Ada Terminal Emulator

Reference Manual

Version 3.0 (including Telnet)

Table of Contents

Table of Contents
Table of Contents

1 Ada Terminal Emulator Package

1.1 Overview

Who needs Yet Another Terminal Emulator? Possibly no one at all. Despite this, the Ada Terminal Emulator package provides a set of terminal emulation capabilities, implemented almost entirely in Ada 95 (or Ada 05)[1], and intended to run under Windows[2] 95/98/NT/2000/XP. It should also run under Windows ME.

In addition to providing simple "dumb" terminal emulation, the package provides full emulation of DEC[3] VT52/VT100/VT102 compatible terminals, including double height and double width characters, smooth scrolling, special graphics, display controls and national replacement character sets. The package also implements a substantial subset of VT220/VT420 and ISO 6429 capabilities.

All source code for the emulator is provided. The emulator can be used "as is", or extended to include additional emulation capabilities. The package consists of the following main components:

- **Terminal_Emulator**: An Ada package that provides facilities for creating, configuring and interacting with terminal windows from an Ada program. Multiple terminal windows can be created from the same program. Each window implements a completely independent terminal emulator. Each can be a separately configured as simple "dumb" terminal, or emulate full DEC VTxxx capabilities. Each window can have complete mouse support (e.g. for scrolling or resizing the window, or for selecting, copying and pasting text).
- **Term_IO**: An Ada package that can be used as a complete and transparent replacement for the standard Ada text-handling package Text_IO. In addition to supporting all the normal file handling capabilities of Text_IO, **Term_IO** replaces the default files (i.e. standard input, standard output and standard error) with one or more terminal windows. It also allows the creation and manipulation of user-defined terminal windows using only Text_IO syntax. Each terminal window can be configured to either strictly implement the semantics of Text_IO as defined in the Ada 95 Language Reference Manual (LRM), or to implement relaxed semantics that allow the user to take advantage of the terminal emulator capabilities in a more intuitive manner.
- **Redirect**: A stand-alone program that allows existing text based applications (including non-Ada applications) to have their standard I/O (i.e. standard input, standard output and standard error) redirected through a terminal window - without recompiling or re-linking. **Redirect** provides full mouse, file and printing support, and also line editing, command and filename completion and command history. It can be used to create a very sophisticated window-based command line interpreter with only a few dozen lines of programming. A complete Ada 95 example is included.
- **Comms**: A stand-alone program that allows the use of a serial communications port as input and output to a terminal window. This enables a PC to be used as a serial terminal.
- **Telnet**: A stand-alone program that allows the use of the telnet protocol as input and output to a terminal window. This allows a PC to be used as a network virtual terminal.

The Ada Terminal Emulator package was developed primarily using GNAT[4] and GWindows[5] under Windows. However, the package may be used in other development environments - GNAT provides facilities to allow Ada program units to be called from other languages.

All **Terminal_Emulator** and **Term_IO** capabilities are fully compatible with Windows 95/98/NT/2000/XP. A very few of the **Redirect** capabilities (such as Unicode handling and named pipes) are only compatible with Windows NT/2000/XP.

[1] The emulator uses one component written in C – an ANSI control sequence parser, developed at MIT. See the acknowledgements.

[2] Windows, Windows 95, Windows 98, Windows NT and Windows 2000 are all trademarks of the Microsoft corporation.

[3] DEC (and Digital Equipment Corporation) is a trademark of Compaq computers.

[4] See the acknowledgements.

[5] See the acknowledgements.

1.2 Feature Summary

1.2.1 Terminal_Emulator

Terminal_Emulator supports the following major features:

- Character attributes include italic, bold, underline, strikeout, reversed, and flashing, as well as foreground and background color, using any windows font (not just fixed pitch fonts).
- Line attributes include double width and double height characters.
- Support for Windows character sets; DEC special graphic character sets; DEC multinational character sets; DEC national replacement character sets and DEC display controls character sets (for displaying control codes as characters on screen).
- Support for DEC VTxxx control sequences, ISO 6429 control sequences and ANSI.SYS control sequences. Support for 7 or 8 bit control sequences.
- Support for DEC user-definable function keys, numeric and application and editing keypads, and language-specific keyboards.
- Separately sizable virtual buffer, screen, view, and scrolling region.
  - Virtual buffer sizes up to 32,767 rows (lines) by 1024 columns (characters).
  - Screen sizes up to 32,727 rows (lines) by 1024 columns (characters).
  - View sizes up to the lesser of the screen size and the capacity of the display.
  - Rectangular scrolling regions of any size up to the screen size.
- Support for smooth (soft) scrolling of the screen or scrolling region.
- Support for separate cursors, character attributes and screen colors for input and output operations.
- Mouse support - select text by character, word, line or rectangular region. Copy and paste between terminal windows and other Windows applications. Resize the screen, the view, or the font size using the mouse.
- File support - load and save virtual buffer to/from text files.
- Printer support – printer setup, page setup, print current selection, print entire buffer.

1.2.2 Term_IO

Term_IO supports all the features of the **Terminal_Emulator**, and adds the following features:

- Complete and transparent replacement for the Ada standard text handling package Text_IO.
- Combined or separate terminal windows for the Text_IO default files (standard input, standard output and standard error).
- User defined terminal windows, which can be used as if they were Text_IO files.
- Either strict adherence to the Ada 95 LRM semantics defined for Text_IO, or relaxed adherence for more intuitive terminal operation.

1.2.3 Redirect

Redirect supports all the features of the **Terminal_Emulator**, and adds the following features:

- Full command line editing.
- Command completion and filename completion.
- Command history.
- Unicode (wide character) support for redirected I/O (not supported under Windows 95/98).
- Support for named or anonymous pipes for redirected I/O (named pipes not supported under Windows 95/98).
- Supports use in either stand-alone or as a filter for other text-processing programs, in a similar fashion to traditional filter programs such as “more”, “sort” etc.

1.2.4 Comms

Comms supports all the features of the **Terminal_Emulator**, and adds the following features:

- Supports the use of COM1 to COM9 as input and output, with configurable serial options.

1.2.5 Telnet

Telnet supports all the features of the **Terminal_Emulator**, and adds the following features:

- Supports the use of the Telnet protocol as input and output, allowing the Terminal Emulator to act as a Network Virtual Terminal.
- Can be used as either a telnet client or a server.
- Configurable port number (defaults to the Telnet standard port number 23).
- Supports the following telnet options:
  - Binary
  - Echo
  - Suppress Go Ahead
  - Status
  - Timing Mark
  - Terminal Type
  - End of Record

1.3 *Copyright and License*

All Ada Terminal Emulator package components (including the documentation) are released under the GNU General Public License. For more details, see the file `COPYING`, included in the distribution.

1.4 Contact

Feedback is welcome, as are bug reports. Bug *fixes* would be even more welcome.

Email: ross@thevastydeep.com

1.5 Acknowledgments

1.5.1 GNAT

The GNU Ada Translator (GNAT) is an excellent and very complete Ada implementation, available for many platforms. It was used to develop this terminal emulator package.

You can download GNAT from https://www.adacore.com/download

The version used for this release was the GNAT Community 2021 edition. The file name for this is **gnat-2021-20210519-x86_64-windows64-bin.exe**.

1.5.2 win32ada

win32ada is an Ada binding to the Microsoft Win32 libraries.

You can download win32ada from from https://github.com/AdaCore/win32ada

The version used for this release was version 23. The file name for this is **win32ada-23.0.0.zip**. Note that there are a few small changes required – see the *win32ada* folder in this release for details.

1.5.3 GWindows

GWindows is a full Win32 RAD GUI Framework, originally developed by David Botton. It is a powerful tool for developing Windows applications in Ada, and is extensively used in this terminal emulator package.

You can download GWindows from https://sourceforge.net/projects/gnavi/files/

The version used for this release was GWindows 1.3,, specifically the version released on 13 Nov 2022. The file name for this is **Gwindows 13-Nov-2022.exe**. Note that there are a few small changes required – see the *GWindows* folder in this release for details.

1.5.4 VTtest

VTtest is a program to test the compatibility (or to demonstrate the non-compatibility) of so-called "VT100-compatible" terminals. Although not required by this terminal emulator package, a modified version can be used to test and demonstrate its capabilities. The source code for this modified version is included.

The original can be obtained from http://dickey.his.com/

The program was originally developed by Per Lindberg, and has been extensively enhanced (and is maintained) by Thomas E. Dickey.

1.5.5 Cygwin

Cygwin is a Linux-like environment for Windows. Cygwin is neither required nor used by this terminal emulator package, but it may be used to compile the VTtest program to run under Windows. A compiled version of the VTtest program is included, but the Cygwin DLL (required to run the VTtest program) is ***not*** included.

For more information about Cygwin, or to obtain a copy of the Cygwin DLL (**`cygwin1.dll`**), visit the Cygwin home page at http://cygwin.com/

Note that the GNAT compiler already contains a copy of the Cygwin DLL. It is in the **`GNAT\bin`** subdirectory and is called **`cyggnat.dll`**. To use the VTtest program you can simply copy this file (e.g. to the **`terminal\bin`** subdirectory) and rename the copy to **`cygwin1.dll`** (do not change the name of the original file – it is needed by GNAT). However, it is probably better to get a current copy from the Cygwin home page.

1.5.6 MIT parser

The “Parser For VT420 Terminal Emulators” was developed at the Massachusetts Institute of Technology (M.I.T.). It is an astoundingly comprehensive and robust ANSI and DEC control sequence parser. This terminal emulator package uses a modified version. The source code for this modified version is included.

The original can be obtained from http://www.cs.utk.edu/~shuford/terminal/mit_vt420_parser.html, a site maintained by Richard S. Shuford.

1.5.7 VT100.net

VT100.net is dedicated to the range of video terminals produced by Digital Equipment Corporation (DEC) from 1970 to 1995. It is a rich source of detailed information about DEC terminals (including programming manuals), and was of great assistance in developing this terminal emulator package.

The VT100.net site is maintained by Paul Williams (http://celigne.co.uk/paul/)

For more information, visit http://vt100.net/

1.5.8 Telnet Protocol

The telnet protocol software used in the terminal emulator was based on software originally developed by Mike Thomas and Paul Higgins of E-Systems, Inc. The original is freely available in the Public Ada Library or from the Simtel repository. The original version was for Ada 83 on a DEC platform, and it has been extensively modified for use in the terminal emulator package.

1.5.9 Special Thanks

A special thanks to Simon Clubley, who perservered with testing many versions of this program, and who made valuable suggestions for improvements.

2 Terminal_Emulator

2.1 Introduction

The package **Terminal_Emulator** provides a comprehensive API for creating and manipulating terminal emulator windows. It provides high-level access to most terminal emulator capabilities.

The API is intended to be simple and intuitive to use. For a simple "Hello World" program, see section 2.4.2. For more complex examples, refer to the demonstration programs described in .

This section is not a complete syntax description of the API – it merely introduces each procedure (or function), describes its purpose, and notes any issues associated with it. Examples are given in this document only to illustrate functionality that may not be immediately evident. Refer to the demonstration programs for complete examples. Refer to the source file `terminal_emulator.ads` for complete details of the syntax of each type, procedure or function.

There are some terminal emulator capabilities that do not have equivalents in the API, and are only accessible via ANSI control sequences. These are usually the more obscure functions that only apply to specific DEC VTxxx modes, and have no general applicability. To invoke functions that do not have corresponding API calls, a string containing the appropriate ANSI control sequence can be constructed and sent to the terminal window. See the description of `Put` section 2.4.1 for an example.

2.2 Basic Concepts

Before describing the API, the following sections define and describe the relationship between buffers, screens, views and regions. A diagram then summarizes the relationships. It is not necessary to read this section before using the basic API capabilities, but it is necessary in understanding some of the more complex API capabilities.

2.2.1 Virtual Buffer

The virtual buffer is where all character information is stored. It is organized as rows (normally limited to 32,767) of columns (normally limited to 1024), each representing a character cell that can contain a single character and all its attributes.

Note for developers: The row limit is due only to a limitation in Windows dialog boxes, and could be changed if required, at the expense of not being able to use the dialog box to resize the virtual buffer. The column limitation is also arbitrary, and can be changed in the source code if required.

The screen, region and view (described in the sections below) are all overlaid onto the virtual buffer. The virtual buffer has the same number of columns as the screen, but typically has many more rows, so information scrolled off the screen is still displayable. Normally, a character is only lost when it is removed from the virtual buffer, not just scrolled off the current screen.

The Terminal_Emulator API provides procedures for manipulating the current size of the virtual buffer. See section 2.4.6.

Note for developers: The virtual buffer is itself implemented as an overlay onto another buffer (of at least the same size as the virtual buffer) known as the real buffer. The real buffer is always managed internally by the Terminal_Emulator package and cannot be accessed via the API, so it is not normally necessary to consider this.

2.2.2 Screen

The screen is the active part of the virtual buffer. Most of the API functions and ANSI escape sequences refer to the manipulation of character cells on the screen. The screen is the only part of the virtual buffer where the cells are individually addressable by users of the API. For example, the cursor is represented as a (column, row) position on the screen.

The screen is always the same width as the virtual buffer, but may have fewer rows. In essence, the screen can slide up and down the virtual buffer as necessary. For example, depending on the use of the scrolling region (see the section below), scrolling the data on the screen up (i.e. when the characters appear to move up) is actually performed by sliding the location of the screen down the virtual buffer. The rows that move off the current screen are still present in the virtual buffer.

The Terminal_Emulator API provides procedures for manipulating the current size and position of the screen within the virtual buffer. See section 2.4.6.

Note that all cursor position and screen position addressing in the API is done via column and row, where column 0, row 0 represents to the top left character on the screen. However ANSI escape sequences address the top left cell as row 1, column 1, so keep in mind that 1 will need to be added to API column and row references to refer to the same screen position in an ANSI escape sequences.

2.2.3 Region

The region is a portion of the screen. The region can be smaller than the screen or the same size - it cannot be larger than the screen. The region is defined relative to the screen, and moves as the screen moves. The region primarily required to implement some of the DEC VTxxx functionality. It is where all scrolling is performed, and normally if the cursor is within the region it is constrained to stay within the region.

The region can be disabled or enabled. When enabled, all scrolling is performed entirely in place within the region, and the screen does not move. When the region is disabled, vertical scrolling is performed by shifting the screen instead, allowing rows scrolled off the screen to be preserved in the virtual buffer.

The Terminal_Emulator API provides procedures for manipulating the current size and position of the region within the screen, and also for enabling and disabling it. See section 2.4.6.

2.2.4 View

The view is a window onto the virtual buffer. It represents the portion of the virtual buffer that is displayed in the terminal window. The view can be smaller than the virtual buffer or the same size. It cannot be larger than the virtual buffer. In essence, the view can slide up and down, or left and right within the virtual buffer as necessary.

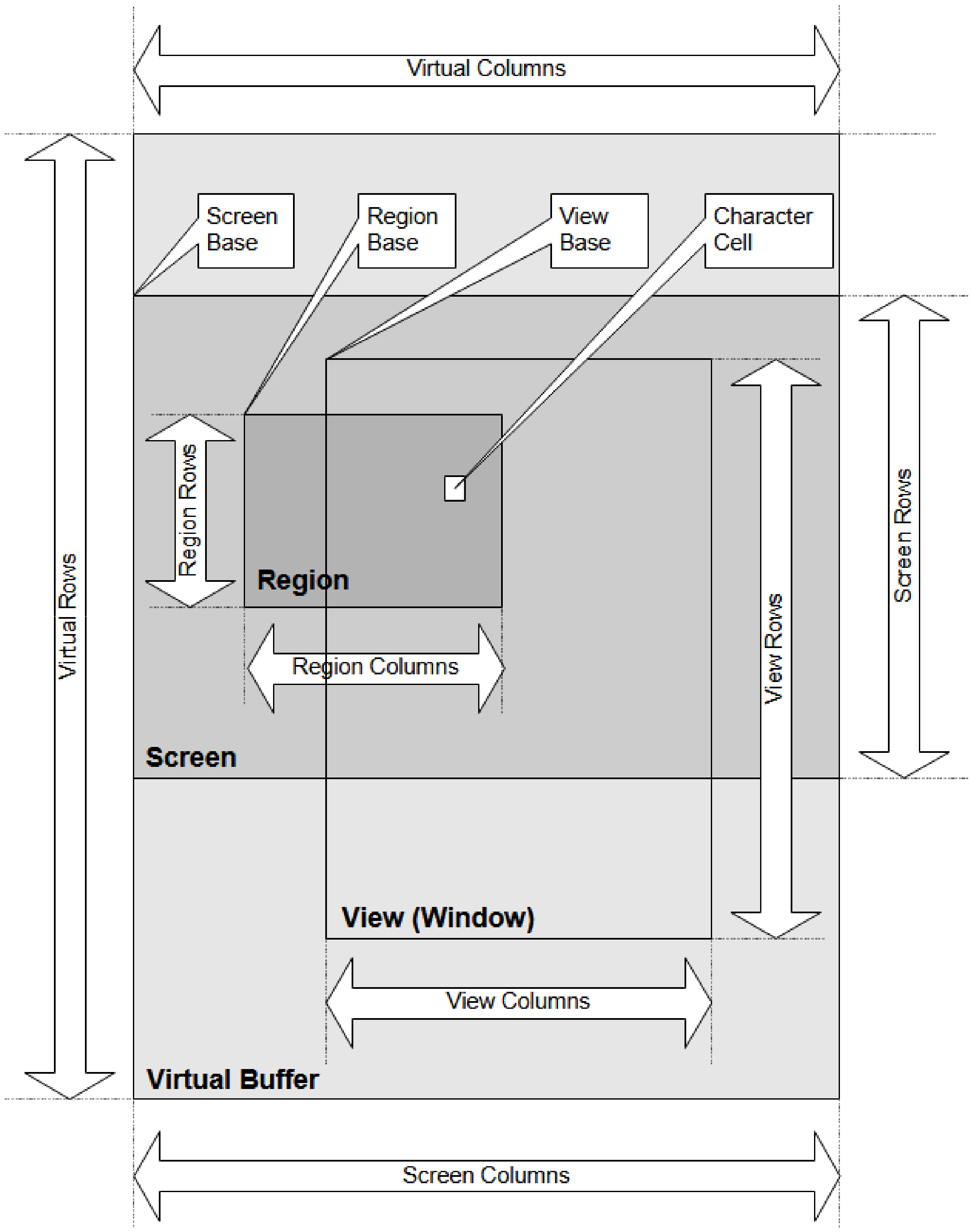
Note that the view is not restricted to the screen, but can overlap it. This means that the terminal window may contain only a portion, or none, of the active screen (or region). This can initially be confusing - since the cursor can only be moved within the screen (or region) there may be character cells on view that cannot be directly addressed via the cursor, or even via the API.

Note that the selection mechanism operates on character cells anywhere in the virtual buffer, not just the current view, screen or region. When selecting with the mouse, only characters currently on the view can be selected initially. But the selection can be extended beyond the current view if necessary, and cells remain selected even if they subsequently move off the view.

The Terminal_Emulator API provides procedures for manipulating the current size and position of the view within the virtual buffer. See section 2.4.6.

2.2.5 Summary

The relationships between the Virtual Buffer, Screen, View and Region is summarized in the following diagram:



2.3 Terminal_Emulator Types

This section describes the types used in the API procedures and functions.

`Terminal`

This type is a limited private type that represents the terminal window. It must be specified in all Terminal_Emulator API calls, and must be explicitly opened (using the `Open` procedure) before any other procedure or function can be used. Many variables of type `Terminal` can be declared in the same application, possibly within different tasks in the application. Each one represents a completely separate terminal window that can be used independently.

`Option`

This type is used by the API wherever a configurable options is required, instead of using a Boolean type. The advantage of this is that the Option type has three states instead of two (i.e. `Yes`, `No` or `Ignore`). When used in conjunction with named parameter association, it allows a very concise means of specifying options without requiring a separate procedure for each, since the default if the option is not specified is usually “ignore”. For example, the `SetOtherOptions` procedure has about a dozen miscellaneous options. Instead of having a dozen procedures to be able to specify each option independently, we just write (for example):

```
SetOtherOptions (Term, HalftoneEnabled => Yes);
```

`Row_Size`

This type is used to specify whether an entire row (line) of characters is single width (occupies one character cell), double width (occupies two adjacent character cells), or double height (see below). It is used in the `SetRowOptions` procedure.

In keeping with the DEC method of specifying double height characters, to make characters double height the same characters must be written to two adjacent rows, in the same column positions. The rows are then configured to display either the upper or lower half of each character. Note that double height characters are also double width.

`Sizing_Mode`

This type is used when specifying how the terminal window responds to resizing (i.e. dragging the window border with the mouse, or maximizing or restoring the window). The window will either adjust the font size to accommodate the new window size (without changing the screen or view size), adjust both the screen size and the view size, or adjust the view size independently of the screen size. If the view size and the screen size differ, the horizontal and vertical scroll bars can be used to show different parts of the screen within the view. Note that the view size can be taller than the screen size (i.e. contain more rows) but not taller than the virtual buffer size. The view size cannot be wider than the screen (or virtual buffer) size.

`Modifier_Key_Type`

This type is used when extended keys are enabled (via the `ExtendedKeys` option in the `SetKeyOptions` procedure). It indicates whether the shift, control and/or alt keys were pressed when the key was pressed. Reading extended keys also requires the use of the `GetExtended` procedure, instead of the `Get` procedure, which returns only keys that can be represented by ASCII characters.

`Special_Key_Type`

This type is used when special (also known as extended) keys are enabled (via the `ExtendedKeys` option in the `SetKeyOptions` procedure). It is used to represent keys (such as the arrow cursor keys and function keys) that cannot be represented by an ASCII character. Reading extended keys also requires the use of the `GetExtended` procedure, instead of the `Get` procedure, which returns only ASCII characters. The special key values are given in the file `terminal_emulator.ads.`

Note for developers: The type `Special_Key_Type` is a renaming of the corresponding type from GWindows. Having this in the Terminal_Emulator package means it is not necessary to explicitly “with” any GWindows packages.

`Ansi_Mode`

This type is used to specify the mode for interpreting ANSI escape sequences. The modes are `PC`, `VT52`, `VT100`, `VT101`, `VT220`, `VT320` and `VT420`. The meaning of each mode is explained in Appendix A.

`Color_Type`

This type is used to represent font and screen colors. Note that colors are not limited to the colors explicitly named in `terminal_emulator.ads` – additional colors can be defined using suitable RGB values.

Note for developers: The type `Color_Type` *is a renaming of the corresponding type from GWindows. Having this in the Terminal_Emulator package means it is not necessary to explicitly "with" any GWindows packages. See the GWindows.Colors package for examples of how to define new colors.*

`Font_Type`

This type is used to refer to a type of stock font, and is used by one variant of the `Open` procedure, and also by the `SetFontByType` procedure. It is an alternative to explicitly specifying a font by name. Note that stock fonts have at least one major limitation – they cannot be resized, which means that font sizing cannot be used with stock fonts. However, they are guaranteed to be available on all platforms, whereas any specific font name may not.

Note for developers: The type `Font_Type` *is a renaming of the corresponding type from GWindows. Having this in the Terminal_Emulator package means it is not necessary to explicitly "with" any GWindows packages.*

`Charset_Type`

This type is used to specify the character set of a font, and is used by one variant of the `Open` procedure, and also by the `SetFontByName` procedure. Note that the `ANSI_CHARSET` character set must be used for DEC VTxxx compatibility. For an explanation of this see section 2.9.14. Also note that the `OEM_CHARSET` must be used to select the Terminal font, since that font does not support an `ANSI_CHARSET`. If the `OEM_CHARSET` is not specified, or another character set type is specified, then Windows may substitute a different font that supports the character set.

Note for developers: The type `Charset_Type` *is just an integer, with a set of constants corresponding to those defined in the package* `common_dialogs.ads`*. Having this in the Terminal_Emulator package means it is not necessary to explicitly "with" this package.*

2.4 Terminal_Emulator Semantics

2.4.1 Basic Operations

This section describes the most fundamental API procedures and functions. Many programs will need no more than these procedures and functions to be able to use almost all the facilities provided by the terminal emulator:

`Open`

The `Open` procedure must be called before any other procedure in the API, to open a terminal window. The parameters to the `Open` procedure include a terminal variable (i.e. a variable of type `Terminal`), the title of the terminal window, the number of virtual buffer rows to allocate, the initial size of the screen (and view), the position of the window, the initial menus to display, and whether or not the window should stay topmost in the Z-order. Note that the terminal window can also be opened but not made visible. This is typically used when additional terminal window options will be configured after it has been created, and it is not desirable for the user to see the window appear and then change (e.g. change size or color) as the additional configuration is performed. The window can be made visible after it is completely configured – see `SetWindowOptions`.

The `Open` procedure also specifies the initial font to use for all text displayed in the terminal window. There are two variants to the procedure – one that accepts a font name, character set and size, and the other that accepts a font type. Specifying a font type uses the Windows stock font of the corresponding type. Such fonts are "built in" to Windows and are guaranteed to be available – but they may not look very good and don't support font sizing. Also, since stock fonts don't usually print well, this variant of the procedure allows a separate font name to be specified for printing. Only use the stock font variant of the open procedure if it cannot be guaranteed that a font specified by name will be available.

The initial cursor position when the terminal window is opened is at the top left corner (column=0, row=0). The initial style is all text attributes (bold, italic, underline etc) turned off. The initial colors are white foreground on a black background.

Examples of a minimal `Open` operation of each variant:

```
Open (Term); -- open a window using a stock font
Open (Term, Font => "Lucida Console", Size => 12); -- use a named font
```

Note that the `ANSI_CHARSET` character set must be used for DEC VTxxx compatibility. For an explanation of this see section 2.9.14.

`Close`

The `Close` procedure can be used to close the terminal window. This may or may not terminate the program, depending on the close options specified using the `SetWindowOptions` procedure. The default is that closing the terminal window does *not* terminate the program. The terminal window can also be closed using the "File -> Exit" menu command, or via the close button on the title bar of the terminal window (unless these have been explicitly disabled). The effect is similar in all cases.

Note for developers: Closing the terminal window using any method will cause a `TASKING_ERROR` exception to be generated from any calls to API procedures that are currently queued and waiting on the terminal, such as `Get` or `GetExtended`. This behaviour is intentional.

`Closed`

The `Closed` function can be used to check if a terminal is currently open or closed. It returns a Boolean value.

`Get`

The `Get` procedure can be used to read characters typed on the keyboard (when the terminal window has the focus). It returns ASCII characters. It will queue until a character is available for reading (i.e. a key is typed).

Keystroke data is queued by the terminal emulator until read. The size of the key buffer is configurable (via the **Size** option in the **SetKeyOptions** procedure). A size of zero means that keyboard input is disabled for that terminal window. This is useful if a program never intends to read from the keyboard – if key input is enabled but the key buffer fills up with unread data, and the terminal window beeps every time a key is pressed to indicate a keystroke has been lost.

Note that if extended keys are enabled (via the **ExtendedKeys** option in the **SetKeyOptions** procedure), and the **Get** procedure is called, it will return a null character (**ASCII.NUL**) whenever the next key in the key buffer is a special key that cannot be represented as an ASCII character. The special key will be lost. To avoid this, use **GetExtended** instead of **Get** whenever extended keys are enabled.

*Note for developers: If a **Get** is currently queued waiting for a keystroke and the terminal window is closed, a **TASKING_ERROR** exception will result. This is deliberate. To handle this, code similar to the following could be used:*

```
begin
   Get (Term, Char);
exception
   when others =>
     do_something_else;
end;
```

A simple procedure could be written that explicitly traps this exception and returns a Boolean flag instead if required.

Put

The **Put** procedure can be used to send text for display in the terminal window. It has two variants – one for sending a single character, and one for sending a string. Using the minimal form of the **Put** procedure, the character or string will be displayed at the current cursor position, and will use the current character attributes and colors.

Note that ANSI control sequences can be constructed and sent either as single characters or as a string. If ANSI processing is enabled (it is enabled by default) then such ANSI sequences will not be displayed, but will be executed by the terminal emulator. ANSI control sequences and characters for display can be combined in the same string.

Examples of a **Put** operation of each variant:

```
Put (Term, ASCII.CR);                -- send carriage return
Put (Term, ASCII.ESC & "[1;1Hhi!");  -- print "hi!" at top left corner
```

The **Put** procedure has some optional parameters: **Row** and **Column** can be used to set the cursor position before the **Put** is performed (the default for these ensures the current cursor position will be used), and **Move** determines whether or not the cursor is moved as a result of the procedure (the default for this is **Yes**). These are just a programmatic convenience – the same effects can be achieved by explicitly setting the cursor position before the **Put**, or by saving the cursor before the **Put** and restoring it afterwards.

2.4.2 Hello World

The following example illustrates most of the basic operations discussed so far. It implements the traditional "Hello World" program using the **Terminal_Emulator** package:

```
with Terminal_Emulator;
procedure Hello_World is
   pragma Linker_Options ("-mwindows");
   use Terminal_Emulator;
   Term : Terminal;
   Char : Character;
Begin
   Open (Term);
   Put (Term, ASCII.ESC & "[12;32HHello, World !");
   Get (Term, Char);
   Close (Term);
end Hello_World;
```

This program opens a terminal window and displays "Hello, World !" in the middle. Note the use of the ANSI standard control sequence `ESC[12;32H` to position the output in the middle of the window.

Some points worth noting:

- There are API calls that can be used instead of the ANSI control sequence to move the cursor.
- The `Linker_Options` indicate that this program is to be linked as a windows mode application, not as a console mode application. The Terminal_Emulator package does not use console I/O, but the default terminal window does look a bit like a console window. However, terminal windows are *much* more functional.
- If a key is pressed in the terminal window, the terminal closes down gracefully. If instead the terminal is closed using the Windows close button or the "File->Exit" menu item, a `TASKING_ERROR` exception will be raised. This occurs because the program is queued at the `Get` procedure, waiting for the user to press a key before closing the terminal, and is the expected behaviour. See the description of the `Get` procedure for details on how to deal with the exception.

2.4.3 Cursor and Text Operations

This section describes the API procedures and functions involved with moving the cursor around the screen, and setting various text attributes and text colors. A set of specific text attributes is referred to collectively as a “style”.

Note that the procedures in this section are mainly intended for use when the `CombinedCursor` and `CombinedStyle` options are in effect, which means there is essentially a single terminal cursor, with a single current style and color. This is the default mode, and the most common mode for most terminal emulation applications (e.g. it is assumed for all the DEC VTxxx emulation modes). When this is not the case, the procedures defined in section 2.4.5 (which differentiate between input and output cursors and styles) should be used explicitly instead.

SetPos

This procedure can be used to set the character position of the cursor. It accepts a row and column parameter. If either parameter is outside the current screen, that particular parameter is ignored. The character position column=0, row=0 is the top left of the screen.

Note that cursor references are always to character positions, not screen cell positions. The two are identical if all rows on the screen are single width, but differ when double height or width characters are used. See the description of `SetRowOptions` for more details.

Note for developers: This procedure is a rename of ***`SetInputPos`****. Refer to section 2.4.5.*

GetPos

This procedure can be used to retrieve the character position of the cursor. It accepts a row and column variable. The character position column=0, row=0 is the top left of the screen.

Note that cursor references are always to character positions, not screen cell positions. The two are identical if all rows on the screen are single width, but differ when double height or width characters are used. See the description of `SetRowOptions` for more details.

Note for developers: This procedure is a rename of ***`GetInputPos`****. Refer to section 2.4.5.*

SetStyle

This procedure can be used to set the text attributes that will be applied to any text written to the terminal using the `Put` procedure. The text attributes that may be specified (in any combination) are:

- Bold
- Italic
- Underline
- Strikeout
- Inverse
- Flashing

Not all attributes need to be specified. Using named parameter associations allows specific attributes to be set or reset independently. The following example turns on the bold attribute, turns off the italic attribute, and leaves the others untouched:

```
SetStyle (Term, Bold => Yes, Italic => No);
```

Note for developers: Windows does not support flashing text natively - it is emulated by the Terminal_Emulator package in software, and can be CPU intensive if there is a lot of flashing text on a large screen. There is a means of disabling the software emulation of flashing text. See the ***`FlashingEnabled`*** *option to the* ***`SetOtherOptions`*** *procedure.*

Note for developers: This procedure is a rename of ***`SetOutputStyle`****. Refer to section 2.4.5.*

SetFgColor

This procedure is used to set the foreground text color that will be applied to text written to the terminal using the `Put` procedure. Note that colors are not limited to those named in `terminal_emulator.ads` – additional colors can be defined using suitable RGB values.

Note for developers: This procedure is a rename of `SetOutputFgColor`. Refer to section 2.4.5.

SetBgColor

This procedure is used to set the background text color that will be applied to text written to the terminal using the `Put` procedure. Note that colors are not limited to those named in `terminal_emulator.ads` – additional colors can be defined using suitable RGB values.

Note for developers: This procedure is a rename of `SetOutputBgColor`. Refer to section 2.4.5.

2.4.4 Font, Keyboard and Screen Operations

This section describes the API procedures and functions involved with slightly more advanced screen and keyboard manipulation.

SetFontByName

This procedure can be used to select a named font, or change the font size or character set. Note that whereas text attributes such as bold and italic are defined per character, the font name, font size and character set apply to all characters in the virtual buffer (apart from the special cases of double width and double height characters). The terminal window will be resized to accommodate the new font name and size.

The font name, font size and character set can be set independently, or at the same time. The following examples illustrate some of the possibilities:

To set the font name without changing the character set or size:

```
SetFontByName (Term, SetFont => Yes, Font => "Lucida Console");
```

To set the character set without changing the font name or size:

```
SetFontByName (Term, SetChar => Yes, CharSet => ANSI_CHARSET);
```

To set the font size without changing the font name or character set:

```
SetFontByName (Term, SetSize => Yes, Size => 14);
```

To set both the font size and the font name:

```
SetFontByName (Term,
   SetFont => Yes, Font => "Courier",
   SetSize => Yes, Size => 24);
```

Font sizes from 1 to 36 point are supported by the terminal emulator, but not all these sizes will be supported by the font itself. If the requested size is not supported, the nearest supported size is substituted. The maximum font size 36 is arbitrary, and can be changed at compile time if required.

Note that the `ANSI_CHARSET` character set must be used for DEC VTxxx compatibility. For an explanation see section 2.9.14.

Note that to specify a font that does not support the `ANSI_CHARSET` (e.g. The Windows Terminal font), the character set must also be explicitly specified (in this case, the `OEM_CHARSET`). Otherwise Windows may substitute another font.

Finally, note that just because a terminal window was opened using a named font, this does not mean that only named fonts can be selected thereafter. A stock font can be selected at any time using `SelectFontByType`.

SetFontByType

This procedure can be used to select a stock font type. Note that whereas text attributes such as bold and italic are defined per character, the font type and size applies to all characters in the virtual buffer (apart from the special cases of double width and double height characters). Note that since stock fonts cannot be resized, the size parameter for stock fonts refers to the size used when printing, not the size used in the terminal window.

As an example, to set the stock font type without changing the size:

```
SetFontByType (Term, SetType => Yes, Font => Default_GUI);
```

Finally, note that just because a terminal window was opened using a stock font, this does not mean that only stock fonts can be selected thereafter. A named font can be selected at any time using `SelectFontByName`.

`ClearToEOL`

This procedure can be used to erase characters on a row (line), from the current cursor position to the end of the line. The erased characters are set to spaces (‘ ‘), with the current style and color.

`ClearKeyboard`

This procedure empties the keyboard buffer, removing any key strokes that have been received by the terminal emulator but not yet read.

`Peek`

This procedure can be used to see if there are any keys in the key buffer waiting to be read. It returns a Boolean value in the `Ready` parameter, and a character in the `Char` parameter. The `Ready` parameter will be set to `True` if there are any characters in the keyboard buffer, or `False` otherwise. If there are any keys in the keyboard buffer, the `Char` parameter will be set to the character value of the first key in the buffer. The key is not removed from the keyboard buffer. Note that if the next character in the keyboard buffer is an extended character, the character returned will be `ASCII.NUL`.

Unlike the `Get` or `GetExtended` procedure, this procedure does not queue.

`UnGet`

This procedure can be used to push a character back onto the key buffer, as if it had never been read. Note that only characters can be pushed back, not extended keys.

`UnGet` will not work if the key buffer has been disabled (by setting `Size` to zero in the `SetKeyOptions` procedure).

Note for developers: one additional key buffer position over and above the requested key buffer size is reserved for `UnGet`. Therefore a single `UnGet` will always succeed. After that, `UnGet` will only succeed if there is space left in the key buffer.

`GetExtended`

This procedure is analogous to the `Get` procedure, and should be used in preference to `Get` whenever extended keys are enabled (via the `ExtendedKeys` option to the `SetKeyOptions` procedure). As well as returning a character value, this procedure can return special key values (such as `Left_Key` or `F1`), and a modifier value. If the special key value is `None` then the key represents a normal character key, and its value is the returned character.

Note that for normal keys, any modifier is incorporated into the returned character as well as being returned. For example, if `*` is typed by holding down the `shift` and `8` keys, the modifier will indicate the shift, but the character will be `*` (not `8`).

Note for developers: If a `GetExtended` is currently queued and the terminal window is closed, a `TASKING_ERROR` exception will result. See the note on the corresponding situation in the description of the `Get` procedure.

`ClearScreen`

This procedure clears the screen. All character cells will be set to the current default text attributes and colors, and the space character (‘ ‘). All cursors will be reset to column=0, row=0.

`SetCursorColor`

This procedure sets the input cursor to the specified color. Note that the actual color of the character cell where the input cursor is currently located is not necessarily this color. The color determined as follows:

- If neither the background nor foreground cell color is the cursor color, the cell character is displayed with its normal foreground color, and with a background color of the cursor color.

- If the foreground cell color is the cursor color then the cell character is displayed with the foreground color of the normal background color, and with a background color of the cursor color.
- If the background color is the cursor color then the cell character is displayed with the foreground color of the cursor color, and with a background color of the normal cell foreground color.

Note that colors are not limited to those named in `terminal_emulator.ads` – additional colors can be defined using suitable RGB values.

SetScreenColors

This procedure sets the foreground and background colors of all character cells on the screen. Note that colors can explicitly be specified using `FgColor` and `BgColor`, or the `Current` option can be used to specify that the screen should be set to the current output colors.

Here is an example of both methods:

```
SetScreenColors (Term, FgColor => Blue, BgColor => Black);
SetScreenColors (Term, Current => Yes);
```

Note that this does not set color values for any characters not currently on the screen.

SetBufferColors

This procedure sets the foreground and background colors of all character cells in the virtual buffer. Note that colors can explicitly be specified using `FgColor` and `BgColor`, or the `Current` option can be used to specify that the screen should be set to the current output colors.

Here is an example of both methods:

```
SetBufferColors (Term, FgColor => Blue, BgColor => Black);
SetBufferColors (Term, Current => Yes);
```

ClearBuffer

This procedure clears the virtual buffer. All character cells will be set to the current default text attributes and colors, and the space character (‘ ‘).

Scroll

This procedure can be used to scroll the screen (or region, if regions are enabled) up or down a number of rows (lines). The procedure accepts an optional parameter specifying the number of rows. The default if not specified is one row.

If regions **are not** currently enabled, the screen will be moved down (which means the characters on the screen will appear to move up) if the `Rows` parameter is positive, and the screen will be moved up (which means the characters on the screen will appear to move down) if the `Rows` parameter is negative. Any characters that are scrolled off the screen will continue to be displayable until they scroll off the virtual buffer (which may be far larger than the screen).

If regions **are** currently enabled, the characters in the region will appear to move up if the `Rows` parameter is positive, and the characters in the region will appear to move down if the `Rows` parameter is negative. The screen is not moved when regions are enabled, and any characters that scroll off the region are lost.

Shift

This procedure can be used to shift the screen (or region, if regions are enabled) left or right a number of columns (characters). The procedure accepts an optional parameter specifying the number of columns. The default if not specified is one column.

If regions **are not** currently enabled, the characters on the screen will appear to move left if the `Rows` parameter is positive, and the characters on the screen will appear to move right if the `Rows` parameter is negative. Any characters that are scrolled off the screen are lost.

If regions **are** currently enabled, the characters in the region will appear to move left if the `Rows` parameter is positive, and the characters in the region will appear to move right if the `Rows` parameter is negative. Any characters that scroll off the region are lost.

2.4.5 Separate Input and Output Cursor and Style Operations

The terminal emulator can be configured to use either a single cursor for all input and output operations (known as the combined cursor), or two separate cursors (known as the input cursor and the output cursor). The default is to use a combined cursor. Separating or combining the cursors can be done via the `CombinedCursor` option to the `SetOtherOptions` procedure.

The following is a summary of the differences between the input cursor and the output cursor when they are separate:

The **input** Cursor:

- Is visible to the user (unless it has been explicitly made *not* visible);
- Can be moved by the user with the mouse or cursor keys (if these are enabled);
- Is the position where keys entered by the user are echoed (if echo is enabled);
- Is the position where pasted characters will appear (if paste to buffer is enabled).

The **output** Cursor:

- Is always invisible;
- Is the position where `Put` characters or strings will appear;
- Is the position where `ClearEOL` erases;
- Is the position where characters loaded from a file will appear.

When the cursors are combined, all operations effectively use the same cursor (in fact they use the **input** cursor).

The terminal emulator can also be configured to use either a single text style for all input and output operations (known as the combined style), or two separate styles (known as the input style and the output style). Similarly for text colors. The default is to use single style and colors. Separating or combining the styles can be done via the `CombinedStyle` option to the `SetOtherOptions` procedure.

When the styles are combined, all operations requiring a text style or color effectively use the same style (in fact they use the **output** style and colors). When the styles are separate, operations that use the **input** cursor use the input style and colors, and operations that use the **output** cursor use the output style and colors (by default).

Many terminal emulation programs will only ever need to use combined cursors and styles. The main use for separating the cursors and styles is to allow the screen to be modified without interfering with what a user may be typing on the terminal. See the **Demo_Cursors** demonstration program for an example.

Section 2.4.3 described the Terminal_Emulator procedures for cursor and text manipulation when the input and output cursors and styles are combined. These procedures can continue to be used when the cursors or styles are separate – they actually operate on the **input** cursor and the **output** style. Many of the procedures described in this section are identical, but can be used to explicitly manipulate either the **input** and **output** cursor and style. If separate cursors or styles are used, it is preferable to explicitly use these procedures rather than those in section 2.4.3. This will avoid potential confusion.

`SetOutputFgColor`

Same as `SetFgColor` (see section 2.4.3) - sets the **output** foreground text color.

`SetOutputBgColor`

Same as `SetBgColor` (see section 2.4.3) - sets the **output** background text color.

`SetOutputStyle`

Same as `SetStyle` (see section 2.4.3) - sets the **output** style.

`SetInputFgColor`

Same as `SetFgColor` (see section 2.4.3), except that it sets the **input** foreground text color.

`SetInputBgColor`

Same as `SetBgColor` (see section 2.4.3), except that it sets the **input** background text color.

`SetInputStyle`

Same as `SetStyle` (see section 2.4.3), except that it sets the **input** style.

`SetInputPos`

Same as `SetPos` (see section 2.4.3) - sets the **input** cursor position.

`GetInputPos`

Same as `GetPos` (see section 2.4.3) - returns the **input** cursor position.

`SetOutputPos`

Same as `SetPos` (see section 2.4.3), except that it sets the **output** cursor position.

`GetOutputPos`

Same as `GetPos` (see section 2.4.3), except that it returns the **output** cursor position.

`PushInputPos`

This procedure pushes the current position of the **input** cursor onto an internal stack. It does not change the current cursor position. The cursor can later be restored with `PopInputPos`.

The internal cursor stack can hold up to 10 elements. There is a separate stack for the **input** cursor and the **output** cursor.

`PopInputPos`

This procedure sets the current position of the **input** cursor to the top element of the internal cursor stack. The element is removed. The procedure accepts three optional parameters: `Discard` can be set to `Yes` to simply discard the top element without using it, `Show` can be set to `Yes` to automatically scroll the view to ensure that the relevant line is on display, and `Force` can be set to `Yes` to scroll the screen to ensure that the relevant line is on the screen. Of course, `Force` and `Show` have no effect if `Discard` is set. Also, if `Force` is not set and the relevant cursor position is no longer on the screen (because the screen has moved) then it is effectively discarded. Note that the screen or view only moves vertically. This ensures that the row will be on the current view or screen (or both), but not necessarily the column.

The major advantage of pushing and popping the cursor, rather than simply getting the current cursor position (e.g. with `GetPos)` and later restoring it (e.g. with `SetPos)` is that cursor values on the internal stack are automatically adjusted to accommodate virtual buffer and screen resizing that the application program may be otherwise unaware of, and also that it is guaranteed the popped cursor will be positioned on the same position even if the screen scrolls, provided that the original position still exists somewhere in the virtual buffer.

`PushOutputPos`

Same as `PushInputPos` (see above), except that it pushes the **output** cursor.

`PopOutputPos`

Same as `PopInputPos` (see above), except that it pops the **output** cursor.

2.4.6 Virtual, Screen, Region and View Operations

This section describes procedures and functions for specifying the size of the virtual buffer, the size and position of the screen within the virtual buffer, the size and position of the view within the virtual buffer, and the size and position of the region within the screen.

Since all these default to sensible values, and are managed within the Terminal_Emulator package automatically as required, it is usually unnecessary to use any of these procedures in most applications.

Before using any of these procedures, review the descriptions of the virtual buffer, screen, view and region, and the relationships between them given in section 2.2.

`SetVirtualSize`

This procedure can be used to resize the virtual buffer. It accepts an optional `Rows` parameter specifying the number of rows in the virtual buffer. If the parameter is not specified, or is less than the current number of rows on the screen, the size of screen (in rows) is used instead.

If not specified in the `Open` procedure, the initial size of the virtual buffer is set to the default values specified in the file `terminal_emulator.ads`.

Note that the number of columns in the virtual buffer is always the same as the number of columns on the screen, and can only be adjusted by adjusting the screen size using the `SetScreenSize` procedure (below).

`SetScreenBase`

This procedure can be used to adjust the current position of the screen within the virtual buffer. It accepts optional `Column` and `Row` parameters. These represent the offset of the top left screen position within the virtual buffer. The top left hand corner of the virtual buffer is Column=0, Row=0. If either parameter value is not specified, it defaults to zero.

The initial value for the screen base is Column=0, Row=0.

Note that currently the screen and the virtual buffer always have the same number of columns, so the only valid value for the `Column` parameter is zero. The largest valid value for the `Row` parameter is the size of the virtual buffer, less the size in rows of the current screen.

`SetScreenSize`

This procedure can be used to adjust the current size of the screen. It accepts optional `Column` and `Row` parameters. If either parameter is not specified, the procedure uses default values for the screen size columns and rows (specified in the file `terminal_emulator.ads`).

If not specified in the `Open` procedure, the initial size of the screen is set to the default values specified in the file `terminal_emulator.ads`.

Note that currently the screen and virtual buffer always have the same number of columns, so adjusting the width of the screen also adjusts the width of the virtual buffer.

The view size will be adjusted automatically if the new screen size is narrower than the current view size, since the view cannot be wider than the screen.

`SetViewBase`

This procedure can be used to adjust the current position of the view within the virtual buffer. It accepts optional `Column` and `Row` parameters. These represent the offset of the top left view cell within the virtual buffer. The top left hand corner of the virtual buffer is Column=0, Row = 0. If either parameter value is not specified, it defaults to zero.

Note that the largest for the `Column` parameter is the width of the virtual buffer, less the size in columns of the current view. The largest valid value for the `Row` parameter is the size of the virtual buffer, less the size in rows of the current view.

The initial value for the view base is Column=0, Row=0.

Note that the view is not limited to the screen. The view may or may not overlap with any part of the screen.

SetViewSize

This procedure can be used to adjust the current size of the view. It accepts optional `Column` and `Row` parameters. If either parameter is not specified, the procedure uses a value corresponding to the current screen size.

Note that the view can never be wider than the screen, or larger than can be accommodated on the display – if a larger value is specified then the largest valid value is used instead.

The initial size of the view is usually set to be the same as the initial size of the screen, unless it is too large to fit on the current display. In this case it will be set to be as large as possible instead.

SetRegionBase

This procedure can be used to adjust the current position of the region within the screen. It accepts optional `Column` and `Row` parameters. These represent the offset of the top left region cell within the screen. The top left hand corner of the screen is Column=0, Row = 0. If either parameter value is not specified, it defaults to zero.

Note that a valid region has to include at least two lines and one column. Therefore the largest valid value for the `Row` is two less than the screen height (in rows), and the largest valid value for the `Column` parameter is one less than the screen width (in columns). If either value is invalid, the request is ignored. The default region base is Column=0, Row=0.

Note that setting a region base does not enable the use of the region. This must be done explicitly via the `Region` option to the `SetScrollOptions` procedure.

SetRegionSize

This procedure can be used to adjust the current size of the region within the screen. It accepts optional `Column` and `Row` parameters. If either parameter is not specified, the procedure uses a value corresponding to the current screen size

Note that a valid region has to include at least two lines and one column. Therefore the smallest valid value for the `Row` is two, and the smallest valid value for the `Column` parameter is one. If either value is invalid, the request is ignored. The region size can be no larger than the screen size. The default region size is the same as the screen size.

Note that a request to set a region size that (when combined with the current region base) would lead to an invalid region is ignored. Therefore it is usually necessary to set the region base first before setting the region size.

Note that setting a region size does not enable the use of the region. This must be done explicitly via the `Region` option to the `SetScrollOptions` procedure.

GetScreenSize

This procedure can be used to retrieve the current size (columns and rows) of the screen.

GetBufferInformation

This procedure can be used to retrieve information about the size of the real buffer, the size and position of the virtual buffer within the real buffer, the size and position of the screen within the virtual buffer, the size and position of the region within the screen, and the size and position of the view within the virtual buffer.

Note that while the `GetScreenSize` procedure can be used to retrieve the screen size, the `GetBufferInformation` procedure is the only way of getting the screen base, or the view or region base and size. However, since the terminal emulator usually manages these internally, they are rarely required.

2.4.7 Setting Options

The Terminal_Emulator package has a multitude of options. Some are used often, some very rarely. Each set of options with related functionality is grouped into a single procedure, but can be set or reset independently. Note that in most cases the default settings of each option will be appropriate, and need not be changed.

SetEditingOptions

This procedure can be used to set options that control the behaviour of the terminal emulator when characters are output to the terminal window, either by typing in the terminal window (when Echo is enabled), or as the result of a **Put** operation:

Wrap

When Wrap mode is enabled, the cursor will move to the first position on the next line when the previous character output was at the last position on the line. Note that the Wrap does not take effect until the next character is output.

The default is that Wrap mode is disabled.

Insert

When Insert mode is enabled, the character in the current cursor position (and all characters to the right of that character on the same row) are moved one position to the right when a character is output. The last character on the row will be lost.

The default is that Insert mode is disabled.

Echo

When Echo mode is enabled, characters typed in the terminal window will be echoed at the current cursor position, using the current style. The cursor position is updated. Echo always occurs at the input (or combined) cursor, using the input (or combined) style and colors.

The default is that Echo mode is enabled.

GetEditingOptions

This procedure can be used to retrieve the current setting of the editing options (Wrap, Insert and Echo). Note that they are returned as Boolean values (not as Option values).

SetRowOptions

This procedure can be used to set options affecting screen rows.

Row

Specifies the row for setting row options. If **Row** is not specified, the row containing the output cursor is used. If the row is specified but is not valid, the request is ignored.

Size

Specifies the size of characters in a row (line). The size applies to all characters in the row. The size can be **Single_Width** (the default), **Double_Width**, **Double_Height_Upper** or **Double_Height_Lower**. Single width characters occupy a single character cell on the screen. Double width characters occupy two adjacent character cells on the screen. Double height characters occupy two adjacent character cells on the screen, and are drawn as the upper or lower half of the complete character. Therefore, to fully draw double height characters, two adjacent rows need to contain the same characters – the upper row should have its size set as **Double_Height_Upper**, and the lower row should have its size set to **Double_Height_Lower**.

Double width or double height rows contain only half the character positions of a normal width line. The second half of the row is not displayed, but is not lost. Rows can be changed from single width to double width and back again at any time.

Note that cursor references always refer to the character position, not the cell position. This means that an API request like **SetInputPos (Term, 3, 0)** will always set the cursor to the fourth character on the first row of the screen (remember references start from zero), whether the row is single width or double width. If the line *is* double width, the actual screen cell position the cursor ends up in will be Column=6, Row=0.

When the cursor is positioned on a double width (or double height) character, the cursor may appear significantly wider than the character, and may in fact partially or completely blank out the next character position. This is normal, and is a consequence of the way double width characters are rendered on the display.

Note that Column and Row references used by **SetRegionBase**, (and most other API requests) refer to the screen cell position, not the character position. This can make the interaction between double width (or double height) rows and regions somewhat non-intuitive. In particular, regions should consist only of rows of the same size. Otherwise the results will probably be unexpected. Also, note that the width of a character is an attribute of the row, not the character. When regions are enabled, the size of a character does not scroll with the character unless the region extends to the full width of the screen row.

SetScrollOptions

This procedure can be used to enable or disable options that affect various scrolling operations for the screen, region or view.

Horizontal

Enable or disable the horizontal scroll bar in the terminal window. This scroll bar can be used to scroll the view horizontally.

The default is that the horizontal scroll bar is disabled.

Vertical

Enable or disable the vertical scroll bar in the terminal window. This scroll bar can be used to scroll the view vertically.

The default is that the vertical scroll bar is disabled.

OnOutput

Enabled or disables scrolling on new output. When enabled and a character is output to the terminal window, the terminal emulator will scroll the view to make the screen row on which the output occurred appear on the view. Note that only vertical scrolling is performed (i.e. the row on which the output occurred will appear, but the column may still be off the view).

The default is that scroll on new output is enabled.

Smooth

Enabled or disables smooth (also known as “soft”) scrolling. When enabled, vertical screen, region or view scrolling is performed by moving up or down one raster line at a time. When disabled, vertical screen, region or view scrolling is performed by moving up or down by whole character cells at a time. The effect of smooth is to make vertical scrolls move slowly and (hopefully) readably.

Smooth scrolling only affects vertical scrolling. Horizontal scrolling (shifting) is always performed by moving left or right by whole character cells.

Even when enabled, smooth scrolling is (temporarily) disabled in some circumstances:

- When the area to be scrolled is a region which includes double width (or height) lines and the region is not as wide as the screen;
- When the area to be scrolled includes selected characters;
- When scrolling with the vertical scrollbar;
- During window resizing

The default setting for smooth scrolling is disabled.

Note to developers: Smooth scrolling can be very CPU intensive. The implementation of smooth scrolling is quite primitive, and the terminal window will not respond to any Windows messages while smooth scrolling is being performed. Therefore, smooth scrolling should be used sparingly, and disabled whenever it is not needed.

Region

Enables or disables Region mode, which determines the use of the scrolling region.

When Region mode is enabled, screen scrolling (and shifting) affects only the area defined by the region. Also, when the cursor is within the region, the cursor stays in the region when a new line or carriage return is performed. Text wrap occurs at the right hand margin of the region, not the screen, and the home position for the cursor is at the left hand margin of the region, not the screen.

When Region mode is disabled, screen scrolling (and shifting) affects the entire screen.

The default setting for Region mode is disabled.

The region must be valid to be enabled. In particular, the region must contain at least one column and two rows.

Note that by moving the cursor with the cursor keys or the mouse (when enabled), and also by using some ANSI control sequences, the cursor may end up outside the region even when the region is enabled. In that case carriage returns, new lines and text wrap temporarily ignore the region until the cursor moves back into it. However, scrolling and shifting the screen still affects only the region.

Enabling or disabling the Region has important implications for the use of the virtual buffer. When the Region is enabled, vertical scrolling affects only the region, even if the region is the same size as the screen. This is significant since it means that lines scrolled off the region are lost. When the Region is disabled, vertical scrolling is performed by shifting the screen instead, allowing rows scrolled off the screen to be preserved in the virtual buffer.

SetMenuOptions

This procedure can be used to enable or disable various terminal window menus.

MainMenu

Enable or disable the display of the main terminal window menu. Refer to section 2.6 for a description of all terminal window menus and menu items.

OptionMenu

Enable or disable the display of the Options menu item in the main terminal window main menu. Note that the main menu itself must be enabled for this menu item to appear. Refer to section 2.6 for a description of all terminal window menus and menu items.

The Option menu item can be enabled or disabled separately to the main menu, since it may be that a terminal application wants to make the File and Format menu items available, but not allow users to change the options of the terminal window, which can affect the way the terminal emulator behaves in ways not expected by the application.

AdvancedMenu

Enable or disable the display of the Advanced menu item in the Option menu of the terminal window main menu. Note that both the main menu and the option menu must be enabled for this menu item to appear. Refer to section 2.6 for a description of all terminal window menus and menu items.

The Advanced menu item can be enabled or disabled separately to the Option menu, since it may be that a terminal application wants to make the File, Format and Options menu items available, but not allow users to change the advanced options of the terminal window, which can affect the way the terminal emulator behaves in ways not expected by the application.

ContextMenu

Enable or disable the display of the terminal window context menu. The Context menu appears when the right mouse button is held down in the terminal window, and contains some commonly used functions. Refer to section 2.6 for a description of all terminal window menus and menu items.

SetCursorOptions

This procedure can be used to set options for the input cursor. There are no user configurable options for the output cursor.

Visible

Enables or disables whether the input cursor is visible.

The default is that the input cursor is visible.

Flashing

Enables or disables whether the input cursor flashes, or is steady. The cursor must be visible for this to have any effect.

The default is that the input cursor is not flashing.

`Bar`

Enables or disables whether the input cursor is drawn as a highlight (by drawing the character on which the cursor is positioned according to the rules described in `SetCursorColor`), or as a thin vertical bar on the left hand side of the character position. The cursor must be visible for this to have any effect.

The default is that the input cursor is a highlight, not a bar.

`SetTitleOptions`

This procedure can be used to affect how the terminal window title bar (sometimes referred to as the “caption”) is displayed.

`Visible`

Enables or disables whether the title bar is visible. Note that if the title bar is not visible, the system menu, maximize/minimize and close window buttons are also not visible.

The default is that the title bar is visible.

`Set`

Enables or disables whether the title will be set by this request (to the value in the `Title` parameter, described below). Otherwise the `Title` parameter is ignored.

By default, the title is not set.

`Title`

If the `Set` parameter is `Yes`, this parameter specifies the title that will be used in the title bar. Otherwise, this parameter is ignored.

The default title is blank.

The following is an example of how to set the title:

```
SetTitleOptions (Term, Set => Yes, Title => "My Title");
```

`SetSizingOptions`

This procedure can be used to set sizing behaviour when the view is resized by the user (i.e. by dragging the window border, or by maximizing or restoring the window).

`Sizing`

This parameter enables or disables the ability for the user to resize the terminal window. It does not affect the ability to resize the window using the API (e.g. via the `SetViewSize` procedure).

`Mode`

This parameter sets the sizing mode. There are three supported sizing modes:

- `Size_Fonts`: When this sizing mode is set (and sizing is enabled), the font size is changed to accommodate the existing view within the resized window. If the window is made smaller, the font is made correspondingly smaller. If the window is made larger, the font is made larger. The terminal emulator supports font sizes from 1 to 36 point, but the currently selected font may not support all font sizes. Only fonts sizes that are supported by the selected font will be used. Note that stock fonts cannot be resized, so this sizing mode is not allowed for stock fonts.
- `Size_Screen`: When this sizing mode is set (and sizing is enabled), the screen size is changed to whatever will fit within the resized window (using the current font size). If the window is made smaller, the screen is made smaller. If the window is made larger, the screen is made larger. The view will be resized along with the screen.
- `Size_View`: When this sizing mode is set (and sizing is enabled), the view size is changed to whatever will fit within the resized window (using the current font size). If the window is made smaller, the view is made smaller. If the window is made larger, the view is made larger. Note that the screen size is not affected. Also note that the view cannot be made any wider than

the screen, or taller than the virtual buffer. If the view is smaller than the screen, the horizontal and vertical scrollbars can be enabled to see the parts of the screen not on the view.

SetAnsiOptions

This procedure can be used to control the parsing of ANSI escape sequences by the terminal emulator.

Note that although parsing ANSI control sequences can be enabled or disabled separately for the input and output streams (see below), there is in fact only a single ANSI parser. If it is enabled on both input and output streams then it is possible that input characters and output characters will intermingle in the parser, causing ANSI control sequences to be misinterpreted or invalidated. Therefore, the ANSI parser would normally only be enabled in one stream at once, and it is most commonly the output stream. One common reason for enabling ANSI control sequences on the input stream is to test ANSI control sequences when manually typed on the keyboard, or cut from another application and pasted into the terminal window.

OnOutput

This parameter enables or disables the parsing of ANSI control sequences in the output stream (e.g. when data is `Put` to the terminal window). If enabled, ANSI control sequences in the output stream are interpreted by the ANSI parser instead of being processed as displayable characters. If disabled, any ANSI control sequences in the output stream are treated the same as any other characters.

Parsing of ANSI control sequences on output is enabled by default.

OnInput

This parameter enables or disables the parsing of ANSI control sequences in the input stream (e.g. when data is typed into the terminal window). If enabled, ANSI control sequences in the input stream are interpreted by the ANSI parser instead of being processed as displayable characters. If disabled, ANSI control sequences in the input stream are treated the same as any other characters. Note that for ANSI control sequences to be interpreted in the input stream, Echo also needs to be enabled.

Parsing of ANSI control sequences on output is disabled by default.

Mode

This parameter sets the mode of the ANSI parser, which may determine how some ANSI control sequences are parsed and interpreted. The supported modes are:

- `PC`
- `VT52`
- `VT100`
- `VT101`
- `VT102`
- `VT220`
- `VT320`
- `VT420`

For details on these modes, refer to Appendix A.

SetMouseOptions

The procedure can be used to determine the way left mouse clicks are interpreted within the terminal window. Right mouse clicks are always interpreted as a request to display the context menu (if the context menu is enabled – see `SetMenuOptions`).

MouseCursor

Enable or disable the mouse cursor mode.

When the mouse cursor mode is enabled, left clicking the mouse within the terminal window is interpreted as a request to move the input(or combined) cursor to the character position under the mouse. Note that this is only possible if the character position is actually on the current screen – just because a character position is on view does not mean it is on the screen. If it is not on the screen, the left click is ignored.

When mouse cursor mode is enabled, selecting with the mouse (see **MouseSelects**, below) must be done by holding down the shift button and left clicking.

The default for the mouse cursor mode is disabled.

MouseSelects

Enable or disable the mouse selection mode.

When the mouse selection mode is enabled, left clicks of the mouse within the terminal window can be used to select the cell position under the mouse. The precise meaning of a left click depends on whether mouse cursor mode (see **MouseCursor** above) is also enabled.

- If mouse cursor mode is disabled, then un-shifted left clicks are interpreted to mean a request to selected the cell position under the mouse. If the shift key is held down while the left button is clicked, left clicks are interpreted as a request to extend the current selection to the cell position under the mouse.
- If mouse cursor mode is enabled, an un-shifted left click is interpreted as a mouse cursor request, and shifted left clicks are interpreted as a mouse selection request – and there is no way to selection an extension.

When selecting, a single left click specifies selection by single character. If the character is already selected, it means to unselect it. A double left click specifies selection by word. A triple left click specifies selection by line. Selecting characters in a rectangular region can be performed by holding down the control key while clicking the left mouse button.

To select more than just what is under the cursor, click the left button (single, double or triple click), but instead of releasing it after the final click, hold the button down and move the mouse. The selection will be extended by character, word, line or region, depending on the number of clicks.

To extend an existing selection (when mouse cursor mode is not enabled), hold down the shift key and left click the mouse. If there is an existing selection, it will be extended to the current character position. The method of extension will depend on the original selection – i.e. it may be by character, word, line or rectangular region.

Note that cell positions do not have to be on the region, screen or view to be selected. They can be anywhere within the virtual buffer. To extend the selection vertically beyond what is currently on the view, hold down the mouse button and move the mouse outside the terminal window. The view will scroll by single rows if the mouse is held within a character cell height beyond the window boundary, or by half the view at once if it is moved further beyond the boundary. This only works vertically. Horizontally, it may be necessary to enable and use the horizontal scrollbar to get the desired character position on view before selecting or extending a selection.

Selected text is displayed with the foreground and background colors inverted. This means that a selected region may be displayed in multiple colors, depending on the foreground color of the cells selected, even if the foreground color is not visible because the character in the cell is currently not using it (such as a space character).

The default for the mouse select mode is enabled.

SetKeyOptions

The procedure can be used to set options relating to the keyboard.

ExtendedKeys

Enable or disable extended key mode, which controls whether extended keys (also known as special keys) can be read by the application program. Extended keys include the cursor keys, function keys, editing keys etc. Refer to the description of **Special_Key_Type**. Note that extended key mode only applies when the ANSI mode of the terminal is set to **PC** (refer to Appendix A).

When extended key mode is disabled, extended keys not otherwise interpreted by the terminal emulator are simply discarded, and cannot be read by an application program. The **Get** procedure can be used to read normal character keystrokes entered in the terminal window. The **GetExtended** procedure can also be used, but will not return extended keys when extended key mode is disabled.

When extended key mode is enabled, extended keys not otherwise interpreted by the terminal emulator are returned to the application. `GetExtended` must be used to read the extended keys - `Get` will return an `ASCII.NUL` character for any extended key, and the actual key will be lost. Refer to the description of `GetExtended`.

The default value for Extended Key mode is disabled.

`CursorKeys`

Enable or disable Cursor Key mode.

When Cursor Key mode is enabled, the cursor keys (Up, Down, Left, Right, Home, End) are interpreted by the terminal emulator, and can be used to move the cursor around the screen. Note that the operation of the Up, Down, Home and End keys may also be affected by other options (see `SetOtherOptions`).

When Cursor Key mode is disabled, cursor keys are not interpreted by the terminal emulator. They may be stored in the key buffer if the ANSI mode is set to `PC` and Extended key mode is enabled. In this case extended keys can be read by an application program (using `GetExtended)`.

The default value for Cursor Key mode is disabled.

`VTKeys`

Enable or disable VT Key emulation mode.

When VT Key emulation mode is enabled, the top row of the numeric keypad is used to simulate the VTxxx keys PF1 to PF4. When VT Key mode is disabled, the top row of the numeric keyboard behaves as usual for a PC, and function keys F1 to F4 are used as the VTxxx keys PF1 to PF4. Note that VT Key mode has no effect if the ANSI Mode of the terminal emulator is not set to one of the VTxxx emulation modes (i.e. the ANSI mode is set to `PC`).

When VT Key emulation mode is enabled, then the Num Lock key will no longer operate as Num Lock, but will instead be interpreted as PF1. However, note that due to the vagaries of Windows keyboard handling, this mode may not work on all platforms. Since there are so many different platforms (i.e. Win98/NT/ME/2000/XP, with various combinations of service packs), the best way to determine if VT Key emulation mode works is by actually trying it. If it works, the Num Lock light will come on when the VT Key emulation mode is enabled, and will stay on even if the Num Lock key is pressed. If VT Key emulation mode does not work on a particular platform, then use the F1 to F4 keys for PF1 to PF4 instead.

Since the VTxxx range of terminals have one additional key on their numeric keypad then (under Windows NT/2000/XP) the '-' key on the PC numeric keypad does double duty in VT Key emulation mode – unshifted, it sends PF4, and shifted it behaves as the '-' key on the VTxxx terminal numeric keypad. This is not possible under Windows 98, since Windows 98 does not appear to support the shift or control modifiers numeric keypad keys. Note also that this does not make SHIFT+'-' the same as the '-' key on the main keyboard. For example, when a VTxxx terminal is set to application keypad mode, the '-' key on the numeric keypad (along with all other numeric keypad keys) send various escape sequences, while the '-' key on the main keyboard remains as just '-'.

When the VT Key emulation mode is enabled and the ANSI mode is set to VT420, F1 to F5 send the appropriate codes for the VT420 keys F1 to F5. In the other VTxxx modes, these keys do nothing.

The default value for VT Key emulation mode is disabled.

Note for developers: VT Keys simulation mode is implemented by checking the num lock key after each key press, and turning it back on if it has been turned off. This achieves the required functionality without having to resort to intercepting the physical scan codes returned from the keyboard driver. Unfortunately, turning num lock on is a platform dependent function in Windows, and this may not work on all platforms. In particular, some versions of Windows 98 and NT require specific service packs to be installed before this functionality will work correctly. For more details, refer to the Microsoft Knowledge Base.

`AutoRepeat`

Enable or disable Auto Repeat mode.

When disabled, normal keystrokes entered in the terminal window do not repeat if the key is held down. This does not apply to special (extended) keys.

When enabled, normal keystrokes entered in the terminal window repeat if the key is held down. The repeat rate will be as configured within Windows.

The default value for Auto Repeat mode is enabled.

`Locked`

Enable or disable the Keyboard Lock.

When the Keyboard Lock is disabled, keystrokes entered in the terminal window will be either interpreted by the terminal emulator or stored in the key buffer (depending on the settings of various other options).

When the Keyboard Lock is disabled, keystrokes entered in the terminal window will not be interpreted or stored in the key buffer. They are lost.

The default value for the Keyboard Lock is disabled.

`SetSize`

This option indicates whether the `Size` parameter (described below) should be used to set the key buffer size. If `Yes`, the size is used. Otherwise it is ignored.

`Size`

The size to be used for the key buffer. Note that for this parameter to be used, the `SetSize` option must be `Yes`. A size of zero disables the keyboard buffer.

The following is an example of how to set the key buffer size:

```
   SetKeyOptions (Term, SetSize => Yes, Size => 100);
```

Note for developers: resetting the size of the keyboard buffer discards any keys in the buffer. An alternative for temporarily disabling keyboard entry is to use the Keyboard Lock, described above.

Note for developers: If the size is not zero, one extra space over and above the requested size is added, and is reserved for UnGet.

`SetWindowOptions`

The procedure can be used to set options relating to the terminal window itself.

`Xcoord`

This parameter can be used to set the horizontal position of the terminal window on the Windows desktop. If not specified, it is ignored.

`Ycoord`

This parameter can be used to set the vertical position of the terminal window on the Windows desktop. If not specified, it is ignored.

`OnTop`

This parameter can be used to set whether the terminal window should stay at the top of the Z-order (i.e. always be on top of other windows). If not specified, it is ignored.

`Visible`

This parameter can be used to set whether the terminal window is visible. If not specified, it is ignored.

`Active`

This parameter can be used to force the terminal window to move to the foreground and acquire the keyboard focus. If not specified, it is ignored. This is a one-off event, and the window may subsequently lose the keyboard focus and/or be moved into the background again.

`CloseWindow`

This parameter can be used to configure whether the terminal window can be closed by the user, either by selecting File->Exit from the menu, or using the close button on the window title bar. If `Yes`, the window can be closed. If **No**, the terminal window cannot be closed by the user, and must be closed by the application program explicitly calling the **Close** procedure. If not specified, this parameter is ignored.

The default setting is that the terminal window can be closed by the user.

CloseProgram

This parameter can be used to specify what happens when the terminal window is closed, either by the user or by an application program calling the **Close** procedure. If **Yes**, the entire application is forcibly terminated. This will close all other terminal windows opened by the application program. If **No**, the terminal window is closed, but the application continues execution. If not specified, this parameter is ignored. When a user closes a terminal window that will terminate the application program, the user is prompted to confirm that this was what was intended. This does not occur when the **Close** procedure is called.

The default setting is that closing a terminal window does not terminate the application program.

SetTabOptions

The procedure can be used to set tab options for the terminal window.

Size

This parameter can be used to specify a default tab size (i.e. a tab stop at every **Size** columns). All other tabs stops will be removed. If **Size** is zero, all tab stops are removed. If not specified, this parameter is ignored.

SetAt

This parameter can be used to set a single tab, at the specified column. If not specified, this parameter is ignored.

ClearAt

This parameter can be used to set a single tab, at the specified column. If not specified, this parameter is ignored.

Note that more than one of these parameters can be used in the same procedure call. For example, the following sets tabs every 8 stops, sets another at column 10, and then clears the first tab set by the Size parameter:

```
SetTabOptions (Term, Size => 8, SetAt => 10, ClearAt => 8);
```

The default setting for tabs is every 8 columns.

SetPasteOptions

The procedure can be used to set options relating to pasting information in the terminal window.

ToBuffer

Enable or disable the Paste To Buffer mode.

When Paste To Buffer mode is enabled, any characters pasted into the terminal window are sent to the screen buffer, at the current input cursor position (i.e. as if the characters had been **Put**, but using the input cursor instead of the output cursor). If the pasted information contains ANSI control sequences, and ANSI processing is enabled on input, then these sequences will be parsed and executed by the terminal emulator.

The default setting for Paste To Buffer mode is disabled.

ToKeyboard

Enable or disable the Paste To Keyboard mode.

When Paste To Keyboard mode is enabled, any characters is pasted into the terminal window are sent to the key buffer (i.e. as if the characters had been typed into the terminal window). The key buffer must have enough free space to contain all the pasted characters, or the paste is ignored. Note that the pasted characters are not echoed in the terminal window, so no ANSI processing is performed on these characters.

The default setting for Paste To Keyboard mode is enabled.

SetOtherOptions

The procedure can be used to set various miscellaneous options relating to the operation of the terminal emulator. These options are usually complex, and the default settings are intended to be the ones appropriate for most applications.

IgnoreCR

Do not process received ACII/CR. They are simply discarded.

IgnoreLF

Do not process received ASCII.LF characters. They are simply discarded.

UseLFasEOL

Enable or disable LF as EOL mode.

When LF as EOL mode is disabled, pressing ENTER on the normal keyboard (or on the numeric keypad when the ANSI mode is set to `PC`) will result in an ASCII.CR being added to the key buffer, and the keystroke will be processed as an ASCII.CR by the terminal emulator (if Echo is enabled).

When LF as EOL mode is enabled, pressing ENTER on the normal keyboard (or on the numeric keypad when the ANSI mode is set to `PC`) will result in an ASCII.LF being added to the key buffer, but the keystroke will still be processed as an ASCII.CR by the terminal emulator (if Echo is enabled).

The default setting for LF as EOL mode is disabled.

Note to developers: See the note on AutoLFonCR, below.

AutoLFonCR

Enable or disable LF on CR mode.

When LF on CR mode is disabled, pressing ENTER on the normal keyboard (or on the numeric keypad when the ANSI mode is set to `PC`) will result in an ASCII.CR being added to the key buffer, and the keystroke will be processed as an ASCII.CR by the terminal emulator (if Echo is enabled).

When LF on CR mode is enabled, pressing ENTER on the normal keyboard (or on the numeric keypad when the ANSI mode is set to `PC`) will result in an ASCII.CR being added to the key buffer, immediately followed by an ASCII.LF. The keystroke will still be processed as an ASCII.CR by the terminal emulator (if Echo is enabled).

When LF on CR mode is disabled, when an ASCII.CR is received by the terminal emulator it is processed as an ASCII.CR.

When LF on CR mode is enabled, when an ASCII.CR is received by the terminal emulator, it is processed as if it were an ASCII.CR followed by and ASCII.LF.

The default setting for LF on CR mode is disabled.

Note to developers: The LF as EOL mode is primarily intended for use by the TERM_IO package, since Ada text I/O typically expects to see a single LF as a line terminator. It is not the same as LF on CR mode, although the two modes may interact. Here is a summary of the interactions:

	LF on CR disabled	**LF on CR enabled**
LF as EOL disabled	CR processed as CR, returns CR	CR processed as CR, returns CRLF
LF as EOL enabled	CR processed as CRLF, returns LF	CR processed as CRLF, returns LF

AutoCRonLF

Enable or disable CR on LF mode.

When CR on LF mode is disabled, when an ASCII.LF is received and processed by the terminal emulator (i.e. if Echo is enabled), it is processed as an ASCII.LF.

When CR on LF mode is enabled, when an ASCII.LF is received and processed by the terminal emulator (i.e. if Echo is enabled), it is processed as if it were an ASCII.CR followed by and ASCII.LF.

The default setting for CR on LF mode is disabled.

`UpDownMoveView`

Enables or disables the Up/Down Keys Move View mode.

If the Up/Down Keys Move View mode is disabled, the view is not moved when the terminal emulator receives an Up or Down key. However, note that the cursor may be moved instead, depending on whether Cursor Keys are enabled (see **`SetKeyOptions`**).

If the Up/Down Keys Move View mode is enabled, the view may be moved up or down when the terminal emulator receives an Up or Down key, as follows:

- If Cursor Keys are not enabled: The view is moved up or down one line (if possible).
- If Cursor Keys are enabled: The view is moved up or down one line (if possible) if the cursor is at the bottom or top of the screen (not the view).

Note that moving the view does not affect the screen, or the location of the cursor on the screen.

The default setting for the Up/Down Keys Move View mode is disabled.

`PageMoveView`

Enables or disables the Page Keys Move View mode.

If the Page Keys Move View mode is disabled, the view is not moved when the terminal emulator receives a Page Up or Page Down key.

If the Page Keys Move View mode is enabled, the view may is moved up or down the number of rows on the current view (if possible) when the terminal emulator receives a Page Up or Page Down key.

Note that moving the view does not affect the screen, or the location of the cursor on the screen.

The default setting for the Page Keys Move View mode is disabled.

`HomeEndMoveView`

Enables or disables the Home/End Keys Move View mode.

If the Home/End Keys Move View mode is disabled, the view is not moved when the terminal emulator receives a Home or End key. However, note that the cursor may be moved instead, depending on whether Cursor Keys are enabled (see **`SetKeyOptions`**).

If the Home/End Keys Move View mode is enabled, the view may be moved up or down when the terminal emulator receives a Home or End key, as follows:

- If Cursor Keys are not enabled: The view is moved to the top or bottom of the virtual buffer.
- If Cursor Keys are enabled: The view is not moved.

The default setting for the Home/End Keys Move View mode is disabled.

Note that moving the view does not affect the screen, or the location of the cursor on the screen.

The default setting for the Home/End Keys Move View mode is disabled.

`HomeEndWithinLine`

Enables or disables the Home/End Keys Within Line mode. This mode only operates when Cursor Keys are enabled.

If the Home/End Keys Within Line mode is disabled, the cursor is moved to the top left corner of the screen when the terminal emulator receives a Home key, and the bottom right corner of the screen when the terminal emulator receives an End key.

If the Home/End Keys Within Line mode is enabled, the cursor is moved to the first character position on the current row (line) when the terminal emulator receives a Home key, and the last character position on the current row (line) when the terminal emulator receives an End key.

The default setting for the Home/End Keys Within Line mode is disabled.

`LeftRightWrap`

Enables or disables the Left/Right Key Wrap mode. This mode only operates when Cursor Keys are enabled.

If the Left/Right Key Wrap mode is disabled, the cursor does not wrap across lines when moved with the cursor keys.

If the Left/Right Key Wrap mode is enabled, the cursor is moved to the first character position on the previous line (row) when the terminal emulator receives a Left Key and the cursor is on the first character position on the line, and to the first position on the next line when the terminal emulator receives a Right Key and the cursor is on the last character position on the line.

The default setting for the Left/Right Key Wrap mode is disabled.

`LeftRightScroll`

Enables or disables the Left/Right Key Scroll mode. This mode only operates when Cursor Keys are enabled, and the Left/Right Key Wrap mode is enabled.

If the Left/Right Key Scroll mode is disabled, the screen does not scroll when the cursor wraps across lines when moved with the cursor keys.

If the Left/Right Key Scroll mode is enabled, the screen is moved down one line if the cursor wraps off the first position on the first line of the current screen, and down one line if the cursor wraps off the last position on the line on the screen.

The default setting for the Left/Right Key Scroll mode is disabled.

`LockScreenAndView`

Enables or disables the Lock Screen and View mode.

If the Lock Screen and View mode is disabled, the screen and view can move independently of one another.

If the Lock Screen and View mode is enabled, the screen and view cannot be moved independently, but are “locked together”. Any request to move (or resize) the view also moves (or resizes) the screen, and vice-versa.

The default setting for the Lock Screen and View mode is disabled.

`CombinedStyle`

Enables or disables the Combined Style mode.

When disabled, the terminal emulator uses separate styles (and colors) for input and output.

When enabled, the terminal emulator uses the same style (and colors) for input and output.

Refer to section 2.4.5 for a discussion of the effects of combined and separate input and output styles.

The default setting of the Combined Style mode is enabled.

`CombinedCursor`

Enables or disables the Combined Cursor mode.

When disabled, the terminal emulator uses separate cursors for input and output.

When enabled, the terminal emulator uses the same cursor for input and output.

Refer to section 2.4.5 for a discussion of the effects of combined and separate input and output cursors.

The default setting of the Combined Cursor mode is enabled.

`FlashingEnabled`

Enables or disables Flashing Text mode.

When disabled, flashing text will not be displayed flashing.

When enabled, the flashing text will be displayed flashing.

Flashing text is not support natively by Windows, and must be emulated in software by the terminal emulator. This can be CPU intensive, especially on large screens with lots of flashing text, which leads to poor terminal emulator performance. When Flashing Text mode is disabled, all Flashing text is displayed as normal text.

The default for Flashing Text mode is enabled.

Note to developers: The flashing cursor is also emulated in software, but it can be disabled separately (see ***`SetCursorOptions`****) and is not affected by this option.*

`HalftoneEnabled`

Enables or disables Halftone mode.

When disabled, bitmapped characters are displayed using normal rendering.

When enabled, bitmapped characters are displayed using halftone rendering.

Halftone rendering can dramatically improve the display of bitmapped characters (especially at small font sizes). However, it can be very CPU intensive, and also is not supported very well by some display drivers (especially at low color depths). Therefore it is not enabled by default. Symptoms that halftone rendering is not supported properly include the screen redrawing v-e-r-y slowly, and also bitmapped characters (such as the DEC special graphics characters) displayed with a different background color to normal text characters. If this occurs, try changing display modes by increasing the color depth, or just disable halftone rendering. Refer to section 2.9.14 for a discussion on characters that are drawn as normal text (i.e. using the selected font), and characters that are actually bitmaps rendered by the terminal emulator.

The default for Halftone mode is disabled.

`SysKeysEnabled`

Enables or disables the System Keys mode.

When disabled, the terminal emulator processes Windows system key presses in the terminal window.

When enabled, Windows processes system key presses in the terminal window.

Windows system keys (such as F10, which selects the menu) are normally processed internally by Windows. When emulating a DEC VTxxx terminal, it is necessary to disable Windows system key processing so the terminal emulator can process the system keys.

The default setting of the System Keys mode is enabled.

Note to developers: Obviously, when this mode is disabled, Windows system keys will no longer respond with their default Windows behaviour. Also, while this enables F10 to be processed (and ALT+F10) which is required for VTxxx modes, there doesn't appear to be any way of stopping Windows processing some other keystrokes, such as ALT+F6 and ALT+F11.

`DisplayControls`

Select the DEC Display Controls character set, and make the terminal emulator display all control codes as displayable characters. Note that using this option also disables the parsing of ANSI control sequences.

Note to developers: This option is very useful for debugging applications. All control characters (as well as all ANSI control sequences) are displayed on the screen instead of being executed.

`DeleteOnBS`

Return ASCII.DEL instead of ASCII.BS when the BACKSPACE key is pressed. Note that this only applies to the BACKSPACE key – CTRL+H will still return ASCII.BS. This option is primarily intended for VT100 emulation, since the VT100 DELETE key is often used to perform the backspace function.

The default setting is that the BACKSPACE key returns an ASCII.BS. To return an ASCII.DEL, you can use CTRL+BACKSPACE.

`RedrawPrevious`

Specifies that when a character is drawn on the current view, the previous character (if there is one) should be redrawn. This dramatically improves the quality of display for italic and/or bold characters. This is because when Windows renders bold or italic characters for fixed width fonts, they may be wider than the standard character cell size, and so overlap into the next character cell. However, it can also slow down the screen updates by up to 30%. This option should be disabled if speed is critical, fixed width fonts are not being used, or italic and/or bold character styles are rarely used.

`RedrawNext`

Specifies that when a character is drawn on the current view, the next character (if there is one) should be redrawn. See the description of `RedrawPrevious` (above) for more details.

2.4.8 Other Operations

This section contains miscellaneous operations that can be carried out on the terminal emulator.

`ScreenDump`

This procedure can be used to retrieve a dump of the current screen contents. It accepts `Column` and `Row` parameters, which indicate where on the screen the dump should begin, and a `Result` variable and a `Length` variable. The screen dump will begin from the specified screen column and row, and the amount of data returned will be the lesser of the length of the `Result` string, or to the end of the screen. To find out the maximum amount of data that can be returned, use the `GetScreenSize` procedure. This means `ScreenDump` can be used to retrieve just a single character, a whole line, any portion of the screen, or the whole screen. The `Length` variable is updated with the length of data actually returned.

`SetPriority`

The terminal emulator creates an Ada task to handle each terminal window. This task performs all terminal emulator processing, all Windows message handling, and all housekeeping functions (e.g. flashing text). The application program operates most efficiently when this task priority is set to a just below the priority of the task or application that uses it. If the priority if the terminal emulator task is equal to or higher than the priority of the task or application that uses it, the internal task may waste time unnecessarily checking for Windows messages, or performing internal housekeeping functions. This can leads to the entire application running slower than necessary. By default, the priority of all the internal terminal windows handling tasks is one less than the system default priority. This ensures that in most cases it will be just lower than the application or task using it. However, if that application or task does not have the system default priority, or changes its own priority dynamically, this procedure can be used to change the priority of the internal terminal emulator task as required.

2.5 Terminal_Emulator Mouse Support

2.5.1 Resizing the Terminal Window

Each terminal window supports resizing by dragging the border (or corners) of the window using the mouse, or via the Windows Maximize/Minimize/Restore buttons. The terminal emulator implements three methods of resizing the terminal window:

- Font sizing;
- Screen sizing; and
- View sizing

Refer to the description of the **`SetSizingOptions`** procedure in section 2.4.7 for a discussion of each sizing mode. Note that Sizing must be enabled if it has been disabled. It is enabled by default. The default sizing mode is View sizing.

Sizing can also be enabled or disabled (and the sizing mode selected) via the menus. Refer to section 2.6.4.

Note that when the sizing mode is set to Font sizing, it can be very difficult to resize the window to use small font sizes. This is because there is a minimum size for the window title bar, and also because the main window menu may changes from a single line to multiple lines during resizing to fit within the new window size. Temporarily disabling both the title and the menu makes it much easier to select small font sizes. The title bar and menu can then be re-enabled if required.

2.5.2 Selecting Text

Each terminal window supports selecting text using the mouse. Selections can be done by:

- Character;
- Word;
- Line;
- Rectangular region.

Refer to the description of the **`MouseSelects`** option to the **`SetMouseOptions`** procedure in section 2.4.7 for a discussion of the various selection mechanisms. Note that mouse selection must be enabled if it has been disabled. It is enabled by default.

Mouse selection can also be enabled or disabled, and the sizing mode changed, via the menus. The menus also have commands for selecting/unselecting the entire virtual buffer, the current view, the current screen or the current region. Refer to section 2.6.2.

2.5.3 Moving the Cursor

Each terminal window supports moving the input (or combined) cursor with the mouse. Refer to the description of the **`MouseCursor`** option to the **`SetMouseOptions`** procedure in section 2.4.7 for a discussion of using the cursor with the mouse. Note that the mouse cursor must be explicitly enabled - it is not enabled by default.

2.5.4 Displaying the Context Menu

The context menu contains some often-used terminal window commands. It can be displayed by right-clicking the mouse in the terminal window. Note that the context menu must be enabled if it has been disabled. It is enabled by default. Refer to section 2.6.6 for a description of the context menu. Also note that if the main menu has been disabled, the context menu can be used to re-enable it.

2.6 *Terminal_Emulator Menu Support*

The Window contains a menu that can be used to select commands and specify configuration options for the terminal window. Note that the Main window menu will only appear if it is enabled. See the `SetMenuOptions` procedure described in section 2.4.7. The default setting is that the Main window menu is enabled. The entries on the main window menu are described in the following sections.

2.6.1 File Menu

The File menu appears whenever the Main menu is enabled.

`New`

This command initializes the entire virtual buffer to the current output style and colors. It will reset both the view and screen to position Column=0, Row=0 of the virtual buffer, and the input and output (or combined) cursor location to Column=0, Row=0 of the screen. The region will be reset to be the entire screen, and Region mode is disabled (refer to the description of the `Region` option to the `SetScrollOptions` procedure).

`Open`

This command resets the entire virtual buffer (like the New command), and then opens an Open File dialog box to allow a filename to be selected. The named file will be loaded into the virtual buffer.

`Save`

This command saves the entire virtual buffer to the current filename. If no filename has been selected yet, it first opens a “Save As” dialog box, allowing a filename to be selected.

`Save As`

This command opens a “Save As” dialog box allowing a filename to be selected. It then saves the entire contents of the virtual buffer to the file.

`Virtual Size`

This command allows the size of the virtual buffer to be modified. A dialog box will open showing the current size, which can be changed. The new size cannot be less than the current screen size, or greater than 32,767.

`Screen Size`

This command allows the current screen (and view) size to be modified. A dialog box will open showing the current screen size (in Columns and Rows), which can be changed. Note that if the screen size is changed using this command, it will set both the screen size and the view size.

For simplicity, when using this dialog box only screen sizes that can be accommodated on the current display are permitted. Note that the size of the view can subsequently be adjusted separately (using windows sizing with the sizing mode set to view sizing).

`Page Setup`

This command opens a “Page Setup” dialog box, allowing page settings to be used when printing to be specified.

`Print`

This command opens a “Print” dialog box, which allows the contents of the current selection, or the entire virtual buffer, to be printed. Note that the print options will default to printing only the current selection if any characters were selected when the dialog box was opened. This may or may not be what is intended – the extent of printing can be changed within the dialog box itself.

`Soft Reset`

This command performs a soft reset on the terminal emulator. A soft reset sets the character set to its default value (Windows character set for PC mode, DEC Multinational for VTxxx modes), turns all text attributes off and resets the text colors to their initial values. It also resets various terminal emulator modes: It resets the region to be full screen (but does not disable the region – use Hard Reset for this), resets the origin mode to absolute, unlocks the keyboard, sets the application keypad mode to numeric, sets the cursor keypad mode to ANSI (only affects VTxxx modes), makes the cursor visible, disables insert, disables wrap and enables key repeat.

`Hard Reset`

This command performs a hard reset on the terminal emulator. A hard reset performs all the functions of a soft reset, and in addition clears all user-defined function keys, resets the terminal to 7 bit mode, disables the use of the region, disables smooth scrolling, turns all LEDs to off, clears the screen and sets the cursor to the home position.

`Exit`

This command closes the terminal emulator window. This may or may not terminate the entire application program, depending on the setting of the `CloseProgram` option. This command may be grayed out, depending on the setting of the `CloseWindow` option. Refer to the description of `CloseProgram` and `CloseWindow` in the `SetWindowOptions` procedure.

2.6.2 Edit Menu

The Edit menu appears whenever the Main menu is enabled.

`Select All`

This command selects all characters in the virtual buffer.

`Select Screen`

This command selects all characters on the current screen. It also provides a useful means of identifying the screen if the view is not showing the current screen.

`Select Region`

This command selects all characters in the current region. It also provides a useful means of identifying the region. It does not indicate, however, whether the region is currently enabled or disabled.

`Select View`

This command selects all characters in the current view (i.e. the portion of the virtual buffer that is currently on display in the window).

`Unselect All`

This command unselects all selected characters.

`Mouse Selects`

This command enables or disables text selection using the mouse. It also indicates (by whether the command is checked or nor) whether mouse selection is currently enabled or disabled.

`Copy`

This command copies the current selection to the clipboard.

`Paste`

This command pastes the current contents of the clipboard to the terminal window. The effect of the paste will be determined by the terminal window option settings. See `SetPasteOptions` for details.

`Clear Buffer`

This command erases all characters on the current buffer to spaces using the current output (or combined) style, scrolls to the start of the buffer, and resets the cursor to the home position.

`Clear Screen`

This command erases all characters on the current screen to spaces using the current output (or combined) style.

Clear to EOL

This command erases all characters from the current output (or combined) cursor position to the end of the line (row). The characters will be set to spaces using the current output style.

2.6.3 Format Menu

The Format menu appears whenever the Main menu is enabled.

`Font`

This command opens a “Choose Font” dialog box, allowing a new font name, character set, and/or font size to be selected. It also allows new text attributes and colors to be selected. However, note that the font name, font size and character set have immediate effect, and that they apply to all characters in the virtual buffer. The font attributes (e.g. bold, underline, strikeout etc) serve only to set the current output (or combined) style for text that is subsequently output.

The dialog box is initialized with the current font name, font size, character set and output (or combined) style and foreground text color. However, note that the font dialog box may be unable to display the current foreground color (this is a limitation of the dialog box), and so the output (or combined) foreground color may need to be reset after using the dialog box.

`Fg Color`

This command opens a “Color” dialog box. The dialog box can be used to select the output (or combined) foreground color. Where possible, the current color is highlighted in the dialog box.

`Bg Color`

This command opens a “Color” dialog box. The dialog box can be used to select the output (or combined) background color. Where possible, the current color is highlighted in the dialog box.

`Set All to Colors`

This command can be used to set the foreground and background colors of all characters in the virtual buffer to the current output (or combined) foreground and background colors.

2.6.4 Options Menu

The Options menu contains some basic user-configurable options for the terminal window. Note that the Options menu will only appear if it is currently enabled. See the `SetMenuOptions` procedure described in section 2.4.7. The default setting is that the Options menu is enabled.

`Show Menu`

This command can be used to enable or disable the display of the main menu. If the main menu is currently enabled, this menu item will be checked. Of course, in practice when the main menu is disabled this menu item can be neither shown nor selected, so this menu item is only used to disable the display of the main menu. However, the same menu item appears on the context menu, and can be used to enable the display of the main menu.

`Show Title`

This command can be used to enable or disable the display of the window title (also known as the caption). If the title is currently enabled, this menu item will be checked.

`Vertical Scrollbar`

This command can be used to enable or disable the display of the vertical scrollbar. If the scrollbar is currently enabled, this menu item will be checked.

`Horizontal Scrollbar`

This command can be used to enable or disable the display of the horizontal scrollbar. If the scrollbar is currently enabled, this menu item will be checked.

`Scroll On Output`

This command can be used to enable or disable the Scroll On Output mode. If the mode is currently enabled, this menu item will be checked. See the description of `SetScrollOptions` in section 2.4.7 for more details on this mode.

Smooth Scrolling

This command can be used to enable or disable Smooth Scrolling mode. If the mode is currently enabled, this menu item will be checked. See the description of `SetScrollOptions` in section 2.4.7 for more details on this mode.

Always on Top

This command can be used to enable or disable the Always on Top mode. If the mode is currently enabled, this menu item will be checked. See the description of `SetWindowOptions` in section 2.4.7 for more details on this mode.

Cursor Visible

This command can be used to enable or disable the visibility of the cursor. If the cursor is currently visible, this menu item will be checked (note that the cursor may not be visible on the current view). See the description of `SetCursorOptions` in section 2.4.7 for more details on cursor visibility.

Cursor Flashing

This command can be used to enable or disable the flashing of the cursor. If the cursor is currently flashing, this menu item will be checked (note that the cursor may not be visible on the current view). See the description of `SetCursorOptions` in section 2.4.7 for more details on cursor flashing.

Window Sizable

This command can be used to enable or disable terminal window sizing. If window sizing is currently enabled, this menu item will be checked (and the window will have a sizing border). See the description of `SetSizingOptions` in section 2.4.7 for more details on window sizing.

Font Sizing

This command can be used to select the `Size_Fonts` sizing mode. If this sizing mode is currently selected, this menu item will be checked. If sizing is disabled, this menu item will be grayed. See the description of `SetSizingOptions` in section 2.4.7 for more details on the window sizing modes.

Screen Sizing

This command can be used to select the `Size_Screen` sizing mode. If this sizing mode is currently selected, this menu item will be checked. If sizing is disabled, this menu item will be grayed. See the description of `SetSizingOptions` in section 2.4.7 for more details on the window sizing modes.

View Sizing

This command can be used to select the `Size_View` sizing mode. If this sizing mode is currently selected, this menu item will be checked. If sizing is disabled, this menu item will be grayed. See the description of `SetSizingOptions` in section 2.4.7 for more details on the window sizing modes.

Tab Size

This command will display a dialog box that can be used to set the tab size. It will show the current default tab setting, which can then be changed. A zero will remove all tab settings. Note that (unlike using the `SetTabOptions` procedure described in section 2.4.7), only the default tab size can be set this way, and any other tab stops will be cleared if a new tab size is selected.

Advanced

The Advanced menu item on the Options menu displays a dialog box that contains advanced user-configurable options for the terminal window. Note that the Advanced menu item will only appear if it is currently enabled. See the `SetMenuOptions` procedure described in section 2.4.7. The default setting is that the Advanced menu item disabled, since most of these options will never need to be changed by the user when using a terminal window application (changing them can cause unexpected behaviour by the terminal emulator):

Advanced Option	**Meaning**
`Wrap`	See `Wrap` in `SetEditingOptions`

`Echo`	See `Echo` in `SetEditingOptions`
`Insert`	See `Insert` in `SetEditingOptions`
`Do not process CR`	See `IgnoreCR` in `SetOtherOptions`
`Do not process LF`	See `IgnoreLF` in `SetOtherOptions`
`Use LF as EOL`	See `UsefLFasEOL` in `SetOtherOptions`
`Auto LF on CR`	See `AutoLFonCR` in `SetOtherOptions`
`Auto CR on LF`	See `AutoCRonLF` in `SetOtherOptions`
`Enable Cursor Keys`	See `CursorKeys` in `SetOtherOptions`
`Combined Text Style`	See `CombinedStyle` in `SetOtherOptions`
`Combined Text Cursor`	See `CombinedCursor` in `SetOtherOptions`
`Up/Down moves View`	See `UpDownMoveView` in `SetOtherOptions`
`Page Up/Page Down moves View`	See `PageMoveMove` in `SetOtherOptions`
`Home/End moves View`	See `HomeEndMoveView` in `SetOtherOptions`
`Screen and View Locked`	See `LockScreenAndView` in `SetOtherOptions`
`Left/Right keys Wrap`	See `LeftRightWrap` in `SetOtherOptions`
`Left/Right keys Scroll`	See `LeftRightScroll` in `SetOtherOptions`
`Home/End stay in line`	See `HomeEndWithinLine` in `SetOtherOptions`
`Enable text flashing`	See `FlashingEnabled` in `SetOtherOptions`
`Allow any screen font`	This option controls the display of fonts in the dialog box displayed by the Format->Font menu item. If enabled, then all installed fonts are displayed in the dialog box, including variable pitch fonts and screen only fonts. If disabled, then only fixed pitch fonts that are both screen and printer fonts are displayed. To accurately print what is on the display, a font should be selected that is both a screen and printer font. Note that there is no equivalent API function – this is not a configuration setting for the terminal emulator – merely for the Redirect dialog box. Via the API, any font name can always be selected (see the description of the `SetFontByName` procedure in section 2.4.4).
`Mouse moves Cursor`	See `MouseCursor` in `SetMouseOptions`
`Enable System Keys`	See `SysKeysEnabled` in `SetOtherOptions`
`Return Extended Keys`	See `ExtendedKeys` in `SetKeyOptions`
`Redraw previous char`	See `RedrawPrevious` in `SetOtherOptions`
`Redraw next char`	See `RedrawNext` in `SetOtherOptions`
`Send DEL on Backspace`	See `DeleteOnBS` in `SetOtherOptions`
`Scroll only within Region`	See `Region` in `SetScrollOptions`
`Display Control Codes`	See `DisplayControls` in `SetOtherOptions`
`VT Keys on keypad`	See `VTKeys` in `SetKeyOptions`
`ANSI Mode`	See `Mode` in `SetAnsiOptions`
`Decode ANSI on Input`	See `OnInput` in `SetAnsiOptions`
`Decode ANSI on Output`	See `OnOutput` in `SetAnsiOptions`

2.6.5 Help Menu

`About`

This menu item displays a dialog box containing copyright information about the terminal emulator.

`Info`

This menu item displays a dialog box containing information about the current internal location and size of the real buffer, the virtual buffer, the screen, the region and the view. It is primarily a debugging aid.

2.6.6 Context Menu

The Context menu contains some common commands used in the terminal window. Note that the Context menu will only appear if it is currently enabled. See the **`SetMenuOptions`** procedure described in section 2.4.7. The default setting is that the Context menu is enabled.

`Copy`

Same as the Format->Copy command described in section 2.6.3

`Paste`

Same as the Format->Paste command described in section 2.6.3

`Show Menu`

Same as the Options->Show Menu command described in section 2.6.4. Note that this command is the only way for the user to enable the main menu if it has been disabled.

`Show Title`

Same as the Options->Show Title command described in section 2.6.4

`Window Sizable`

Same as the Options->Window Sizable command described in section 2.6.4

`Vertical Scrollbar`

Same as the Options->Vertical Scrollbar command described in section 2.6.4

`Horizontal Scrollbar`

Same as the Options->Horizontal Scrollbar command described in section 2.6.4

`Always on Top`

Same as the Options->Always on Top command described in section 2.6.4

`Mouse Selects`

Same as the Format->Mouse Selects command described in section 2.6.3

`Exit`

Same as the File->Exit command described in section 2.6.1

2.7 Terminal Emulator Print Support

The terminal emulator supports printing either via selecting commands from the menus, or via VT52/VT220/VT420 control sequences.

When printing using the menu commands, output is always sent to the printer immediately. See section 2.6.1 for details of the printing commands on the File menu.

When printing using the printing control sequences, some requests generate their output immediately, while others begin spooling their output but do not necessarily print it immediately (this depends in part on whether the printer has been configured to use print spooling, and also on the printer type).

All VT52 and VT220 printing control sequences are supported (and some VT420 sequences). When a printing control sequence is received, the default printer is used unless another printer has previously been selected (e.g. using the Print command on the File menu).

The VT220 control sequence to print the screen generates immediate output. The VT220 control sequence to enable the Auto print or Print Controller modes, or to print the line containing the cursor, cause their output to be spooled, but do not generate immediate output. The reason for this approach is that these control sequences cause individual characters or lines to be printed as they are received and/or displayed, and so generating output immediately would result in multiple pages being printed on most printers. The output is therefore spooled until one of the following occurs:

1. A control sequence is received that would result in immediate output (e.g. print the screen).
2. A menu command is processed that would result in immediate output (e.g. print the current selection), or modify the page setup.
3. The terminal emulator window is closed.

In any of these cases, any spooled output is printed first. To force spooled output to be printed, select either the Print or Page Setup command from the File menu, but cancel the request when the dialog box appears.

2.8 Terminal_Emulator Options

The **Terminal_Emulator** package includes a powerful and user extendable option parser that allows many terminal emulator options to be parsed from text strings. The parser is used by the **Term_IO** package to parse Form parameters, by the **Redirect** program to parse command lines, and by some of the demonstration programs.

Since there are many terminal emulator options that are common to any terminal emulator application program, the following options are "built-in" to the option parser. However, they can be overridden by user defined options if required.

The definition of each built-in option consists simply of identifying the API function or procedure (and specific parameter) that would be used to perform the equivalent configuration function via the API:

Option	Definition
[No]Top	See AlwaysOnTop in SetWindowOptions
[No]MainMenu	See MainMenu in SetMenuOptions
[No]OptionMenu	See OptionMenu in SetMenuOptions
[No]ContextMenu	See ContextMenu in SetMenuOptions
[No]AdvancedMenu	See AdvancedMenu in SetMenuOptions
[No]Title	See Visible in SetTitleOptions
SetTitle=title	See Set and Title in SetTitleOptions
[No]MouseSelect	See MouseSelects in SetMouseOptions
[No]MouseCursor	See MouseCursor in SetMouseOptions
[No]AutoCRonLF	See AutoCRonLF in SetOtherOptions
[No]AutoLFonCR	See AutoLFonCR in SetOtherOptions
[No]UseLFasEOL	See UseLFasEOL in SetOtherOptions
[No]SysKeysEnabled	See SysKeysEnabled in SetOtherOptions
[No]ScrollOnOutput	See OnOutput in SetScrollOptions
[No]HalftoneEnabled	See HalftoneEnabled in SetOtherOptions
[No]Wrap	See Wrap in SetEditingOptions
[No]Echo	See Echo in SetEditingOptions
[No]Insert	See Insert in SetEditingOptions
[No]CursorVisible	See Visible in SetCursorOptions
[No]CursorFlashing	See Flashing in SetCursorOptions
[No]DisplayControls	See DisplayControls in SetOtherOptions
[No]DeleteOnBS	See DeleteOnBS in SetOtherOptions
[No]VTKeys	See VTKeys in SetKeyOptions
TabSize=nnn	See Size in SetTabOptions
VirtualRows=nnn	See Rows in SetVirtualSize
FgColor=color	See Color in SetOutputFgColor[6]
BgColor=color	See Color in SetOutputBgColor
CursorColor=color	See Color in SetCursorColor
ScreenSize=cols,rows	See Columns,Rows in SetScreenSize

[6] When specified on the command line, valid colors are White, Black, Silver, Light_Gray, Gray, Dark_Gray, Red, Dark_Red, Green, Dark_Green, Blue, Dark_Blue, Yellow, Magenta, Cyan, Pink, Orange or the special case of Desktop, which means use the current desktop color.

ViewSize=cols,rows	See Columns,Rows in SetViewSize
Position=x,y	See XCoord,YCoord in SetWindowOptions
FontName=name	See SetName and Font in SetFontByName
CharSet=[Ansi|Oem]	See SetChar and CharSet in SetFontByName
FontSize=nnn	See SetSize and Size in SetFontByName
Ansi=[Input | Output | Both | None]	See OnOutput and/or OnInput in SetAnsiOptions
Mode=[PC | VT52 | VT100 | VT101 | VT102 | VT220 | VT320 | VT420]	See Mode in SetAnsiOptions
Sizing=[None | Screen | View | Font]	See Sizing and Mode in SetSizingOptions
Scrollbar=[Vertical | Horizontal | Both | None]	See Vertical and/or Horizontal in SetScrollOptions
Close=[Window | Program | None]	See CloseProgam and/or CloseWindow in SetWindowOptions
Redraw=[Prev | Next | Both | None]	See RedrawPrevious and RedrawNext in SetOtherOptions

Note that while the option can always be specified in full, most can be abbreviated provided the abbreviation is unambiguous. For example, abbreviating the option FgColor to Fg is valid, whereas abbreviating FontName to Font is not valid, as it would be ambiguous with FontSize.

Options cannot include spaces within the option name or between the option name and its associated parameters. For example, Ansi=both is valid, whereas Ansi = both is not valid.

Parameter values that are strings containing spaces can be quoted with either single or double quotation marks. This is not required if the parameter does not contain any spaces. For example, FontName="Lucida Console" or FontName='Lucida Console' or FontName=Courier are valid, whereas FontName=Lucida Console is not valid.

Each option in a string containing multiple options should be separated by one or more spaces from subsequent options, with no other characters between them. For example, fg=white bg=black is valid, whereas fg=white,bg=black is not valid.

Case is not significant (except within string parameters). For example, fontname is identical with FontName and FONTNAME.

It is possible to specify to the option parser that option names must be preceded by a specific lead character, such as ‘/’ (which would be a common choice when reading option strings from a command line). If not specified, a lead character is not required.

Here are some more valid examples of options strings. They assume that a lead character has not been specified:

```
Scrollbar=Both ScreenSize=80,24 SetTitle="Window Title"
SetTitle=Hello fg=green bg=dark_blue echo nowrap Mode=VT52
notitle Position=100,200 Nocursorvisible nocursorflashing
```

2.9 Terminal_Emulator Related and Child Packages

This section contains a brief overview of some of the more important source packages that make up the terminal emulator.

2.9.1 Terminal_Types

This package defines various types and constants used by the terminal emulator package that must be made visible to the user of the package. In earlier versions, these were defined in the **terminal_emulator** package itself. However, this late definition proved to make it difficult to sensibly restructure the code to improve maintainability, so these definitions were moved to a separate package.

2.9.2 Terminal_Internal_Types

This package defines various types and constants used by the terminal emulator packages, but which are not intended to be visible to the user. There should never be a need to use this package directly.

2.9.3 Real_Buffer

This package implements the fundamental (real) terminal buffer type. The buffer defined here supports cursors, selection types, real, virtual, screen, region and view coordinates, and fundamental operations on these coordinates – such as converting between them. The buffer is a tagged type, and all subsequent buffer types are derived from this type (often by simple extension).

2.9.4 Screen_Buffer

This package extends the real buffer type by adding operations to manipulate the screen buffer, such as scrolling, shifting or moving screen cells. It also adds support for the concept of double width and double height screen cells.

2.9.5 View_Buffer

This package extends the screen buffer type by adding operations to manipulate the view buffer, such as scrolling or shifting the view. It also adds the ability to display the contents of the current view in a GWindows Window. Consequently, the concept of fonts is included.

2.9.6 Scroll_Buffer

This package extends the view buffer type by adding scrolling and resizing capabilities. It also adds basic keyboard processing and selection definition capabilities.

2.9.7 Graphic_Buffer

This package extends the scroll buffer type by adding character (graphic and non-graphic) processing capabilities, selection copy and paste, file load and save, and printing capabilities. The graphic buffer actually implements a fully functional non-ANSI terminal emulator. This package could form the basis of a non-ANSI terminal emulator.

2.9.8 Ansi_Buffer

This package extends the graphic buffer type by adding the ability to interpret ANSI controls in the character processing routines, and also adds keyboard processing intended to emulate DEC VTxxx keyboards using a PC keyboard.

2.9.9 Terminal_Buffer

This package extends the ansi buffer type by adding event handling routines. Although these routines are not called directly by GWindows, there is a one-to-one mapping between these functions and the GWindows events that are required to implement the terminal emulator.

2.9.10 Terminal_Emulator

This package defines the terminal type, and translates all API calls into the appropriate calls on the terminal type, which is defined in this package as a (private) task type. The terminal type also performs all the GWindows (i.e. Windows) event handling required by the terminal window.

2.9.11 Terminal_Emulator.Terminal_Type

This package implements the actual terminal type, including all the API functions and the GWindows callbacks. Note that this is not a child package of the **terminal_emulator** package – it is merely a separate body for compilation purposes.

2.9.12 Terminal_Emulator.Line_Editor

This package implements a line editor. It is used by Redirect, and is available for use by other text handling programs that would like to implement command shell-like command line editing.

Note that the editor is a generic package that must be instantiated with procedures that return the current directory, the current path list and the current list of executable extensions. This is required to use the command completion and file completion processing. If this functionality is not required (or not appropriate), the basic line editing facilities can still be used – simply provide dummy procedures for the generic instantiation, and disable file completion.

When used by the Redirect program, the procedures that are used to instantiate the generic actually send commands to the redirected program, and interpret the returned output. This is necessary when redirecting a command shell, such as `cmd.exe` since there is no easy means of getting at what are essentially the internal variables of the redirected program. When the line editing package to implement a stand-alone (i.e. non redirected) command shell, these procedures would probably return internal values.

2.9.13 Terminal_Emulator.Option_Parser

This package implements the terminal emulator option parser. It is user extendable, and may be used by terminal emulator applications that would like to parse command lines or strings of options. It supports the following types of option:

- True Options (e.g. defining an option called `Opt` sets its result to True when `Opt` is parsed);
- False Options (e.g. defining an option called `Opt` sets its result to False when `Opt` is parsed);
- Boolean options (e.g. defining an option called `Opt` allows both `Opt` and `NoOpt` to be parsed, with the result set to True or False accordingly);
- String options (e.g. defining an option called `Opt` allows options like `Opt=string`, as well as `Opt="a string"` and `Opt='a string'` to be parsed);
- Integer options (e.g. defining an option called `Opt` allows options like `Opt=123` to be parsed);
- Dual Integer options (e.g. defining an option called `Opt` allows options like `Opt=123,456` to be parsed);
- Color options (e.g. defining an option called `Opt` allows options like `Opt=red`, to be parsed. Note that it also includes a special case of `Opt=Desktop`, which returns the current desktop background color);

String containing either single or multiple options can be parsed.

A leading character for all options can optionally be specified. If specified, that character must precede all options. For example, this might typically be '/' if the options were retrieved from a command line.

Note that just because a program uses the Terminal_Emulator API does not mean it will automatically accept terminal emulator options. The option parser must be explicitly created and then invoked from the program.

The Demo_Emulator and Snake programs provide examples of using the options parser to parse options from the command line. Snake also adds a user-defined option.

For other examples on how to extend the options parser to parse user defined options, see the source code of the Redirect program, or the Term_IO package.

2.9.14 Font_Maps

This package implements miscellaneous font-handling operations for the terminal emulator.

The terminal emulator can use any Windows font, and any Windows character set. The standard Windows character set is essentially ANSI compatible. Non-ANSI character sets can also be used, but this should only be done when the ANSI emulation mode is set to `PC` – see Appendix A). This is because when the ANSI mode is set to other values (e.g. `VT100`), the terminal emulator performs various manipulations of the characters in order to reproduce the appropriate characters for the DEC VTxxx modes. This approach was taken to make the terminal emulator capable of using a wide variety of Windows fonts. Many terminal emulators are limited to using only special fonts that already contain the necessary characters.

The DEC Multinational character set, National Replacement Character (NRC) sets, Display Controls character set (which makes control characters visible on the screen) and the Special Graphics character sets all contain characters that are either unavailable in the Windows ANSI fonts, or which are available in ANSI fonts but which correspond to different binary codes.

This package adopts two separate methods for handling these character sets:

- Character substitution. This method is used to reproduce the Multinational and NRC character sets. The package maps the characters sent to the terminal emulator into the appropriate character in the ANSI character set – this is why only ANSI Windows character sets should be used when the terminal emulator is in the VTxxx ANSI emulation modes. If non-ANSI Windows character sets are used in these modes, the substitution will result in unexpected characters being displayed.
- Bitmapped characters. This method is used for the Display Controls and Special Graphic characters. When these characters need to be displayed, they are not drawn as normal characters – the package contains bitmaps that correspond to each character, and these are stretched and rendered on the terminal window in the same size and shape as the surrounding characters. This rendering can be CPU intensive, and the result may not look as nice as the “real” characters (especially at small font sizes) but it does mean that any Windows font can be used even when the Display Controls character set or the Special Graphics character set is selected.

The package also contains support routines used for double width and double height characters. These characters are not drawn as normal characters – they are drawn by taking the bitmap of the equivalent normally sized character (which may itself be a bitmapped character) and then stretching, clipping and rendering them on the terminal window appropriately sized for the surrounding characters. Again, this rendering can be CPU intensive, and the result may not look as nice as the normal sized characters but it does mean that any character in any font can be made double width or double height.

Finally, this package also contains support for emulating non-American DEC keyboards. Such keyboards generate different binary codes in some circumstances. While support for this is implemented in the package, the only example actually implemented is for a couple of the keys on the French/Belgian keyboard. After implementing support for this feature it turned out that the DEC keyboards and the standard Windows keyboards are so different in layout and labelling that mapping individual keys is (in most cases) quite pointless. However, this feature can be used if required. Since the terminal emulator has no equivalent of the DEC “setup” mode, the terminal emulator assumes that the keyboard setup required is the same as the nationality of the last selected NRC set.

2.9.15 Filename_Completion

This package implements the filename completion capabilities used by the terminal emulator line editor.

2.9.16 Ansi_Parser

This package implements the control sequence parser. It is essentially a thick binding to the MIT parser described in section 6.6. Note that the MIT parser processes all control sequences (i.e. ANSI, DEC, ANSI.SYS and ISO 6429), not just ANSI defined control sequences.

2.9.17 MIT_Parser

This package implements the control sequence parser. It is essentially a thin binding to the MIT parser described in section 6.6.

2.9.18 Sizable_Panels

This package derives a new window type from the GWindows “drawing-panel” type. It adds some functionality to the standard GWindows drawing panel type required by the terminal emulator.

2.9.19 Win32_Support

This package implements thick bindings to some common Win32 functions not provided by GWindows.

2.9.20 Common_Dialogs

This package implements thick bindings to some standard windows dialog boxes (e.g. “file open” etc).

2.9.21 Terminal_Dialogs

This package implements thick bindings to some custom dialog boxes used by the terminal emulator.

2.9.22 Protection

This package implements simple types that can be used to achieve mutual exclusion and semaphore semantics.

3 Term_IO

3.1 Introduction

The **Term_IO** package provides a replacement for the standard Ada text-handling package Text_IO. In addition to supporting all the normal file handling capabilities of Text_IO, **Term_IO** replaces the default files (i.e. standard input, standard output and standard error) with one or more terminal windows. It also allows the creation and manipulation of user-defined terminal windows using only standard Text_IO syntax. The terminal windows can be configured to either strictly implement the semantics of Text_IO as defined in the Ada 95 Language Reference Manual (LRM), or to implement relaxed semantics that allow the user to take advantage of the terminal capabilities in a more intuitive manner.

The aim of **Term_IO** is to make as much **Terminal_Emulator** functionality as possible available to programs using only Text_IO syntax and semantics. Ideally, this would mean:

- **Term_IO** syntax should be identical with Text_IO syntax - a valid Ada program that uses Text_IO should remain a valid Ada program when all references to Text_IO are replaced with references to **Term_IO**.
- **Term_IO** semantics should be functionally identical with Text_IO semantics - the behaviour of a valid Ada program that uses Text_IO should be functionally identical to the behaviour of the program when all references to Text_IO are replaced with references to **Term_IO**. Essentially, this means that the program itself should not be able to detect any functional difference when compiled and executed using Text_IO to when it is compiled and executed using **Term_IO** (of course the *user* of the program will detect a difference). The term “functional” is used because it would be possible to write a program that can detect non-functional differences such as memory space utilized or I/O performance achieved.

Term_IO *can* achieve both of these objectives, but at the cost of not actually being able to provide very much **Terminal_Emulator** functionality. The difficult one is the semantic criteria. For example, to meet this criteria in full all ANSI control sequence processing would have to be permanently disabled (since any ANSI control sequences consumed by the terminal emulator would change the data stream). So the approach that **Term_IO** takes is that while full Text_IO syntactic equivalence is maintained, full Text_IO semantic equivalence can only be achieved by resetting various flags in the source code. There are instructions within the file `term_io.adb` on how to do this. Assuming these flags remain set, **Term_IO** still meets the semantic criteria apart from the following limitations:

- ANSI control sequence processing is enabled on output to default files. This may change the way a program behaves if subsequent input is dependent on program output that includes ANSI escape sequences.
- All default files output to a single terminal window. This may change the way a program behaves if subsequent input is dependent on whether the program outputs data to standard error or to standard output.
- The default files implement non-strict semantics, and so (by default) do any files that are created by the program but are actually implemented as terminal windows. This may change the way a program behaves if it is dependent on an exception being raised in various situations described in the LRM (e.g. when attempting to read a character from a file created as an output file). See section 3.4 for the differences between strict and non-strict semantics.

3.2 Term_IO Terminal Windows

3.2.1 The Default Terminal Window

Term_IO normally creates a single terminal window that is used by all the default files (standard input, standard output and standard error). Anything typed in this terminal window can be read by reading from standard input, and anything written to standard output or standard error appears in this terminal window. **Term_IO** can easily be changed to create separate windows for each default file if desired, but this requires a source code change – see the file `term_io.adb`.

Note that the default terminal window is invisible until a file operation is performed on one of the default files, such as `Get` or `Put`. A `Flush` can be used to make the window visible without actually performing any I/O. Once visible, the default terminal window stays visible until the application terminates.

The default file terminal window is titled “Standard Input/Output/Error” Closing this terminal window will force the application to terminate.

3.2.2 Additional Terminal Windows

An application can create additional terminal windows by calling `Create` or `Open` with appropriate `Form` options – these are identical operations for terminal windows. The `Form` options can also be used to set various terminal emulator options for this terminal window. See section 3.3 below for the syntax of the `Form` options. Each terminal window can have either Strict or non-strict LRM semantics. The default is non-strict semantics.

Additional terminal windows can be created either as `In_File` or `Out_File` mode files. Refer to section 3.4 for a description of the semantics associated with each mode. Note that when using strict semantics, an input terminal window can only be read from (not written to) and an output terminal window can only be written to (not read from). When using non-strict semantics, a terminal window can be read from or written to whether it is an input or an output terminal window – but the file mode changes the semantics associated with the terminal window.

Additional terminal windows are always visible as soon as they are created - unlike the default terminal window.

By default, additional terminal windows are titled according to the name specified in the `Create` or `Open` call.

By default, additional terminal windows can be closed without forcing the application to terminate.

3.3 Term_IO Form Options

To specify that a terminal window is to be opened (as opposed to a normal file) in a manner compatible with `Text_IO`, the `Form` parameter is used. The `Form` parameter can be specified in either an `Open` or a `Create` operation.

To open a terminal window instead of a file, the first 7 characters of the `Form` parameter must be exactly the characters `TERM_IO`. Anything other than this means the entire request (including the `Form` parameter) is treated as a normal file operation. After the `TERM_IO` (separated by spaces) additional terminal window options can be included in the form string. Terminal windows respond to all normal `Text_IO` operations, including a request to retrieve the current form options.

Since `Term_IO` uses the options parser provided as part of the Terminal_Emulator package, all the standard options described in section 2.8 are also available to `Term_IO`. `Term_IO` defines one additional option:

Option	**Definition**
[`No`]`Strict`	Specifies whether or not the terminal window should enforce strict Ada 95 LRM semantics. The default is to use non-strict (relaxed) semantics, which are more intuitive to use than the strict interpretation of the LRM. See section 3.4.

Examples of valid `Form` strings are:

```
TERM_IO
TERM_IO Wrap Strict Fg=Yellow Bg=Red ScreenSize=50,20
TERM_IO noStrict insert fontname=”Lucida Console”
```

3.4 Term_IO Semantics

When using strict LRM semantics, input terminal windows (i.e. additional terminal windows opened with file mode `In_Mode`) can be used only for input (e.g. `Get`), and output terminal windows (i.e. additional terminal windows opened with file mode `Out_Mode`) can be used only for output (e.g. `Put`).

When using relaxed semantics, both input terminal windows and output terminal windows can be used for either input (e.g. `Get`) or output (e.g. `Put`). However, the semantics applied are slightly different in each case. The following table describes the differences between strict and non-strict semantics, for input and output modes:

Operation	Mode	Strict (LRM)	NoStrict (relaxed)
`End_Of_File`	`In_File`	STATUS_ERROR exception if terminal window has been closed, otherwise False.	Returns True if the terminal window has been closed, otherwise False.
	`Out_File`	STATUS_ERROR exception if terminal window has been closed, otherwise MODE_ERROR exception.	Same as In_File
`End_Of_Line`	`In_File`	Returns True if next unread character is line mark (ASCII.LF)	Same as Strict
	`Out_File`	MODE_ERROR exception.	Return True if the current column position is equal to the current line length, otherwise False.
`End_Of_Page`	`In_File`	Returns True if next unread characters are line mark (ASCII.LF) and page mark (ASCII.FF)	Return False.
	`Out_File`	MODE_ERROR exception.	Return True if the current column position is equal to the current line length and the current line position is equal to the current page length, otherwise False.
`Flush`	`In_File`	MODE_ERROR exception.	Terminal Window made visible (standard I/O windows are normally invisible until they receive and input or output request).
	`Out_File`	Terminal Window made visible (standard I/O windows are normally invisible until they receive and input or output request).	As above.
`Get (character)`	`In_File`	Get next character from keyboard buffer. Do not return line mark (ASCII.LF) or page mark (ASCII.FF) – get another character.	Same as Strict
	`Out_File`	MODE_ERROR exception	Get next character from keyboard buffer. Return any character, including line mark (ASCII.LF) and page mark (ASCII.FF).
`Get (string)`	`In_File`	Get (length of string) characters from keyboard buffer, ignoring line mark (ASCII.LF) and page mark (ASCII.FF).	Get (length of string) characters from keyboard buffer, including line mark (ASCII.LF) and page mark (ASCII.FF).
	`Out_File`	MODE_ERROR exception	Same as In_File
`Get_Immediate (Available parameter not specified)`	`In_File`	Get next character from keyboard, including line mark (ASCII.LF) and page mark (ASCII.FF).	Same as Strict
	`Out_File`	MODE_ERROR exception	Same as In_File

Get_Immediate (Available parameter specified)	**In_File**	If character available in key buffer, get next character from keyboard, including line mark (ASCII.LF) and page mark (ASCII.FF). Otherwise return ASCII.NUL.	Same as Strict
	Out_File	MODE_ERROR exception	Same as In_File
Get_Line (string)	**In_File**	Get up to (length of string) characters from keyboard buffer, stopping when line mark (ASCII.LF) encountered.	Same as Strict
	Out_File	MODE_ERROR exception	Same as In_File
Line_Length	**In_File**	MODE_ERROR exception	Return number of screen columns.
	Out_File	Return number of screen columns.	Same as In_File
Page_Length	**In_File**	MODE_ERROR exception	Return number of screen rows.
	Out_File	Return number of screen rows.	Same as In_File
Look_Ahead	**In_File**	Read the next character and Return End_Of_Line as True if it is line mark or page mark, otherwise False. Do not return line marks and page marks.	Same as Strict
	Out_File	MODE_ERROR exception	Set End_Of_Line as True if current column position is equal to the current line length, or read the next character and set End_Of_Line if current column position is equal to the current line length, or it is line mark or page mark. Otherwise End_Of_Line is False. Returns lines marks and page marks.
New_Line	**In_File**	MODE_ERROR exception	Write specified number of line marks (ASCII.LF).
	Out_File	Write specified number of line marks (ASCII.LF). If page length exceeded, write page mark (ASCII.FF).	Same as In_File
New_Page	**In_File**	MODE_ERROR exception	Write page mark (ASCII.FF).
	Out_File	Write page mark (ASCII.FF).	Same as In_File
Put	**In_File**	MODE_ERROR exception	Write the character, string or item.
	Out_File	Write the character, string or item.	Same as In_File
Put_Line	**In_File**	MODE_ERROR exception	Write string, followed by line marks (ASCII.LF).
	Out_File	Write string, followed by line marks (ASCII.LF). If page length exceeded, write page mark (ASCII.FF).	Same as In_File

`Set_Col`	`In_File`	Read characters from the terminal until the current column matches the specified column.	Set the output cursor to the specified column in the current row (line).
	`Out_File`	Write a line mark (ASCII.LF) if the current column is greater than the specified column, then write spaces until the column matches the specified column.	Same as In_File
`Set_Line`	`In_File`	Read characters from the terminal until the current line matches the specified line.	Set the output cursor to the current column in the specified row (line).
	`Out_File`	Write a page mark (ASCII.FF) if the current line is greater than the specified line, then write line marks (ASCII.LF) until the current line matches the specified line.	Same as In_File
`Set_Line_Length`	`In_File`	MODE_ERROR exception	Set line length and screen width to specified number of columns.
	`Out_File`	Set line length and screen width to specified number of columns.	Same as In_File
`Set_Page_Length`	`In_File`	MODE_ERROR exception	Set page length and screen height to specified number of rows.
	`Out_File`	Set page length and screen height to specified number of rows.	Same as In_File
`Skip_Line`	`In_File`	Read characters until line mark (ASCII.LF) encountered.	Same as Strict
	`Out_File`	MODE_ERROR exception	Write a line mark (ASCII.LF).
`Skip_Page`	`In_File`	Read characters until page mark (ASCII.FF) encountered.	Same as Strict
	`Out_File`	MODE_ERROR exception	Write a page mark (ASCII.FF).

When using strict semantics, all terminal window semantics should be as defined in the Ada 95 LRM.

When using non-strict semantics, input and output terminal windows are more flexible, and are in some ways quite similar. The difference between input and output terminals can be summarized as:

- How `Skip_line` and `Skip_page` are handled. These differences are retained so that an existing program expecting to read from a real file will correctly wait for appropriate input to be typed into the terminal window;
- How `Get` handles line marks and page marks. For input terminals, line mark (ASCII.LF) and page mark (ASCII.FF) are not usually returned. For output terminals, `Get` returns both line marks (ASCII.LF) and page marks (ASCII.FF);
- How `End_Of_Line` and `End_Of_Page` are handled. For input terminals, these operations must read the next character to see if the appropriate line mark and/or page mark characters are next. For output terminals, these operations check if the current column and/or line positions indicate end of line or end of page. Note that this is only possible if `Set_Line_Length` and/or `Set_Page_Length` have been called to specify the size of the screen – otherwise the terminal window is regarded as an "unbounded" file.
- How `Look_Ahead` is handled. For input terminals, this operation must read the next character to see if the appropriate line mark and/or page mark characters are next in order to set the `End_Of_Line` flag. However, these characters are not returned - ASCII.NUL is returned instead. For output terminals, this operation sets the `End_Of_Line` flag if the current column position is at the end of line (and returns an ASCII.NUL). If not, this operation also reads the next character to see if the appropriate line mark and/or page mark characters are next, in

order to set the `End_Of_Line` flag. However, for output terminals these characters are actually returned.

Note that when using non-strict semantics, using ANSI control sequences can cause problems with the normal Text_IO style line and column handling (which is also implemented by **Term_IO)**. This is because **Term_IO** does not inherently understand such ANSI sequences – data that looks like normal displayable text output will causes the package to update its internal line and column positions as normal. However, the output may in fact be an ANSI control sequence that is either not output at all, or that changes the screen row or column in unpredictable ways. The only solution to this is to explicitly reset the line and column whenever the program needs to depend on Text_IO style line and column handling. They will remain valid up until the next ANSI control sequence is parsed.

3.5 Term_IO Related and Child Packages

3.5.1 Term_IO.Integer_Text_IO

The **Term_IO** analogue of Ada.Text_IO.Integer_Text_IO, which is required by the LRM.

3.5.2 Term_IO.Float_Text_IO

The **Term_IO** analogue of Ada.Text_IO.Float_Text_IO, which is required by the LRM.

3.5.3 Gnat_IO

The **Term_IO** analogue of Gnat.IO, which is provided with the GNAT Ada compiler.

3.6 Term_IO Demonstration Program

There is a demonstration program for **Term_IO** (`demo_term_io`) in the demos directory of the distribution. See section B.6 for more details.

3.7 Converting existing Ada programs to use Term_IO

Preparing an existing Ada program to use **Term_IO** instead of Text_IO is simple, but requires editing the original program source files (or perhaps many source files). The following changes need to be made:

- Replace references to Text_IO (Ada.Text_IO) with **Term_IO**
- Replace references to Integer_Text_IO (Ada.Integer_Text_IO) with **Term_IO.Integer_Text_IO**
- Replace references to Float_Text_IO (Ada.Text_IO.Float_Text_IO) with **Term_IO.Float_Text_IO**
- Replace references to Gnat.IO with **Gnat_IO**.
- (Optional) add a line `pragma Linker_Options (”-mwindows”);`[7]
- Recompile.

Note that there may be many such references in the source files. However, provided all the references in the “with” clauses are replaced, other references should be detected by the compiler and can then be replaced relatively easily.

Note that it is possible that an Ada program contains a program unit (or even a variable or type) called Text_IO that is actually nothing to do with Ada.Text_IO. Of course, any such references should not be replaced.

The sources for the example programs provided with GNAT, with appropriate modifications made to use **Term_IO,** are included in the `termio\examples` subdirectory of the distribution.

Note that all the GNAT example programs are included in the distribution, even those that do not use Text_IO (e.g. the `use_of_import` example). Such examples cannot be modified to use **Term_IO**, but they can instead be run using the **Redirect** program (e.g. `redirect use_of_import`) to achieve a similar result.

[7] When using **Term_IO** instead of Text_IO, the program can be made a Windows mode program since it no longer needs to use the console I/O subsystem.

4 Redirect

4.1 Introduction

The **Redirect** program can be used to execute another program, redirecting that programs standard input, output and error to a terminal window. The program whose I/O is redirected can be a windows mode or console mode application, provided it is "well behaved"[8]. Not all applications are well behaved. One notable example is the Windows 95/98 command shell `command.com`. This program does not fully support redirection of its I/O. By contrast, the Windows NT/2000/XP command shell `cmd.exe` *is* well behaved.

When redirected to a terminal window, the program's input and output can be edited, viewed, scrolled, selected, copied, pasted, printed and/or saved to a file.

Redirect has a wide range of configuration options, which can be set from within the program or on the command line. For example, try:

```
redirect /virtualrows=5000 /fg=yellow /bg=dark_green cmd.exe /Y
```

Note that some programs and commands use **LF** as a line terminator, not **CRLF,** which can cause problems when such commands are redirected – typically, instead of appearing like this:

```
output line 1
output line 2
output line 3
```

The output may appear like this:

```
output line 1

             output line 2

                          output line 3
```

In this case, add the `/cookout` option to the **Redirect** command (this option is described in the next section). For example:

```
redirect /virtualrows=5000 /fg=yellow /bg=dark_green /cookout cmd.exe /Y
```

The redirected program has full access to the terminal capabilities provided by the Terminal_Emulator program, and can use all supported ANSI control sequences. Therefore, **Redirect** also provides an environment for executing many applications that were designed to work under the ANSI.SYS device driver that existed under Windows 95/98, but which is not provided under Windows NT/2000/XP.

In addition, **Redirect** provides facilities to perform printing, selection, cut and paste, mouse support, command line editing, filename completion and command history. These are closer in style to the equivalent facilities provided by Xterm and Unix shell programs than those provided by the Windows command shell, and furthermore are available for use by user written applications - even very simple ones. See the `minimal` example provided for a bare bones program that makes full use of these facilities to provide a very simple but fully functional command shell.

Redirect can also be used as a filter. If no command is specified for redirection, the program simply accepts data from standard input, echoes it in a terminal window, and puts the data to standard output. This can be useful for capturing large volumes of output in a terminal window with full scroll/select/cut/paste/load/save capabilities.

For example, under Windows NT/2000/XP, when using `cmd.exe` in a redirected window (i.e. started using the command `redirect cmd.exe`) you can execute commands like:

```
dir | redirect | sort | redirect
```

Under Windows 95/98, you can do the same, provided you are using `win95cmd.exe` in a redirected window (i.e. started using the command `redirect win95cmd.exe`) and have installed the

[8] To Redirect, a "well behaved" program is one that use its standard input, standard output and standard error files for I/O, and does ***not*** use console I/O. Note that using console I/O is not equivalent to being a console mode application – most console mode applications do not use console I/O at all, and some windows applications ***do*** use console I/O. Console I/O is a hangover from the days of DOS, probably intended only for support of legacy DOS applications. Fortunately, the use of console I/O is becoming increasingly rare.

appropriate registry fix (see section 4.3 for more details on `win95cmd.exe`). If you are using the default Win95/98 command interpreter `command.com` then the above command will not work, but simpler commands will work, such as:

```
dir | redirect
```

4.2 Redirect Command Line Options

Since **Redirect** uses the options parser provided as part of Terminal_Emulator, all the standard Terminal_Emulator options described in section 2.8 (with the exceptions noted below) are available as command line options.

Note that when specified on the command line, each option must be preceded by a '/' character to distinguish it from any command to be executed. There must be no space between the '/' and the option, but each option must be separated by at least one space character.

Note that (depending on how the command is executed and the command shell used), string parameters may need to be quoted with multiple quotation marks (e.g. `"""a string"""`). This may only occur for double quotes – single quotes may still work (e.g. `'a string'`).

An example of a Redirect command with command line options:

```
redirect /fontname='Courier New' /virtualrows=20000 cmd.exe /Y
```

In this example the command to be executed is `cmd.exe /Y`. The `/Y` is not interpreted as a command line option to **Redirect** itself because processing of options always stops as soon as anything is encountered that is not a valid command line option (in this case, the string `cmd.exe`). The remainder of the command line is assumed to be part of the command to be executed.

The new or modified options added by **Redirect** (note that the leading '/' is not shown, but must be included) are as follows:

Option	Definition
`[No]Insert`	Disable or enable the Terminal_Emulator.Line_Editor Insert mode. Note that this option redefines existing Insert option, which applies to the Terminal_Emulator itself – this can no longer be specified on the command line. Within the line editor, pressing the Ins key also toggles the line editor insert mode. Note the line editor must be enabled to use insert mode. The default value is enabled.
`[No]Edit`	Disable or enable the use of the Terminal_Emulator.Line_Editor package. When the line editor is disabled, keystrokes typed in the terminal window will not be buffered and cannot be edited – they will be sent to the redirected process keystroke by keystroke. The default value is enabled.
`[No]Completion`	Disables or enables filename and command completion. Note that the line editor must be enabled to perform filename or command completion. The default value is enabled.
`CompleteChar=`nnn	Specifies the character the line editor will interpret as a request for filename or command completion. The character must be specified as a decimal value. The default value is to use the TAB key (i.e. decimal value 9). The line editor must be enabled to use filename or command completion.
`EraseChar=`nnn	Specifies the character the line editor will interpret as a request to erase the whole current line. The character must be specified as a decimal value. The default value is to use the ESC key (i.e. decimal value 27). The line editor must be enabled to use the erase function.
`[No]Console`	Enable or disable the creation of a hidden console for the redirected process to inherit. Many redirected processes, especially console mode processes, require a console. If they do not inherit one from Redirect they will create one themselves. Creating the console can be disabled where it is known that the redirected process is a windows application. The default is enabled. Note that when the console is created, it is unfortunately always created as visible. Redirect then hides it, but on some systems the console may appear momentarily before it can be hidden.
`NamedPipes`	Forces the use of named pipes instead of anonymous pipes. The default is to use anonymous pipes. Note that named pipes cannot be used on Windows 95/98.

AnonPipes	Forces the use of anonymous pipes instead of named pipes. The default is to use anonymous pipes.
History=nnn	Specifies the number of commands that the line editor should save in the history. History is disabled altogether if the number is zero. The default history is 10 commands. History commands are recalled with the up and down arrow keys. The line editor must be enabled to use command history.
High	Force the program to set its Windows priority to High. The default is to set the priority to Normal. Use this option if the redirected program seems slow, or the terminal window itself is sluggish to respond to window messages (e.g. scrolling with the scrollbar).
CookOut	Forces redirect to simulate some aspects of Unix "tty" I/O when processing output from the redirected process. Specifically, any LFs output from the captured process are translated into CRLFs before being sent to the terminal window.
CookIn	Forces redirect to simulate some aspects of Unix "tty" I/O when preparing input to be sent to the redirected process. Specifically – for edited input the lines are terminated with LF instead of the usual CRLF, and for unedited input any CRs are translated to LFs before being sent to the process.
Unicode	Force Redirect to treat all output from the redirected program as Unicode. Redirect tries to detect Unicode output automatically. However, there are some circumstances (such as when the output arrives as individual characters) when Redirect cannot tell if the output was supposed to be Unicode or not. Note that Unicode cannot be used on Windows 95/98.
XmitUnicode	Force Redirect to send all input to the redirected program as Unicode. Note that Unicode cannot be used on Windows 95/98.
DirCommand=command	Specifies the command that the line editor will use to interrogate the redirected program for the current directory during command and filename completion. Default is **"cd"**, assuming the redirected program is cmd.exe or compatible. The command is only used if both the line editor and command history are enabled.
ExtnCommand=command	Specifies the command that the line editor will use to interrogate the redirected program for the current list of program extensions during command and filename completion. Default is **"echo %pathext%"**, assuming the redirected program is cmd.exe or compatible. The command is only used if both the line editor and command history are enabled. Note that on Windows 95/98 a value for pathext may need to be manually set, since it may not be set automatically.
PathCommand=command	Specifies the command that the line editor will use to interrogate the redirected program for the current list of paths to search for executable programs during command and filename completion. Default is **"echo %path%"**, assuming the redirected program is cmd.exe or compatible. The command is only used if both the line editor and command history are enabled.
SetTitle=title	Redirect does not redefine the SetTitle option, but it always sets the terminal window title after processing any command line options, so a SetTitle on the command line will essentially be ignored.

4.3 Specific Redirect issues with Windows 95/98

By default, **Redirect** is a windows mode application that opens a console when it needs one. However, this has implications where redirection is applied to **Redirect** itself and it is either the first or the last in a sequence of redirected commands. An example is the command:

```
dir | redirect | sort | redirect
```

The first issue is that the command will not execute at all under the Windows 95/98 command interpreter **command.com** unless **Redirect** is linked as a console mode application. However, it *can* be executed as a windows mode application under **`win95cmd.exe`**[9], which is a WinNT/2000/XP style command interpreter that runs under Win95/98.

The second issue is that (if it executes at all) the command logically should always produce output to the command shell in which it was executed. But it may not - the second instance of **Redirect** does not always get given a valid file handle for standard output. This appears to be a Windows bug - the command NEVER works as expected when using a normal command shell either in Win95/98 or WinNT/2000/XP (e.g. using either **`win95cmd.exe`** or **`cmd.exe`**) but under Win95/98 it SOMETIMES works as expected when using a redirected command shell (e.g. **`redirect win95cmd`**), and under WinNT/2000/XP it ALWAYS works as expected when using a redirected command shell (e.g. **`redirect cmd.exe`**).

One way of solving all such issues is that **Redirect** can be turned into a console mode application by a simple edit of the source code – see the file **`redirect.adb`**. Unfortunately, this also forces the program to open a console in cases where it is neither necessary nor desirable for it to do so (e.g. if started using the Windows "Start -> Run..." command. Hence the default is to make the program a windows mode application, and accept the minor (and somewhat obscure) limitations.

[9] To locate a copy of **`win95cmd.exe`**, use an Internet search engine such as http://www.google.com and search for “win95cmd”. Win95cmd works reasonably well as an alternative to **command.com**, but note that there is also a registry setting that has to be applied to make it work properly – the registry patch is included in the “redirect” directory as **Win95cmd.reg**. Simply double click on this file to apply the registry setting. Also, note that Win95cmd should not be used to build or install the terminal emulator – for more detail, see Appendix C.

5 Comms

5.1 *Introduction*

The **Comms** program can be used to open a terminal window that uses a serial communications port as input and output. This allows the PC to be used as a serial terminal. The **Comms** program has full access to the terminal capabilities provided by the Terminal_Emulator program, and can use all supported ANSI control sequences. Therefore, **Comms** provides an alternative to a stand-alone terminal.

Multiple instances of **Comms** can be run simultaneously, provided each one uses a different serial communications port.

5.2 *Comms Command Line Options*

Since **Comms** uses the options parser provided as part of Terminal_Emulator, all the standard Terminal_Emulator options described in section 2.8 are available as command line options.

Note that when specified on the command line, each option must be preceded by a '/' character to distinguish it from any command to be executed. There must be no space between the '/' and the option, but each option must be separated by at least one space character.

Note that (depending on how the command is executed and the command shell used), string parameters may need to be quoted with multiple quotation marks (e.g. `"""a string"""`). This may only occur for double quotes – single quotes may still work (e.g. `'a string'`).

The new options added by **Comms** (note that the leading '/' is not shown, but must be included) are as follows:

Option	Definition
`[No]Rts`	Disable or enable RTS handshaking. Note that `Rts` is normally used in conjunction with `Cts`. The default value is disabled.
`[No]Dtr`	Disable or enable DTR handshaking. Note that `Dtr` is normally used in conjunction with `Dsr`. The default value is disabled.
`[No]Cts`	Disable or enable CTS flow control. Note that `Cts` is normally used in conjunction with `Rts`. The default value is disabled.
`[No]Dsr`	Disable or enable DSR flow control. Note that `Dsr` is normally used in conjunction with `Dtr`. The default value is disabled.
`[No]Xin`	Disable or Enable XON/XOFF processing on receive (i.e. send XOFF when receive buffer fills up, and send XON when it empties again). Note that `Xin` is normally used in conjunction with `Xout`. The default value is disabled.
`[No]Xout`	Disable or Enable XON/XOFF processing on transmit (i.e. stop transmission when XOFF received, resume when XON received). Note that `Xout` is normally used in conjunction with `Xin`. The default value is disabled.
`Com=`n	Use serial communications port COMn. COM ports 1 to 99 are supported.
`BaudRate=[` `300` | `600` | `1200` | `2400` | `4800` | `9600` | `14400` | `19200` | `38400` | `56000` |	Specify the baud rate. Note that not all baud rates will be supported on all hardware. The default baud rate is 9600.

57600 | 115200 | 128000 | 230400 | 256000 | 480800 | 921600 | 1000000 | 1500000 | 2000000 | 3000000]	
StopBits=[**1**|**2**]	Specify the number of stop bits. The default is 1 stop bit.
DataBits=[**7**|**8**]	Specify the number of data bits. Note that if 7 data bits are selected, the terminal emulator will not recognize 8 bit ANSI control sequences. The default is 8 data bits.
Parity=[**none** | **even** | **odd** | **mark** | **space**]	Specify the parity. The default parity is none.
High	Force the program to set its Windows priority to High. The default is to set the priority to Normal.
PasteSize=nnn	The maximum number of characters that can be stored in the paste buffer. This represents the maximum number of characters that can be pasted in a single paste operation. The default is 2048. Increase if you need to paste large amounts of text.
PasteGroup=nnn	The number of characters sent as a group as a result of a paste operation – can be as low as 1. The default is 15. Reduce if the recipient program has a small keyboard buffer.
PasteDelay=nnn	The delay in milliseconds to wait after the last character echoed as a result of sending each group of pasted characters – this is intended to reduce the speed of a paste to something approximating typing speeds, to accommodate recipient programs (such as text editors) that may be slow to process incoming pasted characters, especially when there is no flow control on the serial interface. The default is 250. Increase for low baud rates.
[**No**]**LockScreen**	Disable or enable the Terminal_Emulator LockScreenAndView option. Note that this can also be enabled or disabled on the Options->Advanced menu. The default value is disabled.

For example:

```
comms /virtualrows=5000 /fg=yellow /bg=green /com=1 /baudrate=19200
comms /com=2 /cts /rts /stopbits=2 /databits=7 /parity=even
comms /com=3 /dtr /dsr /baudrate=230400 /pastesize=5000
comms /xin /xout /baud=300 /pastegroup=1 /pastedelay=500 /lock
```

The Comms program enables two new entries on the **File** menu:

DTR

Selecting this entry toggles the state of DTR. The current state of DTR is indicated next to the menu entry.

Pulse DTR

Selecting this entry pulses DTR either off or on for 100ms, depending on its current state.

5.3 Comms Ymodem supported

The **Comms** program adds YModem and Ymodem-1k support. A new **YModem** menu entry is added, with the following entries:

Send

This opens a dialog box allowing a file to be specified for sending using the original YModem protocol, which sends data in packets of 128 bytes.

Note that the receiver must be started before this command is used.

Send 1k

This opens a dialog box allowing a file to be specified for sending using the YModem-1k protocol, which sends data in packets of 1024 bytes.

Note that the receiver must be started before this command is used.

Receive

This opens a dialog box allowing a file to be received using either the original YModem or the YModem-1k protocol. If no filename is specified, the file name used by the sender will be used. The file will be saved in the current directory.

Note that the sender must be started before this command is used.

6 Telnet

6.1 Introduction

The **Telnet** program can be used to open a terminal window that uses the telnet protocol as input and output. This allows the PC to be used as a network virtual terminal. The **Telnet** program has full access to the terminal capabilities provided by the Terminal_Emulator program, and can use all supported ANSI control sequences.

Multiple instances of **Telnet** can be run simultaneously, either as telnet clients or as telnet servers.

6.2 Telnet Command Line Options

Since **Telnet** uses the options parser provided as part of Terminal_Emulator, all the standard Terminal_Emulator options described in section 2.8 are available as command line options.

Note that when specified on the command line, each option must be preceded by a '/' character to distinguish it from any command to be executed. There must be no space between the '/' and the option, but each option must be separated by at least one space character.

Note that (depending on how the command is executed and the command shell used), string parameters may need to be quoted with multiple quotation marks (e.g. `"""a string"""`). This may only occur for double quotes – single quotes may still work (e.g. `'a string'`).

The new options added by **Telnet** (note that the leading '/' is not shown, but must be included) are as follows:

Option	Definition
`Client`	Specify that this telnet instance is to act as a telnet client. The client will attempt to make an active connection to the specified telnet host and port. This is the default.
`Server`	Specify that this telnet instance is to act as a telnet server. The server will listen on the telnet port and accept connections from telnet clients. Only one simultaneous connection at a time is supported, although the program is easily modified to support multiple connections if required. Note that the sever merely displays anything received from the telnet client in the terminal emulator window, and sends anything entered in the terminal emulator window to the currently connected client. The default is that the telnet instance is a client, not a server.
`Port=`nnn	Specify the telnet port. Specified in decimal. Defaults to 23.
`Host=`[host_name | ip_address]	Specifies the name or IP address of the host telnet server for telnet clients. Can be specified as a name, or as an IP address in "dot" notation. For example: `gigantor` or `100.2.3.4`. Defaults to `localhost`. Also note that any text left on the command line after all options have been parsed is interpreted as the host. This means that if the host is the last thing specified on the command line, the `/host=` is not required. Note that on some systems, `localhost` does not resolve properly, so you have to explicitly specify the hostname. You can find the hostname in the Control Panel System settings (it may be called Device name), or by using the Windows `hostname` command in a command line window.
`Terminal=`name	Specify the type of terminal. This does not affect the terminal emulator mode, which must be specified separately using the `Mode` option. It only affects the name reported in response to the telnet "send terminal type" request. Defaults to `DEC-VT100`, although for many UNIX telnet servers a more suitable terminal type is probably `ANSI`.
`Escape=`nnn	Specify the escape character that can be used to allow telnet commands to be entered in the terminal window. A value of zero disables the escape character, which means that telnet options can only be specified on the command line. To send the escape character as data, there is a special telnet command that can be used. Refer to the section on telnet commands for more details. The escape character is specified in decimal.

	Defaults to `^]` (i.e. decimal 29 or CTRL+] or ASCII.GS).
`Debug=[ None | Data | Options | Controls | All ]`	Specify the output of debugging information. `Data` means print each sent and received data byte. `Options` means produce messages in the terminal window during option negotiation about what options are in effect or not. `Controls` implies `Options` and also produces messages in the terminal window of the actual telnet control controls received. `All` implies `data` and `controls` and also generates reams and reams of output on standard output about the state of the telnet processor. The default is `None`.
`Binary=[ None | Input | Output | Both ]`	Specify whether to negotiate (offer or demand) a binary connection on the input connection, the output connection, or both connections. The default is not to use a binary connection. Note that this is a recommendation only – if the other end of the connection demands a binary connection, the telnet program will accommodate the demand.
`/Mode=[ Line | Char ]`	Specify whether to enter character-at-a-time or line-at-a-time mode. Note that unlike the telnet command `mode char`, negotiation of the mode is not performed – the command-line option option simply sets the mode in the telnet client. It is intended to be used with servers that are not real telnet servers, but simply echo all characters, including any telnet commands. This can make it difficult for the client to enter the mode successfully using normal telnet negotiation.
`High`	Force the program to set its Windows priority to High. The default is to set the priority to Normal.
`SetTitle=`title	Telnet does not redefine the SetTitle option, but it always sets the terminal window title after processing any command line options, so a SetTitle on the command line will essentially be ignored.

For example:

```
telnet gigantor
telnet /terminal=ANSI gigantor
telnet /client /host=100.2.2.2 /port=24 /fg=yellow /bg=green
telnet /server /binary=both
telnet /server /escape=0 /debug=options
telnet /client /host=myhost /mode=vt420 /terminal='DEC-VT420'
```

6.3 Telnet Commands

6.3.1 Overview

All telnet commands are entered by first entering the escape character (by default `^]`) and then typing the command in response to the `Telnet>` prompt. Note that when `telnet` is acting as a server, it will not recognize the escape character until a connection is established.

All commands (and the telnet command mode itself) are terminated by a carriage return. To enter another telnet command, the escape character must be entered again.

A command consists of one or more command words, separated from each other by at least once space. Only the first character in each word of the command is significant.

To send the escape character as data, the command `SEND ESCAPE` can be used.

Commands may be in upper or lower case.

If an incorrect command is entered, `telnet` will respond with `Unrecognized telnet command`.

Valid Command Format Examples:

`SEND ARE YOU THERE <CR>`

`S A Y T <CR>` (equivalent to the above command)

`suppress goahead local <CR>`

`s g l <CR>` (short form of above)

Invalid Command Format Examples:

`  SEND ABORT OUTPUT <CR>` (spaces before first word of first command)

`sendbreak` (no space between command words)

Note that in most circumstances, without a full understanding of the implications of the various telnet command sequences and options, the only commands that should be used routinely are `OPEN` and `CLOSE`.

Many of the commands have to do with setting or resetting telnet options. If it is important to verify the success or failure of such commands, the /`debug=options` or /`debug=controls` command line parameter should be specified.

6.3.2 Command Descriptions

6.3.2.1 HELP or ?

The HELP command simply lists all the known commands. The command `?` is a synonym for the HELP command.

6.3.2.2 OPEN

The OPEN command allows the user to establish a connection to a remote TELNET. The parameter after the OPEN command specifies the network address of the remote TELNET's host computer. OPEN accepts parameters, as follows:

```
OPEN [ host_name | ip_address ]
```

`host_name`	: The remote host name (e.g. `gigantor.forefront.com.au`).
`ip_addr`	: The remote host IP address in "dot" notation (e.g. `100.1.2.3`).

If no parameter is specified, the current default is used.

6.3.2.3 CLOSE

The CLOSE command closes a previously opened TELNET connection. The response message to a successful close command is "connection closed".

6.3.2.4 RESET

The RESET command ensures the transport level connection is closed and resets all buffers and state information to start-up/default status. It does not affect the current setting of the port, terminal or escape character.

6.3.2.5 RESET OPTIONS

The RESET OPTIONS command attempts to de-negotiate all local and remote options currently in effect.

6.3.2.6 NEGOTIATE OPTIONS

The NEGOTIATE OPTIONS command attempts to negotiate all desirable local and remote options. Note that this happens automatically when a telnet connection is started, and it is not usually necessary to manually initiate negotiations.

6.3.2.7 ECHO or ECHO LOCAL

The ECHO LOCAL command will request the local TELNET to echo data characters received from the remote TELNET back to that remote TELNET. Note: TELNET will not allow remote echoing on both sides of the connection at once as this would imply an endless loop of retransmissions. If one side of the TELNET connection is performing remote echoing and both sides have agreed to suppress sending the TELNET GO-AHEAD, then TELNET will go into character-at-a-time instead of the default line-at-a-time transmission mode.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.8 ECHO REMOTE

The ECHO REMOTE command will request the remote TELNET to echo data characters received from the local TELNET back to the local TELNET. Note: TELNET will not allow remote echoing on both sides of the connection at once as this would imply an endless loop of retransmissions. If one side of the TELNET connection is performing remote echoing and both sides have agreed to suppress sending the TELNET GOAHEAD, then TELNET will go into character-at-a-time instead of the default line-at-a-time transmission mode.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.9 QUIT ECHO or QUIT ECHO LOCAL

This command will stop the local TELNET from echoing characters to the remote TELNET.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.10 QUIT ECHO REMOTE

This command will stop the remote TELNET from echoing characters to the local TELNET.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.11 SUPPRESS GOAHEAD or SUPRESS GOAHEAD LOCAL

The default I/O mode of TELNET is half duplex. This option can be used to increase efficiency on a full duplex terminal or I/O device by putting the local TELNET in full duplex mode. This option should not be used on a half duplex "lockable" keyboard terminal. If one side of the TELNET connection is performing remote echoing and both sides have agreed to suppress sending the TELNET GOAHEAD, then TELNET will go into character-at-a-time instead of the default line-at-a-time transmission mode.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.12 SUPPRESS GOAHEAD REMOTE

The default I/O mode of TELNET is half duplex. This option can be used to increase efficiency on a remote full duplex terminal or I/O device by putting the remote TELNET in full duplex mode. This option should not be used on a remote half duplex "lockable" keyboard terminal. If one side of the TELNET connection is performing remote echoing and both sides have agreed to suppress sending the TELNET GO-AHEAD, then TELNET will go into character-at-a-time instead of the default line-at-a-time-transmission mode.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.13 QUIT SUPPRESS GOAHEAD or QUIT SUPPRESS GOAHEAD LOCAL

This command will stop the local TELNET from suppressing the transmission of the TELNET GOAHEAD signal.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.14 QUIT SUPPRESS GOAHEAD REMOTE

This command will stop the remote TELNET from suppressing the transmission of the TELNET GOAHEAD signal.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.15 BINARY or BINARY LOCAL

This command causes TELNET to begin binary transmission (if supported by the remote TELNET) on the output connection. One of the significant consequences of binary transmission is that no end-on-line conversion is ever performed.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.16 BINARY REMOTE

This command causes TELNET to request that the remote TELNET to begin binary transmission (if supported by the remote TELNET) on the input connection. One of the significant consequences of binary transmission is that no end-on-line conversion is ever performed.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.17 QUIT BINARY or QUIT BINARY LOCAL

This command causes TELNET to cease binary transmission on the output connection.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.18 QUIT BINARY REMOTE

This command causes TELNET to request that the remote TELNET cease binary transmission on the input connection.

If there is a TELNET connection currently established, any required option negotiations are initiated automatically. If not, the state of the option will be noted as “desirable” and negotiated next time a connection is established.

6.3.2.19 TRANSLATE CR or TRANSLATE CR OUTPUT

This command affects the TELNET end-of-line convention on the output connection. It requests that TELNET Translate CRs on the output connection into the pair CR/LF (other than those CRs that are already part of either a CR/LF or CR/NULL pair).

The opposite (see QUIT TRANSLATE CR OUTPUT) is to translate CRs (other than those CRs that are already either a CR/LF or CR/NULL pair) into CR/NULL pairs.

6.3.2.20 QUIT TRANSLATE CR or QUIT TRANSLATE CR OUTPUT

This command affects the TELNET end-of-line convention on the output connection. It requests that TELNET Translate CRs on the output connection into the pair CR/NULL (other than those CRs that are already part of either a CR/LF or CR/NULL pair).

The opposite (see TRANSLATE CR OUTPUT) is to translate CRs (other than those CRs that are already either a CR/LF or CR/NULL pair) into CR/LF pairs.

6.3.2.21 TRANSLATE CR INPUT

This command affects the TELNET end-of-line convention on the input connection. It requests that TELNET Translate CR/NULL pairs on the input connection into the pair CR/LF.

The opposite (see QUIT TRANSLATE CR INPUT) is to translate CRs/NULL pairs into a single CR.

6.3.2.22 QUIT TRANSLATE CR INPUT

This command affects the TELNET end-of-line convention on the input connection. It requests that TELNET Translate CR/NULL pairs on the input connection into a single CR.

The opposite (see TRANSLATE CR INPUT) is to translate CRs/NULL pairs into a CR/LF pair.

6.3.2.23 SEND ABORT OUTPUT

This command allows the current process to (appear to) run to completion while not sending the output to the user. Only TELNET commands that can be processed in a timely manner will continue to be processed.

6.3.2.24 SEND ARE YOU THERE

This command causes the remote TELNET to return a message indicating the remote TELNET is operational.

6.3.2.25 SEND BREAK

This code provides a signal outside the USASCII set which is currently given local meaning within many systems. It indicates the "break" or "attention" key was hit. It is not a synonym for the Interrupt Process (IP) standard representation.

6.3.2.26 SEND ERASE CHARACTER

This command instructs the remote user process to delete the last undeleted character from the stream of data being supplied by the user.

6.3.2.27 SEND ERASE LINE

This command instructs the remote user process to delete the last undeleted line from the stream of data being supplied by the user.

6.3.2.28 SEND INTERRUPT PROCESS

Many systems provide a function that suspends, interrupts, aborts, or terminates the operation of the user process. The IP is the standard representation for this function. This is likely to be utilized by users who have an unwanted or runaway process.

6.3.2.29 SEND SYNCH

This command will essentially eliminate all data already received by the remote but not processed.

6.3.2.30 SEND END RECORD

This command causes TELNET to transmit an End-of-Record control. Incoming End-of-Records are always ignored by TELNET.

6.3.2.31 SEND STATUS REQUEST

This command causes TELNET to transmit a request for the remote TELNET to send its current status. The received status message is not used, but will be displayed if the `/debug=controls` option is in effect.

6.3.2.32 SEND STATUS UPDATE

This command causes TELNET to transmit its current status to the remote TELNET.

6.3.2.33 MODE CHAR

This option can be used to tell both the local and the remote TELNET to suppress the TELNET GOAHEAD and enable remote TELNET ECHO. When one side of the TELNET connection is performing remote echoing and both sides have agreed to suppress sending the TELNET GO-AHEAD, then TELNET will go into character-at-a-time mode.

Note that this is different from the `/mode=char` command line option, which does not use option negotiation with the remote TELNET, but instead causes the local TELNET to behave as if character-at-a-time mode had been successfully negotiated. The command-line version is intended to be used with servers that are not full TELNET servers, but simply echo their input character by character. In such cases, performing a TELNET negotiation would not succeed.

6.3.2.34 MODE LINE

This option can be used to tell the remote TELNET to enable both the local and the remote TELNET GOAHEAD and suppress remote TELNET ECHO. This causes TELNET to go into line-at-a-time mode.

6.4 Telnet Negotiable Options

The **Telnet** program supports only a bare minimum of negotiable options. These are:

- Binary
- Echo
- Suppress Go Ahead
- Status
- Timing Mark
- Terminal Type
- End of Record

This is adequate to allow **Telnet** to be used as a simple VT100 compatible network virtual terminal in nearly all circumstances. More negotiable options may be added in future.

6.5 Using Telnet to emulate a PC Client

By default, **Telnet** emulates a VT100 compatible network virtual terminal (i.e. a VT100 client). When using **Telnet** to connect to a server that expects a PC client (e.g. many Telnet BBSs) **Telnet** can be configured to emulate a PC compatible network virtual terminal instead.

This is especially the case when the server expects to be able to use so-called “ANSI Graphics”. To render these graphics correctly, the terminal emulator ANSI mode may need to be set to `PC`, and the Windows Terminal font should be specified – which means the OEM character set must also be specified. The terminal type may also need to be set to something other than `DEC-VT100` (the default value). For example, it may need to be set to `ANSI` or even `ibmpc`. An appropriate command to do all this would be similar to the following:

```
   telnet /mode=PC /fontname=terminal /charset=oem /terminal='ANSI' <host>
```

A batch file specifying these options (`pc_telnet.bat`) is provided. A symptom that PC emulation is required instead of VT100 emulation is that ANSI graphics are drawn with blanks on some lines, or with consecutive lines mis-aligned. This is because PCs and VT100s differ in the interpretation of some ANSI control sequences. Conversely, a symptom that VT100 emulation is required instead of

PC emulation is that cursor keys are not interpreted correctly. Unfortunately, there are a few instances of telnet servers that expect the keyboard behaviour of a VT100, but the screen behaviour of a PC. This cannot be accommodated with this version of **Telnet**, so it may be necessary to decide between accurate screen drawing or correct keyboard behaviour in such cases.

Appendix A. ANSI emulation

A.1 Introduction

The terminal emulator implements various ANSI and DEC control sequences, as well as other control sequences defined for the DOS device driver ANSI.SYS, and in ISO 6429.

The interpretation of many control sequences, and also some other behaviour of the terminal emulator, depends on the mode the ANSI parser is configured to run in. The modes, and their effects, are defined in the sections below.

Note that the following sections do not contain a description of the effect of the control sequences supported in each mode. Refer to the file **`ANSI_Parser.adb`** for a description of all control sequences support by the ANSI parser. The ones currently supported by the terminal emulator (all VT52/VT100/VT102 control sequences, a large subset of VT220 and VT420 control sequences, and quite a few ISO 6429 control sequences) are marked with an asterisk ('*'). The following sections only identify the differences between the various modes.

Note that the current implementation of the control sequences is "permissive". This means (for example) that even if the ANSI mode is currently set to **`VT52`**, a valid VT420 escape sequence will still be processed unless it conflicts with a VT52 control sequence. However, even though this is currently the case, the correct ANSI mode should always be selected for the terminal emulator, or the results may be unexpected. Subsequent versions of this terminal emulator may make the parsing of ANSI sequences "restrictive" rather than "permissive".

Note that although the terminal emulator supports many command sequences defined for ISO6429 terminals, there is no specific mode setting for this emulation mode. **`VT420`** mode should be used for general emulation except when specific PC, VT52 or VT100/VT101/VT102/VT220/VT320 emulation is required.

A.2 All modes

In all modes, the terminal emulator returns ASCII.NUL when CTRL+SPACE is pressed, and (by default) an ASCII.BS when the BACKSPACE key is pressed, and an ASCII.DEL when CTRL+BACKSPACE is pressed. The option **`DeleteOnBS`** option can be used to force the BACKSPACE key to return an ASCII.DEL.

In all modes, the character attribute concealed (or "invisible") is shown using strikeout. This was an arbitrary decision - concealed text is an ISO 6429 attribute, and is not normally supported by DEC terminals.

A.3 PC mode

`PC` mode is intended to be suitable for most use, except when specific VTxxx emulation is required. Unless otherwise stated below, this mode is similar to the **`VT100`** mode.

When this mode is set, any Windows font and any character set can be used - the terminal emulator does not remap characters, so non-ANSI Windows character sets will be displayed correctly in this mode.

ANSI.SYS control sequences are processed, with the following exceptions:

- Set Mode – text mode requests are implemented, but graphics modes requests are ignored.
- Set Keyboard Strings – not implemented.

Setting **`PC`** mode also has the following effect on the operation of the terminal emulator:

- The form feed character (ASCII.FF) is interpreted as a form feed (in the VTxxx modes, form feed is interpreted as a line feed – i.e. the same as ASCII.LF).
- **`ESC [ 2 J`** causes the cursor to move to the home position. In the VTxxx modes, the cursor position is left unchanged.

- `ESC [ s` control sequences are interpreted as Save Cursor (for ANSI.SYS compatibility). In the VTxxx modes, `ESC [ s` control sequences are interpreted as Set Left and Right Margin.
- Extended (special) keys can be enabled, and read using `GetExtended`. In the VTxxxx modes, extended keys return specific sequences defined by ANSI or DEC, or nothing.
- The '+' key on the numeric keypad returns '+'. In the VTxxx modes, this key is used to simulate the application keypad ',' key.

A.4 All VTxxx modes

In all VTxxx modes, the DEC Multinational character set is used by default. Because of the font mapping performed by the terminal emulator, only ANSI Windows character sets should be used while in any VTxxx mode.

The following are common to all the VTxxx modes (i.e. `VT52`, `VT100`, `VT101`, `VT102`, `VT220`, `VT320` and `VT420`):

- The form feed character (ASCII.FF) is interpreted as a line feed (i.e. the same as ASCII.LF).
- `ESC [ 2 J` leaves the cursor position unchanged. In PC mode, the cursor is moved to the home position.
- `ESC [ s` control sequences are interpreted as Set Left and Right Margin.
- Extended (special) keys return the specific sequences defined by ANSI or DEC, or nothing at all.
- `ESC [ 3 h` selects the Display Controls character set (to GL and GR), and disables all ANSI processing on input and output. This is equivalent to the `DisplayControls` option to the `SetOtherOptions` procedure, or the `/DisplayControls` command line option. Note that ANSI processing must be re-enabled manually. The Display Controls character set displays all control codes as viewable characters, and is very useful in debugging programs that generate control sequences.
- Support is included for both 7 and 8 bit control sequences.
- The DEC editing keypad is implemented as follows (but note that some keys may be redefined by specific terminal emulator options such as the `HomeEndMovesView` or `PageView` options to the `SetOtherOptions` procedure, or the `CursorKeys` option to the `SetKeyOptions` procedure):
  - **Find** is the Home key;
  - **Select** is the End key;
  - **Insert Here** is the Insert key;
  - **Remove** is the Delete key;
  - **Prev Screen** is the Page Up key;
  - **Next Screen** is the Page Down key.
- The DEC application keypad is implemented using the numeric keypad. Note that Numlock has to be on to enable the use of the numeric keypad. Note that the '+' key on the numeric keypad is used as the application keypad ',' key. If VT Key emulation mode is enabled, then the top row of the numeric keypad can be used to send PF1 to PF4, and SHIFT+'-' can be used to send the '-' application keypad key (Windows NT/2000/XP only). If VT Key emulation mode is disabled, the function keys F1 to F4 must be used to send PF1 to PF4, and the top row of the numeric keypad acts as normal for a PC.
- The PC keys F6 to F12 can be used as DEC keys F6 to F12. To simulate DEC keys F13 to F20, use CTRL+F5 to CTRL+F12 (e.g.PC key CTRL+F12 simulates DEC key F20). These keys work in all VTxxx modes, even though a VT100 did not have any function keys.
- The PC keys F6 to F12 can be used as DEC user definable keys (note that the user defined values are only sent when SHIFT is also held down, as on a DEC keyboard). Since the DEC keys F13 through to F20 are implemented as CTRL+F5 through to CTRL+F12, to send the DEC user definable key defined for F13 requires using CTRL+SHIFT+F5 on the PC keyboard.

- When the LEDs on the DEC keyboard are turned on or off by using control sequences, the terminal emulator indicates their states by appending **`L1`**, **`L2`**, **`L3`**, and/or **`L4`** to the window title. Obviously the title bar has to be enabled for this to have effect.

A.5 VT52 mode

The `VT52` mode should be used for VT52 emulation. The following notes apply to the `VT52` mode:

- **`ESC F`** and **`ESC G`** control sequences are interpreted as requests to select the Multinational character set, or the Special Graphics character set (respectively).
- The terminal responds to an identify request (e.g. **`ESC Z`**) with **`ESC / Z.`**
- **`ESC V`** is interpreted as Print Line containing Cursor.
- Cursor (arrow) keys send VT52 sequences (unless otherwise disabled, such as by the **`CursorKeys`** option to **`SetKeyOptions`**).
- Numeric keypad keys send VT52 sequences. Note that Numlock has to be on to enable the Numeric keypad.
- F11 sends ASCII.ESC, F12 sends ASCII.BS, F13 (CTRL+F5) sends ASCII.LF.
- If VT Keys mode is enabled, the top row of the numeric keyboard are used to simulate PF1 to PF4 and send VT52 sequences, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and sent VT52 sequences, and the top row of the numeric keyboard behaves as normal.

A.6 VT100 mode

The `VT100` mode should be used for VT100 emulation. The following notes apply to the `VT100` mode:

- **`ESC V`** is interpreted as Start of Guarded Area.
- Cursor (arrow) keys send ANSI sequences (unless otherwise disabled, such as by the **`CursorKeys`** option to **`SetKeyOptions`**).
- Numeric keypad keys send ANSI sequences. Note that Numlock has to be on to enable the Numeric keypad.
- Function keys F1 to F4 (PF1 to PF4) send ANSI sequences.
- F11 sends ASCII.ESC, F12 sends ASCII.BS, F13 (CTRL+F5) sends ASCII.LF.
- **`ESC ( 1`** selects Multinational character set to G0.
- **`ESC ( 2`** selects Special Graphics character set to G0.
- **`ESC ) 1`** selects Multinational character set to G1.
- **`ESC ) 2`** selects Special Graphics character set to G1.
- **`ESC [ c`** (Primary Device Attributes Request) causes the emulator to send **`ESC [ ? 1; 2 c`** (indicating a VT100 with AVO option).
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal.

A.7 VT101 mode

The `VT101` mode should be used for VT101 emulation. It is identical to the `VT100` mode, except for the following:

- **`ESC [ c`** (Primary Device Attributes Request) causes the emulator to send **`ESC [ ? 1; 0 c`** (indicating a VT101).
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal.

A.8 VT102 mode

The `VT102` mode should be used for VT102 emulation. It is identical to the `VT100` mode, except for the following:

- `ESC [ c` (Primary Device Attributes Request) causes the emulator to send `ESC [ ? 6 c` (indicating a VT102).
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal.

A.9 VT220 mode

The `VT220` mode should be used for VT220 emulation. The following notes apply to the `VT220` mode:

- `ESC V` is interpreted as Start of Guarded Area.
- Cursor (arrow) keys send ANSI sequences (unless otherwise disabled, such as by the `CursorKeys` option to `SetKeyOptions`).
- Numeric keypad keys send ANSI sequences. Note that Numlock has to be on to enable the Numeric keypad.
- Function keys F1 to F4 (PF1 to PF4) and send ANSI sequences.
- F11, F12, F13 send VT420 control sequences.
- `ESC [ c` (Primary Device Attributes Request) causes the emulator to send `ESC [ ? 62; 1; 2; 6; 8; 9 c` (indicating a VT220).
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal.

A.10 VT320 mode

The `VT320` mode should be used for VT320 emulation. It is currently identical to the `VT220` mode, except for the following:

- `ESC [ c` (Primary Device Attributes Request) causes the emulator to send `ESC [ ? 63; 1; 2; 6; 8; 9 c` (indicating a VT320).
- Function keys F1 to F4 send nothing.
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F4 send nothing. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal.

A.11 VT420 mode

The `VT420` mode should be used for VT420 emulation. It is currently identical to the `VT220` mode, except for the following:

- `ESC [ c` (Primary Device Attributes Request) causes the emulator to send `ESC [ ? 64; 1; 2; 6; 8; 9 c` (indicating a VT420).
- If VT Keys mode is enabled, the top row of the numeric keyboard is used to simulate PF1 to PF4, and function keys F1 to F5 can be used to send the sequences defined for the VT4xx terminal F1 to F5 keys. If VT Keys mode is disabled, function keys F1 to F4 are used to simulate PF1 to PF4 and the top row of the numeric keyboard behaves as normal. The sequences defined for the F1 to F4 keys of a VT4xx cannot be sent.
- The PC keys F1 to F5 can be used as DEC keys F1 to F5. This is in addition to the F6 to F12 keys, which work as DEC keys F6 to F20 in all VTxxx modes. See section A.4.

- The PC keys F1 to F5 can be used as DEC user definable keys F1 to F5 (note that the user defined values are only sent when SHIFT is also held down, as on a DEC keyboard). This is in addition to the F6 to F12 keys, which work as DEC user definable keys in all VTxxx modes. See section A.4.

Appendix B. Demonstration Programs

B.1 *Hello_World*

This program is the traditional “Hello World” program written for the Ada terminal emulator. It is described in section 2.4.2.

B.2 *Demo_Emulator*

This program provides a very simple terminal emulator. It demonstrates reading and writing to the terminal, and setting text attributes and fonts.

Note that this program also demonstrates the terminal emulator options parser. The program accepts and parses all the built-in terminal emulator command line options defined in section 2.8. Note, however that when options are parsed on the command line from a command shell (such as `cmd.exe`), quoting string options may have to be done using single quotes (‘) or else three successive double quotes. Also, the options must be preceded by a ‘/’ character. For example:

```
demo_emulator /settitle='my title'
demo_emulator /settitle= """my title"""
```

The only non-intuitive part of the demonstration is the Screen_Dump procedure. To demonstrate this, place the cursor anywhere on the screen, and type CTRL+S. The data from the cursor to the end of the screen will be sent to the program (dumped), the screen will be cleared, and the data will be sent back to the terminal.

B.3 *Demo_Cursors*

This program provides an example of the potential advantages of using separate input and output cursors and styles.

B.4 *Demo_Regions*

This program provides an example of the effects of using scrolling regions. It also demonstrates soft scrolling

B.5 *Demo_Multiple*

This program provides an example of the use of multiple terminals (within multiple tasks) from a single Ada program.

One of the terminal windows provides a demonstration of the rendering of double width and double height characters, scrolling, and also italic and bold fonts. While this window is open, try selecting another font (using the Format menu) to see how they look.

Other terminal windows can be used to assess the speed of the terminal emulator when using single character I/O, when using string I/O, and when simultaneously moving the cursor around the screen. Note that on a slow machine, the response time of the entire application may be slow – each of these windows is driven by a different task inside the application, and each task just writes to their respective terminal windows as fast as they possibly can.

B.6 *Demo_Term_IO*

This program provides an example of an Ada program that uses the **Term_IO** package instead of Text_IO. It demonstrates user created input and output terminal windows, as well as the default file terminal window.

When running the demo, please pay close attention to the instructions given in the various windows. When there are multiple windows on display, make sure you enter input into the correct window.

B.7 Snake

This program is a simple (but addictive) game of “snake”. It demonstrates the use of extended or special keys, dynamically resizing the screen and view, parsing command line options and adding user-definable options to the terminal emulator.

Note that the program expects to use the Windows “Wingdings” font to draw the snake. If this font is not installed, the snake may appear as a jumble of L, J, K and I characters. Use the command line option `/NoWingdings` to force the program to use a normal font – but the snake will not look as happy when it eats an egg.

B.8 Minimal

This program is not actually a terminal emulator program – it is a very simple text-oriented command interpreter that uses only standard I/O. It reads lines from standard input and executes each line as a command using an operating system function call. It also implements a minimal number of built-in commands using operating system function calls. It is intentionally an almost completely trivial program. It is intended to provide a demonstration of how the **Redirect** program can be used in conjunction with a “dumb” text-based program to take advantage of the terminal emulator functionality. When executed in conjunction with **Redirect** (i.e. `redirect minimal`), the program can be used just as if were a Windows application, with command line editing, command and filename completion, command history, mouse support and printing support thrown in – but without actually having to do write any code.

Appendix C. Installing and Compiling the Programs

Assuming that the correct versions of the GNAT Ada compiler and GWindows are already installed (see section 1.5 for the versions required), compiling and installing the terminal emulator is easy.

Decide where the terminal emulator is to be installed. The GNAT directory is recommended (probably `C:\GNAT`), but it can be anywhere. Unzip the distribution (make sure the “use folder names” or equivalent option is selected when unzipping). The unzipped structure will be as follows:

`terminal`	Main directory (install batch files and makefile).
`terminal\doc`	Subdirectory for documentation
`terminal\doc\html`	Subdirectory for HTML documentation
`terminal\emulator`	Subdirectory for the source of the Terminal_Emulator packages.
`terminal\emulator\mit_parser`	Subdirectory for the sources of the C based MIT Parser.
`terminal\term_io`	Subdirectory for the Term_IO source packages.
`terminal\term_io\examples`	Subdirectory for the GNAT examples as modified to use Term_IO
`terminal\redirect`	Subdirectory for the source of the Redirect program.
`terminal\comms`	Subdirectory for the source of the Comms program.
`terminal\telnet`	Subdirectory for the source of the Telnet program.
`terminal\demos`	Subdirectory for source of the demonstration programs.
`terminal\bin`	Subdirectory for all compiled programs.
`terminal\vttest`	Subdirectory for the sources of the C based VTtest program.

```
In the terminal subdirectory, enter the command install.
```

However, note that if there is a version of “sh” (e.g. /bin/sh.exe or similar) anywhere in the current path, this can cause problems. This may be the case (for example) if a package like GtkAda has been installed. To resolve this, temporarily edit the path environment variable to remove references to the directory containing “sh”, or simply rename that directory. This only needs to be done during the installation. Once installed, “sh” can be restored.

Also, note that if the terminal emulator is compiled and built under Win95cmd, then the MIT parser builds incorrectly (note that this is a problem related to using pipes under Win95cmd - it is nothing to do with the MIT parser itself). A symptom of this problem is that the ***.tbl** files in the **emulator\mit_parser** subdirectory have zero length. As a result, the entire terminal emulator may appear to build and install correctly – it even runs – but it will ***not*** process any ANSI control sequences !. Although Win95cmd is fine as a day-to-day command interpreter, the terminal emulator itself should only be built under **command.com** or **cmd.exe**.

The `install` just invokes the command `make` (installed with GNAT) with specific options. The `make` command can be used directly instead. Here are the options that can be specified:

`OPTIMISE=1`	Use the GNAT Optimiser.
`DEBUG=1`	Generate additional debugging information - do not use with RELEASE.[10]
`RELEASE=1`	Strip symbols (saves space) - do not use with DEBUG.

PROFILE=1	Compile with profiling enabled (for use with the GNU Profiler) - do not use with RELEASE.

The recommended way to install the terminal emulator is make RELEASE=1. This is all the install batch file does. Adding the OPTIMISE=1 option can help reduce the size of terminal emulator executables. However, note that compiling the package Terminal_Emulator.Font_Maps with optimisation specified could take a v-e-r-y long time, even though it is not an especially complex package. It always compiles correctly, but seems to elicit some pathological behaviour from the GNAT compiler.

Installing the terminal emulator registers two new standard packages (using the GNAT command **gnatreg**): EMULATOR and TERM_IO. This means that terminal emulator programs can be compiled without having to explicitly specify the location of the **Terminal_Emulator** or **Term_IO** source or binary files.

The **make** command compiles the MIT parser, the **Terminal_Emulator**, **Term_IO**, **Redirect**, **Comms, Telnet** and the demonstration programs. For convenience, the compiled versions of the **Redirect, Comms, Telnet** and the demonstration programs are copied into the **bin** subdirectory. It may be useful to add the **bin** subdirectory to your PATH environment variable.

The **make** command does ***not*** compile the GNAT examples - there is a separate makefile for this. Go to the **examples** subdirectory and **make** these programs manually.

The **make** command also does ***not*** compile the VTtest program - since *VTtest must be compiled under Cygwin*. However, a compiled version of VTtest is already included in the distribution. Unfortunately, a copy of the Cygwin DLL **cygwin1.dll** (also required) cannot be included for licensing reasons - but it is readily available (see section 1.5.5 for more details). To compile VTtest under Cygwin, first execute **./configure** in the vttest directory, then **make**.

After installing, remember to add the “bin” subdirectory to your PATH environment variable (or copy the contents to another directory that is already on your current PATH). Note that the subdirectory should be BEFORE the normal Windows subdirectories to avoid executing the Windows telnet program instead of the Ada telnet program. Alternatively, you could rename the Ada telnet program – e.g. to “atelnet.exe”.

[10] When using the debug option (i.e. DEBUG=1) under Windows 98, the terminal emulator sometimes behaves erratically. For example, it crashes on certain key combinations, such as F4 followed by Keypad_Minus. This behaviour does not occur on other platforms, or on Windows 98 when the program is compiled without the debug option. Therefore, do not use the DEBUG option unless it is specifically required for debugging.

Appendix D. Notes

6.6 *MIT Parser*

The software commonly referred to throughout this document as the “MIT Parser” is properly known as the “PARSER FOR VT420 TERMINAL EMULATORS”. It is the only part of the terminal emulator not written in Ada. It was originally intended that this emulator be implemented completely in Ada, but the MIT Parser turned out to be such an astonishingly comprehensive piece of software that it seems a shame not to use it, and totally unnecessary to duplicate it.

The only changes required to the MIT Parser are very minor - mostly the addition of some control sequence definitions for ANSI.SYS or ISO 6429 support. The modified source code is included.

6.7 *VTtest*

The VTtest program is a very comprehensive test program for VTxxx terminals, but it was written to run under Unix, not Windows.

The easiest way to compile and run VTtest under Windows is to use Cygwin. If Cygwin is installed, VTtest can be compiled and executed using the terminal emulator telnet program. Simply log into a Cygwin shell and execute the Cygwin telnet daemon in debug mode (e.g. by executing the Cygwin command **`/usr/sbin/in.telnetd -debug`**). Then, from within a DOS command shell, use the terminal emulator telnet program to log in to the local host (e.g. **`telnet`** or **`telnet localhost`**). Once logged in to a Cygwin shell from telnet, simply execute **`vttest.exe`** from within the telnet session. Depending on the version of Cygwin you have installed, it may be necessary to use the “-x” parameter to VTtest (i.e. **`vttest.exe -x`**). See below for more details on the “-x” command line parameter. If VTtest can be run this way, the remainder of this section can be ignored.

If Cygwin is not installed, VTtest can still be run under Windows provided that the Cygwin DLL itself (**`cygwin1.dll`**) is present somewhere in the current path. VTtest cannot be compiled, but this is not necessary since an executable version is included in the distribution. However, there are a couple of significant provisos:

- VTtest was not intended to run with its I/O redirected through pipes.
- VTtest makes assumptions about the default UNIX tty mode, and does not work correctly if the tty is not configured as it expects. It also changes the tty mode dynamically during execution to do things like turning echo and raw/cooked mode on or off. Some tests fail if this doesn’t work (e.g. when the I/O is redirected through pipes).

As an alternative to rewriting VTtest to resolve both these issues, there are a couple of minimal source changes that can be used to at least get VTtest to function with its I/O redirected through pipes. The changes consist mainly of a couple of extra “flush” operations, and some rework of some instances where the program redefines the standard error file handle to do some unusual I/O. These changes are only enabled when a new VTtest command line option is specified (‘-x’). This means that VTtest can still be used exactly as its maker intended if necessary – although some things won’t work. The modified source code is included.

With these changes made, VTtest can be run under Windows with its I/O redirected. Specifically, this means it can be run under **Redirect**. This provides a handy way of testing the terminal emulator, and also demonstrating some of the capabilities of the terminal emulator. To compensate for the lack of tty support, various **Redirect** options can be used. The following options are recommended for most purposes:

```
   redirect /adv /noedit /uself /echo /cookout /nosyskeys /mode=vt100 /screen=80,24 vttest -x
```

A batch file specifying these options **(`vt_test.bat`)** is provided. Using these options, there are a couple of tests that may appear to fail. Specifically, this occurs wherever the success of the test depends on VTtest being able to change the tty mode. Manually changing the terminal window options or running Redirect with other options makes these tests succeed, but may make other things fail. For example, to get the test of Line Feed/New Line mode to work (VTtest test number 6.2), the **`UseLFasEOL`** option can be temporarily turned off just before the test and turned on again afterwards. Alternatively, **Redirect** can be run without this option set at all (i.e. remove the command line option **`/uself`**) but this causes other problems because VTtest itself expects LFs and not CRs in most circumstances when it says “press RETURN”. Such translation gets done under UNIX by the tty, but when using **Redirect** without `/uself`, CTRL+J has to be used to explicitly generate an LF instead of using ENTER (which normally generates a CR). Note that sometimes VTtest actually wants to see a CR, such as in test number 6.2.

Note that it is particularly important when running VTtest to ensure that the correct ANSI mode and screen size is selected for the terminal emulator. Otherwise some of the tests (particularly the keyboard tests) will fail. Refer to Appendix A.

6.8 Windows 95/98

Under Windows 95/98, when using the standard command interpreter `command.com`, the environment variable `pathext` is not created automatically. So before starting **Redirect**, execute the following command (or something similar):

```
set pathext=.bat;.exe;.com
```

Without this, **Redirect** filename/command completion may not work correctly.

The standard Windows 95/98 command interpreter `command.com` cannot itself be redirected (i.e. run using `redirect command.com`). Obtain a copy of `win95cmd.exe` instead (see section 4.3 for more details).

On Windows 95/98, underlining of text may not be rendered correctly. This appears to be when Windows simulates an underlined font by drawing a line underneath a non-underlined font. In this case Windows sometimes draws the underline outside the bounding size of the non-underlined character - and this means it can get overwritten when the character below the underlined character is drawn. It is difficult to detect precisely when this will occur, and it may occur with only some sizes of the same font. If the text underline is not visible, or seems to appear and disappear (e.g. when the cursor is moved over the character, forcing it to be redrawn) then try a different font size or a different font. The problem does not seem to occur on Windows NT/2000/XP.

6.9 Windows NT/2000/XP

When redirecting `cmd.exe`, the command extensions introduced in Windows NT may need to be disabled (i.e. use the command `redirect cmd /Y` rather than `redirect cmd`).

If command extensions are enabled, Windows mode applications (e.g. `notepad.exe`) may not display their window until ENTER is pressed several times in the terminal emulator window. It does not affect console mode applications (such as `mem.exe`).

Never use `command.com` with **Redirect**. Always use `cmd.exe`. The `command.com` shell does not work properly with redirected I/O.

6.10 Still To Do

This section contains a (partial) list of things left to do on the terminal emulator. They may or may not be made available in subsequent versions:

- User configurable menus.
- Smooth scrolling for horizontal scroll.
- Implement an Edit->Find menu option (with regular expressions support).
- Implement partial command matching in the command history.
- Implement an option for saving command history between sessions.
- Implement more control sequences (e.g. DEC user definable characters, ANSI.SYS style key redefinition).
- Improve performance when rendering bitmapped and double size characters.
- Improve the choose font dialog box so it doesn’t override the current foreground color.
- Remove the restriction on 32,767 rows imposed by the select virtual buffer size dialog box.
- Improve Unicode support.
- Improve font rendering of underlined fonts.

- Implement Print support for double height, double width and bitmapped characters (currently printed as normal size non-bitmapped characters).