&

This operator interprets its arguments as logical objects. If both are true the result is true. Otherwise the result is false. This is the 'and' operation.

||

This is the logical 'if' operation. If either of the arguments is true the result is true.

It could be that in the future more operations are added to this list. It is allowed to use parentheses to group subexpressions. An expression may continue over more than one line. In that case the end of a line that is to be continued should be formed either by a backslash character or an <escape> character. On some systems the backslash cannot be used to 'escape' the end of line, because this would interfere too much with the path name conventions of the local file system. In that case only the escape character may be used to 'escape' the end of line. This escape character can be typed in with the sequence 'ctrl-H escape' The above continuation may not occur in the middle of a name or between the characters that indicate an operation. Note that for numerical purposes the empty string is interpreted as zero. When the user needs to introduce a string

there are two possibilities: In the first the string is enclosed between double quotes as in "string". The backslash or the <escape> characters can be used to 'escape' characters like the double quote itself a dollar sign or a linefeed. In the second possibility the string starts with an alphabetic 'word character' and runs till the first character that is not part of a word. Excluded from this possibility are all those words that are interpreted as operators, so it is best to always use the notation with the double quotes. Numbers may be entered as regular numbers. They are always interpreted as decimal numbers.

19.2 Flow control

A good macro language needs of course flow control. The statements that take care of the flow control are:

while/endwhile
: The word 'while' should be followed by a blank and then an expression that is formed according to the above rules for expressions. As long as this expression evaluates into true the statements between the 'while' and its matching 'endwhile' statement will be executed, otherwise execution continues after the matching 'endwhile' statement.

if
: The if should be followed by one or more blank spaces and then an expression. The rules for this expression are identical to the rules for the expression in the while statement. If the expression has the logical value true the statements after the if are executed. If the expression has the value false execution continues either after a matching else or after a matching endif.

else
: This statement is used together with an if and an endif statement. When execution reaches an else statement (without having been sent there directly from an if statement) the statements between the else and the matching endif are skipped. When execution reaches the else because of a 'false' expression in an if statement the statements between the else and the endif will be executed.

endif
: Needed to terminate a range of statements that come with an if statement. The occurrence of an if statement without a matching endif statement

is a fatal error: the execution of the macro will be stopped. An endif statement that is superfluous is ignored.

goto
This should be followed by the name of a label. Control is passed to the statement after the label. If the label is not found execution of the macro will be halted and an error message will be given.

label
This should be followed by a name which is then interpreted as the name of the label.

return 'returncode'
This statement causes the termination of the current macro. This doesn't generate an error condition as abnormal termination of a macro would do (like when running into a syntax error). On the other hand whatever comes after the word 'return' is seen as a return code. It may be a complete expression as in the right hand side of a 'set' command. The result of this expression is put in a dedicated variable that goes by the name returncode. If no return statement is used the returncode will be set to an empty string when the macro is finished. The variable returncode can be used to send information to the parent process. See also p. 115.

19.3 The 'first' command

A command that has very close connections with the macro's is the command 'first'. Its syntax is:

```
first pattern
```

in which the pattern should be acceptable to the local file system. This statement returns no value. After this command has been given the various files that match this pattern can be obtained with the successive use of the variable $nextfile. This variable returns the logical values true and false depending on whether a new file was found. The information about a file can then be obtained with the variables $nextatt, $nextdate, $nextname and $nextsize (see p. 114). This is shown in the following example:

```
home
first $1
```

```
while $nextfile
set i = ( $nextname // "    " ) << 12
set j = ( " " // $nextsize ) >> 7
"$i  $nextatt  $j  $nextdate\n"
endwhile
```

This macro expects one argument. This argument is used as a pattern in a 'first' command. After this each file that is found in the successive use of $nextfile has its name, attributes, size and date put in the current buffer in a nice format. This formatting works in a rather simple manner. In the variable i the name is concatenated with a string of 12 blanks. After this only the leftmost 12 characters are taken from this string. The command `"$i  $j  $k  $nextdate\n"` puts all variables and some intermediate blanks as text in the current buffer and the `\n` indicates that a newline should be started after this.

Stream editing

A stream editor is an editor that processes a file taking its instructions from another file. This can be very useful when a fixed editing task has to be used regularly, like the conversion of one dialect of a language to another. STEDI provides this facility in three ways. The first way involves the macro facility. This can be looked up in the chapter on macro's. It is by far the most useful facility of the ones mentioned here. The second way is given by the I command in which the I stands for 'input'. When the I# command is issued from the command line (# should be a digit in the range of 1-8) STEDI treats the lines in the corresponding buffer as input for the command line. It starts at the 'current line' in this buffer (the line in which the cursor is when the user switches to this buffer) and proceeds executing statements either till there are no more lines, till a search with the . option is not successful or till a command results in a fatal error condition. There are several restrictions that should be taken into account:

- Any line that starts with either a $*$ or a # is skipped. It is seen as commentary.
- Neither a fold line nor the lines within a closed fold are executed.
- Only the first 255 characters of a command can be copied to the command line.
- If a line ends with an <escape> character or a backslash (only on systems for which the backslash has no special function in the file system) it is assumed that the carriage return that followed it has been interpreted wrong, with the result that the command is now spread over more than one line. Therefore a <return> character is added and the contents of the next line are considered to be part of the same command.
- Commands that are executed from a stream are not entered in the command history. The last command in the command history will be the input command itself.

It can happen occasionally that the user may want to abort a stream command when he sees that things are not proceeding as planned. This can be done by pressing both shift keys simultaneously. STEDI will then halt the execution of the input command. The cursor in the stream file is left in the line with the next statement that would be executed if the error condition had not occurred. This is different from the abort after a 'fatal' error condition. In

that case the cursor is left in the line that caused the error. In either case it is rather easy to proceed execution as a new input command will start execution at the line of the cursor. A good example of such a condition would be a lengthy set of actions that is interrupted due to lack of memory. During a stream action STEDI doesn't execute garbage collections, unless forced to do so with the 'garbage' command, so one second pause after such an interrupt may be sufficient to clean up the memory, after which one can continue by using the Ctrl-R command.

The example below is a stream program that removes all lines that contain the string 'Remove me' from a file. The file is supposed to reside in buffer 1 and the stream file should be in buffer two:

```
 u-
*    This turns off the screen updates
 alt-1
*    Go to buffer 1.
 /Remove me/>.
*    Search forward for the string.
*    The period option makes the stream stop
*    when the string is not found any more.
 deleteline
*    delete this line.
 alt-2
*    Go to buffer 2 (the buffer with the stream!)
 home
*    Go to line 1. This means that the next line
*    to be executed is line 2.
```

Of course the above could be obtained easier with a macro. The I command allows however a few possibilities that the macro's cannot offer. One is that a stream script can modify itself. In rare occasions this can be handy (but don't complain when you get in trouble). The other more useful feature is that because a stream script isn't a macro the variables that are defined in it will not be removed after the script has terminated. This means that it offers the opportunity to set a number of variables that the user would like to have around. Note however that if the stream script is called from a macro all its variables will disappear again when the macro is finished.

If stream files invoke themselves or other stream files in a recursive fashion a crash may result if the algorithm that was used doesn't terminate in time. There should however be enough space for at least 16 recursions. This may depend somewhat on the instructions that are used at the deepest levels.

No attempts have been made to forbid certain actions as this could restrict the user needlessly. If during a particularly complicated operation STEDI starts malfunctioning it is left to the good taste of the user to determine whether he was asking for trouble or whether a serious shortcoming of STEDI has been found. Anyway the author would like to know about it.

The third stream facility in STEDI is something that is usually interpreted as stream editing. At the startup of the editor the user indicates that a special file contains edit instructions and should be used to act upon at least one of the other files. With STEDI this is done with the -i or -x parameter in the command tail at startup. The file after this parameter is interpreted as a macro. The macro is loaded and all other arguments after the name of the macro are passed as arguments to the macro. The editor starts up, the macro is executed and the file in buffer 1 is saved. After this the editor terminates execution and returns to the command processor. Whether there will be an enormous visual display during the execution of the macro depends on whether the screen updates are switched off in the macro.

Execute an external command

Programs may be executed from within the editor. Likewise, if the editor is being run from a shell, it is possible to execute a shell command from within the editor provided the shell is equipped for such action. Under MS-DOS this means that at startup the environment variable COMSPEC must contain the name of the command line processor. This variable must be set in the file autoexec.bat (see also the chapter on running STEDI) (MS-DOS users can skip immediately to the section on 'escape to shell'). A shell program on the Atari-ST should set the systems variable _shell_p with the address of the routine that should handle such a call. People using such a shell can also skip to the section 'escape to shell'.

Programs or commands are executed from the command line, using the exclamation mark (`!`) command. The syntax is:

```
!progname [command tail]
```

With this command the editor will execute the program in the file 'progname'. If the operating system has conventions for special extensions those will be honored. For instance the extensions .tos and .ttp indicate to the atari ST that the program isn't a GEM program. In that case the program is started with the mouse off and the text cursor on, while otherwise the normal cursor is turned off and the mouse is activated. A possible command tail is passed to the program in the same way as is done from MS-DOS or a regular command shell. On the Atari this corresponds to the contents of the dialog box that appears on the screen when a .ttp program is executed from the desktop. The difference from the desktop lies in the possibility to pass a command tail to any program, independently of the extension in the program name. The environment string passed to this new program is a copy of the string that was passed to the editor when it was started up.

This command can be particularly useful when running from a floppy disk. In the cycle (1) entering the editor, (2) reading a file, (3) writing the file after editing, (4) compiling, (5) make a test run, one can save the first two steps if the steps 4 and 5 are done from the editor.

When the external program is finished, the editor will produce a bell sound (which of course cannot be heard if the sound level is set to be very low) and wait for further action. No attempt is made to give a visual prompt, as that may overwrite an error message or other precious output. After completion

of the program, the editor flushes its input buffer before waiting for further action. Thus, it is quite safe to accidentally press a key while your program runs. This will not cause an immediate return to the editor upon completion of the program. It is also possible that one doesn't know whether the program has finished. Also in that case one may freely press any key without the risk of loosing a later error message.

When the program ends, hitting any key (with the exception of the status keys) will cause a return to the editor at the point where one was when the program began. A message in the message line will tell how long the execution of the program took from the moment that the control was given to either the operating system or the shell to load the program up to the moment the editor got the control back and started waiting for a key. A possible return code will also be displayed. Often compilers give a negative return code when an error was detected, but if the error messages were also written to the screen this is superfluous information. The more important messages are the ones from the operating system that may indicate problems such as lack of memory or that the file was not found. These errors are presented in text format while all other errors - mostly user defined - are given by number.

Sometimes the user may not like it that STEDI waits for the pressing of a key upon completion of a program. In that case he can use a variation of the `!` command in which the `!` is followed by a – sign. The minus sign indicates that there will be no waiting. It is also possible to set a flag that indicates to STEDI that there should never be any waiting. This is done with the command

```
set waitflag = off
```

When the wait flag is off one may issue a command that waits anyhow by putting a + sign after the `!`. The wait flag can be turned on again with the command

```
set waitflag = on
```

To see what the contents of the wait flag are one may use the command

```
show waitflag
```

21.1 Escape to shell

When a shell program has been used to start up the editor, and the shell program has taken the appropriate measures necessary to allow the editor to tap into its command capabilities, the editor will detect this. Then the editor

will not submit the program itself but pass the combination of progname + command tail as a single string to the shell for further processing. Settings concerning a mouse and/or the cursor are restored as much as possible to what STEDI found at startup. After the command has been executed STEDI will test its environment and reinitialize it completely if the need arises. It is clear that such an interaction with a shell will greatly enhance the power of the ! statements as full use can be made of the possibilities of the shell (like aliasing, shell scripts, search paths, etc.). On MS-DOS and UNIX this is rather standard. An example of a shell that does this on the Atari according to the standard is the GPSHELL . Under MS-DOS the escape to shell can only be done if the environment variable COMSPEC points to the command shell to be used. This is usually the program 'COMMAND.COM'. The environment variable should have been set by including the following line in the file AUTOEXEC.BAT:

```
set COMSPEC=c:\command.com
```

if the file COMMAND.COM is located in the root directory of the drive c. If the file is to be found elsewhere this should be reflected in the value of COMSPEC.

The issuing of external commands that are rather complicated can be made much easier in combination with either the key redefinitions, the learn mode or the macro's. One could for instance make the simple macro:

```
save
set name = $filename ^= "."
!-cc -o $name.exe -iD:\include -DDEBUG -C $name.c
!$name.exe
```

If this macro is called cc (or sits in the file cc.mac) the command

```
x cc
```

will save the contents of the current buffer, create the variable name as the basename of the file without its extension, have the shell call cc with the proper parameters so that the new file is compiled. If the compilation is successful the program is executed (on the atari the extension should be .prg or .ttp). This can give a very quick turnaround during debugging. If output redirection is possible one could have compilation errors caught in a file and have another macro to load this file, read in which line the first error was found and jump to this line. This second macro would depend on how the compiler produces its error messages.

It is also possible to bind the execution of the macro 'cc' to a single key like Ctrl-P via the key redefinitions. The whole execution of the program in

the current buffer would then be reduced to pressing one key combination! Those people who like to work with the mouse could tie the execution of such a macro to an item in the mouse menu. How this can be done is discussed in the chapters on the mouse p. 92 and the key redefinitions p. 164.

Screen control

STEDI has a number of commands to control the appearance of the screen. Some commands control the color of the screen, others allow the user to simultaneously display two files, and again another command can show the hexadecimal representation of all characters in a buffer. These commands are treated in order.

22.1 Screen color

The color of the screen is by default black and white. Depending on the computer it may be white characters on a black background, or black characters on a white background (if the main colors of your screen happen to be black and white). The command line is represented in the inverse colors. The simplest color control is with the Alt-C command. It exchanges the color attributes of the message line and the text screen. This is remembered as a 'toggle' command. It is written in the default file with the DW command (p. 52) and upon startup STEDI sees whether it should flip the color of the screen. On some systems it is possible to ask for the color of the screen, in which case an attempt is made to fix this color exchange in an absolute sense. On other systems this parameter doesn't involve an absolute color.

For some color systems there is a more flexible way of controlling the color. The commands

```
c1 color1 color2
c2 color1 color2
```

sets the colors for the text screen (c1) and the message line (c2) respectively. The first color is the foreground color i.e., the color of the characters, while the second color is the color of the background. On the computers in the PC family the foreground color can be a number in the range of 0 to 15 (4 bits in total) while the background color can be a number in the range of 0 to 7 (three bits). The meaning of the bits in these numbers are:

bit 0 If set the color blue is on.
bit 1 If set the color green is on.
bit 2 If set the color red is on.
bit 3 If set the high intensity attribute is on.

So the commands

```
c1 7 0
c2 1 7
```

sets the text screen to white characters on a black background and the message screen has blue characters on a white background.

All the above variables are stored in the default file when it is written.

22.2 Split screen

Sometimes it can be quite handy to show two files on the screen next to each other. Some editors carry this idea even further, but stamp sized windows are rarely very useful. The commands that control the splitting of the screen are mostly given from the command line:

F

This restores the screen to the representation of a single buffer. The buffer which was the 'current buffer', which means that it had the cursor in it, will be the buffer that is shown.

FH

The screen is split horizontally. The message line moves up a number of lines to serve as a divide between the two windows. If the editor was in the single window representation the top window will show the old buffer and the bottom window will show the buffer whose number is one higher. For purpose of counting buffer 1 comes after buffer 8. The split between the buffers will be about even.

FV

The screen is split vertically. The message line stays at the bottom of the screen. If the editor was in the single window representation the left window will show the old buffer and the right window will show the buffer whose number is one higher. For purpose of counting buffer 1 comes after buffer 8. The split between the buffers will be about even.

FH#

Same as the FH command, but the top window will now have the specified number of lines (if possible).

FV#
: Same as the FV command, but the left window will now have the specified number of columns (if possible).

F+#
: This moves the divide between the windows down or to the right by the specified amount.

F-#
: This moves the divide between the windows up or to the left by the specified amount.

After the screen has been split there are various ways to go from one buffer to another. If one wants to go to the other window this can be done either by the **Alt-F** key combination, by clicking the mouse (if any) in it, or by the Alt-number combination in which the number is the number of the other buffer. If the number is the number of a buffer that isn't on display at the moment, the window with the cursor will change to that buffer. So there is no need to have the two windows show buffers with sequential numbers.

The mouse cursor can move freely from one window to another. Only when a button is clicked will the editor determine in which window this was and then make that the current window and take the appropriate action for it.

There are no provisions for having both windows display parts of the same buffer.

22.3 Special representations

Sometimes it would be nice to see what the hexadecimal value of the characters in a buffer is. This is mainly so for the editing of binary files. This can be done with the **Alt-H** key combination. It is described in full in the chapter on hex codes on p. 145.

Another special screen representation can be obtained with the Alt-T key combination. It is meant mainly to inspect in files what the true nature is of all blank spaces on the screen (blanks, tabs, ASCII zero, no character at all). It is described in full in the chapter on the tabs on p. 96.

The sort command

When preparing lists it can be necessary that the list is in a given order. To sort a long list by hand can be quite time consuming, and a macro could do the job, but also it will be very slow. Therefore STEDI has been equipped with a sort command. There are several options to make the command rather practical. The main command is the command O (for order, the s has been taken already by save and set) from the command line. If this is the whole command, the whole buffer is ordered in a lexicographic way. When this is done the first time there is usually a big scare: All the empty lines come first, so it is most likely that the user gets a blank screen in front of him. The more interesting things are usually near the end of the file. The various commands are:

O
: The whole buffer is sorted on a line by line basis.

O#1,#2
: The whole buffer is sorted, but from each line only the **character** ranges indicated by the given numbers (inclusive) are considered for determining which line comes first.

O:#1,#2
: The whole buffer is sorted, but from each line only the **column** ranges indicated by the given numbers (inclusive) are considered for determining which line comes first.

OF#[**char**]
: The whole buffer is sorted, but from each line only the the field indicated by the given number is considered when comparing two lines. Fields are separated by field separators. The default field separator is a comma, but if the user prefers a different field separator he can specify it after the number (so 'char' is optional). For changing the order of the fields one should either use regular expression replacements, or make a more sophisticated macro.

OFN#[**char**]
: This command is as the above, but before comparing the indicated fields STEDI tries to interpret the fields numerically. If both fields are numbers

in the range of -2^{31} to $2^{31}-1$ the compare will be arithmetic, rather than lexicographic. (This allows one to sort lists of number without getting 10 to come before 9).

OR...
The R should be before the above options. It indicates that only the range from mark to cursor will be sorted. Only whole lines are considered, so the lines with the mark and the cursor are included entirely in the sort.

O<...
The less than sign should be the first character after the O. All other options may follow it. This indicates that the ordering will be backwards (largest comes first).

Closed folds are taken along in the sort as if they were a single line. No attempt is made to look inside the closed fold.

Example: We would like to sort the closed folds inside a buffer, but the folds have different characters for their first three characters (but no hash sign(#)). The command to give is then:

```
OF3#
```

Here we indicate that the field separator is the hash sign, and that we are interested in the third field (the first field is made up by the first three characters, and the second field is the empty field between the two hash marks). If a field doesn't exist, as may be the case in lines that aren't closed fold lines, it is considered to be empty.

23.1 About the algorithm

The sort algorithm that was selected is a recursive merge sort. This sort needs a time which is strictly proportional to $n\,^2\log n$ and has, unlike quicksort, no bad behaviour for special cases. As often the compare of two lines can be rather slow (when comparing ranges of columns) it is important to minimize the number of compares, and also here a merge sort wins (quicksort has less overhead and therefore it can win in speed when the compare is trivial). The main drawback of the merge sort is that it needs some extra memory. In the case of STEDI this is roughly 1.5 pointers per line to be sorted. This means that when there is no free memory the sort command cannot be executed. STEDI will give the message "no memory" and execution is aborted (for macro's).

Comparing ranges of columns is rather slow, because each time two lines are compared they have to be brought to screen representation. This isn't necessary for the other modes. It is impractical to work out all the lines first and then compare them. This could pose enormous memory requirements. If the user would like to speed up such sorting he could expand all tabs before the sorting command is given, do the sorting in the character mode, and then put the tabs back in. This can be done provided that the tabs are 'trivial' (no essential blanks inside tabs as is sometimes the case with the first three characters in the fold lines, no strings of blanks inside character strings that have to be printed to the screen, etc.).

Miscellaneous commands

24.1 Date

Often it is needed to put the current date in a file. At later times it will then be clear in what order parts of a program were made. This can be an invaluable tool for maintaining complicated software. The command <**ctrl**> – (control key and minus sign) puts the date in the text at the current position of the cursor. The date is given in the European format but with the month represented by three characters to avoid all misunderstanding. The date 23-jan-1990 was put in the text with the <ctrl> – command.

24.2 Display last message

Sometimes a message is removed from the screen before the user has realized that there was an error message. In that case the last error message can be recalled with the **Alt-M** key combination. This will display this message.

24.3 File searches

There is a special command to allow the user to work his way through all files in the file system that match a given pattern (like in MS-DOS or UNIX the pattern *.c). Its syntax is

```
first pattern
```

This command sets up all the internal variables for the search. After this each time the value of the variable 'nextfile' is asked the next file will be looked for. The first file should be looked for also with the reading of 'nextfile'. If its value is 1 (= true) there is another file and its properties can be found in the variables nextname (the name), nextatt (the attributes), nextdate (the date and time) and nextsize (the size in bytes). The attributes are presented in terms of a string of six characters. The meaning of the characters is (from the left to the right) format:

a The object has been modified since the last archiving operation.

d The object is a subdirectory.

v The object is a volume label.

s The object is a systems file.

h The object is a hidden file.

w The user has writing rights to this object.

If a character isn't apllicable it is replaced by the character −. A regular file is usually represented by the string 'a − − − − w' indicating that the user may write to it and that it has been modified recently. The date is presented in such a format that it can be sorted in a lexicographic way. For an example, see p. 114.

24.4 Garbage collections

Normally garbage collections (the rearranging of the contents of the buffers to minimize the use of memory) are executed when STEDI waits for input. This way the user may never notice these garbage collections. There are two exceptions to this rule. The first is rather passive: When a file has all its tabs expanded into blank spaces it may be necessary for STEDI to execute a garbage collection during this expansion. In that case the message 'No more memory' appears but STEDI doesn't give the control back to the user. Instead it executes a full garbage collection. When this is done the message disappears and STEDI continues with the expansion of the tabs.

The second exception is more active. When macro's or stream scripts are run there is no waiting between the commands. This gives STEDI no chance to rearrange the memory while waiting for input from the user. If the macro (or stream script) involves instructions that use the memory in a rather fragmenting way eventually the message 'No more memory' may appear and STEDI would stop the execution of the macro. To avoid this the user can force a full garbage collection by the command 'garbage' in the command line (or as one of the statements in his macro or stream script.

24.5 Message

The command

```
message "string"
```

puts the given string in the message line, using the regular message mechanism of STEDI. This command can be handy for macro's.

Sometimes a message goes by too fast. If one would like to see the last message after it disappeared the key combination **Alt-M** will make the most recent message reappear.

24.6 Pause

To debug a particularly difficult macro (or stream script) one can insert the 'pause' command. This command is like an ordinary command from the command line. It expects a single parameter which should be a positive number. Internally this number is multiplied by 100 to obtain the number of milliseconds that STEDI will wait before continuing execution. Example:

```
pause 10
```

makes STEDI wait for one second before continuing.

Hex code

STEDI allows some rather extensive possibilities for entering special characters into the text directly via hex code or with the search and replace commands. To enter characters in the text by their hexadecimal (ASCII) code, one may use the **Ctrl-H** command. This command has two variations :

1. If the Ctrl-H is followed by two hexadecimal digits, the corresponding code between 00 and FF is entered as a single character. It will be shown on the screen as the pixel image in the current font that belongs to that code. For example, a formfeed would be entered by Ctrl-H followed by 0c.

2. If the Ctrl-H is followed by another control-character combination (or escape or backspace or return), the ASCII code of this character is entered in the text. It will be represented on the screen by the corresponding pixel image in the current font. An example is Ctrl-H followed by Ctrl-L for a formfeed.

The ctrl-H command is sometimes useful for making a file with control sequences by which a printer can be set in a selected mode. Note however that if the hexadecimal code 0a (linefeed) or 0d (return) is entered in the text, it will be read as a linefeed next time the file is read. It would be very difficult to edit such a file.

If the Ctrl-H is followed by a character that doesn't fulfil the above requirements, the Ctrl-H is canceled and the character is put in the text as if the Ctrl-H was never pressed. If there was already a single hexadecimal digit in the input buffer, it will be put in the text first. One can see whether the hex mode is active by the representation of the dirty bit in the status line. The little open circle will be replaced by the same little closed circle that is used for representing tabs if Ctrl-H has been pressed and the editor is waiting for a hexadecimal or control character.

This hex mode command can also be used in the command line and it can even be used to put special characters in file names. It cannot be used however to 'escape' a slash (/) in the search command. For this purpose, there is a second way to enter special characters that is only valid for the command line. In this case the character 'escape' fulfils a role similar to the Ctrl-H, except for that any interpretations of the character that it 'caught' are made during the execution of the command. After typing an escape in the command line,

the next character is taken 'at face value', even if it is a backspace. If two hexadecimal digits follow the escape, the whole of escape and the two digits is interpreted internally as a single character. The search routines will interprete escape followed by a slash (/) as the character slash (/) and will not interprete it as the delimiter of a search or replacement string. Instead of the escape character one may also use the backslash character. The difference is that this backslash character doesn't allow the user to type in special characters like a backspace immediately after it. This would have to be done again with the Ctrl-H command.

The main difference between these modes is the moment of translation. The Ctrl-H is translated immediately, the sequence with the escape key is interpreted later (during the search/replace) or not at all.

As in normal text editing, it is sometimes convenient to know what characters actually are present in a file if the ones that just appear blank on the screen. The **Alt-T** key allows you to tell what characters are actually there. The possible characters that are displayed on the screen as blanks are blanks themselves, tabs, and the null ASCII character. On IBM-PC-like computers also the character number 255 is represented by a blank space. In addition if there is no character at all to be displayed, the screen remains blank. After pressing the Alt-T, each of these 'white space' characters are given a unique representation. Blanks are displayed as small empty circles which are superscripted relative to other characters. Tabs are represented as small filled circles, also superscripted. Since a tab can represent several spaces depending on tab stops, the intermediate spaces in the range of a tab are not real characters and they remain blank. Finally the null ASCII character is given the representation of a small filled circle which is subscripted, so as to be able to distinguish it from a tab. When the 255-th character has to be presented this way it is shown as a colon. When no character is present at all, this place on the screen remains blank. The Alt-T commands serves as a toggle between the mode in which all characters are given unique representations and the mode in which all 'white space' appears as blanks. On terminals the characters that are used for displaying the blanks, tabs and ASCII zeroes will be in the regular ASCII set. The blanks become a period, the tabs a greater than sign (>) and the ASCII zeroes become an underscore.

On most terminals it would not be very practical to try to present the non-ASCII characters. Even if the extended character set of the VT100 is used there aren't 256 different characters. For representing more than the standard character set the workstations and the micro computers clearly have an advantage. It is however possible to obtain the hex code of each character in a buffer by toggling the **Alt-H** key combination. Normally only characters

are presented that have a legal screen representation. On terminals the non-ASCII characters aren't considered legal characters. Sending some of them over would mess up the terminal considerably. When the Alt-H combination is pressed once the screen changes drastically: At the position of each character there is now its first hexadecimal digit. After the Alt-H combination has been used again one gets the second hexadecimal digit. Hitting the Alt-H for the third time gives the original screen back. So the capital A would look like

```
A   4   1
```

in the three representations, because the hexadecimal representation of the character A is 41.

25.1 Binary editing

Sometimes it can be very useful to edit a binary file. A very popular use is the changing of default (path)names in compilers that aren't versatile enough to pick new (path)names up from the environment. This can cause a compiler to look for some of its files in a 'more sensible' place.

A file can be read in as a binary file when the buffer is first placed in the 'raw' mode. This is done with the **Alt-R** command. In the raw mode a file is read without interpretation of the linefeeds and carriage returns in it. It is read with 64 characters per line and the display will look rather messy. Many of the bytes in a binary file correspond to non-ASCII characters. On the micro's and the workstations they can still be shown on the screen. It takes however a very experienced hacker to read the code segment of an executable file. On the other hand a study of the text strings in a program may be very revealing.

When searching for a string in a binary file there can be a problem when the screen representation of the searched for string has part of the string at the end of one line and the rest at the beginning of the next line. STEDI has put the file in the buffer, using its normal lines mechanism and it chopped it up into pieces of 64 characters. The following macro can restore such strings so that the characters that are in the next line will be moved.

```
set n = strlen $1 - 1
while $n
  set a = $1 << $n
  set b = $1 =^ $a
  //"$a"\n"$b"/=/$1\n/
  set n = $n - 1
endwhile
```

This way we search for the string 'head' end-of-line 'tail'. We take a to be n characters from the left of the argument and b is the rest of the argument. When n becomes zero we finish. The macro needs one argument, so it is called with the command

```
x reorder string
```

assuming that the macro got the name 'reorder'. The effect could be that

```
abcdstr
ing0123
```

is changed into

```
abcdstring
0123
```

It is absolutely no problem that the length of the lines has been changed now. After this macro has been executed one can either search for the string in the normal way, or try to replace it. It is actually rather easy to change the above macro, so that it uses two arguments and makes the replacement immediately. If you don't understand the above macro's you should consult the chapter on macro's p. 119.

One reminder: when changing a binary file, make sure not to add characters between code segments. This can upset all kinds of offsets, causing the program to crash (if you are lucky, because worse things can happen).

Keyboard transformations

The editor has a very flexible capability of redefining the keyboard layout. These keyboard redefinitions are not only for rearranging the keys with which the commands are executed, but whole sequences can be assigned to a single key. This allows the user to define his own powerful and custom made commands. For example, if the user prefers to use different keys than the default ones for particular actions, new keys can be assigned to the actions. Or if a particular command is not exactly what a user is used to, it can be changed to his liking. One can also access all the special characters built into the display fonts such as those for national character sets using the keyboard redefinition feature. Commands can be designed that are only activated after hitting several keys as well. This is of use when emulating another editor with which one might be familiar. Of course such a flexible environment has some strict rules. In this chapter the keyboard redefinition capabilities of STEDI are explained in detail.

The way the keyboard can be reprogrammed is somewhat involved as a special file is required which defines the new layout. Such a file can be made in the editor, but then must be translated into a binary file that in turn can be read by the editor to reconfigure the keys. An outline of what you have to do to redefine some keys of the keyboard is as follows.

First you must create a file which makes the key change assignments. The syntax of this file will be explained below but the general form of a key redefinition statement in this file is similar to an equation. On the left side a list of key codes are given, then an equals sign, and then a second list of key codes. This has the effect of assigning the actions represented by the key codes on the right side of the statement to the key strokes on the left side.

Next you must translate this file into a binary file which the editor can read. For this purpose, a utility program is provided called 'KEYCOMP' with an extension that is systems dependent (.EXE for MS-DOS, .TTP for the Atari-ST and no extension for UNIX). This program takes the key redefinition file as input, and it creates the binary file needed to be read in by the editor. Then, to make the key redefinitions become effective, the binary file must be read into the editor using the K command from the command line. This command causes STEDI to read in the file and store its contents in a special buffer. Finally, if the DW command is issued, the contents of this buffer will become part of the default file so that at a later startup the new keyboard layout will be available

again. A subsequent reading of the default file will not affect the keyboard layout. It can only be changed when the default file is read at startup time or by using the K command.

The use of KEYCOMP, the compiler for keyboard transformation programs, allows the editor to skip much error checking. Also, unpredictable results may occur if a file is read by the editor that was not made by the key compiler as a keyboard transformation file with the K command. There is some error checking and the binary files can even be exchanged between machines with a different byte ordering scheme (there are limits though). In what follows, first, the syntax of the key redefinition file will be explained. Then through a collection of examples, the key codes will be explained and the capabilities of this feature explored. Finally the 'K' command and the use of the key redefinition file compiler KEYCOMP will be explained. For those of you who can't wait to try out this feature, you may wish to skip to the end of the chapter after reading only a few examples.

26.1 Syntax

Now we turn to the syntax for redefining single keys or sequences of keys. This syntax has the following rules:

1. Each redefinition consists of a left hand side and a right hand side. The key(s) in the left hand side will be replaced by the sequence in the right hand side. The left hand side and the right hand side are separated by an equals sign (=). Both the left hand side and the right hand side must contain at least one key.

2. Each sequence of keys consists of single key codes, separated either by a plus sign (+) a comma (,), a tab or blank spaces.

3. Each redefinition is terminated by a semicolon (;).

4. Each single key code is a string of 8 hexadecimal digits. This is either a string that is an exact replica of the code that is given by STEDI's keyboard handler when a character is read out (a must for the left hand side keys) or something that looks sufficiently like such a string that the keyboard processor of the editor will accept it for further treatment (right hand side). If this last condition is not met, the editor may not recognize what you want.

5. Linefeeds and carriage returns are seen as irrelevant. This means that one redefinition can run over several lines. The limit to the length of a

redefinition is set to 509 keys. This limit comes from the buffer sizes in the program KEYCOMP only. If it turns out to be a real problem the user could change the size of these buffers and retranslate KEYCOMP. There is also a limit on the length of the whole redefinition ASCII file: the compiler must be able to read it in in a single read statement, so it must fit in memory. This should not be much of a limitation.

6. Two types of commentary are allowed. First, when the compiler runs into a colon (:) it considers any characters that follow as commentary until the terminating semicolon is reached. Second, any text between a matching pair of (any type of) brackets or parentheses is considered as commentary. The commentary isn't allowed to break up the field of the 8 digits. This commentary is very useful for noting what key a key code corresponds to. There are no escape characters inside the commentary so if this commentary is to include a ')' it cannot be enclosed between '(' and ')' but other types of brackets must be used.

7. Instead of the 8 character key codes a number of keywords is allowed. In addition single characters will also be translated into the appropriate 8 character hex code. The backslash serves as an escape character if any special character is to be used. This mode of using mnemonics and single characters is the preferred mode for writing readable key redefinition files, although sometimes one has to resort to the hex codes in the right hand side of the redefinitions and the left hand sides should nearly always use the 8 digit hex codes. We will see why.

The above specifications define the entire language used for keyboard reconfiguration. Most of the rest of this chapter is devoted to examples which will clarify exactly what the key codes needed for the redefinitions are, and how to use them. Should you want to relax some of the constraints on this language for your own personal use, the source code to the program KEYCOMP (programmed in the C language) is included on the distribution disk, so you may make your own modifications.

Once the keyboard file is stored in STEDI's buffer, whenever the editor receives a key code from the keyboard, it will work its way through the list of key redefinitions from the front to the back until it finds a complete match. No attempts are made to sort this list or to change the order in any other way. This allows for very tricky redefinitions. Of course, if no match occurs, then the normal action of that key code will be enacted. Beware though, in the case of actions assigned to multiple key strokes, the redefinition may preclude the possibility of using action assigned to a subset of such key strokes. This will be made clear in the following paragraphs. Now let's turn to some examples:

```
: Example 1: Some combinations of the left shift
  key, the alternate key and function keys are used
  to enter a number of blanks into the text;

0A000801(left-shift+alt+F1)=blank;
0A000802(left-shift+alt+F2)=blank+
                            blank: 2 blanks;
0A000803(left-shift+alt+F3)=blank+blank+blank;
0A000804(left-shift+alt+F4)=blank+blank+blank+
                            blank: 4 blanks;
: Etc. ;
```

In this first example, you see how a key redefinition statement looks. In this case, on the left hand side of each statement is just one 8 digit hexadecimal key code followed by some commentary explaining what key that code corresponds to. These codes stand for the successive function keys in combination with the left-shift and the alternate keys. On the right hand side in each case are one or more mnemonic codes which all correspond to the same key code - that for the space bar or blank space. Of course, the right hand side is much simpler than the left hand side. Typing the key codes at the left is not a very easy task, apart from the problem that one may not have documentation available containing these codes. Therefore STEDI contains a special command, the **Ctrl-K** command. When the cursor is in the text window and the Ctrl-K is pressed, the cursor will vanish and STEDI waits for the next key. The full code of the next key pressed will then be entered in the text. This holds for combination key codes using the shift, alternate and control keys as well. This key code is the code after the keyboard processor of STEDI has processed it. If for some reason or another you would like to see the raw code of a key, as given by the operating system you may type Ctrl-J, followed by the key combination of your choice. The result is now system dependent and STEDI won't recognize it any more. Sometimes it can be handy though for people who make software that is tailored to the possibilities of a specific keyboard.

The reason that we put the left hand side in terms of the cumbersome hexadecimal code is rather simple. This way it may contain full information about the shift, alternate, control and capslock keys. In addition we can make up our own keys as we will see in the sequel that there are several bits that can be used for flags and masking. At the right hand side we want to put some commands for STEDI. This is then a much better defined set of codes, hence the practicality of using the mnemonics. If on the other hand we want to do fancy things on the right hand side one can also put the hexadecimal codes there. Similarly it is possible to use mnemonics in the left hand side.

If there are several redefinitions and some of them happen to have the same left side only the first one is effective. The redefinitions are stored in the same order as you have specified them. This can have unexpected consequences with redefinitions of more than on key at the left hand side.

```
: Example 2: The order of the keys
        (What to avoid....);
04000247 (ctrl-G) + 00000072 (r) = .....
04000247 (ctrl-G) + 00000072 (r)
                  + 00000065 (e) = .....
04000247 (ctrl-G) = .....
```

Of the above left hand sides only the first one is relevant. Once Ctrl-G and an r have been typed the first redefinition will be executed. So the second redefinition never gets a chance. The third one won't do anything either: After a Ctrl-G has been pressed the editor runs into the Ctrl-G + r redefinition and decides to wait in order to see whether there will be more. After the next character there are two possibilities: (i) it is the character r and there is a match. (ii) It isn't an r and the search goes on further through the table. But because we have now two characters in the buffer it cannot match a single character sequence any more. So number three cannot match.

26.1.1 Mnemonics

Before looking at some more examples, lets first include the list of mnemonics. This list can be found also in the sources of the file KEYCOMP.C which is on the distribution disk. If you would like to introduce more mnemonics you only have to add them to the list and translate the file again. Be sure though that they are in strict lexicographic order (you can use the sort command (p. 138) if you are not sure). The rules about the hexadecimal codes are given further on in the text. All mnemonics are taken in a case insensitive way but they must be in capitals in the table.

after_read A special code for in the left hand side. Used to define action after reading a file.

alt_0 to alt_9 Like the regular commands Alt-number.

alt_a to alt_z Like the Alt-character commands.

alt_eq Code for Alt-= to jump to a matching bracket.

alt_fun0 to alt_fun9 For the PC: to terminate a learn sequence or to execute it.

backslash The backslash character (not to be used as an escape code in the redefinition file.

backspace

blank

clear Clear the current buffer.

close Close the current fold.

close_all Close all folds.

colon The colon character (avoids problems with the colon that indicates commentary in the redefinition file).

comma The comma character.

command A special entry to the command line of STEDI. The same redefinition should also contain 'endcommand'. All characters in between are seen as a command for the command line, but they won't be entered in the command history, and they won't be shown on the screen.

copy Copy the current range.

copy_block Copy the current block.

creturn A carriage return.

ctrl_0 to ctrl_9 Terminate a learn sequence or execute it.

ctrl_a to ctrl_z Like the Ctrl-character commands.

ctrl_fun1 to ctrl_fun10 For the PC: Go to the corresponding tag.

ctrl_min Put the date in the current buffer.

ct_delete Delete the current line.

ct_down Move screen up by one line, hold cursor.

ct_home Go to home position on current screen.

ct_left Scroll one page to the left.

ct_right Scroll one page to the right.

ct_up Move screen down by one line, hold cursor.

cut Cut/yank current range.

cut_block Cut/yank current block.

delete Delete the current character.

deleteline Delete the current line.

delete_mark Delete the mark.

down Move cursor down.

end Put cursor at the end of the buffer.

endcommand See the 'command' mnemonic above.

enter The enter key.

escape The escape key.

exchange Exchange the mark and the cursor.

flags000 - flags111 Set the flags to the given binary position.

flag1on-flag3on Turn one flag on.

flag1off-flag3off Turn one flag off.

fun1 to fun10 The first ten function keys.

fun11 to fun20 The first ten function keys, together with a shift key.

goto_mark Go to the mark.

help Same as pressing the help key (shift-F1 on PC).

home Put cursor at the beginning of the buffer

insert Insert a new line.

learn0-learn10 Terminate a learn sequence or execute it.

left Move the cursor to the left.

linefeed Generate a linefeed code (has usually the same effect as a carriage return).

mark Place a mark.

ml_men1 to ml_men30 Generate a left mouse button click in the corresponding mouse menu item.

mr_men1 to mr_men30 Generate a right mouse button click in the corresponding mouse menu item.

nocode Dummy code which has no effect. This way a right hand side can be 'empty'.

num_0 to num_9 Numeric key pad 0 to 9 (if it can be distinguished (cannot be done on most PC's).

num_add Atari:Numeric plus.

num_div Atari:Numeric slash.

num_dot Atari:Numeric period.

num_enter The enter key.

num_lb Atari:Numeric left bracket.

num_lf Atari:Numeric linefeed.

num_min Atari:Numeric minus.

num_mul Atari:Numeric star.

num_period Atari:Numeric period.

num_plus Atari:Numeric plus sign.

num_rb Atari:Numeric right bracket

num_slash Atari:Numeric slash.

num_star Atari:Numeric star.

num_sub Atari:Numeric minus.

on_exit Special code for actions to be undertaken when the editor is terminated.

open Open the current fold (if the cursor is on one).

open_all Open all folds.

page_down Move the cursor one page down.

page_up Move the cursor one page up.

paste Paste the contents of the active yank buffer into the text.

paste_block Paste the contents of the active yank buffer into the text using the block mode.

plus The character +.

quit Leave the editor.

read Read a file (same as F8).

return The return key.

right Move the cursor to the right.

save Save the contents of the current buffer, using the name of the buffer.

save_quit Save the contents of the current buffer and leave the editor.

sc_home Move the cursor to the left bottom position on the screen.

sc_left Move the cursor to the previous character. See tabs as one character, and go to the previous line if necessary.

sc_right Move the cursor to the next character. See tabs as one character and go to the next line if necessary.

semicolon The semicolon character for insertion in the text.

sh_down Go to the next page.

sh_home Go to the end of the buffer.

sh_insert Insert a new line after the current line.

sh_left Go to the beginning of the current line.

sh_page_down Move the cursor to the next line but keep its position fixed on the screen (scrolls screen up).

sh_page_up Move the cursor to the previous line but keep its position fixed on the screen (scrolls screen down).

sh_right Go to the end of the current line.

sh_up Go to the previous page.

startup Special code to define a sequence of keys to be executed when the editor is started. This code will be executed after the files from the command line have been read. This code is not executed in the streamer mode.

tab The tab character.

tag0 to tag10 Go to the corresponding tag. 0 and 10 are identical.

undo Undo the last sequence of deletes (or clear buffer).

up Move the cursor up.

write Write file. Similar to shift-F8.

yank Yank/cut the current range.

yank_block Yank/cut the current block.

The numeric key pad codes cannot be generated easily on some computers, because in the past they were generated by having a special status key and then keys on the regular pad were used together with this status key. This is still the case on the PC. The numeric key codes given by the keyboard processor of the PC mimic the corresponding regular keys.

There are more mouse events that can be constructed. This is explained later. Keys that have no direct binding in STEDI are not mentioned in the above table.

When you are taking a key redefinition file from one type of computer to another type, the right hand sides with their mnemonics should still be valid. It may be that the other computer has a completely different keyboard with entirely different capabilities. Hence it could be necessary to reprogram the left hand sides (usually the lesser amount of work).

In the next example, the left and right arrow keys will be replaced by Ctrl-Shift-left-arrow and Ctrl-Shift-right-arrow respectively and vice versa. The Ctrl-Shift-left/right-arrow keys cause the cursor to move according to the actual characters that are in the text as in some editors, and not according to the screen. Some people may prefer this.

```
: Example 3 : redefinition of normal keys;

00001002(right-arrow)            = sc_right;
00001003(left-arrow)             = sc_left;
0600100E(Ctrl-Shift-right-arrow) = right;
0600100F(Ctrl-Shift-left-arrow)  = left;
```

What we see here is not at all dangerous as might be expected. The output of the keyboard processor is not given back to its own input, so there is no danger of loops. Hence you may freely interchange the action of two keys as was done above. The next example is a little more sophisticated (and useful).

```
: Example 4 : Put editor in wordwrap
              and auto indent mode;

0200101B(left-shift+escape)
    + 00000032 (followed by 2) =
   command a plus  endcommand(autoindent on)
   command w w 7 2 endcommand(wordwrap at 72);
```

The left hand side consists now of two keys. This means that after you press shift-escape the editor will wait to see what comes next. If this is the character 2 it will generate the given sequence. If it is a character that doesn't occur in any sequence the message 'Key not active' is given in the message line. You can also see what is the best way of generating commands that are executed via the command line. The special mnemonics 'command' and 'endcommand' cause the characters in between to be treated as a command from the command line. This command will not be entered in the history though.

26.1.2 The meaning of the codes

Before we can understand the use of the flags and the masks we have to know what the various fields in the hex codes stand for. The meaning of the different fields may depend on what other fields look like, so it isn't entirely trivial. Each byte of the key code (eight bits or two hexadecimal digits) is treated separately.

byte 3 (leftmost) :
This byte contains bits that indicate the status keys. In addition three bits have been reserved for special flags that can be controlled by the user. The bits are:

bit 0
right shift key (hex code of byte is 01)

bit 1
left shift key (hex code of byte is 02)

bit 2
Control key (hex code of byte is 04)

bit 3
Alternate key (hex code of byte is 08)

bit 4
Caps Lock key (hex code of byte is 10)

bit 5 - 7
User flag 1 to 3 respectively.

A combination of status keys results in the sum of their codes. For example, if both the Alternate key and the left shift key were pressed, the value of byte 3 becomes 0A or 10 in decimal notation. The best way to become familiar with these bit patterns is to try out the Ctrl-K command with various key combinations.

byte 2 :
This byte contains either a flag to indicate that we have a mouse event, or it contains masking information. If the 0 bit (hex code 01) is set we have a mouse event and more bits may be set. This is explained in the part about the mouse events. The masks are explained in the part about flags and masks. For the next two bytes we assume now that the mouse bit is off.

byte 1 :
Contents are in hexadecimal:

0 Byte 0 contains a character to be put in the text.

1 Byte 0 contains the ASCII code of the corresponding alternate key code. So 0141 is Alt-A as 41 is the ASCII code of the character A.

2 Byte 0 contains the ASCII code of the corresponding control key code. Unlike regular control combinations STEDI takes them apart and stores them as a control bit and an ASCII character. On systems

that don't support the combination of a digit and the control key such combinations can be generated with the alt-function key combination.

4 Byte 0 contains the number of a function key, counting from 1 up (so F1 is 0401).

8 Byte 0 contains the number of a function key, but in addition a shift key was pressed.

C Byte 0 contains the number of a function key, when both a shift and the control key were also pressed. On systems that don't support such key combinations this code is given by the Ctrl-Function key combination.

10 Byte 0 contains the code for a special key or key combination. This may involve the arrow keys, the delete key, the backspace, the return and enter keys and more.

1E Byte 0 should be zero. The 'endcommand' code.

1F Byte 0 should be zero. This establishes a direct connection to the command line. The keys following are used to build up a command for the command line that is executed after the 'endcommand' code (see under 30). This 'endcommand' code must be part of the same key redefinition. The command thus formed will not be put in the command history.

20 Byte 0 contains the ASCII code of a key in the numeric key pad. This information isn't really used by STEDI.

byte 0 (rightmost) :
This byte contains either the ASCII code of a key or a simulated ASCII code. It is a 'real' ASCII code if byte 1 is zero.

Although only the key codes as explained above can be returned from the keyboard, the editor knows a few more codes internally. These codes cannot possibly come from the keyboard so STEDI can use them for signal passing. Some of these codes are:

08FF0000 Inactive key. Code is given as for instance a sequence with several keys at the left hand side gives no match. Its only effect is that the message 'Key not active' is placed in the message line.

10000000 STARTUP code. A lhs with this code makes that its right hand side is executed at startup, after the command tail has been interpreted

and its files have been read. The code isn't executed when the editor is run in stream mode. In the right hand side this codes is a good 'null' code in the sense that it has absolutely no effect, not even a message.

12000000 AFTER_READ code: When a file is read into a buffer this code is put into an input buffer that is read before the next user input is processed. This can be used for instance to call a macro that studies the extension of the name of the file just read and to take corresponding action.

18000000 ON_EXIT code: special left hand side of which the right hand side is to be executed when the editor is left.

x4000000 Absolute setting of the flags field. x is an even hexadecimal digit.

26.1.3 Flags and masks

The top three bits in the left most byte are reserved for user defined flags. Such flags can be used to indicate that a sequence was interrupted for some input, or to toggle between various modes. One common use is to allow the temporary disabling of a particularly drastic key redefinition scheme to allow other users who aren't used to this scheme to type anything. Internally the flags are treated as if they were status keys like the shift, control and alternate keys. The difference is that the user exerts software control over them. The flags are set either by providing an absolute bit pattern (with the mnemonics flagsxxx), by turning one flag on (with flagxon) or turning one flag off (with flagxoff). The flags are numbered 1 to 3 and the leftmost bit corresponds to flag three.

So let's try an example. The key sequence shift-escape followed by the number one will put the current range between mark and cursor inside a fold, properly commented for the language TEX. Of course a fold needs a name, so in the middle it has to wait for for the user to type a name. After the name is complete the user presses again shift-escape followed by the number 1 and the sequence will be finished. For storing the name we use the learn buffer 0 as we don't want to upset the paste buffers.

```
: Example 5 : Put a marked range in a fold
              Mark should be at end of range!;

0200101B (left-shift+escape)+00000031 (number 1)=
    flag3on                    (Switch flag 3 on)
```

```
    insert                      (make the new line)
    % blank blank # \[ blank    (start of line)
    command 1 0 endcommand      (start learning);

8200101B (left-shift+escape)+80000031 (number 1)=
    learn0        (finish learning)
    flag3off      (switch back to normal)
    blank colon   (finish the opening fold line)
    exchange      (go to the end of the range)
    insert        (make new line)
    % blank blank # \] blank    (start other line)
    learn0        (replay the typing of the name)
    blank colon   (finish this line)
    close         (close the fold {optional});
```

As you can see, we have two entries for the sh-esc+1 code sequence. The second time however we demand that the top bit, corresponding to the third flag, has been set.

The attentive reader will have noticed the **left**-shift, and may wonder about the right shift key (and about the simultaneous activities of the other flags). It would be rather annoying to have to provided the same code up to 24 times (left, right, left+right and this times 4 positions for the two other flags, and times two for the CapsLock key). For this we have masking. Basically a mask tells that a certain redefinition is active regardless the settings of some flags or status keys. Masks are only relevant at the left hand side of a redefinition. They should not occur at the right side. The mask should be in the leftmost 7 bits of byte 2. These are called bit 1 to 7 and 7 is the leftmost bit. The meaning of the bits is:

bit 1
: This means that either one or both shift keys should be pressed. It is irrelevant which shift key. Hex code = 02.

bit 2
: It doesn't matter whether the control key is pressed. Hex code = 04.

bit 3
: It doesn't matter whether the alternate key is pressed. Hex code = 08.

bit 4
It doesn't matter whether the CapsLock key is pressed. Hex code = 10.

bit 5
It doesn't matter whether flag 1 is on or off. Hex code = 20.

bit 6
It doesn't matter whether flag 2 is on or off. Hex code = 40.

bit 7
It doesn't matter whether flag 3 is on or off. Hex code = 80.

One may combine masks by adding the corresponding hex codes. To make the scheme of the fold lines complete we make now the complete left sides as they could look in a real redefinition scheme:

```
: Example 5a : The proper left hand sides;

0272101B (shift+escape) + 00700031 (number 1) =
    etc....

8272101B (shift+escape) + 80700031 (number 1) =
    etc....
```

We have added the mask 72 (=02+10+20+40) which tells that we don't care whether it is the left shift, the right shift, or both. Neither do we care whether the CapsLock has been pressed or what the status of the other flags is. The 70 is similar but now we don't need the shift keys (on most keyboards at least). One could of course make the scheme independent of one flag and dependent of the other. If non of the redefinitions have masked out flag 2 we could set up a toggle to temporarily disable all redefinitions:

```
: Example 6 : Toggle flag 2;

02B21018 (shift+backspace mask 1+3) = flag2on:
    If flag 2 is off, turn it on;

42B21018 (shift+backspace mask 1+3) = flag2off:
    If flag 2 is on, turn it off;
```

If all other redefinitions don't like flag 2 then none of them will work when flag 2 is turned on.

26.1.4 The mouse

The codes for the mouse are kind of special. When a key is processed it is tested first for bit 16 (bit 0 of byte 2). If the bit is set all the above rules don't apply any more with the exception of the flags and the masking of the flags. The mouse information is restricted to the bytes 0, 1 and 2. Byte 0 (the right most byte) contains the column in which the mouse event took place. This is the column in terms of a character position and it starts counting at one for the left most column. Byte 1 contains the number of the line of the mouse event. It starts also counting at one for the top line on the screen. Sometimes these positions aren't reported as they are considered to be unimportant or they concern the dead parts in the command line. The interesting byte is byte 2. Its information is:

bit 0
: The mouse flag. Its on when you get here.

bit 1
: Menu indicator. If this bit is set, the event took place in the status line, so it needs special processing. In case of the dead zones in the menu bar the position isn't reported. When in the active regions of the menu bar the line number is still left empty and the column number is the number of the item in the menu, counting from 1. Outside the menu bar both the line and the column numbers are reported.

bit 2
: Set if the left (=first) mouse button is pressed.

bit 3
: Set if the right (=second) mouse button is pressed.

bit 4
: Reserved.

bit 5 - 7
: A possible mask for the user defined flags.

When more than one mouse button is pressed at the moment of the event there will be a corresponding number of bits. This allows the user to check for special action.

The main use of these mouse events is in a redefined mouse menu. This can be coupled to the installation of different characters in the menu bar. This is explained in the chapters on the mouse (p. 88) and the variables (p. 111).

```
: Example 7 : Redefine position in the mouse menu
              Close all other folds;

00070008 (position 8 between P and 1 ) =
command u - endcommand
    command s e t blank x \= $ l i n e endcommand
    close_all
    command \[ $ x endcommand;
```

The above macro notes first the number of the current line in the variable 'x', then closes all folds and opens just enough folds to go back to the line the cursor was in originally. The net effect is that all folds except for the current fold are closed.

26.2 The K command and keycomp

The K command which is used to read in the compiled key redefinition file is a command line command with the following syntax:

```
K filename
```

where 'filename' must be a file made by the keyboard compiler KEYCOMP. Some checking is performed to see if this is so. This command installs the contents of the file 'filename' into the buffer of the editor used for keyboard redefinitions. The command

```
K0
```

removes all keyboard redefinitions and restores the original layout.

Although there is no need to use a fixed extension for key redefinition files the preferred extensions are .k for the text files that are typed in (like the above examples) and .key for the translated versions. The key compiler KEYCOMP prefers these extensions as its use is either

```
keycomp filename
```

or

```
keycomp filename1 filename2
```

In the first case the 'filename' should be without extension. The extension .k is appended to obtain the name of the input file *filename*.k and that file is read. The output file will then be *filename*.key. In the second case the names and the extensions are arbitrary:

```
keycomp foo bar
```

The file 'foo' is read and the file 'bar' is written. On systems that use a desktop in which programs are run by double clicking with the mouse a dialog box will ask for the command tail. In MS-DOS the name of the key compiler is KEYCOMP.EXE and it can be run with the above commands. The same holds for command shells that run inside window systems.

The sources of the key compiler are present on the distribution disk. Apart from being an example of how one could use folds it serves a second purpose. It allows the user to add to the mnemonics if he wishes to do so. There is however no guarantee that the key compiler can be translated with the C-compiler you use. It has been tailored a little bit to several systems. For the PC it will translate with the WATCOM-C compiler (version 7.0) and for the Atari-ST it translates well with Turbo-C 1.1. The use of a number of standard library was avoided as there were some rather disappointing experiences with some compilers and the quality of their libraries. The result is that rather low level functions were used, that are rather direct hooks into the file system. This gives high speed (not very important here), compact code (nice) and reliability. It presents a small problem with respect to portability. Anybody who likes to translate it does so at his own risk.

Running STedi

This chapter covers the topic of starting up STEDI and also how to exit from the program. The help feature is also introduced.

27.1 Starting up STedi

27.1.1 From a desktop

When STEDI is used from a desktop, one may just double-click on its icon when its services are needed. It will then start and prompt the user by means of a file selector. If the user would like to edit an already existing file, he may select this file using this file selector. Its rules are usually explained in the manual of the computer or the windowing system. If a new file is to be made one may either type its name in the name field, followed by a <Return> or a click in the box marked OK, or just press a <Return>. In either case STEDI will greet the user with the message 'New File', as it was unable to find an already existing file with the name specified (or lack thereof).

Now it is possible to edit one or more files and then exit STEDI again. One may also run external programs such as compilations from STEDI making it unnecessary to continue to exit and re-enter the editor. This is explained in the chapter on executing external program p. 131.

27.1.2 From a command processor

One can run STEDI from a command processor by typing its name followed by a number of file names. STEDI can read up to eight files during startup. There is also the possibility to tell STEDI where to find its default file, its help file and the user defined macro's. If the first parameter given to STEDI is -d (or -D), then the next parameter is interpreted as the name of a directory. STEDI will make this directory its default directory and look there for its default file, its help file and macro's when the user wants to execute them. So the command

```
c:\bin\stedi -d c:\bin foo.c bar.c
```

will start a copy of STEDI which is found in the directory `c:\bin`, and the -d parameter along with the following path name informs STEDI that it can also find its default file, its help file and the macro's there. Finally the command

instructs STEDI to read the files `foo.c` and `bar.c` for editing. The file `foo.c` will be put in buffer 1 and the file `bar.c` will be put in buffer 2. In most shells this can be done simply by defining the alias

```
alias e "c:\bin\stedi -d c:\bin"
```

After this, one may call STEDI via the command

```
e foo.c bar.c
```

This command accomplishes exactly the same task as the command above that was fully written out. On MS-DOS one would include the directory `c:\bin\` in the path variable in the file autoexec.bat, after which one would accomplish the above with:

```
stedi -d c:\bin foo.c bar.c
```

UNIX-like systems would have a similar statement in a 'login' or 'resource' file. The above is still not very handy. Therefore STEDI will also recognize several variables in the environment. They are:

STEDIDFT
: This variable indicates the path of the directory where the default file can be found.

STEDIHLP
: This variable indicates the path of the directory where the help file can be found.

STEDIMAC
: This variable indicates the path of the directory where the macro's can be found.

On MS-DOS these environment variables can be set in the file autoexec.bat with the statements:

```
set STEDIDFT=C:\BIN\
set STEDIHLP=C:\BIN\
set STEDIMAC=C:\BIN\
```

For UNIX-like systems or shells one specifies such variables usually in a login script or a resource script (.cshrc for the c-shell or gpshrcu.sh for the GP-shell on the Atari-ST). One should use the setenv command in those UNIX-like environments.

The complete set of command line options is:

-d

This option must be the first parameter. The next parameter is then interpreted as a path name. In this path STEDI will look for the default file stedi.dft and –if needed– for the help file stedi.hlp. Also macro's are taken from this directory when needed. See also the more extensive description above.

-i or -x

The parameter after the -i parameter is interpreted as the name of a macro. This macro is read and all parameters after the name of the macro are passed as arguments to the macro. The macro will be automatically executed and after the execution buffer 1 is saved and execution is halted. This is called stream editing. It is used to modify a file according to a fixed set of predetermined rules. See also the chapters on stream editing p. 128 and macro's p. 119.

-r

The file following this parameter is read in the 'raw' mode. In this mode there is no interpretation of linefeeds or tabs. See also the part on write modes p. 22

-v

The file following this parameter is read in the 'view only' mode. This prevents the accidental change of an output file that is read into STEDI for viewing.

-#

When # is a number the file following this option is read and its current line will become the line indicated by the given number. When the number is greater than the number of lines in the file the cursor is placed in the last line of the text.

+#

The file following this option is read from the byte position indicated by the number. The first indicated number of bytes is skipped.

All above parameters can be given either in upper case or in lower case. The r, v and line number options can be combined. Each option must still be preceded by its own minus sign, but several of such parameters may be given before the actual file name is given. Example:

```
stedi -d c:/bin foo.c -v -800 bar.c -i action
```

This command starts STEDI, STEDI will look in c:/bin for its default file, put the file foo.c in buffer 1, the file bar.c is put in buffer 2 and buffer 2 is put in the 'view only' mode while the cursor is put at line 800. The file action.mac is looked for and if found it is executed as a regular macro.

It is allowed to use so called wildcard characters in the names of the files that should be read by STEDI. The full pattern is given to the local file system and each successive match is read into a separate buffer until either there are no more matches, or all 8 buffers have been filled. If there were options specified they apply to all files that are read in with this pattern. If a pattern is used for the name of a macro it will not be interpreted and the results can be unpredictable. The order in which the files match the pattern depends on the file system. Some systems give a purely lexicographic order, while MS-DOS and GEMDOS like to use the order in which the files appear in the directory on the disk which is at first the order in which they were written to disk, but after the directory has been used intensively the order can be quasi random.

When using the environment to define default directories there is no need to have the three variables to point at the same directory. If both the environment has been set and the -d option is used, the -d option takes precedence (most likely it has been typed by hand so it should override the defaults). In all cases STEDI will first look in the current directory for any of the files it needs, whatever the default settings. The rationale behind this is that a local file is usually project bound and should be preferred. This can avoid very nasty surprises.

Finally something more about the environment. When variables are used (for instance in macro's) and the user specifies a variable that was neither defined before, nor a predefined variable STEDI will search the environment for this variable. This search is done case insensitive. If the variable is found in the environment its contents are returned. If the variable doesn't occur in the environment the message variable is undefined is given. See also the chapter on macro's p. 119.

27.2 Exiting STedi

There are several ways to exit STEDI . You may use the function keys, the command line, or the mouse. with the function keys you may press **shift-F10** which is the command to exit the editor. If any changes have been made to files that are in any of the buffers without saving them, STEDI will query to

make sure that you indeed want to abandon the changes. You may answer 'y' (yes) to leave the editor without saving the changes, or 'n' (no) which results in no action. You then can save the files desired and leave the editor. The function key F10 allows you to save the file in the current buffer and to quit all at one stroke. Again if files in other buffers are unsaved, you will be queried concerning them.

In the command line you may exit the program by typing the command 'q' for quit or the whole word 'quit'. This is equivalent to the shift-F10 command. You may also enter the command 'sq' standing for 'save and quit', which is the same as the command F10. In addition the command 'savequit' will have the same effect.

With the mouse, there is only one command available for exiting the editor. Clicking on the 'Q' in the mouse menu bar performs the same action as pressing shift-F10.

For the options on writing while saving a file, see the chapter on Reading, Writing and Printing p36.

27.3 The help facility

If you press the <Help> key and STEDI is able to find the file stedi.hlp either in the current directory or the default help directory (see above), then this file will be displayed. If buffer eight is available, the help file will be read into that buffer, or if not the first lower buffer available will be used. The buffer will be automatically set in 'view only' mode so that the help file will not be written away if accidentally altered. (For more information on the view only mode, see the chapter on the status bar.) The help file consists of a very brief summary of all STEDI commands to serve as a reminder. Pressing the <Help> key a second time will cause the editor to return to the buffer which was being edited before the help facility was called. Each time the user asks for help this way the help file is read in freshly. This allows for experimentation with respect to what is read in it.

List of messages and their meanings

This chapter is something like an anti panic chapter. It explains very shortly what each message means, and how serious it is.

 # bytes: #/# screen: #,# lines: #
See the equals command (=) p. 16.

 # character(s) in learn buffer #
When a learn sequence is completed (p. 108) this message will appear.

 # lines per formatted printerpage. Offset = #
The setting of the page layout is reported after setting it with the pp command. See p. 47.

 # lines read.
After a file has been read the number of lines is reported.

 # lines sent to the printer.
After a file has been printed the number of lines is reported. This number is reported after STEDI has finished sending the lines and before it returns control to the user. The printer may still be printing as it may have a buffer of its own.

 # lines written.
After a file has been written the number of lines is reported.

 # replacement(s)
After a replacement this indicates the number of successful replacements.

'file' not found
The user tried to read a file but it wasn't there.

'name' is illegal name
The given name isn't a proper name for a variable (p. 111).

'var' = contents
The answer to a show command if the variable has a value.

'var' is undefined
The answer to a show command if the variable is undefined.

Abort execution? (Y/N)
When both shift keys are pressed during the execution of a macro the user is asked whether the execution should be halted.

Access denied.
A file cannot be read, due to a lack of rights.

Already learning in a lower buffer.
You cannot start learning in a higher buffer if a lower learn sequence is already running (p. 108).

Autoindent at column #(+)

Autoindent off.

Autoindent on.
Regular messages corresponding either to a change in the settings of the auto-indent mode or a request to what the auto-indent status is currently.

Buffer is ViewOnly.
Reply to an attempt to write a buffer that is in the View-Only mode (see p. 46).

Buffer made read-only!
When a file cannot be read in its entirety, due to a lack of memory, this message will be displayed. STEDI will automatically turn on the View-Only mode to prevent accidental writing of a truncated file (see p. 41).

Buffer not found
A buffer was searched by name, and it couldn't be located.

Buffer unsaved. Empty anyway? (Y/N)
The clear buffer command is given (shift-F9) and the current buffer has its dirty flag on. You can answer with Y or N.

Cannot create file 'name'
Somehow the file system refuses to create this file properly.

Cannot delete running macro
It is possible to execute the md command (p. 119) from a macro. It is however not allowed to kill or remove a currently active macro.

Cannot find file 'name'
Either the default file cannot be located, or a file with key redefinitions as specified in the K command (p. 165) cannot be found by the file system.

Cannot find help file. Quitting is with shift-F10.
If the help file cannot be located the least the editor can let you know is how you can get out.

Cannot find macro 'name'
The requested macro couldn't be located. Do you have the proper default directory or environment variable STEDIMAC?

Cannot make a proper backup.
Somehow the old file cannot be renamed so that it gets a .bak extension.

Cannot move old file to backup.
The name of the old file corresponds to that of an object of which you aren't allowed to change the name into a name with the extension .bak.

Cannot read defaults from learn buffer.
You aren't allowed to read a default file from a learn buffer (p. 110).

Cannot remove old backup.
A previous backup file cannot be removed. Is it readonly?

Cannot save a buffer without a name
Before a buffer can be saved, the buffer must have a name.

Columns must be in range 1-255
A reply to a sort command with an illegal request for a column range.

Command too long.
The command you just entered is longer than 255 characters after the substitution of the variables in it.

Cursor not in a fold line.
You cannot open a fold if the cursor isn't on the fold line.

Cursor not inside a legal fold.
You cannot close the current fold if the cursor isn't inside a fold.

Cursor not on a bracket.
You cannot search for a matching bracket if the cursor isn't on a bracket.

Cursor not on a word.
The cursor isn't on a word and you are trying to do something with the current word.

DFT file from 'path'
A reply to the setting of the path for the default file with the DD command (p. 53).

Deallocation error
Some internal error has messed up the line allocation mechanism inside STEDI. If you can reproduce this the author would like to know how. Anyway it is time for panic. See what you can save, but be smart and don't write it over your regular files (don't use the save command). If that can be done you should leave the editor and start it up again.

Defaults written to 'fullname'
The reply to a DW (defaults write) command (p. 52);

Detabbed # line(s)
The number of lines in which STEDI had to look for tabs to execute a TE (tabs expand) command (empty lines don't count) (see p. 95).

Drive not connected.
You were asking for the available disk space on a drive that isn't known to the operating system.

Drive not found.
Message returned by the file system. The requested file couldn't be found for the indicated reason.

Else without if
Each else statement must belong to an if statement and each if statement can have at most one else. See the chapter on macro's p. 119.

Empty substring
A substring in a regular expression is a string enclosed by double quotes. You used the meaningless sequence "".

Endwhile without while
Each endwhile statement in a macro's should correspond to a while statement earlier in the text (see macro's p. 119).

Error #
While returning from executing a shell command this error number was reported by the operating system (see 'Execute an external command' p. 131).

Error while reading key redefinitions Hit any key to abort
This message can occur only during startup. It means that something is wrong with your default file.

Error while reading macros. Hit any key to abort
This message can occur only during startup. It means that something is wrong with your default file.

Error while reading.
A general mishap reported by the file system during reading of a file. Reading is aborted.

Error while writing. Disk full?
The operating system reported an error during writing. The most common reason for such an error is that the disk is full.

Error while writing. File unreliable.
An error is reported during the writing of the default file. It would be best to remove it as it has only partially been written.

Evaluation error
The evaluation of the right side of a set statement or the expression in an if or while statement ran into some problems. Check your expression carefully (p. 121).

Execution time = #.# sec.
After the execution of an external command STEDI always reports the time to an accuracy of tenths of a second.

Expression too complicated during matching
An internal stack overflow was reported during the matching of a regular expression. For this to happen there should be a few hundred objects in the match.

Expression too complicated.
The pattern for a regular expression is too complicated to be translated. This is quite some expression. The author would like to see it (if you think it is altogether reasonable to have STEDI process this expression).

File exists already. Write anyway ? (Y/N)
The write command was issued with the name of a file that exists already. STEDI wants to know for sure that the user doesn't mind that this name is used nevertheless. A .bak version will be made if the backup setting asks for it.

File not found.
A reply of the operating system when a file has to be read and cannot be located.

Fold lines may not be changed.
You undertook an attempt to alter a line that represents a closed fold. This is not allowed (p. 64).

Fold not found.
If you look for a fold by name it can always happen that the fold doesn't exist (typing error?).

Fold line skipped
A paste in block mode was done and one of the lines that should have gotten a new piece in it was a closed fold line. The fold line wasn't changed!

Higher buffer called from a lower one!
An attempt at replaying a higher learn buffer from a lower one was intercepted (see p. 108)

If without endif
For each if statement in a macro there must be an endif statement. According to STEDI there is no matching endif for the if statement of which

the line number is given in regular way for macro error messages (see the chapter on macro's p. 119).

Illegal end of expression
Either a regular expression pattern wasn't completed properly or an expression that had to be evaluated wasn't complete.

Illegal end of macro
The evaluation of an expression ran into the end of the macro. This can happen when the end of line has been escaped with either an escape character or a backslash and there are no more lines.

Illegal if/while nesting
This could involve a sequence of statements like: if, while, endif, endwhile. Such a sequence of statements is of course illegal.

Illegal key redefinition file.
The file read with the K command (p. 148) cannot be recognized as a proper key redefinition file.

Illegal name for variable
The name of the variable you tried to use didn't conform to the rules (see the chapter on variables p. 111).

Illegal option in field 'string'
The option refers to an option in the command that was used to start STEDI. See the chapter on running STEDI p. 167.

Illegal position for)
The parentheses inside an expression to be evaluated aren't at the right place.

Illegal repetitor
A repetitor in a regular expression was found wanting. See the definition of repetitors p. 80.

Illegal sequence of operators
The operators in an expression to be evaluated were in an improper order. See p. 121 about the syntax of the operators.

Illegal sequence starting with 'string'
The regular expression isn't formed properly. The offending object is shown.

Illegal string
A string (enclosed in double quotes) had an illegal ending (maybe the closing double quote is missing). This message is reported by the routine that evaluates expressions.

Illegal substring
A substring (string enclosed in double quotes in a regular expression) wasn't formed properly. Check your expression.

Illegal use of @
A digit was expected after the character during the compilation of your regular expression. See the chapter on regular expressions p. 83.

Improper defaults file.
The default file cannot be right. It is way too short. You'd better throw it away.

Improper use of @ in rhs
A digit was expected after the character in the right hand side of a replacement by means of regular expressions. See the chapter on regular expressions p. 83.

Improper use of nonnumerical string
An arithmetic operation was attempted on a nonnumerical string. See the part about operators in expressions p. 121.

Internal error!
This is a sign of STEDI getting into trouble. If you can reproduce this one, please report it to the author. This particular message is generated by the routine that performs the operations for the evaluation of an expression.

Invalid default file. Hit any key to abort
This message can occur only during startup. It means that something is wrong with your default file.

Invalid load format.
This message is reported by the operating system when the program you

tried to execute by means of the ! command isn't a proper executable program.

Invalid range in string
Inside a group of characters in a regular expression (indicated by straight brackets) a range of characters wasn't formed right. See the part on character groups p. 79.

Irregular ending
A regular expression statement ends in an escape of the end of line and there are no more lines in the macro (or the command line).

Key not active
A key combination was used that isn't bound to anything. When this happens suddenly to your favorite key redefinitions you better check (i) the default file. Did you load the right one. (ii) have the redefinitions been loaded? (iii) Did you add a redefinition that messes up your scheme (see example 2 p. 152) (iv) what happened to the flags? (v) if all else fails, reread the chapter on key redefinitions p. 148. (vi) Don't Panic! (vii) Start from scratch.

Label 'name' not found
A goto statement without a label is quite serious in a macro. (Goto's are quite serious anyway, but sometimes they are very handy).

Line too long.
First source: Line allocation mechanism. Somehow a request for a line of more than 16000 characters was made. The author would like to know how you did it. Second source: The command you gave was expanded into a command of more than 255 characters after the variables were substituted.

Line too long. cut up? (Y/N/Q)
During reading of the input a line of more than 255 characters has to be constructed. If you don't want the editor to cut it up the reading will be stopped. It is usually an indication that you are reading a file without linefeeds/carriage returns. This can be either a binary file or a file from some word processor with its own ideas about character files. Usually one can still do much with such a file by putting STEDI in the raw mode (Alt-R) before reading the file. See also p. 41.

Macro 'name' added
Message after a successful 'mc' command to create a macro (p. 119).

Macro 'name' deleted
Message after a successful 'md' command to delete a macro (p. 119).

Mark not found.
The presence of the mark was needed (either to jump to the mark or to use it for a 'ranged' action). However it couldn't be found. The most likely reason is that it is hiding inside a closed fold.

Mark removed.
The answer to either the 'delete mark' command (shift-F1 on systems with a help key) or an operation that made it impossible to keep the position of the mark (like deleting the line with the mark in it).

Mark set.
The answer to the 'put mark' command.

Missing / in replacement
The syntax of a replacement is much stricter than that of a search command. The trailing slash should also be present.

More than 9 parameters
A macro may have at most 9 parameters (p. 120).

Name longer than # characters
The name of a buffer, including the path name is longer than a built in maximum value. Try to keep the path name simpler.

Negative number of chars?
One of the operations 'string take/make left/right' had to take/make a negative number of characters. See the rules for the operators on p. 121.

New file.
At startup a file was requested that didn't exist. The buffer does get the name and this message is displayed.

No = in set
A set statement needs the equals sign. See the chapter on variables p. 111.

No linefeeds in block or column mode
In block or column mode it makes no sense to try to replace an end of line.

No lines to be sorted
There are no lines to be sorted (for instance in an empty buffer).

No mark.
The presence of the mark was needed (either to jump to the mark or to use it for a 'ranged' action). However there is no mark.

No match
A search returns unsuccessful.

No memory for writing.
The write operation needs a buffer to make the writing efficient. If there is no memory to be allocated for this buffer, there are two possibilities: (i) the screen memory is borrowed. (ii) the above message is displayed. On most systems it is too dangerous to borrow the screen memory. It is possible with the Atari-ST. If the above message is given you should try to clear the undo buffer or one of the paste buffers. If that doesn't help, clear another buffer (press shift-F9 twice to also clear the undo buffer). If that cannot be done either you can try to remove all macro's and the key redefinitions. That may make some space available. If that doesn't help either you are in trouble.

No memory
(Without period) Issued at startup if there isn't enough memory to even start the editor up.

No memory.
A given operation couldn't be performed. This can be a temporary matter if the operation was restructuring the memory considerably. In that case it may be sufficient to wait a second and give the command again, so that it may finish. The garbage collections are quite fast. If it happens during the running of a long macro one could try to insert a 'garbage' command at some strategic spots in the macro to force a garbage collection (p. 142), as these are usually only done when the computer waits for input. If the garbage collection isn't sufficient you should try to empty some buffers first before continuing.

No more handles.
A message from the file system. It says that a file couldn't be opened for reading or writing for the mentioned reason. It indicates usually that some other program has been wrecking havoc on the operating system and that it is time to reset the computer. STEDI pleads innocence.

No more words.
A command like 'goto next word' (Ctrl-W) has been given but either the end of the file has been reached or the only characters left are seen as illegal characters for a word. See the word definitions on p. 97

No name specified in goto
A goto statement needs the name of a label. It's bad enough already! For the syntax see p. 126.

No other match
The search statement comes to the conclusion that the only match is the match at the position that the cursor occupies already.

No reading in fold line
When the cursor is inside a closed fold line reading would damage this line, so it is forbidden (see p. 64).

No rewrapping in fortran mode.
Rewrapping makes no sense in the fortran mode. So it isn't done! See the section on rewrapping p. 102.

No slot for help file. Quitting is with shift-F10.
All buffers are filled (or have a name), so the help file cannot be read. If you don't know how to get out it tells you at least how it can be done. See the part about the help facility p. 171.

No word wrap set.
Rewrapping a paragraph can of course only be done when the word wrap mode has been set. The current status of the word wrap can be inspected with the WW command (p. 101).

Not enough memory. Buffer made View-Only.
During the reading of a file the memory got exhausted. As the file has only been read partially it is better to make it difficult to write the file away. If a file is too big one can edit it in parts and concatenate these

parts while writing so that the eventual output forms a new file that is too large. See the options of the read and write commands p. 38 and p. 41.

Not in backwards mode
Regular expressions can only be used in one of the forward search modes. Discussions about how to interpret a backward search with regular expressions have still not been finished. The opinions are divided.

Not in raw mode
In raw mode all special tab representations are switched off. Hence it would not serve any purpose to try tabbing, trimming or detabbing. Therefore it is forbidden.

Numerical value expected.
When setting some variables it is necessary to provide them with a numerical value. The string that was given couldn't be translated in one. See p. 116.

Path not found.
A message from the file system. The path specified with the name of the file to be read couldn't be located.

Printer not available
If you try to print something and either there is no printer or the printer is off, this is the message. It may also be that you write to the wrong port. See the chapter on printing p. 47.

Printer not ready. Continue ? (Y/N)
A message if during writing the printer 'times out', indicating that during a given period no character could be sent to the printer. This could be due to a lack off paper, or the user took the printer off line to interrupt a listing. One can continue printing (for instance after refilling the paper tray) by typing Y. Typing N aborts the printing.

Printing to PRN:/AUX:/COM:
The answer to the setting of the printer port with the 'P=' command. It is also the answer to the request for the printer port status. See the chapter on printing p. 47.

Range copied.
The message after the successful copying of a range to a yank buffer.

Range cut.
The message after the successful cutting/yanking of a range to a yank buffer.

Raw writing? (Y/N)
When a write command is given and the buffer is in the raw mode this writing should always be verified. If you switch the buffer accidentally to the raw mode and the file would just be written it is a lot of work to put all the linefeeds back in. See the chapter on writing p. 41.

Reading discontinued
The answer to the request to truncate lines longer than 255 characters during reading was no. STEDI cannot put this file in its buffers (see p. 41).

Replace? (G/Y/N/Q)
When using the veto option for a replace, each time the editor finds a match it will wait and ask the user what to do.

Return code = #
The return code when an external command has been executed. If the return code is zero, it isn't reported and when it is negative it is reported as an error.

Stopped
The execution of a macro was interrupted by the user.

Syntax error.
Some simple commands result only in this vague message when their syntax isn't right. You better check the corresponding pages in the manual.

Tabbed # line(s)
The number of lines in which STEDI had to look in to see whether tabs should be placed in them while executing a 'tab' command (empty lines don't count) (see p. 95).

Tag # placed
After the command <# this is the message (# is a single digit). See the section on tags p. 57.

Tag not found.
A 'goto tag' command was given but the tag could not be located. See the section on tags p. 57.

There are # bytes free on drive 'char'
The reply to the ? command. See the section on free disk space p. 49.

There is a macro with this name already!
There can only be one macro with a given name. If you insist on this name you have to delete the existing macro first with the 'md' command p. 119.

Too long word for autorewrap.
The current word is so long that the word wrap would get in trouble. This trouble would manifest itself by getting a new line, every time a character is added to the word.

Too many character groups
The number of character groups in your pattern is larger than a predefined maximum (at least 10). Try to rethink your pattern.

Too many characters in learn buffer #
The learn buffers have a capacity of 100 characters each. If the above message is shown you should stop the learning in the given buffer, as it has no use to continue it. See the chapter on the learn buffers p. 108.

Trimmed # line(s)
The number of lines in which STEDI had to look for superfluous blanks and tabs while executing tt (tab trim) command (empty lines don't count) (see p. 95).

Truncate too long line? (Y/N)
A paste command results in a line being longer than 255 characters. Should this line be truncated (with loss of characters) or should this part of the paste operation be aborted?

Twice else in one if
Each if can only have one else in a macro.

Undefined variable # in rhs
In a regular expression replacement there is a variable in the right hand

side that wasn't specified in the left hand side, so during the replace operation it will be undefined. See the chapter on regular expressions p. 83.

Unexpected end of expression
A pattern in a regular expression is terminated abnormally.

Unknown command 'string'
The command line command you gave isn't recognized.

Unknown option in set
The set command knows only two options: the -L and the -G option. Apparently you asked for another option. See the chapter on variables p. 111.

Unmatched
A group of characters in the pattern of a regular expression is opened but the closing bracket is missing.

Unmatched brackets
The brackets in an expression aren't matched.

Unmatched parentheses
Some subexpression in the pattern of a regular expression has unmatched parentheses. Check the statement.

Unmatched while or if in macro 'name'
When a macro is put in the inner buffers all the jumps are set up. During this phase some links couldn't be made.

Unsaved buffer(s) 'digits' Leave anyway? (Y/N)
The message that is displayed indicates that there is still some unsaved work and that you are trying to leave the editor. The digits are the numbers of the buffers that have their dirty flag on. So the string '13' means that the buffers 1 and 3 have still unsaved work in them.

User interrupt. Abort execution? (Y/N)
The user has pressed both shift keys during the execution of a macro. He can tell now whether he wants to abort execution.

Value should be ON or OFF
You tried to set a variable to an illegal value. This variable can have only one of the reported values. See p. 116.

Value should be ON, OFF or SINGLE
You tried to set a variable to an illegal value. This variable can have only one of the reported values. See p. 116.

Variable stack overflow during evaluation
During the evaluation of an expression the intermediate result became too long. Try something simpler.

Word wrap at column #

Word wrap in fortran mode.

Word wrap off.
Either an answer after the status of the word wrap has been changed or a report on its status after the WW command has been given. See p. 101.

Wrong version default file.
The default file belongs to a different version of STEDI (or to a version on a different computer). It should be removed or renamed before STEDI is willing to start up.