NAME

twander – File Browser

OVERVIEW

Wander around a filesystem executing commands of your choice on selected files and directories. The general idea here is that `twander` provides GUI facilities for navigating around your filesystem, but **you** define the commands you want available via "Command Definitions" (in the Configuration File). In other words, `twander` can't do anything useful until you've defined some commands. This document describes how to install and start `twander` as well as it's various startup options.

This document also describes the format and content of a `twander` Configuration File. You will find an example Configuration File called `.twander` in the distribution tarball. All the entries in that file are commented out, so you'll need to uncomment and edit the ones you want to work with.

If you're new to `twander` and want to know why this program is better and different than whatever you're using at the moment, take a moment to read the section called **DESIGN PHILOSOPHY** toward the end of this document first.

Similarly, if this is the first time you've worked with `twander`, there is a section near the end of this document entitled **INSTALLING** `twander` which describes how the program should be installed.

You can get the latest version of the program and documentation from the `twander` homepage:

```
http://www.tundraware.com/Software/twander/
```

You should check this site periodically for program updates, bug fixes, and enhancements.

Also, you are strongly encouraged to join the `twander` mailing list where you'll find help and answers to questions you have about this program. Details of how to do this can be found toward the end of this document in the section entitled, **GETTING HELP: THE** `twander` **MAILING LIST.**

SYNOPSIS

twander [-cdhqrtv] [startdir]

OPTIONS

startdir

Directory in which to begin. (default: directory in which program was started)

If this directory does not exist or cannot be opened, `twander` will display an error message and abort.

-c path/name of Configuration File

Specify the location and name of the configuration file. (default is ~/.twander)

If this file does not exist or cannot be opened, `twander` will display a warning to that effect but continue to run. This is reasonable behavior because `twander` provides a command to reload the Configuration File without exiting the program (which you would presumably do after fixing the Configuration File problem).

-d debuglevel

Start in debug mode dumping the items specified in the debuglevel. (default: debuglevel=0/debug off)

`twander` is able to selectively dump debugging information to stdout. ´debuglevel´ is understood to be a bitfield in which each bit specifies some kind of debugging information or behavior. ´debuglevel´ can be specified in either decimal or hex (using the form 0x#) formats. The bits in the bitfield are defined as follows:

Bit	Hex Value	Meaning
0	0x001	Dump Internal Options & User-Settable Options
1	0x002	Dump User-Defined Variables
2	0x004	Dump Command Definitions
3	0x008	Dump Key Bindings
4	0x010	Display, Do Not Execute, Commands When Invoked
5	0x020	Dump Directory Stack As It Changes
6	0x040	Dump Command History Stack After Command Executes
7	0x080	Dump Contents Of Program Memories As They Change
8	0x100	Dump Contents Of Filter/Selection Wildcard Lists As They
9	0x200	Dump Associations Table
10	0x400	Reserved/Unused
11	0x800	Dump Requested Debug Information And Exit Immediately

These bits can be combined to provided very specific debugging information. For example, ´-d 0x80f´ will dump (to stdout) all the Internal Options, User-Settable Options, User-Defined Options, Command Definitions, and Key Bindings and then terminate the program.

-h Print help information on stdout.

-q Quiet mode - suppresses warnings. (default: warnings on)

-r Turn off automatic refreshing of directory display. (default: refresh on)

Normally `twander` re-reads and displays the current directory every few seconds to reflect any changes that might have occurred to that directory's contents. This option is useful on slow machines (or slow X connections) and/or when working with very large directories. In this situation, the frequent updating of the `twander` display can make the program unacceptably slow and unresponsive. In this case you can still force an update manually with the REFRESH function (default assignment is to the Control-l key).

-t Turn off quoting when substituting built-in variables. (default: quoting on)

Anytime `twander` encounters a reference to one of the built-in variables which do string replacement (DIR, DSELECTION, DSELECTIONS, MEM1-12, PROMPT:, SELECTION, SELECTIONS) in a command, it will replace them with **double quoted** strings. This is necessary because any of these can return values which have embedded spaces in them. By quoting them, they can be passed to a command or script as a single item. The -t option disables this behavior and replaces the built-in variable with unquoted literals.

-v Print detailed version information.

OTHER WAYS TO SET `twander` OPTIONS

In addition to these command line options, there are two other ways you can set `twander` program features. If you prefer, you can set the command line options via the environment variable, TWANDER. That way you don't have to type them in each time you start the program. Say you set the environment variable this way on Unix:

```
export TWANDER=-qt
```

From then on, every time you run the program, the -q and -t options would be invoked (No Quoting, No Warnings) just as if you had typed them in on the command line.

The second way to set these (and MANY more) Program Options is by setting the appropriate entries in the Configuration File. This is covered later in this document.

`twander` evaluates options in the following order (from first to last):

- Internally set default value of options
- Options set in the Configuration File
- Options set in the TWANDER environment variable
- Options set on the command line

This means, for example, that the environment variable overrides a corresponding setting in the Configuration File, but the command line overrides the environment variable. Furthermore, there are many Program Options which can **only** be set/changed the Configuration File and are not available in either the environment variable or on the command line.

This also means that options set on the command line are not read until after the Configuration File has been processed. So, the -q argument on the command line will not inhibit warnings generated during the reading of the Configuration File. This is best done by adding the statement, WARN=False, at the top of the Configuration File.

If the Configuration File is reloaded while the program is running (see the READCONF key below), any options set in the file will have the last word. This allows you to edit the Configuration File and have your changes reflected in a running instance of `twander`, but it also means that the environment variable/command line arguments are ignored after initial program startup.

THE TITLE BAR

`twander` displays a lot of information about the state of the running program in the main window title bar. From left to right you will see:

- Program name and version number.

- Login name and machine/domain of current user.
- The path of the directory you are currently viewing. If this path length is greater than 60 characters, `twander` will show the last 60 characters of the path length, prepended with "..." to show that it is truncated.
- Any "filter" that is limiting which files you see. If you've toggled the filter, you will see the word "NOT" before the filter string.
- An indication of whether or not "dotfiles" are currently hidden.
- The total number of files in this directory. The ".." entry is not included in this count.
- The total size of all the files in this directory. The size of the ".." directory is not included in this total.
- They key used to sort the display. If a reverse sort is selected, you will see "-" appended to the end of the key to indicated this. "Sort By: NAME-" means you are doing a reverse sort by name.
- An indication of whether or not directories and ordinary files are being separated in the display.
- An indication of whether or not automatic refreshing is enabled. Anytime refreshing is actually underway, whether automatic, manual, because you are executing a command that forces a refresh, or just because you changed directories, you will see the '*' character appended to this field. Ordinarily, this happens so quickly you will not see it. However, on really large directories and/or very slow disks like CDROMs, you'll see the asterisk stay on for some time. During refresh, the program is locked from user input. On very long refreshes, it can appear to be "hung". This indicator is there just to let you know the program is busy refreshing and all is well.

KEYBOARD USE

By design, `twander` allows you to do almost everything of interest using only the keyboard. Various `twander` features are thus associated with particular keystrokes which are described below. It is also very simple to change the default key assignments with entries in the Configuration File, also described below.

NOTES ON KEYBOARD ARROW/KEYPAD BEHAVIOR AND TEXT DIALOG EDITS

Generally, the arrow and keypad keys should do what you would expect on the system in question. On Windows systems, particularly, there ought to be no odd arrow/keypad behavior.

X-Windows is somewhat more problematic in this area. Just what an arrow key is "supposed" to do depends on how it's been mapped in your X server software. Testing `twander` on various X servers showed quite a bit of variability in how they handled the arrows and keypad. So ... if you're running in an X Windows universe and arrows or keypad do nothing, or do strange things, look into your key maps, don't blame `twander`.

There are several features of `twander` that will present the user a text entry dialog. These include the CHANGEDIR and RUNCMD features as well as the {PROMPT:...} Built-In Variable (all described below).

Any time you are entering text in such a dialog, be aware that the text can be edited several ways - You can edit it using the arrow/keypad editing assignments which are enabled/normal for your operating system, OR

you can use emacs-style commands to edit the text. For instance, Control-a, Control-k will erase the text currently entered in the dialog.

DEFAULT KEYBOARD AND MOUSE BINDINGS

Here, ordered by category, are the default keyboard and mouse bindings for `twander`. The general format is:

Description (Program Function Name)

Default Key Assignment

Default Mouse Assignment (if any)

The "Program Function Name" is the internal variable `twander` uses to associate a particular feature with a particular keystroke or mouse action. You can ignore it unless you intend to override the default key assignments. This use is described below in the section entitled, **Key Binding Statements.**

It is important to realize that `twander` key-bindings are **case-sensitive.** This means that ´Control-b´ and ´Control-B´ are different. This was a conscious design decision because it effectively doubles the number of Control/Alt key combinations available for the addition of future features.

The default bindings chosen for `twander` features are all currently **lower-case.** If your program suddenly stops responding to keyboard commands, check to make sure you don't have CapsLock turned on.

NOTE: Some `twander` features are doubled on the mouse. These mouse button assignments are documented below for the sake of completeness. However, **mouse button assignments cannot be changed by the user,** even in the Configuration File.

General Program Commands

This family of commands controls the operation of `twander` itself.

Clear History (CLRHIST)

Control-y

Clears out various program histories including the All Visited Directories list, the Directory Stack, the Command History, and the last manually-entered values for CHANGEDIR and RUNCMD. The 12 Program Memories are not cleared - they have specially dedicated key bindings for this purpose.

Decrement Font Size (FONTDECR)

Control-[

Decrease font size.

Increment Font Size (FONTINCR)

Control-]

Increase font size.

These two features allow you to change the display font sizes while `twander` is running. But, you may not immediately get the results you expect. `twander` internally keeps track of separate font sizes for the main display, the main menu text, and the help menu text. When you use the two

font sizing commands above, `twander` subtracts or adds 1 to each of these three values respectively. On systems like Windows using TrueType fonts, this works as you would expect, because every font is effectively available in every size. However, in systems like X-Windows or Windows using fixed-size fonts, you may have to press these keys repeatedly until `twander` finds a font matching the requested size.

This can also cause some parts of the display to change but not others. Suppose you are running on X-Windows and have specified that the main display is to use a 12 point font, and that menus and help should use 10 point font. Let's also suppose that the next font available larger than 12 point is 16 point. If you press FONTINCR twice, both the menu text and help text will jump to 12 point, but the main display text will remain unchanged. Why? Because pressing FONTINCR twice tells `twander` to set the main display to 14 point (12+1+1) which does not exist, and the menu and help text to 12 point (10+1+1) which does exist, so that change is visible.

The "User-Settable Options" Help Menu displays the font metrics (name, size, weight) you've currently specified. Pressing FONTDECR/FONTINCR changes the size specification and this will be reflected in that menu. However, most systems do some form of "best match" font substitution - if you ask for a font that does not exist, the system will use the "closest matching" font as a substitute. This means the font you see specified in the Help Menu is not necessarily the font you're actually using. You're more likely to run into this when running on a Unix/X-Windows system (where not all the fonts are available in all sizes/weights like they are on Windows TrueType) as you change the font size with FONTDECR/FONTINCR.

Reloading the Configuration File (READCONF) will reset the fonts to either their default values or any font sizes specified in the Configuration File.

Display Command Menu (MOUSECTX)
Right-Mouse-Button

Displays a list of all available commands in a pop-up menu near the mouse pointer. If no commands are defined, this feature does nothing at all. This means commands can be invoked one of three ways: Directly via the Command Key defined in the Configuration File, via selection in the Command Menu at the top of the GUI, or via selection from the Command Menu.

Windows users should note that, unlike Windows Explorer, the `twander` Command Menu does not change the set of currently selected items. It merely provides a list of available commands. This allows the command chosen via the Command Menu to operate on a previously selected set of items.

Display Directory Menu (MOUSEDIR)
Shift-Right-Mouse-Button

Displays a list of all the directories visited so far in a pop-up menu near the mouse. This means that you can navigate to a previously visited directory in one of two ways: Via a selection in the Directory Menu at the top of the GUI or via a selection from this pop-up menu.

Display History Menu (MOUSEHIST)
Control-Shift-Right-Mouse-Button

Displays a list of all commands executed so far (including those entered manually) in a pop-up menu near the mouse pointer. If the Command History is empty, this command does nothing. This means you can repeat a previously entered command via the History Menu or this mouse command. (You can also repeat the last manually entered command by pressing RUNCMD - it

will preload its text entry area with the last command you entered by hand.)

Display Shortcut Menu (MOUSESC)
Alt-Control-Left-Mouse-Button

Displays a list of all user-defined directory shortcuts in a pop-up menu near the mouse. The menu also has "canned" navigation shortcuts to go up a directory, back a directory, to the home directory, to the starting directory, and to the root directory. On Windows systems with the Win32All extensions, there is also a shortcut to the Drive List View.

Display Sorting Menu (MOUSESORT)
Alt-Shift-Right-Mouse-Button

(Note that on Windows you must press Alt **then** Shift **then** the Right-Mouse-Button for this to work. Windows appears to care deeply about keystroke order.)

Displays a list of all the sorting options in a pop-up menu near the mouse.

Quit Program (QUITPROG)
Control-q

Exit the program.

Re-Read Configuration File (READCONF)
Control-r

Re-read the Configuration File. This allows you to edit the Configuration File while `twander` is running and then read your changes in without having to exit the program. This is handy when editing or changing Command Definitions.

Program Options are set back to their default each time a Configuration File is about to be read (initially or on reload) just before the Configuration File is parsed. This means commenting out or removing a Program Option Statement (see relevant section below) in the Configuration File and then pressing READCONF causes that option to be reset to its default value. STARTDIR defaults to either its internal default ($HOME or ./) or to the value given in the Environment Variable/Command line.

Refresh Display (REFRESH)
Control-l

Re-read the current directory's contents and display it. This is most useful if you have turned off automatic directory refreshing with either the -r command line flag or setting the AUTOREFRESH Program Option to False.

Toggle Autorefreshing (TOGAUTO)
Control-o

Toggle Autorefreshing on- and off. This is handy if you are about to enter a very large directory and/or a very slow disk (like a CDROM). With very large or slow directory reads, `twander` can end up spending all its time doing re-reads of the directory and never give you time to do anything there. If you find this is consistently the case, then you need to increase REFRESHINT. But for

the occasional adventure into very large/slow directories, just toggling Autorefresh off is more convenient.

The state of the Autorefresh feature is displayed on the main window title bar.

Toggle Details (TOGDETAIL)
Control-t

Toggle between detailed and filename-only views of the directory.

Toggle Between Normalized And Actual File Length Display (TOGLENGTH)
Control-0

By default, the program "normalizes" file sizes and expresses them in bytes, Kilobytes, Megabytes, or Gigabytes rather than showing their actual size. This is done everywhere a file size is displayed: on individual files, the total files size displayed on the title bar, and the drive sizes in Win32 Drive List View. This key binding invokes a feature that toggles these size displays between normalized and actual. See the ACTUALLENGTH configuration option below to set the default as you prefer it.

Toggle ´win32all´ Features (TOGWIN32ALL)
Control-w

As described later in this document, `twander` provides enhanced features for Windows users who also install Mark Hammond's ´win32all´ extensions for Python on Windows. This key binding will toggle those advanced features on- and off. This is useful if you happen to be examining a very large directory. The ´win32all´ features, while handy, can be computationally expensive and make updates of a directory with many entries somewhat slow. This toggle is provided as a means to temporarily disable the advanced features when viewing such a directory.

Directory Navigation
This family of commands controls movement between directories. If you attempt to navigate into a directory that does not exist or which does not have appropriate permissions, `twander` will display a warning message and remain in the current directory. This is **unlike** the case of a non-existent or unreadable directory specified when the program is first started. In that case, `twander` reports the error and aborts.

Change Directory (CHANGEDIR)
Control-x

This is a shortcut that allows you to directly move to a new directory/path - i.e., Without having to navigate to it.

Unless you have set the MAXMENU option to 0, CHANGEDIR keeps track of your last manually entered directory and presents it as a default when you press CHANGEDIR again. You can then move to that directory, edit the string to specify another directory, or delete it and enter an entirely new directory. Directories can be edited with either the arrow and keypad keys defined on your system or by emacs editing commands like Control-a, Control-k, Control-e, and so forth.

Go To Home Directory (DIRHOME)
Control-h

If the "HOME" environment variable is defined on your system, this will move you to that

directory. If the "HOME" environment variable is not defined, this command will move to the original starting directory.

Go Back One Directory (DIRBACK and MOUSEBACK)

Control-b

Control-DoubleClick-Left-Mouse-Button

`twander` keeps track of every directory visited and the order in which they are visited. This command allows you to move back successively until you get to the directory in which you started. This feature is implemented as a stack - each "backing up" removes the directory name from the visited list. The "Directory" menu (see **MENU OPTIONS** below) implements a similar feature in a different way and keeps track of all directories visited regardless of order.

Go To Root Directory (DIRROOT)

Control-j

Go to the root directory.

Go To Starting Directory (DIRSTART)

Control-s

Go back to the original directory in which `twander` was started.

Go Up To Parent Directory (DIRUP and MOUSEUP)

Control-u

Control-DoubleClick-Right-Mouse-Button

Move to the parent of the current directory ("..").

Display Drive List View (DRIVELIST)

Control-k

This is a Windows-only feature which displays a list of all available disk drives. Details about each drive are also displayed if you have details enabled. In order for this feature to work, you must be running on Windows AND have the ´win32all´ package installed, AND the USEWIN32ALL Program Option must be True (default condition,) AND you must not have toggled these features off with the TOGWIN32ALL key described above. For more details about Drive List View, see the section below entitled, **ADVANCED WINDOWS FEATURES.**

Selection Keys

This family of commands controls the selection of one or more (or no) items in the current directory.

Select All Items (SELALL)

Control-Comma

Select every item in the current directory. The ".." entry at the top of the directory listing is not included. (We almost never want to include the parent directory when issuing a command on "everything in this directory". If you do wish to include the "..", do the SELALL command first, then click on ".." while holding down the Control key.)

Invert Current Selection (SELINV)
Control-i

Unselects everything which was selected and selects everything which was not. As with SELALL, and for the same reason, the ".." entry is never selected on an inversion.

Unselect All Items (SELNONE)
Control-Period

Unselect everything in the current directory.

Select Next Item (SELNEXT)
Control-n

Select next item down in the directory.

Select Previous Item (SELPREV)
Control-p

Select previous item up in the directory.

Select Last Item (SELEND)
Control-e

Select last item in the directory.

Select First Item (SELTOP)
Control-a

Select first item in the directory. This will always be the ".." entry, but it is a quick way to get to the first part of a very long directory listing which does not all fit on-screen.

Mouse-Based Selections

The mouse can also be used to select one or more items. A single-click of the left mouse button selects a particular item. Clicking and dragging selects an adjacent group of items. Clicking an item and then clicking a second item while holding down the "Shift" key also selects an adjacent group of items. Finally, a group of non-adjacent items can also be selected. The first item is selected with a single left mouse button click as usual. Each subsequent (non-adjacent) item is then selected by holding down the "Control" key when clicking on the item.

Scrolling Commands

If a given directory's contents cannot be displayed on a single screen, `twander` supports both vertical and horizontal scrolling via scrollbars. This capability is doubled on the keyboard with:

Scroll Page Down (PGDN)
Control-v

Scroll down one page in the directory listing.

Scroll Page Up (PGUP)
Control-c

Scroll up one page in the directory listing.

Scroll Page Right (PGRT)
Control-g

Scroll to the right one page width.

Scroll Page Left (PGLFT)
Control-f

Scroll to the left one page width.

Command Execution Options

This family of commands causes `twander` to actually attempt to execute some command you've chosen:

Run Arbitrary Command (RUNCMD)
Control-z

This is a shortcut that allows you to run any command you'd like without having to define it ahead of time in the Configuration File. It is more-or-less like having a miniature command line environment at your disposal.

You may enter a number of different things in the RUNCMD dialog. You may type literal text or refer to any of the variable types (User-Defined, Environment, or Built-In) supported by `twander` just as you do in writing Command Definitions (see below). This makes it easy to enter complex commands without having to type everything literally. For example, if you would like to copy all the currently selected files to a new directory, press RUNCMD and enter (on Unix):

```
cp [SELECTIONS] newdir
```

`twander` understands the variable reference syntax here just as it does in a Command Definition. This also gives you a single way of referring to environment variables, regardless of OS platform. Recall that in Unix-like shells, an environment variable is in the form "$NAME", but on Windows it is in the form "%NAME%". Instead if having to keep track of this difference, you can just use a `twander` Environment Variable reference. For instance, assuming the EDITOR environment variable is set, this command works the same on both systems:

```
[$EDITOR] [SELECTIONS]
```

Built-in variables are most often used when manually entering commands So, RUNCMD also understands some "shortcut" references to many of the built-ins. You may use:

```
[D]  for  [DIR]
[DN] for  [DSELECTION]
[DS] for  [DSELECTIONS]
[SN] for  [SELECTION]
[SS] for  [SELECTIONS]
[1]  for  [MEM1]
[2]  for  [MEM2]
```

```
[3]  for  [MEM3]
[4]  for  [MEM4]
[5]  for  [MEM5]
[6]  for  [MEM6]
[7]  for  [MEM7]
[8]  for  [MEM8]
[9]  for  [MEM9]
[10] for  [MEM10]
[11] for  [MEM11]
[12] for  [MEM12]
```

Of course, the full form is also fine as well.

This shortcut feature is only supported in RUNCMD!!! Configuration File entries must use the full form of all built-in variables. This is a conscious design decision to help enforce some consistency and clarity in the Configuration Files.

Unless you have set the MAXMENU option to 0, RUNCMD keeps track of your last manually entered command and presents it as a default when you press RUNCMD again. You can then run the command again exactly as you last entered it, you can modify it before running the command again, or you can delete it and enter an entirely new command. Commands can be edited with either the arrow and keypad keys defined on your system or by emacs editing commands like Control-a, Control-k, Control-e, and so forth.

Also see the section below entitled, **Program Option Statements,** to understand the CMDSHELL option. This option greatly simplifies running command-line programs from RUNCMD so their output can been seen in a GUI window. This is particularly handy on Unix.

As with command definitions in a Configuration File, you can tell `twander` to force a display refresh after the command has been initiated. You do this by beginning the command with the ´+´ symbol. So, for example, if you enter,

```
+mycommand
```

`twander` will initiate the command, wait AFTERWAIT seconds (default: 1), and then update the display. See the discussion below entitled, **Forcing Display Updates In Command Definitions** for a more complete explanation.

This feature may be used in combination with CMDSHELL escaping (also described in the **Program Option Statements** section below) and the two characters may appear in any order at the beginning of the command line you enter.

Run Selected File / Move To Selected Directory (SELKEY and MOUSESEL)

Return (Enter Key)

DoubleClick-Left-Mouse-Button

If the selected item is a Directory, `twander` will move into that directory when this command is issued. If the selected item is a file, `twander` will attempt to execute it. Whether or not the file is actually executed depends on how the underlying operating system views that file.

In the case of Unix-like operating systems, the execute permission must be set for the user running `twander` (or their group) for the file to be executed.

On Windows, the file will be executed if the user has permission to do so **and** that file is either executable or there is a Windows association defined for that file type. For example, double-clicking on a file ending with ".txt" will cause the file to be opened with the ´notepad´ program (unless the association for ".txt" has been changed).

If `twander` determines that it is running on neither a Unix-like or Windows system, double-clicking on a file does nothing.

Run User-Defined Command
User-Defined (Single Letter) Key

Each command defined in the Configuration File has a Command Key associated with it. Pressing that key will cause the associated command to be run. If no command is associated with a given keystroke, nothing will happen when it is pressed.

Directory Shortcuts
`twander` provides a way to directly navigate into a frequently-used directory using a single keystroke. You can define up to 12 such "Directory Shortcuts" in the Configuration File. Each of the definitions is associated with one of the following 12 keys:

Navigate Directly To A Directory (KDIRSC1 ... KDIRSC12)
F1 ... F12

Pressing one of these keys changes to the directory associated with it in the Configuration File. For more information on this topic, see the discussion of the Configuration File below entitled, **Directory Shortcut Statements**

Assign Current Directory To One Of The Shortcut Keys (KDIRSCSET)
Control-8

As discussed in **Directory Shortcut Statements** , the directory shortcut keys are associated with particular directories in the `twander` Configuration File. However, it is possible to temporarily assign them to something else while the program is running. This is handy if you need to temporarily "remember" one or more directories so you can jump back to them with a single keystroke. Think of it as a way to "override" the directory shortcut assignments defined in the Configuration File.

To do this, press the KDIRSCSET key (Default: `Control-8`). You will be presented with a dialog that asks you to specify which Directory Shortcut you want overwritten with the **current directory.** You may only enter a number from 1 to 12. You will see an error message if you try to enter anything else.

Any such reassociation of a directory shortcut is temporary. Directory shortcuts are set back to the values specified in the Configuration File if you restart the program or reload the Configuration File (Default: `Control-r`).

Program Memories
If you've used GUIs before, you're probably familiar with the idea of a program "Clipboard" - a temporary holding area which is used when cutting, copying, and pasting files. This is a good idea, but has several limitations. First, most systems only have a **single clipboard.** It would be mighty handy to have muliple Clipboard-like storage areas for keeping track of several different operations at once. Secondly, when you

copy or paste something to a conventional Clipboard, **its old contents get overwritten.** There is no way to keep appending items to the Clipboard. Finally, items usually can only be cut or copied to the Clipboard **from the current directory.** It would be nice if we could not only keep adding things to the Clipboard, but be able to do so as we navigate around the filesystem.

`twander` addresses these concerns by means of 12 separate "Program Memories". As you use `twander`, you can add (append) the names of any directories or files in the currently viewed directory by selecting them and then using the appropriate `twander` MEMSETx key (see below). To take advantage of this feature, you write Command Definitions (or manually issue a command via the RUNCMD key) which reference the contents of a Program Memory using one of the [MEMx] Built-In Variables. (See the section below on entitled, **Program Memory Built-Ins** for more details in how to apply Program Memories).

`twander` provides key combinations for selectively setting and clearing particular Program Memories as well as a key combination for clearing all Program Memories in a single keystroke:

Clear Selected Program Memory (MEMCLR1 - MEMCLR12)
Control-F1 ... Control-F12

Clear (empty) the selected Program Memory.

Clear All Program Memories (MEMCLRALL)
Control-m

Clear (empty) all 12 Program Memories at once.

Set Selected Program Memory (MEMSET1 - MEMSET12)
Alt-F1 ... Alt-F12

Append the **the full path names** of the currently selected files and directories to the Program Memory desired.

Sorting Options

`twander` provides a variety of ways to sort the display. These can be selected with either a keystroke or from the Sorting Menu (see below). The meaning of the sort depends on whether or not you are in Drive List View (see **ADVANCED WINDOWS FEATURES** below). The table below summarizes the keys associated with sorting and their meaning in the two possible views:

Key	Program Function Name	Sort Order In Normal View	Sort Order In Drive List View
Shift-F10	SORTBYNONE	No Sort	No Sort
Shift-F1	SORTBYPERMS	Permissions	Label/Share String
Shift-F2	SORTBYLINKS	Links	Drive Type
Shift-F3	SORTBYOWNER	Owner	Free Space
Shift-F4	SORTBYGROUP	Group	Total Space
Shift-F5	SORTBYLENGTH	Length	Drive Letter
Shift-F6	SORTBYTIME	Time	Ignored
Shift-F7	SORTBYNAME	Name	Ignored
Shift-F11	SORTREV	Reverse Order	Reverse Order
Shift-F12	SORTSEP	Separate Dirs	Ignored

An easy way to remember these is that the function key number for the primary sort keys corresponds to the column position of the key in a detailed display. For instance, Shift-F1 sorts by column 1, Shift-F2 by column 2, and so forth.

These sorting options are available **whether or not details are currently available.** For example, you can toggle details off, but still sort by one of the now invisible details such as Owner, Length, and so on.

SORTREV reverses the order of the sort.

SORTSEP toggles whether or not directories and files should be grouped separately or displayed in absolute sort order. If enabled, directories will be displayed first, then files. If the sort is reversed via SORTREV, and SORTSEP is enabled, the directories will appear **after** the files, sorted by whatever sort key has been chosen. SORTSEP is not meaningful in Drive List View and is ignored.

You'll find the currently selected sorting options displayed in the program title bar.

Wildcard Features

Although `twander` provides a very rich set of keyboard and mouse selection commands, selecting a group of files out of list of hundreds or thousands in a large directory can be tedious. If the files/directories you want to select have some lexical commonality **in their names OR details** you can have `twander` select them for you using so-called "Regular Expressions".

You can do this in one of two ways. A wildcard "filter" only **displays** files that match the specified regular expression. A wildcard "selection" **selects** (highlights) the matching entries from the currently displayed list. The general idea is to use filters to limit the number of files you'll even see in the `twander` interface and then optionally choose from among them with a wildcard-based selection.

For example, suppose you initiate a wildcard-based selection (SELWILD) with the text, `tar`. This would select every file or directory in the current display where the string "tar" **appeared anywhere on the line for that file/directory.** This is very important: Wildcard matching takes place anywhere on the visible line. So, if you have details turned on, the match can occur anywhere on the permissions, links, group, owner, and so on. Obviously, if you have details turned off, the match can only occur on the name of the file or directory since that's all that is visible.

This is a purposeful design decision because it allows you to make selections on more than just the name. Say you enter the following in the FILTERWILD dialog:

```
drwx------
```

`twander` would display only the entries that are directories with no permissions enabled for group or world users.

The matching string above could also filter/select other entries (not having the permissions just described), if say, this string appeared in their name ... a rather unlikely scenario, but not impossible. If we want to get **real** specific about which entries we want selected, we need to enter a "regular expression" in the wildcard dialog. Regular expressions are a far more powerful pattern-matching tool than simple text strings and will allow you to do some fairly amazing selections. For example, this regular expression selects all entries which contain a string beginning with "Ju" followed by any other character, a single space, and ending in "0":

```
Ju. 0
```

So, for instance, this would select files with date details (or names, or anything else on the line...) like "Jun 01", "Jul 03", and "Jul 09".

No matter what you specify, a literal matching string or a regular expression, the ".." entry of the currently viewed directory is never selected for wildcard processing. This is a "special" entry that is always present regardless of filtering and never selected with wildcard-based selections.

Notice that these regular expressions are **not** the same thing as the filename "globbing" wildcards commonly used with Unix and Windows shells. If you enter constructs like "*.txt" or "*.tar.gz", you will not get the results you expect. In fact, these specific examples will cause `twander` to grumble and present a warning message.

For an excellent tutorial on Python-compliant regular expressions, see:

```
http://www.amk.ca/python/howto/regex/
```

By default, these wildcarding tools will select an entry when your regular expression matches anything on the displayed line. This allows you to make selections based on any visible column of information. This "match anywhere on the line" semantic is possible because `twander` automatically massages the regular expression you provide to make "any match on the line" true. There may be times when you want to provide very specific regular expression definitions which seek a match at specific locations. In that case, you can prevent the program from fiddling with your regular expression, by beginning it with the double-quote (") character. `twander` understands this to mean that your regular expression is to be treated literally without modification. (It only throws away this leading escape character.)

Suppose we changed our example above slightly and used this regular expression:

```
"^drwx------
```

Now `twander` would select **only** the directories without any group and world access because:

- The leading double-quote (") forces literal interpretation of the regular expression - i.e. It turns off "match anywhere" semantics as just described.

- The carat (^) at the beginning of the actual regular expression "anchors" the match to the start of the line. For a match to be declared (and for `twander` to select an item) the regular expression must be satisfied at the beginning-of-line.

Because regular expressions can get complicated and tedious to type in, any such expression you use is saved in a history available via the Filter and Select Menus (see below). There is also provision for predefining frequently used wildcards in your Configuration File (see section below on `WILDFILTER=` and `WILDSELECT=` configuration statements) so you don't have to type them in manually each time you start the program - you can just select them from the relevant menu.

A few points to keep in mind when using wildcard features:

- By default, wildcard matching is case-insensitive on Win32 systems and case-sensitive everywhere else. This is because Windows systems allow case in filenames and attributes, but it is not significant - i.e., The case of a filename or attribute is ignored on Windows systems. You can control this explicitly with the `WILDNOCASE` configuration option. If you set `WILDNOCASE=False`, it will force all wildcard filters and selections to be case-sensistive. Setting it to `True` makes the wildcarding case-insensitive. This option is available for both Unix and Win32 systems so you can set the behavior you like anywhere.

- If you escape a wildcard to force `twander` to treat it exactly as you defined it, the case-sensitivity set by default or `WILDNOCASE` is ignored. Escaped wildcards are always treated **exactly** as you enter them and they are matched against the filename and/or details exactly as they appear.

- Wildcard-based filters are applied against the **entire contents** of the current directory to determine which files match and should be displayed. But, wildcard-based selections are done against the **currently visible** files. This is important if you do a filter and then a selection wildcard. The first will select which files to display. The second will select which ones to highlight from the displayed list.

Display Files Matching A Regular Expression (FILTERWILD)
Control-equal

This will present you with a dialog to enter your regular expression matching criteria described above. After you enter it, `twander` will only display the files that match. The filter is reset (to no filtering) when you manually referesh the directory - with `REFRESH` (default: Control-l - or change directories.

Toggle Active Filter (TOGFILT)
Control-minus

Pressing this once "inverts" any filter currently active. It means "show me the files that **don't** match the filtering regular expression." Pressing it again returns the filter to its normal meaning. This is handy when you want to display everything **except** a group of files. You first filter for the files you don't want and then press `TOGFILT` which will display everything except these files.

Select Files Matching A Regular Expression ´Wildcards´ (SELWILD)
Control-\

This will present you with a dialog to enter your regular expression matching criteria described above. After you enter it, `twander` will select (highlight) the files that match.

You can also "invert" your selections by using the `SELINV` key described previously. This is useful when you want to select everything **except** a group of files. Select the ones you don't want with a selection wildcard andthen press the `SELINV` key.

Selections remain in effect until you make another manual selection, clear all selections, or run a command that forces a directory refresh after it runs - i.e., Commands defined with a leading "+".

Display Selection Wildcard Menu (MOUSEFILTERSEL)
Alt-Control-Middle-Mouse-Button

Display Selection Wildcard Menu (MOUSEWILDSEL)
Alt-Control-Right-Mouse-Button

(Note that on Windows you must press Alt **then** Control **then** the mouse button for this to work. Windows appears to care deeply about keystroke order.)

These keys popup a list near the current mouse cursor of any previously used filtering or selection wildcards respectively. Selecting one of the entries therein pops-up a dialog that allows you to edit the wildcard before actually doing another wildcard filter or selection. This allows use to modify previous wildcards for new use.

Hiding Dotfiles

By convention on Unix and many other systems, files or directories whose names begin with a dot ('.') are usually used as configuration files (directories). Unless you specifically want to edit a configuration, you typically do little or nothing with these files. Since there can be quite a few of them on a modern system, it's helpful to be able to block them from view.

By default, dotfiles are not hidden, but this can be changed with the HIDEDOTFILES configuration option.

By default, files or directories whose names begin with a period (".") are considered dotfiles. You can change this dotfile "introducer" string with the DOTFILE configuration option.

Toggle Dotfile Hiding (TOGHIDEDOT)

Control-9

This toggles dotfile hiding, on- and off. The program starts up with dotfiles visible or hidden as defined by the HIDEDOTFILES program option. Thereafter, TOGHIDEDOT can be used to make these files and directories visible or not.

Unlike Wildcard Filters (which test the entire displayed line), dotfile hiding is triggered only by the **name** of the file or directory.

If you change DOTFILE to some other string, be aware that the test to see if a file (or directory) name starts with this string is **case-sensitive.** If you set DOTFILE to "De", it will not hide files starting with "de", for example.

The current state of dotfile hiding is displayed in the window title bar, immediately after the Filter information.

Note that even though you cannot see the files with this option enabled, commands you write can still operate on these files. For example, if you define a command that does something like:

```
c cleandot rm .*~
```

This command will remove any backup (~) files, whether or not you can see them in the interface.

MENU OPTIONS

Although `twander` is primarily keyboard-oriented, several menu-based features are also implemented to make the program more convenient to use. These menus appear at the top of the `twander` display window, above the directory listing.

Invoking A Menu

A menu can be invoked in one of several ways. You can click on it, you can press its associated "Accelerator Key" combination, or you can use the "Mouse Shortcut" to cause a copy of the menu to pop-up near the mouse pointer. The Accelerator Keys are shown in parenthesis next to the menu names below and the Mouse Shortcuts are similarly shown below in square brackets. All menus have Accelerator Keys defined, but only some menus have associated Mouse Shortcuts.

Detaching A Menu

The first item in each menu is a dashed line ("----") which indicates that it is a "tearoff" menu. Clicking on the dashed line will detach the menu from `twander` allowing it to be placed anywhere on screen. Even

when detatched, these menus remain current and in-sync with `twander` as it continues to run. You can also tear off multiple instances of these menus if you'd like copies of them at several locations on the screen simultaneously.

Managing The Size Of Dynamic Menus

A number of these menus have "dynamic" content - their content changes as the program runs. For example, the Directory, History, Filter and Select menus all keep some sort of "history" of what the program has done. Their content thus grows longer as the program is used.

On Windows systems, if such menus grow too long to physically fit on screen, up- and down- scrolling arrows automatically appear at the top- and bottom of the menu respectively. This is not a feature of the Unix Tk implementation, so menus which grow too large are simply truncated to fit the screen on Unix-like systems.

There are two User-Settable Options options available to help you manage the maximum size of dynamic menus (see the section below on the Configuration File which explains how such options are actually set. The MAXMENU option specifies the maximum number entries **that will be displayed** in any dynamic menu. (`twander` internally tracks MAXMENUBUF number of items for each dynamic menu.) This defaults to 32 as is intended to keep the menu size reasonable.

If you set MAXMENU=0, it means you are **disabling** all dynamic menus. It also means that no interactive dialog will "remember" your last manual entry. For example, with MAXMENU set to 0, `twander` will not keep track of your last manual entries for the CHANGEDIR, FILTERWILD, SELWILD, and RUNCMD dialogs.

MAXMENUBUF specifies the size of the internal storage buffer for each dynamic menu regardless of how many entries are actually displayed. i.e. MAXMENUBUF determines how many dynamic events each menu tracks internally regardless of how many are actually visible in the menu at any moment in time. It defaults to 250 and probably never needs to be changed. If you set MAXMENUBUF to be less than MAXMENU, then this smaller value will determine the maximum size of the displayed menu. Setting MAXMENUBUF to 0 is equivalent to setting MAXMENU to 0.

Commands Menu (Alt-c) [Right-Mouse-Button]

Every command defined in the Configuration File is listed in this menu by its Command Name. The association Command Key is also shown in parenthesis. Clicking on an item in this menu is the same as invoking it from the keyboard by its Command Key. This is a convenient way to invoke an infrequently used command whose Command Key you've forgotten. It is also handy to confirm which commands are defined after you've edited and reloaded the Configuration File. The commands are listed in the order in which they are defined in the configuration file. This allows most frequently used commands to appear at the top of the menu by defining them first in the Configuration File. If no commands are defined, either because the Configuration File contains no Command Definitions or because the Configuration File cannot be opened for some reason, the Commands Menu will be disabled (grayed out).

History Menu (Alt-h) [Shift-Control-Right-Mouse-Button]

`twander` keeps track of every command you attempt to execute, whether it is an invocation of a Command Definition found in the Configuration File or a manually entered command via the RUNCMD key. (default: Control-z) This is done whether or not the command is successfully executed.

This feature provides a quick way to re-execute a command you've previously run. When you select a command to run this way, a dialog box is opened, giving you an opportunity to edit the command before running it again.

One important point of clarification is in order here. If you run one of the commands defined in your Configuration File, it is stored in the History **after all variable substitutions have been made.** But, manually entered commands are stored in the History **literally as typed -** i.e., Without variable substitution. This allows you easily reuse a manually entered command in another directory or context. (Presumably, Command Definitions in the Configuration File are written in such a way so as to be useful across many different directories and contexts. Running such a command again is simply a matter of pressing its associated letter key once more. By storing the resolved version of the command in the History, you can see what the command actually did.)

The History Menu is emptied and grayed out when you press the CLRHIST key. (default: Control-y)

Directories Menu (Alt-d) [Shift-Right-Mouse-Button]

`twander` keeps track of every directory visited. The previously described command to move "Back" one directory allows directory navigation in reverse traversal order - you can back up to where you started. However, this feature "throws away" directories as it backs up, sort of like an "undo" function.

The "Directories" menu provides a slightly different approach to the same task. It keeps permanent track of every directory visited and displays that list in sorted order. This provides another way to move directly to a previously visited directory without having to manually navigate to it again, back up to it, or name it explictly using the Change Directory command.

Unless MAXMENU is set to 0, the Directory Menu shows the last MAXMENU directories visited in alphabetically sorted order (unless you change MAXMENUBUF to be smaller than MAXMENU). "Visited", in this case, is stretching things a bit.

The Directory Menu is emptied and grayed out when you press the CLRHIST key. (default: Control-y)

Shortcut Menu (Alt-u) [Alt-Control-Left-Mouse-Button]

This menu provides a way to access any of the Directory Shortcuts defined in the Configuration File. It also provides a number of "canned" navigation shortcuts to go up a directory, back a directory, to the home directory, to the starting directory, and to the root directory. On Windows systems using the Win32All extensions, there is also a shortcut to the Drive List View.

Filter Menu (Alt-f) [Alt-Control-Middle-Mouse-Button]

(Note that on Windows you must press Alt **then** Control **then** the Middle-Mouse-Button for this to work. Windows appears to care deeply about keystroke order.)

This menu provides a list of all previously used filtering "wildcard" regular expressions. Any regular expressions defined in the Configuration File (see below) using the "FILTERWILD = " statement will also appear in this menu. This saves you the tedium of constantly having to enter complex regular expression syntax every time you wish to do wildcard-based selections.

Selecting something from this menu brings up a dialog box which allows you to edit the selected wildcard before using it.

Bear in mind that the size of the displayed menu is governed by the MAXMENU and MAXMENUBUF Configuration File options (see below). i.e., Only the last MAXMENU number of wildcards are actually displayed on the menu.

The Filter Menu is emptied and grayed out when you press the CLRHIST key. (default: Control-y) This history is **not** cleared if the Configuration File is reloaded.

Select Menu (Alt-l) [Alt-Control-Right-Mouse-Button]

(Note that on Windows you must press Alt **then** Control **then** the Right-Mouse-Button for this to work. Windows appears to care deeply about keystroke order.)

This menu provides a list of all previously used selection "wildcard" regular expressions. Any regular expressions defined in the Configuration File (see below) using the "SELECTWILD = " statement will also appear in this menu. This saves you the tedium of constantly having to enter complex regular expression syntax every time you wish to do wildcard-based selections.

Selecting something from this menu brings up a dialog box which allows you to edit the selected wildcard before using it.

Bear in mind that the size of the displayed menu is governed by the MAXMENU and MAXMENUBUF Configuration File options (see below). i.e., Only the last MAXMENU number of wildcards are actually displayed on the menu.

The Select Menu is emptied and grayed out when you press the CLRHIST key. (default: Control-y) This history is **not** cleared if the Configuration File is reloaded.

Sorting Menu (Alt-s) [Alt-Shift-Right-Mouse-Button]

(Note that on Windows you must press Alt **then** Shift **then** the Right-Mouse-Button for this to work. Windows appears to care deeply about keystroke order.)

This menu provides a way to select any of the available sorting options. It is context-sensitive and will show entries appropriate to what kind of "view" the program is currently displaying. That is, it will show options which make sense for both "normal" view as well as "Drive List View" (see the **ADVANCED WINDOWS FEATURES** section below).

You'll find the currently selected sorting options displayed in the program title bar.

Help Menu (Alt-l) [No Mouse Shortcut]

This menu provides information about various internal settings of `twander` including Internal Program Variables, User-Settable Options, Keyboard Assignments, User-Defined Variables, Command Definitions, and Associations. It also has an About feature which provides version and copyright information about the program.

For the most part, this help information should fit on screen easily. However, very long Command Definitions will probably not fit on-screen. In this case, if you are curious about just how `twander` is interpreting your Command Definitions, invoke the program with the relevant debug bit turned on and watch the output on stdout as `twander` runs.

THE `twander` CONFIGURATION FILE

Much of `twanders` flexibility comes from the fact that it is a **macro-programmable user interface.** The program itself does little more than provide a way to navigate around a filesystem. It must be configured (programmed) to actually do something with the files you specify. This is done via a "Configuration File". This file is also used to set Program Options and change keyboard assignments. Although the program will run without a Configuration File present, it will warn you that it is doing so with no commands defined.

LOCATION OF CONFIGURATION FILE

By default, the program expects to find configuration information in **$HOME/.twander (%HOME%\.twander on Windows)** but you can override this with the -c command line option.

(Recommended for Windows systems - see the section below entitled, **INSTALLING** `twander`)

Actually, `twander` can look in a number of places to find its Configuration File. It does this using the following scheme (in priority order):

- If the -c argument was given on the command line, use this argument for a Configuration File.
- If -c was not given on the command line, but the HOME environment variable is set, look for the a Configuration File as $HOME/.twander.
- If the HOME environment variable is not set **and** a -c command line argument was not provided, look for a file called ".twander" in the directory from which `twander` was invoked.

CONFIGURATION FILE FORMAT

`twander` Configuration Files consist of freeform lines of text. Each line is considered independently - no configuration line may cross into the next line. Whitespace is ignored within a line as are blank lines.

There are several possible legal lines in a `twander` Configuration File:

```
Comments
Program Option Statements
Key Binding Statements
Directory Shortcut Statements
Wildcard Statements
Variables And Command Definitions
Associations
Conditional Processing Statements
The Include Directive
```

(See the ".twander" file provided with the program distribution for examples of valid configuration statements.)

Everything else is considered invalid. `twander` will respond with errors or warnings as is appropriate anytime it encounters a problem in a Configuration File. An error will cause the program to terminate, but the program continues to run after a warning. For the most part, `twander` tries to be forgiving and merely ignores invalid configuration statements (after an appropriate warning). It only declares an error when it cannot continue. This is true both when the program initially loads as well as during any subsequent Configuration File reloads initiated from the keyboard while running `twander`.

The following sections describe each of the valid Configuration File entires in more detail.

Comments

A comment is begun with the "#" character which may be placed anywhere on a line. Comments may appear freely within a Configuration File. `twander` strictly ignores everything from the "#" to the end of the line on which it appears without exception. This means that "#" cannot occur anywhere within a User-Defined Variable Definition, Key Binding Statement, or Command Definition (these are described below). Comments **can** be placed on the same line to the right of such statements.

It is conceivable that the "#" character might be needed in the Command String portion of a Command Definition. `twander` provides a Built-In Variable, [HASH], for exactly this purpose. See the section below entitled, **Variables And Command Definitions,** for a more complete description.

Program Option Statements

Many of twanders internal program defaults can be overriden in the Configuration File using Program Option statements. These statements look just like the User-Defined variables described later in this document except twander recognizes the variable name as a Program Option rather than an arbitrary variable. Program Option Statements thus take the form:

```
Option Name = Option Value
```

The Option Name is case-sensitive and must be entered exactly as described below. For instance, twander understands "AUTOREFRESH" as a Program Option, but will treat "AutoRefresh" as a User-Defined Variable.

The Option Value is checked to make sure it conforms to the proper type for this variable. The Type can be Boolean, Numeric, or String.

A Boolean Option must be assigned a value of True or False. These logical values can be in any case, so TRUE, TRue, and tRue all work.

A Numeric Option must be a number 0 or greater. Numbers can also be entered in hexadecimal format: 0xNNN, where NNN is the numeric expression in hex.

A String Option can be any string of characters. **Quotation marks are treated as part of the string!** Do not include any quotation marks unless you really want them to be assigned to the option in question (almost never the case).

Furthermore, as described above, you cannot use the ´#´ symbol as part of the string assignment because twander always treats this character as the beginning of a comment no matter where it appears.

For consistency with other Configuration File entries, Program Option Statements may have a blank Right Hand Side. Such statements are simply ignored. This is convenient when you want to leave a placeholder in your Configuration File but don't actually want to activate it at the moment. However, be careful - depending on what precedes the statement, you'll get different settings for the option in question. For example:

```
# This effectively sets BCOLOR to its default value when
# the Configuration File is reloaded
BCOLOR =

# But this means the value of BCOLOR is set to red

BCOLOR = red
BCOLOR =
```

In other words, you should think of Program Option Statements with a blank Right Hand Side as comments - present but ignored.

Other than this basic type-checking, twander does no further validation of the Right Hand Side of a Program Option Statement. It is perfectly possible to provide a RHS which passes twanders type validation but which makes no sense whatsoever to the program. Entries like this cause everything from a mild twander warning to a spectacular program failure and Python traceback on stdout:

```
# A Nice Way To Clobber twander
BCOLOR = goo
```

The following sections document each available Program Option using this general format:

```
Option-Name [Type] (Default Value)
```

ACTUALLENGTH [Boolean] (False)

By default, file sizes , total directory size (on the title bar), and drive sizes in Win32 Drive List view are "normalized". They are expressed in bytes, Kilobytes, Megabytes, or Gigabytes. This keeps the display from getting cluttered with the longer strings required to display the actual lengths in bytes. If you want the program to display the actual lengths for these items by default, set ACTUALLENGTH=True in your Configuration File. You can also "toggle" between normalized and actual size display with the TOGLENGTH key (default: Control-0).

ADAPTREFRESH [Boolean] (True)

Whenever a directory is read, the time to do so is tracked. If that time is less than the current value of REFRESHINT - i.e., The directory read took less than REFRESHINT milliseconds to complete - nothing special happens. But, if the actual directory read time takes **longer** than REFRESHINT milliseconds, `twander` adjusts the value of REFRESHINT upwards. That way, you're guaranteed to have time after the read completes to actually do something.

This dynamic adjustment takes place on every directory read. If you go to a slow directory and REFRESHINT gets dynamically adjusted to, say, 25 seconds, when you go back to a faster/smaller directory, REFRESHINT will be reset to its default value. The changing value of REFRESHINT is **not** shown in the program options help menu. The value there is the one set by default or set in the Configuration File. Think of this as the "base" value for REFRESHINT.

If ADAPTREFRESH is set to False, then adaptive refresh timing is disabled and a directory refresh will be attempted every REFRESHINT milliseconds.

AFTERCLEAR [Boolean] (True)

Tells `twander` to clear any selections in the GUI if a command forces a display refresh after it completes. (See the AFTERWAIT and **Forcing Display Updates In Command Definitions** sections below). This is done because a command that forces a display update is probably changing the content of the current directory (otherwise, why bother with the update?), and the current selections may no longer be relevant.

Setting AFTERCLEAR to False, will leave the current selections alone after doing a command with a forced update.

AFTERWAIT [Numeric] (1)

It is possible to define commands so that a display refresh is forced after a command is invoked (see the section below entitled, **Forcing Display Updates In Command Definitions**). The AFTERWAIT option tells `twander` how long to wait after the command has been initiated before actually doing the refresh. The idea here is to give the command some time to complete before updating the display.

AUTOREFRESH [Boolean] (True)

By default, `twander` regularly re-reads the current directory to refresh the display with any changes. If you are running on a very slow machine or slow connection between the X-Windows server and client, set this option to False. You can manually force an update at any time using the REFRESH key. (default: Control-l)

BCOLOR [String] (Black)

Selects the main display Background Color.

CMDMENUSORT [Boolean] (False)

By default, `twander` populates the command menu on the menu bar (and the popup command menu) with commands in the order in which they are defined in the Configuration File. This was done so that you could define the most important or most frequently used commands first and they would thus conveniently appear at the top of the menu list. However, if you prefer your command list to be sorted, set the `CMDMENUSORT` option to `True`.

Note that the `Command Definitions` help menu ignores `CMDMENUSORT` and is always presented in sorted order.

CMDSHELL [String] ()

This option is primarily intended for people running `twander` on Unix-like operating systems like FreeBSD and Linux. As described in the **GOTCHAS** section below, running a command line program or script requires some extra effort if you want to see the results presented in a GUI window. Typically, you need to run these commands in some kind of ´xterm´ context so that the results will be visible, possibly using a shell as well. So, it's common to see Command Definitions like:

```
x MyCommand   xterm -l -e bash -c 'stuff-for-my-command'
```

In fact, on Unix, the need for this idiom is so common, it's best to define some variables for this. If you look in the example ´.twander´ Configuration File provided in the program distribution, you'll see something like (comments removed):

```
SHELL       = bash -c
VSHELL      = [XTERM] [SHELL]
XTERM       = xterm -fn 9x15 -l -e
```

Now the Command Definition above becomes:

```
x MyCommand [VSHELL] 'stuff-for-my-command'
```

That's all well and good for Command Definitions, but what happens when you want to **manually enter a command** via the RUNCMD key? (default: Control-z) You have to manually enter the gobbledy-gook above, or at least start your command with [VSHELL] (since RUNCMD understands variable references).

The CMDSHELL option is a way to automate this. You can assign it to any literal text. That text

will be **automatically prepended to any command you enter manually.** In this case you could do either of the following in the Configuration File:

```
CMDSHELL = xterm -l -e bash -c

           - or -

CMDSHELL = [VSHELL]  # Assuming VSHELL is defined previously
```

Now every time you enter a command, this will be placed in front of your text before command execution commences.

To disable CMDSHELL operation **permanently,** just remove the statement above from your Configuration File. If you want to leave it in as a placeholder, but deactivate CMDSHELL, use the following statement:

```
CMDSHELL =
```

You also may want to occasionally use RUNCMD to do something without CMDSHELL processing, even though that feature has been defined in the Configuration File. You can disable CMDSHELL operation on a per-RUNCMD basis. Just begin your entering your command with the double-quote (") character. `twander` understands this to "escape" CMDSHELL processing.

As a general matter, CMDSHELL allows you to prepend **anything you like** before a manually entered command - literal text, references to variables, or even the name of a script the system will use to execute your command. Whatever value you use for CMDSHELL will appear in the Command Menu history for each manually initiated command which used this feature - i.e., All the manual commands that **did not** escape the feature.

DEBUGLEVEL [Numeric] (0)

This is another way to set the debugging level you desire (the other way being the -d command line argument). For example, say you want to always dump the current Command Definitions to stdout when the program starts - perhaps you want to redirect this output to a file or printer. Just add this line to your Configuration File:

```
DEBUGLEVEL = 0x004
```

DEFAULTSEP [String] (==>)

This is the string that separates the prompting text and the default response in `{PROMPT: ...}` and `{YESNO: ...}` Built-In Variables. You may change this to any string you like, though doing so is not recommended. Changing DEFAULTSEP will require you to edit any Configuration Files that use these Built-Ins with default responses. In no case should the delimiter string include any of the characters, `[ ]{ }` since these are used as delimiters in the `twander` configuration language.

DOTFILE [String] (.)

It is a convention on Unix (and other systems) that files or directories whose names begin with a period are program configuration files (directories). `twander` has the ability to hide these so-

called "dotfiles". (See the section above entitled **Hiding Dotfiles** for the details.) `twander` treats any file or directory whose name begins with the string defined by DOTFILE as a dotfile for this purpose.

For example, if you set `DOTFILE=Xyz`, all files or directories whose names begin with "Xyz", will be hidden when you tell `twander` to hide dotfiles. Notice that if you change this option from its default, you may use any string to be the dotfile "introducer", but it is always treated with case-sensitivity. For instance, in our example, files beginning with "XYZ" would not be hidden.

FCOLOR [String] (green)

Selects the main display Foreground (Text) Color.

FNAME [String] (Courier)

Selects the main display Font Name.

FSZ [Numeric] (12)

Selects the main display Font Size.

FWT [String] (bold)

Selects the main display Font Weight. This can be assigned to: normal, bold, italic, or underlined. Depending on your system, other values may also be possible.

FORCEUNIXPATH [Boolean] (False)

Ordinarily, Built-In Variables and Program Memory References in a command definition are replaced with strings that list one or more files and/or directories. When this substitution is made at runtime, these strings contain the path separator character appropriate for the underlying operating system ("/" for Unix and "\" for Windows).

If you set FORCEUNIXPATH to True, `twander` will **always** use the Unix path separator character("/") in these substitutions.

This option is primarily useful when writing command definitions with Unix tools under Windows (such as cygwin) that are fussy about path separator conventions.

This option is only relevant on Windows systems. It is ignored on other operating systems.

HBCOLOR [String] (lightgreen)

Selects the help menu Background Color.

HEIGHT [Numeric] (600)

Initial vertical size of the `twander` window in pixels.

HFCOLOR [String] (black)

Selects the help menu Foreground (Text) Color.

HFNAME [String] (Courier)

Selects the help menu Font Name.

HFSZ [Numeric] (10)

Selects the help menu Font Size.

HFWT [String] (italic)

This selects the help menu Font Weight.

HIDEDOTFILES [Boolean] (False)

Sets whether the program hides "dotfiles" by default. (See previous section entitled **Hiding Dotfiles** for the details of this feature.) This value can be toggled with the TOGHIDEDOT (default: Control-9) key binding.

MAXMENU [Numeric] (32)

Maximum number of entries to **display** in any dynamic menu. This keeps the menu size reasonable. Internally, `twander` keeps track of way more than this number of dynamic entiries (see the MAXMENUBUF option below).

MAXMENUBUF [Numeric] (250)

Maximum number of items `twander` **tracks internally** for each dynamic menu. This value need normally not be changed. It is present only to bound how much memory `twander` consumes for this task.

MAXNESTING [Numeric] (32)

Number of times a Command Definition is processed to dereference all variables. For example, suppose you have this:

```
FOO = bax
BAM = x[FOO]

x mycmd [BAM] [SELECTION]
```

When you press the x key, the `twander` command interpreter has to process the line repeatedly until all variables are resolved:

```
[BAM] [SELECTION]   -> x[FOO] [SELECTION]
x[FOO] [SELECTION]  -> xbax [SELECTION]
xbax [SELECTION]    -> xbax selected-item
```

So, in this case, it took 3 iterations to do this. MAXNESTING merely sets the maximum number of times this is permitted. We have to do this to stop runaway definitions like this:

```
FOO = x[FOO]
```

This kind of construct will cause `twander` to iterate MAXNESTING number of times and then give up with a warning about exeeding the nesting (dereferencing) limit.

A 32 iteration limit should be plenty for any reasonable Command Definitions. If you set MAXNESTING to 0, `twander` will not allow **any** variable dereferencing, **including the Built-In Variables.** This is probably not what you want.

MBARCOL [String] (beige)

Selects the Menu Bar color.

MBCOLOR [String] (beige)

Selects the menu Background Color.

MFCOLOR [String] (black)

Selects the menu Foreground (text) Color.

MFNAME [String] (Courier)

Selects the menu Font Name.

MFSZ [Numeric] (12)

Selects the menu Font Size.

MFWT [String] (bold)

Selects the menu Font Weight.

NODETAILS [Boolean] (False)

Prevents details from ever being displayed.

NONAVIGATE [Boolean] (False)

Prevents the user from navigating out of the starting directory. Command Definitions and commands initiated manually via RUNCMD (default: Control-z) can still "see" other directories, the user just cannot move elsewhere with any of the `twander` navigation commands.

The NODETAILS and NONAVIGATE commands are **not** security features. They can easily be defeated by editing the Configuration File. They exist to make it easy for you to create ´twander' configurations for technically unsophisticated users.

Say you want to define a few simple commands for your boss to use which won't challenge his or

her feeble managerial mind ;) By defining these commands and setting both NODETAILS and NONAVIGATE to TRUE, you really limit what can be done with `twander`. They can't wander off into other directories and get lost, or worse yet, clobber files they don't understand. There are no details to confuse them. Your instructions for using the program thus become, "Select the files you're interested in and press P to print them, M to mail them to Headquarters.." and so on.

Again, **these are NOT security features.** Anyone with even very modest technical skills can thwart these limitations. But, it is suprising just how effective these can be in simplifying life for technically challenged users.

QUOTECHAR [String] (")

As described below, `twander` ordinarily quotes most Built-In Variables as it replaces them during command processing. This is useful because modern operating systems allow file and directory names to have spaces in them. Such names must be quoted for most programs to understand them as a single entity on a command line.

By default, the double-quote char is used for this purpose. You can suppress quote processing by using the -t command line argument. This does nothing more than set QUOTECHAR to an empty string. Unfortunately, since the RHS of a Program Option Statement cannot be blank, you cannot disable quoting with this option. However, you **can** set the quotation character to be anything else you like, such as a single-quote. In fact, you can set QUOTECHAR to any **string of characters** you like and they will faithfully be used on either side of a Built-In Variable replacement.

REFRESHINT [Numeric] (5000)

Nominal time in milliseconds between automatic directory refreshes (if AUTOREFRESH is True). This time is **really** nominal and should not be used with any accurate timing in mind. REFRESHINT=8000 says that the refresh interval will be nominally 8 seconds (and certainly more than the default of 5 seconds), but it can be off this nominal value by quite a bit.

If you run `twander` on a slow system (or have a slow link between X-Client and X-Server) you might want to increase this value substantially. You can get into the situtation where just as one refresh completes, its time to do the next one, and the `twander` will seem really sluggish and unresponsive. By lengthening the time between automatic updates, the amount of unresponsive behavior is reduced. Of course, this also means that any changes in the currently viewed directory will also take longer to appear in the `twander` display.

SORTBYFIELD [String] (Name)

Specifies which field is to be used as the sort key. May be one of the fields below under "Sort Key" (case-insensitive). The equivalent field name for Drive List View (see **ADVANCED WINDOWS FEATURES** section below) is shown in the second column, however these may **not** be used as arguments for SORTBYFIELD. For example, if you plan to start the program in Drive List View and want to sort by Drive Type, use: SORTBYFIELD=Links.

Sort Key	Equivalent Drive List View Field
No Sort	No Sort
Permissions	Label/Share String
Links	Drive Type
Owner	Free Space

```
Group           Total Space
Length          Drive Letter
Time            Drive Letter
Name            Drive Letter
```

SORTREVERSE [Boolean] (False)

Specifies whether to reverse the sort order or not. If True and SORTSEPARATE is also True, then the directory list will appear at the **end** of the display in addition to being reverse ordered.

SORTSEPARATE [Boolean] (True)

Determines whether directories and files should be separated or mingled in absolute sort order in the display. By default, they are separated with directories sorted according to SORTBYFIELD order but appearing before any files in the display.

This option is ignored in Drive List View.

STARTDIR [String] (Directory In Which Program Started)

This allows you to force a starting directory of your choice no matter where the program actually is launched. This is useful for day-to-day operation - perhaps you always want to start in your home directory. STARTDIR is also handy in tandem with the NODETAILS and NONAVIGATE options to force a user to the only directory which they should be using.

STARTX [Numeric] (0)

Initial horizontal offset of the `twander` window in pixels.

STARTY [Numeric] (0)

Initial vertical offset of the `twander` window in pixels.

SYMDIR [Boolean] (True)

This option causes symbolic links that point to directories to be treated as directories for purposes of sorting. This is relevant when "separated" sorting is selected - i.e., When the directories are sorted separately from files. If SYMDIR is set to False, then symbolic links will be sorted as files, regardless of what the link points to.

USETHREADS [Boolean] (False)

`twander` defaults to using normal "heavy weight" processes for running commands on Unix. Many Unix implementations also support a "threaded" process model. Setting USETHREADS to True on such systems will cause `twander` to use threads, rather than processes to launch user-defined commands. There are some known issues with thread-based operations (hence the reason this option defaults to False). These are discussed in the **GOTCHAS** section below.

This option applies only to Unix-like operating systems. Windows commands are **always** run as a thread - this is the only process model Windows supports..

USEWIN32ALL [Boolean] (True)

Windows only. If ´win32all´ is installed, determines whether its features should be used (see section below entitled, **ADVANCED WINDOWS FEATURES** for details).

Normally, this option should be left alone. However, if you have ´win32all´ installed on your system for some other reason, but don't want it used by `twander`, set this option to False.

The main reason to do this would be on a slow machine with very large directories. The advanced features of ´win32all´ come at a computational price. This is especially noticeable when it is computing the attributes, ownership, and size in a directory with hundreds (or more) of entries. Typically, you would just use the TOGWIN32ALL key (default: Control-w) to temporarily disable these features before entering such a directory. However, if your starting directory is in this category, setting USEWIN32ALL=False might not be a bad idea.

WARN [Boolean] (True)

Determines whether interactive warnings should be displayed as `twander` encounters them (while parsing a Configuration File or just in normal execution).

Setting this option to False is the same thing as using the -q command line option with one important difference: The Configuration File is parsed before the command line is parsed. Even if you have -q on the command line (or in the TWANDER environment variable), if there is an error in your Configuration File, you will see warning messages at program startup time. Putting WARN=False at the top of your Configuration File will suppress this.

It is not recommended that you operate normally with the -q flag or with WARN=False. `twander` is pretty forgiving in most cases and when it does warn you about something, there is a good reason for it - you probably want to know what the problem is.

WIDTH [Numeric] (800)

Initial horizontal size of the `twander` window in pixels.

WILDNOCASE [Boolean] (True On Win32 / False Elsewhere)

Set's whether or not case is significant in wildcard filtering and selection. If True, case is ignored, if False, case is significant in these wildcard operations.

A few general notes about Program Options are worth mentioning here:

- You can set the same option multiple times in a single Configuration File - `twander` pays no attention. However, only the **last** (the one nearest the end of the file) instance of that Program Option Statement actually takes effect. This is handy if you want to temporarily change something without modifying your existing configuration. Just add your temporary change at the end of the file. When you're done with it, just remove it. No need to edit and re-edit your preferred configuration...

- The font colors, weights, and sizes available for your use will vary somewhat by system. For instance, Windows TrueType fonts are effectively available in every size and weight. On the other hand, most Unix-like systems have a more limited palette of fonts and colors with which to work. Most systems should support obvious color names like, red, white, blue, yellow, beige, and so on. Many also support

colors like lightgreen, lightblue, etc. At a minimum, you should be able to use normal, bold, italic, and underline for font weights.

Most systems attempt some kind of "best fit" font matching. If you specify a font size/weight/name that does not exist, the system will try to find what it thinks is the closest match. This is usually ugly, so try to specify font information for things that actually exist on your system.

If your setting in the Configuration File seems not to work, take a look at the command window in which you started `twander` (or start it from one manually, if you're using a GUI shortcut to start it). Attempts to use unavailable colors and weights will cause Python/Tkinter to dump traceback information on stdout.

- Although you can use proportionally spaced fonts with `twander`, the result is pretty ugly. `twander` assumes a fixed width font when it calculates display formatting. Variable-width fonts will cause your display to be ragged and hard to read.

- If you set MAXMENU or MAXMENUBUF to 0, it disables both dynamic menu content **and** of the last manual entry in the dialogs associated with CHANGEDIR (default: Control-x), FILTERWILD (default: Control-=), SELWILD (default: Control-\), and RUNCMD (default: Control-z).

- Changing MAXMENU and then reloading the Configuration File only changes **the number of items visible on the various dynamic menus.** `twander` actually keeps track of more than this internally (governed by the MAXMENUBUF option).

  Say MAXMENU is set to 4, but you've actually visited 20 different directories and issued 30 commands. You'll only see 4 of each on the associated menus. But, if you edit MAXMENU to now be 32 and reload the Configuration File, you will see all 20 directories and 30 commands on their respective menus.

- At first glance, the ability to set QUOTECHAR to any arbitary string may seem silly, but it actually has a purpose. As good as the `twander` macro capability is, it is still a fairly simple language. Really complex tasks will need to be handed off to some other scripting language (like Python!). It may be useful to delimit Built-In Variables (which indicate your selections via the `twander` interface) in such a way that your script knows where they came from. So, say you set QUOTECHAR=+++ and you have a Command Definition like this:

  ```
  x mycmd MyPythonScript [DSELECTIONS] other stuff
  ```

  When MyPythonScript runs, it can immediately tell which arguments came from `twander` (the ones that are in the form +++dir+++ or +++file+++) and which arguments are just other stuff.

  You probably won't need this often, but its nice to have.

- STARTX and STARTY are relative to the (0,0) origin that Tk uses for window placement. In High-School algebra most of us got used to seeing (0,0) in the lower-left corner of a graph. Tk has a rather different view of this and STARTX and STARTY are relative to the **upper-left corner of the screen.**

Key Binding Statements

No program that runs in many operating environments can satisfy everyone's (anyone's!) idea of what the "correct" key bindings should be. An emacs user, vi user, BSD user, and Windows user are going to differ considerably on what keys should be bound to what feature. `twander` ships from the factory with a set of default key bindings, but it also provides a mechanism for changing these bindings via entries in the

Configuration File.

This feature is available only for **Keyboard Assignments.** Mouse Button Assignments may not be changed by the user. An attempt to do so in the Configuration File will cause `twander` to display a warning and ignore the offending line.

It is not difficult to override the default keyboard bindings by adding entries in the Configuration File. Doing so requires some familiarity with how Tkinter names keystrokes. Good resources for learning this exist abundantly on the Internet, among them:

```
http://www.pythonware.com/library/tkinter/introduction/index.htm
http://www.nmt.edu/tcc/help/pubs/lang.html
http://www.cs.mcgill.ca/~hv/classes/MS/TkinterPres/
```

(As an aside - Tkinter is nothing more than a Python interface to the Tcl/Tk windowing system. The "real" naming conventions for keystokes can be found in the many sources of Tk documentation, both in print and on the Internet.)

Keyboard binding assignments look just like variable definitions in the Configuration File. (The `twander` Configuration File parser automatically distinguishes between Key Binding Statements and Variable Definitions or other legitimate statements. This means you can never use one of the program function names as one of your own variable names.) Key Binding Statements thus take the form:

```
Program Function Name = Tkinter Keystroke Name
```

Changing the default bindings is therefore nothing more than a matter of assigning the appropriate Program Function Name (found in parenthesis next to the description in the default descriptions above) to the desired keystroke.

Examples of all the default key bindings are shown as comments in the ".twander" example Configuration File supplied in the program distribution. The easiest way to rebind a particular function is to copy the relevant line, uncomment the copy, and change the right side of the assignment to the new key you'd like to use.

It is important to observe several rules when rebinding keys:

- It is best if keyboard navigation commands are all Control or Function keys. If you assign a navigation or selection function to a single keystroke, it may conflict with a user-defined command. If you assign it to a keypad/special key it may conflict with that key's normal GUI behavior.

- The Tkinter keynames should placed on the right side of the "=" symbol **without any quotation marks.**

  ```
  # Incorrect
  QUITPROG = '<F3>'

  # Correct
  QUITPROG = <F3>
  ```

- The Program Function Name variables (the left side of the assignment) may not be used as names for your own user-defined variables elsewhere in the Configuration File. In fact, `twander` will never even recognize such an attempt. For example, suppose you try to do this:

```
QUITPROG = something-or-other
```

Because you want to be able to reference [QUITPROG] in a subsequent Command Definition. `twander` will actually interpret this as just another key binding command, in this case binding the program function QUITPROG to "something-or-other" - probably not what you intended. Moreover, if you have a Command String somewhere with [QUITPROG] in it, `twander` will declare and error and abort because it has no User-Defined variable of that name in its symbol table.

- When you're done making changes to the Configuration File, be sure to either restart the program or reload the Configuration File to assign the new bindings.

- Be aware that `twander` does no sanity testing on the assignments you change. If you assign a particular `twander` function to an illegal or silly key string, the program will probably blow-up spectacularly. At the very least, that program feature will probably be unusable, even if `twander` manages to run.

Directory Shortcut Statements

`twander` provides a mechanism for directly navigating into one of 12 frequently used directories. 12 keys, KDIRSC1 ... KDIRSC12 (default: F1 ... F12) have been set aside for this purpose. Directory Shortcut Statements are entries in the Configuration File which associate one of these keys with a particular directory path. These statements are in the form:

```
DIRSCxx = path

   where, xx is a number from 1-12
```

So, for example, if you want to enter "C:\Documents And Settings" when you press the F5 key, you would add this to your Configuration File:

```
DIRSC5 = c:\Documents And Settings
```

There are several subtleties to Directory Shortcuts you should understand:

- You can end the path with slash or not - `twander` will understand the entry either way.

- If there is no path on the righthand side of a Directory Shortcut Statement, this is the same as having no definition at all for that key:

```
# This "undefines" shortcut #5
DIRSC5 =
```

- `twander` does absolutely no checking of what you enter to the right of the equals sign. If you enter something silly for the shortcut path, you will probably get a warning that the directory cannot be opened when you try to run that shortcut.

- Keep the Program Function Names **(KDIRSC1 ... KDIRSC12)** which are used for Key Binding, distinct in your thinking from the Directory Shortcut Names **(DIRSC1 ... DIRSC12)** which are used for defining the shortcuts.

- If you enter a Directory Shortcut Name that is invalid or out of range - examples include, DIRSC01 and DIRSC13 - `twander` treats them like a User-Defined Variable as described below.

Wildcard Statements

As discussed above, twander provides powerful regular expression-based "wildcard" filtering and selection capabilities via the FILTERWILD (default: Control-=) and SELWILD (default: Control-\) commands. These regular expressions can be complex and tedious to enter by hand each time you need them. You can pre-define frequently needed wildcard strings in your Configuration File using the following statements:

```
WILDFILTER = regular-expression-string
WILDSELECT = regular-expression-string
```

The regular expression will then be pre-loaded into either the Filter or Select Menus respectively when twander starts. This makes it easy to use or modify complex wildcards over and over. You may place as many of these as you like in your Configuration File. (Though the menus will be limited to displaying MAXMENU number of items - see the section above on Program Option Statements.)

Variables And Command Definitions

Most programs "ship from the factory" with a pre-defined set of features or commands. twander comes with **no built-in commands!** Instead, it comes with a mechanism which allows you to specify your own **Command Definitions.** By means of a simple and very powerful macro lanuage, you "program" twander and equip it with commands of your own choosing. For example, you might define commands to copy, delete, edit, and move the files or directories you choose. Perhaps you have a specialized shell script for doing backups. It's a simple matter to write a twander Command Definition that will pass the names of the files and directories you've selected to that backup script. You might combine this with twander's Program Memory feature to keep a running list of the files and directories you want to backup and then finally issue the backup command when you're ready. Best of all, commands you define this way are always a single keystroke. This means that once you've programmed twander to suit your needs, actually using it is very fast and convenient.

Command Definitions are built out of literal text and may also have any combination of several variable types: User-Defined Variables, Environment Variables, Execution Variables, and Built-In Variables.

User-Defined Variables are variables you define in the configuration file. They can hold any string of text you desire.

Environment Variables are set in the shell you use to invoke twander. This makes it easy to write a generic command definition that acts based on something set uniquely for each user in that user's environment. You can only read, not change, Environment Variables in twander.

Execution Variables are set by running a program - pretty much any program will do. (Unix users will be familiar with this if they've ever used shell "backtick" quoting.) This makes it easy to construct a twander command that is defined in whole or in part by some external program.

Built-In Variables are a set of variables defined by twander itself. There are two general kinds of Built-Ins. The first kind are used to let your command know (at runtime) which file or files you have currently selected in the twander interface. The other kind of Built-Ins are used to prompt you during command execution. There are also a few other Built-Ins described below.

Just When Does A Variable Get Evaluated?

Before getting into the mechanics of variables and command definitions, it's important to emphasize one point: Variables get "evaluated" (read) **when a command is actually run.** Older versions of twander evaluated variables at the time a configuration file was read. However, as we'll see below, by waiting until the command is actually run to evaluate its variable references, we can do some nifty things.

User-Defined Variables And Environment Variables

User-Defined Variables are defined using the syntax:

```
Variable Name = Replacement String
```

Environment Variables are referenced using the syntax:

```
[$VARIABLE]
```

Say we have a configuration line like this,

```
EDITOR = emacs blah blah blah blah
```

Later on, when defining a command, instead of typing in "emacs blah blah blah blah", you can just refer to the variable [EDITOR] - the brackets indicate you are **referring** to a previously defined variable.

Similarly, suppose you have an environment variable called "EDITOR" which indicates your preferred editing program. Our definition could thus become:

```
EDITOR = [$EDITOR] blah blah blah blah
```

Why bother with this? Because it makes maintaining complex Configuration Files easier. If you look in the example ".twander" Configuration File provided in the program distribution, you will see this is mighty handy when setting up complex "xterm" sessions, for example.

Here are several other subtleties regarding User-Defined Variables:

- `twander` variable definitions are nothing more than a string substitution mechanism. Suppose you have a variable definition that refers to another variable:

  ```
  NewVar = somestring [OldVar]
  ```

  It is important to realize that this only means: "If you encounter the string ´[NewVar]´ **in a subsequent Command Definition,** replace it with the string ´somestring [OldVar]´."

  In other words, no evaluation of the right side of the expression takes place when a variable is **defined.** Evaluation of a variable only takes place when the variable is **referenced** (in the Command String portion of a Command Definition) at the time the command is run. The Command Definition parser will continue to dereference variable names until they are all resolved or it has reached the maximum nesting level (see next bullet).

- User-Defined Variables may be **nested** up to 32 levels deep (this default can be changed via the MAXNESTING Program Option). You can have constructs like:

  ```
  Var1 = Foo
  Var2 = Bar
  FB = [Var1][Var2]
  ```

  Later on (when defining some command) when `twander` runs into the variable reference [FB], it will keep substituting variables until all [...] references have been resolved or it hits the nesting limit (The default is 32, but you can change it with the MAXNESTING option). This limit has to be imposed to catch silly things like this:

```
Var = a[Var]
```

This recursive definition is a no-no and will be cause `twander` to generate an error while parsing the Configuration File and then terminate.

Your variable definitions can also nest other kinds of variables (Environment and Built-Ins). So, constructs like this are perfectly OK:

```
Var1 = [$PAGER]
Var2 = command-arguments
V    = [Var1]  [Var2]  [DSELECTION]
```

- In the example above, notice that since the right-hand side of User-Defined Variables is literally replaced, we have to make sure there is space between the various variable references. If we used [Var1][Var2][DSELECTION] we would get one long string back instead of a command with arguments and a list of selected items.

- Variable references are only significant on the right hand side of an assignment statement:

```
Var1     = Foo
My[Var1] = bar
```

This does **not** create a variable called "MyFoo". It creates a variable called "My[Var1]" and sets its value to "bar". This is both confusing and useless because you can never dereference this variable, because ...

- Variable references cannot be nested. Using our example above, suppose we later want to get the value ("bar") of variable "My[Var1]". That variable reference would look like this: `[My[Var1]]` and this is **not** permitted. A variable reference may only contain a text string, not references to other variables.

- Variables must be **defined before they are referenced** (in a Command Definition). You can, however, include not-yet defined variable name in another User-Variable Definition so long as all these variable are defined by the time they appear in a Command String. The following is OK because all variables are defined by the time they are actually needed:

```
Var1 = foo
Var2 = [Var3]    # This is just a string substitution, not a reference
Var3 = bar
MyVar = [Var1][Var2]

# Now comes the Command Definition
# If we put this before the Variable Definitions above,
# it would be an error.

x mycommand [MyVar]
```

- Variable Names are case-sensitive - [EDITOR], [Editor], and [editor] all refer to different variables.

- The "#" character cannot be used in either the variable name or the replacement string since doing so begins a comment.

- The "=" is what separates the Variable Name from the replacement string. Therefore, the "=" cannot ever be part of a Variable Name. A Variable Name cannot begin with "$" (see next bullet). Other than these minor restrictions, both Variable Names and Replacement Characters can be any string of characters of any length. Good judgment would suggest that Variable Names should be somewhat self-descriptive and of reasonable length - i.e., Much shorter than the replacement string!

- A Variable Name must never begin with "$". This is because a Command Definition containing a string in the form [$something] is understood by `twander` to be a reference to an **Environment Variable,** named "something". If you do this:

```
$MYVAR = some-string
```

  You will never be able to subsequently reference it because, [$MYVAR] tells `twander` to look in the current environment, not its own symbol table to resolve the reference. However, note that "$" symbol may appear anywhere else but the first character of a variable name. So, for example, MY$VAR is fine.

- Unlike previous versions of `twander`, Variable Names may be redefined. This makes it more convenient to exploit the ability for `twander` to process the contents of a Configuration File conditionally (see the **Conditional Processing Statements** section below).

  For example, you can set a variable to some default value, and then override it if a condition is satisfied:

```
# Assume we're running on a Unix-like system

MyEditor = [$EDITOR]

# Override this if we're on Windows

.if [.OS] == nt
        MyEditor = write
.endif
```

Execution Variables

Execution Variables are a special case of User-Defined Variables. However, instead of setting a variable to some string of text, you tell `twander` to **run a program** and set it's results to the variable:

```
TODAY = [`date`]
```

Now, suppose you define a command with `[TODAY]` in it somewhere. When you later run that command, `[TODAY]` will be replaced by the output of the "date" command. In other words, Execution Variables allow you to run any external program you like, and have that program's output substituted into the definition of a command. Several further points are worth noting here.

- Notice that Execution Variables are delimited by backticks, not single-quotes.

- If you have something like `[`program`]` in a Command Definition, it will be replaced with any text that "program" produces as it runs. That text will have any trailing newline stripped. Notice that this is different that most `twander` variables that are evaluated once when the Configuration File is first read in.

This is true wherever the execution variable appears - either as the right-hand-side of a variable statement or explicitly inline within a command definition. You can confirm this by looking at the `User-Defined Variables` and `Command Definitions` help menus. Anything referencing an execution variable will show the command to be executed when the variable is actually referenced at command invocation time.

- Suppose you want to populate an Execution Variable with a program that returns multiple lines of text. You'll need to strip all the newlines out of the output in that case. To do this, you can use a second form of an Execution Variable: `[`-program`]`. The leading minus sign tells `twander` to strip all newlines when doing the replacement. For example, let's define a command that lists all the files in the current directory:

```
a mycommand echo "[`-ls`]"  # We need the double-quotes
                              # to make echo work right
```

Command Definitions

The heart of the `twander` configuration process is creating of one or more **Command Definitions.** These definitions are the way user-defined commands are added to a given instance of `twander`. A Command Definition consists of three fields separated by whitespace:

```
Command-Key  Command-Name  Command-String
```

The **Command Key** is any single character which can be typed on the keyboard. This is the key that will be used to invoke the command from the keyboard. Command Keys are case-sensitive. If "m" is used as a Command Key, "M" will not invoke that command. Command Keys must be unique within a given Configuration File. If `twander` finds multiple Command Definitions assigned to the same Command Key, it will associate the **last** definition it finds with that Command Key. A Command Key can never be "#" which is always understood to be the beginning of a comment.

The **Command Name** is a string of any length containing any characters. This is the name of the command which is used to invoke the command from the Command Menu. Command Names are case-sensitive ("command" and "Command" are different names), but they are not required to be unique within a given Configuration File. That is, two different Command Definitions may have identical Command Names associated with them, though this is not ordinarily recommended.

The **Command String** is any arbitrary string which is what `twander` actually tries to execute when the command is invoked.

A Simple Command Definition

In its simplest form, a Command Definition looks like this:

```
# A simple Command Definition
m  MyMore  more somefile
```

This command can be invoked pressing the "m" key on the keyboard or selecting the "MyMore" entry from the Command Menu - either directly from the menu or from the Command Menu Pop-Up. No matter how it is invoked, `twander` will then execute the command, "more somefile".

The problem is that this command as written actually will not give you the result you'd like (...well, on X-Windows - is does work on Windows as written). (For more details on why, see the **GOTCHAS** section below.) It turns out that starting a non-GUI program like ´more´ in a new window needs some extra work. What we want to do is run ´more´ inside a copy of ´xterm´. Now our command looks like this:

```
# Our command setup to run as a GUI window
m MyMore xterm -l -e more somefile
```

Forcing Display Updates In Command Definitions

You are likely to define commands that change the contents of the currently-viewed directory somehow. For instance, commands that rename, create, or delete files in the current directory all have this effect. When such a command is run, it means that the `twander` display is "out of sync" with the actual disk contents until the next refresh cycle - automatic if AUTORFRESH is enabled, manual otherwise.

Placing ´+´ symbol to the beginning of the Command String tells `twander` that, when the command is run, a display refresh should be forced afterwards. Not immediately afterwards, but AFTERWAIT seconds (default: 1) later. Why? To give the command in question a chance to complete before updating the display. For instance,

```
r removelogs +rm -f *log
```

This means that when the ´r´ key is pressed, the command, "rm -f *.log" is run, and then, AFTERWAIT seconds later, `twander` will force a display update. This happens regardless of the current AUTOREFRESH settings.

This feature is handy, but has some practical limitations. If this feature updates the display before a command actually completes (i.e., the command you've launched takes longer than AFTERWAIT seconds to complete), the final state of the directory will not be displayed. The idea here is to use this feature for "quicky" updates between more conventional display refreshes, whether via AUTOREFRESH or manually.

By default, anytime you run a command that uses this feature, any selections in the GUI are cleared. This is because a forced update presumably is required because the command changes something in the current directory. In that case, the current selections may no longer be relevant. If you wish to disable this behavior, set the AFTERCLEAR program option to False.

User-Defined Variables In A Command String

The last example works quite nicely. But, we're probably going to end up using the string "xterm -l -e" over and over again for any shell commands we'd like to see run in a new window. Why not create a User-Defined Variable for this string so we can simplify its use throughout the whole Configuration File? Now, our command looks like this:

```
# Our command enhanced with a User-Defined Variable.
# Remember that the variable has to be defined *before*
# it is referenced.

XTERM = xterm -l -e                 # This defines the variable
m MyMore [XTERM] more somefile      # And the command then uses it
```

Environment Variables In A Command String

This is all very nice, but we'd really like a command to be generic and be easily used by a variety of users. Not everyone likes the "more" program as a pager. In fact, on Unix-like systems there is an environment variable ($PAGER) set by each user which names the paging program that user prefers. We can refer to environment variables just like any other variable as explained previously. Now our command looks like this:

```
# Our command using both a User-Defined Variable and
```

```
# an Environment Variable to make it more general

XTERM = xterm -l -e
m MyMore [XTERM] [$PAGER] somefile
```

Execution Variables In A Command String

We can further extend the power of Command Definitions by using an Execution Variable to define part of the command. Suppose we want a command that will let us examine all the text files in the current directory:

```
# Our command using User-Defined, Environment, and
# Execution Variables

XTERM = xterm -l -e
m MyMore [XTERM]  [$PAGER]  [`-ls *.txt`]
```

Built-In Variables In A Command String

It would also be really nice if the command applied to more than just a single file called "somefile". The whole point of `twander` is to allow you to use the GUI to select one or more directories and/or files and have your Command Definitions make use of those selections. `twander` uses a set of **Built-In Variables** to communicate the current directory and user selections to the any commands you've defined. Built-In Variables are referenced just like User-Defined Variables and Environment Variables and may be inserted any appropriate place in the Command String. In our example, we probably want the command to pickup whatever item the user has selected via the GUI and examine that item with our paging program. Now our command becomes:

```
# Our command in its most generic form using
# User-Defined, Environment, and Built-In Variables

XTERM = xterm -l -e
m MyMore [XTERM]  [$PAGER]  [DSELECTION]
```

The "DSELECTION" built-in is what communicates the currently selected item from the GUI to your command when the command actually gets run.

Selection-Related Built-Ins

`twander` has a rich set of Built-In Variables for use in your Command Definitions. The first group of these is used to convey your current directory and items which you've selected to a Command Definition:

- **[DIR]**

  [DIR] is replaced with the current directory `twander` is viewing.

- **[DSELECTION]**

  [DSELECTION] is replaced with the full path name of the item currently selected in the GUI. If more than one item is selected, [DSELECTION] refers to the last item in the group (the bottom-most, not the most recent item you selected).

- **[DSELECTIONS]**

  [DSELECTIONS] is replaced with the full path name of **all** items currently selected in the GUI.

- **[SELECTION]**

  [SELECTION] is replaced with the name of the currently selected item in the GUI. The path to that file is **not** included. As with [DSELECTION], if more than one item is selected in the GUI, the name of the last item in the group is returned for this variable.

- **[SELECTIONS]**

  [SELECTIONS] is replaced with the names of **all** items currently selected in the GUI. The path to those names is not included.

Prompting And Special-Purpose Built-Ins

There are also several special-purpose Built-In Variables which are used for creating more powerful Command Definitions.

Note: The PROMPT and YESNO Built-Ins use {} as delimiters, not [].

- **[HASH]**

  Because `twander` always recognizes the "#" as the beginning of a comment, there is no direct way to include this character in a Command String. It is conceivable that some commands (such as ´sed´) need to make use of this character. The [HASH] built-in is provided for this purpose. Anywhere it appears in the Command String, it will be replaced with the "#" at command execution time. Unlike all the other Built-In Variables, [HASH] is never quoted when it is replaced in a Command String (regardless of whether the -t command argument is used or how the QUOTECHAR Program Option is defined).

- **{PROMPT:Prompt-String==>default}**

  {PROMPT:...} allows you to insert an interactive prompt for the user anywhere you'd like in a Command String. The user is prompted with the "Prompt String" and this variable is replaced with their response. If they respond with nothing, it is interpreted as an abort, and the command execution is terminated. This makes commands extremely powerful. For instance, say you want to create a group copy command:

```
# Copy a group of items to a location set by
# the user at runtime
UnixCopy  = cp -R
Win32Copy = copy

# Unix Version
c UnixCP [UnixCopy]  [DSELECTIONS]  {PROMPT:Enter Destination}

# Win32 Version
C Win32CP [Win32Copy]   [DSELECTIONS]  {PROMPT:Enter Destination}
```

  You can also provide a default response to the question. The prompt is separated from the default by the ´==>´ string. This default separator string can be changed to anything you like with the

DEFAULTSEP option.

This feature is useful when you want to provide the user the most-likely response to the prompt:

```
c UnixCP [UnixCopy]  [DSELECTIONS]  {PROMPT:Enter Destination==>/my/home/dir}
```

When the prompt is presented to the user, the default value is pre-loaded into the response field. The user can either accept or edit that string.

- **{YESNO:Question-String==>Yes|No}**

  {YESNO:...} allows you to prompt the user with a dialog containing a Yes/No question and buttons for their response. If the user presses "Yes", command interpretation/execution continues. If the user presses "No", the command is aborted. This is handy when you want to make sure the user really wants to run the command before continuing. For instance, suppose you define a recursive file/directory deletion command. Before running it, it's good to prompt the user to confirm their intentions:

  ```
  D BigDelete {YESNO:Are You Absolutely Sure About This?} rm -rf [SELECTIONS]
  ```

  You can also provide a default response to the question. It must be either "Yes" or "No" (case-insensitive). Anything else will produce an error. The prompt is separated from the default by the ´==>´ string. This default separator string can be changed to anything you like with the DEFAULTSEP option.

  This feature is handy because you can pre-select the most likely response to the dialog:

  ```
  D BigDelete {YESNO:Are You Absolutely Sure About This?==>No} rm -rf [SELECTIC
  ```

Using Variable References Within Prompting Built-Ins

You may have guessed that there is something special about the Prompting Built-In Variables. After all, they use a different delimiter pair than all other variables in the `twander` configuration language. That's because you can include references to other variables within a Prompting Built-In like this:

```
PromptYN  = Are You Sure You Want To Do This?
DefaultYN = No

a mycommand  {YESNO:[PromptYN]==>[DefaultYN]} SomeDangerousCommand
```

A more sophisticated use of this would be when creating a "rename" command. You often want to rename a file by changing only a few of its characters, not the whole file name. Instead of forcing the user to type the whole name in over again, why not just do this:

```
Prompt = New File Name?
r rename mv [SELECTION] {PROMPT:[Prompt]==>[SELECTION]}
```

Now when the user runs the command, the default string will be the name of the file to be renamed. They can move around inside the dialog box created by `{PROMPT: ...}` at runtime to edit the existing file name to taste.

You can also use Execution Variables inside a prompting Built-In:

```
d setdate SomeDateCommand {PROMPT:Set Date To: ===>[`date`]}
```

Program Memory Built-Ins

As described previously, `twander` implements an advanced notion of a Clipboard called "Program Memories". There is a corresponding group of Built-In Variables which allows the contents of these memories to be used in a Command Definition:

- **[MEM1] ... [MEM12]**

  Return the file/directory names currently stored in the indicated memory. For example, to move all the files/directories currently named in the first Program Memory to the current directory we could define a move command like this:

```
m move mv [MEM1] ./
```

Notes On Built-In Variable Use

- Built-In Variables which return a directory name do **NOT** append a path separator character ("/" or "\") to the end of the name even though it is visible in the GUI. This provides maximum flexibility when defining commands. It is up to the command author to insert the appropriate path separator character where needed. (NOTE: Earlier releases of `twander` **did** include the trailing path separator and you may have to edit older Configuration Files accordingly. This change was necessary because certain commands like Unix ´cp´ will not work if given a source directory with the path separator included.)

  For example, another way to express the full path of the currently selected item is:

```
# Unix Path Separator
UPSEP = /

#Win32 Path Separator
WPSEP = \

[DIR][UPSEP][SELECTION]

          - or -

[DIR][WPSEP][SELECTION]
```

  Be aware that, because of `twander` quoting rules, such constructs will result in strings like:

```
"/mydir"/"myfile"

          - or -

"C:\mydir"\"myfile"
```

  This should not generally be a problem with the various Unix shells, and may work for some Windows commands. However, some Windows programs (noted in ´notepad´) reject this kind of filename when passed on the command line. The workaround (and a generally easier way to do this sort of thing), is to use the [DSELECTION] built-in which returns the full path name of an item as a single quoted string.

- All User-Defined, Environment, and Execution Variables are processed each time a command is **run.** This is especially important for Execution Variables. The variable will be "executed" each time the Command Definition in which it is referenced is run.

- Similarly, Built-In Variables are resolved **on each command invocation,** i.e - at command runtime. The Built-Ins will always reflect the current set of files selected in the user interface.

- The results of all built-ins (except HASH) are put inside double-quotes when they are replaced in the Command String. This default is recommended so that any built-in substitutions of, say, filenames with spaces in them, will be properly recognized by your commands. You can suppress the addition of double-quotes by using the -t command line option when starting `twander`.

- Any of the variable types may appear multiple times in the same Command String. For example, suppose you want to define a generic Unix copy command:

```
g gencopy cp -R  {PROMPT:Enter Source}  {PROMPT:Enter Destination}
```

  When the user presses "g" (or clicks on "gencopy" on the Command Menu), they will be presented with two prompts, one after the other, and then the command will run.

Associations

Most X-Windows desktops and Microsoft Windows support the idea of "associations". That is, based on the name of a file, they "associate" an application that can handle it. So, for example, a filename ending in ".txt" is handled by a text editor, a filename ending in ".ps" is handled by a PostScript processing program, and so on. This is handy inside of visual interfaces because you can double-click on a file and the interface can infer which program to load to process that file.

The problem is that the various X desktops and Microsoft Windows don't all handle associations the same way. Some lighter X-Windows desktop may not even have associations at all. In order for to remain portable across operating systems, and work more-or-less the same way everywhere, association support has been implemented directly within `twander` itself.

All you have to do is tell `twander` which program to use for a given file "type". A "type" is defined as a group of files whose names match a so-called "wildcard" (more on that in a moment). You do this by adding association statement to the Configuration File:

```
# Associations are in the form:
#     ASSOC file-type-string command-to-handle-this-type-of-file

ASSOC *.txt emacs [SELECTION]
```

Thereafter, when `twander` runs, the "emacs" command will be loaded to process any file whose name ends in ".txt" when the user selects that file and either double-clicks on it or presses "Enter". On Windows systems, this check is done in a case-insensitive way, so the association above would match files ending in, ".txt", ".TXT", ".Txt", and so on. On Unix-like systems, the check is case-sensitive and the type string must match exactly.

Notice that the "handler command" can consist of pretty much anything that you can use in a command definition (as described in the previous sections). For instance, you can do things like:

```
EDITOR = emacs -fn 10x20

ASSOC *.txt {YESNO:Are You Sure You Want To Edit This File?} [EDITOR] [SELECTION]
```

You can also insert special association command that will be used if no other explicit association matches. Think of this as a "default" association:

```
ASSOC *.pdf   mypdfreader          [SELECTION]
ASSOC *.ps    mypostscriptprogram  [SELECTION]
ASSOC  *      myfineeditor         [SELECTION]   # Default association
```

In this example, if you double-click or press "Enter" on any file not ending in either ".pdf" or ".ps", the default association action will be taken: The file will be opened with `myfineeditor`.

You can also define a list of file types to be **excluded** from association processing. This is handy if you want to use the default association feature for everything except a particular set of file types. This feature is primarily useful on Windows systems where you want to define your own default action, but want a few particular types of files to use the underlying Windows associations.

To do this, put one or more statements in the following form in your Configuration File:

```
ASSOC ! space-separated-list-of-file-types
```

For example:

```
ASSOC  *     myfineeditor [SELECTION]
ASSOC  !     *.txt *.pyo *.ps
```

With this configuration, all files would, by default, be handled with `myfineeditor` **except** files whose names end with `.txt`, `.pyo`, or `.ps`. These excluded file types would be handed to the underlying OS for processing when they are selected.

Note that exclusion has higher precedence than any explicit association, not just the default association. If you do this:

```
ASSOC *.pdf   mypdfreader [SELECTION]
ASSOC !      .pdf
```

You are effectively masking the explicit association for `.pdf` files.

You can also remove a previously defined association by leaving the right-hand-side of the `ASSOC` statement blank:

```
# This example first defines, and then removes an association
# for .pdf files:

ASSOC *.pdf   mypdfreader [SELECTION]
ASSOC *.pdf

# This one removes any exclusions you might have previously defined

ASSOC !
```

```
# This one removes any default association you might have previously defined

ASSOC *
```

This feature is primarily useful when you want to define associations conditionally. That is, you can remove an association if a particular conditional block is true. A typical use for this might be to get different (or remove) associations based on what OS you're running. (See the section below entitled: **Conditional Processing Statements**).

Association Wildcards

Associations are built around the idea of a file "type". You want files of the same type handled by the same application program. On Windows systems, this has traditionally been the set of characters the follow the period at the end of the filename. But this convention is not consistently used on Unix-like systems. `twander` lets you use a fairly powerful "wildcarding" system to define what is common about the names of all files of a given type. Unix users will recognize this as the shell `globbing` wildcards. Here they are implemented for both Windows and the Unix-like systems in the same way. The only difference is that, on Windows, the check for a match ("is this file of type ...?") is done without regard to case, whereas on the Unix-like systems, case matters.

If you are unfamiliar with Unix-style shell globbing, many references can easily be found on the web. Here is a summary of the "meta" characters supported:

```
*         Matches everything - strings of any length with any characters

             Example:  *.text matches all filenames ending in ".text"

?         Matches a single character

             Example:  foo.??? matches all filenames beginning with "foo."
                       and ending with any three characters.

[list]    Matches any characters in the list

             Example:  foo*[tT] matches all filenames beginning with "foo",
                       with any number of characters following, and
                       ending with either the letters "t" or "T".

[!list]   Matches any character NOT in the list

             Example:  foo*[!tT] matches all filenames beginning with "foo",
                       with any number of characters following, and NOT
                       ending with either the letters "t" or "T".

Lists can also be ranges.  For example:

              [a-z]        Matches any lowercase letter
              [A-Z]        Matches any uppercase letter
              [0-9]        Matches any numeric digit
```

So, why bother with this? Because sometimes you want associate an action with a set of files whose names are similar but vary in some known way. For instance, suppose you have a database program that produces

files named "data01, data02, data03, ..." and so on. Instead of having to write a separate association for each different possible filename, you can just do this:

```
ASSOC   data??  MyFineDatabaseProgram [SELECTION]
```

A Few Association Subtleties

- The `ASSOC` keyword itself is case-sensitive - you must enter it entirely in upper-case. However, the order of `ASSOC` statements is unimportant. `twander` distinguishes `ASSOC` statements by their unique file "type" strings. Well ... this is true so long as you make sure the type strings **are** unique! Suppose you put this in your Configuration File:

  ```
  ASSOC *.text emacs [SELECTION]
  ASSOC   *xt  ci [SELECTION]
  ```

  A file whose name ends in ".text" will match **both** of these associations. So which one does `twander` use? There is no way to tell - it's a nonsense condition when a given file type matches more than one association. The moral of the story here is to make sure each of your `ASSOC` statements associates a completely unique file type. Be really careful about this when using complex wildcards to specify the file "type". It's easy for wildcards to overlap in their definition of a filename and you'll end up with more than a single, unique association for a given type of file.

- The "default" association - if defined - will only be applied if no explicit association for a given selection is found and the file in question is not **executable.** That way, you can still double-click (or press "Enter") on executable files to run them without the default association getting in the way. Of course, if you've explicitly defined an association for the type of executable file selected, then that **will** be applied to the selected file. This can be handy if, say, you don't want to actually run the executable, but just edit it when you double-click on it.

- Be careful which Built In Variables you use in an association handler definition. Suppose you do this:

  ```
  ASSOC *.txt emacs [SELECTIONS]
  ```

  Note the use of the multiple selection Built In Variable, `[SELECTIONS]` (as opposed to the single selection `[SELECTION]` used in the previous examples). What happens when you double-click or press "Enter" when multiple files have been selected in the `twander` interface? Well, it depends. The program decides which association to use based on the **last** filename you have selected. Suppose, in order, you select "foo.c", "bar.py", and "baz.txt". Since the last file selected ends with ".txt", the handler defined above will match and **all** the files will be processed using this association. This may not be what you want.

  Even stranger things can happen if the last filename selected is an executable and you've defined an association for it:

  ```
  ASSOC *.py python [SELECTIONS]
  ```

  Say you select "foo.c" and "bar.py". Since "bar.py" is the last file in the multiple selection, it will match the association above. This will effectively produce a command that like this:

```
python foo.c bar.py
```

Which is, um ... bogus.

As a general matter, multiple selection Built Ins are fine if you are specifying an association handler that does things like editing, viewing, printing, and so on. But be wary of them if the handler runs some language processor or other program that expects the content of its arguments to be in a particular format.

Associations Differences Across Platforms

For the most part, `twander` associations work pretty much the same way on all systems . There are, however, some slight differences between the Unix-like systems and Windows.

- On Unix-like operating systems `twander` ignores the underlying associations (if any) of the system and/or X desktop. It only observes its own associations. That's because there is no consistent association mechanism across the many OS and desktop variants in use on those platforms.

  But Microsoft Windows is a different matter. All modern variants of these systems have consistent built-in support for association. `twander` was designed to "play nice" with the underlying associations defined in the Windows registry. It works very simply: An association defined in `twander` will take precedence over the native Windows association. Say you define this:

  ```
  ASSOC .txt myowneditor [SELECTION}
  ```

  Then double-clicking "foo.txt" will cause "myowneditor" to be invoked, even though Windows, by default, associates "notepad" with text files.

  If no matching `twander` association is found for a particular selection, then the execution request is handed to Windows, and it's association for that file type (if any) will be applied.

  This is a very handy feature. You may wish to temporarily or permanently change which program is associated with a given file type when you're running `twander`. Instead of having to fiddle around with reassociating things in Windows, you can just edit the `twander` Configuration File.

- As noted in the previous section, matching the file "type" is case-sensitive on Unix-like systems and case-insensitive under Windows. This is because, although Windows observes case in file names, they are not signficant. Similarly, file types listed in the association exclusion list are treated in a case-insensitive way on Windows systems.

- On Unix-like systems, if you attempt double-click or press Enter on a file that cannot be executed, `twander` will display an error. This happens anytime the file is non-executable and you've not defined an association that matches it.

  On Windows, `twander` does **not** do this. It simply passes the request on to Windows. It does this in case there is a native Windows association that applies for that file. If Windows cannot execute the file, it will present an error message of its own.

Conditional Processing Statements

Most of `twanders` power lies in its ability to be customized to each different user and operating system via its Configuration File. To make this a bit easier to manage, the `twander` configuration language

recognizes so-called "Conditional Processing Statements". These statements give you the ability to write a single Configuration File which automatically tailors itself to run `twander` properly wherever you are running.

The general idea is to define a "Condition Block" which begins by doing a logical test. If that test evaluates to True, all statements in the block are included in the current configuration. If the test evaluates to False, all statements to the end of the block are ignored.

A Conditional Block always begins with a "Condition Test Statement" and ends with the ".endif" statement. Conditional Processing Statements may be nested without limit. `twander` keeps track of which '.endif' matches which Condition Test Statement. Like all Configuration File entries, whitespace is ignored when processing Conditional Statements and you are free to indent (or not) as you see fit.

Condition Test Statements are one of three types:

```
#####
# Existential: True if FOO or $FOO are defined
#####

 .if [FOO]
   ...
.endif

 .if [$FOO]
   ...
.endif

#####
# Equality: True if FOO or $FOO are literally
# the same as the test-string
#####

.if [FOO] == test-string
        ...
.endif

.if [$FOO] == test-string
        ...
.endif

#####
# Inequality: True if FOO or $FOO are literally
# not the same as the test-string
#####

.if [FOO] != test-string
        ...
.endif

.if [$FOO] != test-string
        ...
.endif
```

To make it easy to create conditional blocks based on the type of system you're running, `twander`

automatically pre-defines two variables which provide information about your system: **.OS** (typically: nt, posix) and **.PLATFORM** (typically: freebsd4, linux-i386, win32). You should run `twander` and examine the "User-Defined Variables" section of the Help Menu to see how these variables are set on your system.

These predefined variables show up as "User Defined Variables" in the various `twander` Help and Debug outputs, but they begin with a period to remind you of their intended role. They will thus also sort first in the User-Defined Variables section of the Help Menu.

Several things about Conditional Processing Statements are worth noting:

- Whitespace is mandatory after the ".if" statement - .if[FOO] is syntactically incorrect. However, you need no whitespace on either side of a "==" or "!=" test.

- All these tests involve either a User-Defined Variable or an Environment Variable, never a Program Option or Built In Variable.

- A Condition Test Statement always involves a variable **reference** ("[FOO]", never just "FOO") because we want the **contents,** not the name, of the variable for the test.

- The Right Hand Side of an (in)equality test is just a string comparsion - no variable expansion is done:

  ```
  .if [FOO] == string[BAR]
  ```

  This will not work as you might expect because the contents of variable FOO are literally compared to the string, "string[BAR]". Note too that this comparison is case-sensitive.

- The ".endif" statement must appear on the line by itself. Nothing other than whitespace may precede it, and nothing (other than whitespace or a comment) may follow it.

See the example ".twander" file provided in the distribution for some extended examples of using conditionals in your Configuration File. Also see the **GOTCHAS** section below for further discussion.

The Include Directive

You may include other files in your Configuration File with the following directive:

```
.include path-to-file
```

You may place as many of these statements in your Configuration File as you wish. The only syntactic requirement is that there must be whitespace between the directive and the file path. `twander` makes no attempt to validate that path, and you will see an warning message if the file you specify cannot be opened.

The most common reason to do this is to maintain a "standard" configuration in a separate file which is controlled by the system administrator. This is especially handy on larger systems with multiple users. The system administrator provides a read-only copy of the standard configuration in a place anyone can read it. Everyone is free to use (but not modify) that standard configuration. You are then free to add to, or even override the standard configuration content with statements of your own following the ".include". Suppose you have the following "standard" Configuration File available on your system:

```
# Contents of /usr/local/etc/.twander.global

SHELL   = bash -c
XTERM   = xterm -fn 9x15 -l
```

```
VSHELL  = [XTERM] -e [SHELL]

DIRSC1  = /usr/local
DIRSC2  = /usr/sbin

t terminal   [XTERM]
```

Now, you can create your own personal Configuration File which takes advantage of this standard file, but augments it with additional configuration information of your choosing:

```
# Contents of $HOME/.twander

.include /usr/local/etc/.twander.global

DIRSC2  = /etc

l ls       [VSHELL] 'ls -al | [$PAGER]'
```

Keep in mind that `twander` reads the contents of its Configuration File **in order.** In this case, it means that all of "/usr/local/etc/.twander.global" is read and **then** the rest of "$HOME/.twander" is read. If something is defined more than once, the **last definition** is what is used. In this case, DIRSC2 is overriden in the local Configuration File and is ultimately assigned to "/etc". Similarly, you can override previous definitions for User-Defined Variables and even Command Definitions.

The program checks to see if you attempt to do a "circular" include. For example, say file "A" `.includes` file "B" and file "B" then `.includes` file "A". This wold create a neverending "circle" of included files. If `twander` detects this, it will display an error describing the problem and skip the offending line.

ADVANCED WINDOWS FEATURES

As shipped from the factory, `twander` runs pretty much identically on various Unix variants (FreeBSD, Linux) and Windows. However, `twander` is written to take advantage of Mark Hammond's ´win32all´ Python extensions if they are present on the system. These extensions add many Windows-specific features to Python and allow `twander` to provide quite a bit more Windows-centric information about files, directories, and drives. You do **not** have to install ´win32all´ for `twander` to operate properly on your Windows system. Installing this package just means you'll get even more `twander` features on Windows than you would otherwise. If you've installed ´win32all´, you can toggle these features on- and off with the TOGWIN32ALL key described above.

Getting ´win32all´

You can get the ´win32all´ extensions one of two ways. If you've installed the Active State version of Python for Windows, (http://www.activestate.com/Products/ActivePython/) ´win32all´ is already installed on your system. If you installed the standard Python release for Windows (http://www.python.org/download/download_windows.html), you must add ´win32all´ to your installation. You'll find the extensions and painless installation instructions at: http://starship.python.net/crew/mhammond/

New Features Supported With ´win32all´

One important note is in order here: The features enabled by ´win32all´ are only available on "true" Windows systems like Windows 2000 and Windows XP. Earlier versions of Windows like Win98 and WinME emulate portions of the Win32 API and do not implement the advanced security features found in the NTFS

file system. Therefore, as noted below, some of these features will not work on any of the older 16-bit Windows operating systems. `twander` handles this gracefully without blowing-up so you can safely have ´win32all´ installed on one of these older systems to take advantage of the features that do work.

Once you have these extensions installed, `twander` will automatically enable three new features otherwise unavailable.

- When viewing file/directory detail information, the owner and group names will be the actual names reported by the operating system rather than the filler values normally seen in those fields (´win32owner´ and ´win32group´). (Does not work on older Windows systems like Win98.)

- Instead of showing Unix-style file permissions (which don't mean much under Windows), systems with ´win32all´ installed will show the so-called "file attributes" maintained by the operating system. Each detailed entry in the display will have one or more of the following attributes displayed in what is normally the Unix permissions field:

```
d - Directory
A - Archive
C - Compressed
H - Hidden
N - Normal
R - Read-Only
S - System
```

- A top-level "Drive List View" is enabled if ´win32all´ is installed. This shows you a list of all currently available drives reachable by the system, and information about those drives. For locally attached drives, the drive label is shown. For network-attached drives, the share string is shown. The drive type (CD/DVD, Fixed, Ramdisk, Remote, Removable) is shown as are the free, and total space statistics. As is the case with other `twander` displays, these details can be toggled on- and off via the TOGDETAIL key.

  You can enter the Drive List View in a number of ways:

  1) Select the ".." from the root directory of any drive.
  2) Enter the string "\\" from the CHANGDIR dialog.
  3) Press the DRIVELIST key. (default: Control-k)
  4) Start `twander` using "\\" as the starting directory argument, either on the command line or using the Configuration File STARTDIR option.

  The "Drive List View" is available on all Windows variants, however the free/total space values will be incorrect on older systems like Win98.

Notes On Drive List View

The Drive List View is a little different than the usual file/directory view. Program behavior (semantics) is thus also slightly different than usual in several ways:

- While in Drive List View, the various Built-In Variables which return the current selections will return **the name or names of the selected drive(s)** (without a trailing slash) just as you would expect them to in a normal file/directory view. This allows you to write commands which take drive names (letters) as an argument. The [DIR] Built-In returns an empty string in this view.

- Normally, as you navigate around a file system, `twander` sets its own program context to the current directory. This is why you can write Command Definitions using only the file/directory name currently selected - `twander` knows the current directory. When you are in Drive List View, the notion of "current directory" has no real meaning. So, `twander` treats the directory from which you entered Drive List View as the "current directory" while in that view.

- By default, `twander` automatically rereads the current view about every 3 seconds. This is fine for a file/directory view but would be annoyingly slow in the Drive List View since it takes a moment or two to get the status of any floppy disk drives attached to the system. Instead of forcing the user to listen to (and wait for) the floppy drive status to be determined every 3 seconds, `twander` **only reads the drive information once when it enters Drive List View.** This means if a drive is connected or a floppy is inserted into the system while in Drive List View, this fact will not be automatically noted. You can force a manual update of the Drive List View by pressing the REFRESH key. (default: Control-l)

- The TOGWIN32ALL key (default: Control-w) is disabled in Drive List View. Drive List View is only available in ´win32all´ mode and toggling that mode off makes no sense here.

- The SELALL (default: Control-comma) and SELINV (default: Control-i) features work slightly differently in Drive List View than they do otherwise. Ordinarily, these features never select the first item of a file/directory display because it is always the ".." entry pointing to the directory parent. In Drive List View, the first entry **is** an entry of interest - usually, but not always, Drive A: - so these two keys **do** select it as is appropriate.

Disabling ´win32all´ Features

You can toggle these features on-and off using the TOGWIN32ALL key. (default: Control-w) You can also permanently disable them by setting the USEWIN32ALL option to False in the Configuration File. This allows you to leave ´win32all´ installed on your system if you need it for other reasons but don't want these features enabled in `twander`

GOTCHAS

There are several tricky corners of `twander` which need further explanation:

Program Starts Very Slowly

`twander` attempts to determine the name of the host on which it is running at program startup. This is used in the title bar display. It first looks to see if the environment variable HOSTNAME is set, and uses that value if it is. If this variable is not set, `twander` does a socket call to see if it can determine the hostname that way.

Either of these methods works fine, but the socket call can be very slow if the network is misconfigured or malfunctioning. If `twander` is starting very slowly, try setting HOSTNAME explicitly in your environment - this will prevent the socket call from ever taking place. A simple way to do this with ´ksh´ or ´bash´ is:

```
export HOSTNAME=`hostname`
```

(Note the backticks used to execute the ´hostname´ program and assign its results to HOSTNAME.)

Be aware that ´bash´ claims to automatically set this variable when it starts. However, it does not appear to export it properly on some systems (noted on FreeBSD 4.7 with ´bash´ 2.05b). In this case, you have to do this manually as just described even when using ´bash´

On Windows, environment variables are set via the System Properties menu.

Program Loads Slowly

`twander` is a fairly large Python program and can take a few seconds to load and initialize, especially on older, slower systems. You can speed this up a bit by creating an optimized byte-code version of the program as follows (make sure you have appropriate administrative permission to do this):

1) Go to the directory where the twander.py file is located.
2) Type the following command: python -O
3) Once Python is loaded type: import twander
4) Exit `twander`.
5) Exit Python by pressing Control-d on Unix or
   Control-z on Windows.
6) You will now see a new file in this directory: twander.pyo
   This file should be significantly smaller than twander.py.
7) Now you can run the program by entering: python twander.pyo
   on Unix/Windows or pythonw twander.pyo on Windows.
8) You have to repeat this procedure each time you install
   a new version of twander.py

Cannot Enter Certain Directories On Windows

Windows allows file/directory names to contain non-ASCII characters. Python, as shipped, defaults to ASCII only and grumbles mightily when it is asked to deal with a string containing characters with ordinal values greater than 127 (i.e., 8-Bit "extended" ASCII). The solution to this problem is to enable Python to handle non-ASCII strings. This is done by editing a file called "site.py". This file is normally found in:

```
C:\Program Files\PythonXX\Lib
```

Where "XX" is the actual version of Python you're running.

Open this file with an editor and look for the following text:

```
encoding = "ascii" # Default value set by _PyUnicode_Init()

if 0:
    # Enable to support locale aware default string encodings.
    import locale
```

Change the **if 0:** statement to **if 1:** and the problem will disappear.

Getting Command Results Displayed In A New Window

When you invoke a command via `twander` (whether via a command definition in the Configuration File or the keyboard shortcut), you generally want it to run in a new window. This turns out to be tricky on Unix-like systems. If the program you are running is GUI-aware, this should not be a problem. However, if you are using `twander` to run a command line program or script, you have to take extra care in the formulation of the Command String. In the case of Unix-like systems you have to invoke the command so that it runs in some GUI context. Say you want to use a pager like ´less´ to view files. You would expect that this entry might do it:

```
V   view    less    [DSELECTIONS]
```

Sadly, this will not work, at least not the way you expect. If you started `twander` from a terminal session and use the command above, it will work, but the results will appear in the invoking terminal window, **not** in a new window as you might expect. If you started `twander` from a GUI or disconnected it from the initiating terminal with a ´nohup´ ... & invocation, you will get **no** output. This is not a `twander` problem, it is innate to how command line programs run under Unix shell control.

To achieve the desired results, you have to create a new GUI window in which your command can run and display results. The easiest way to do this is to run your command in a new ´xterm´ window like this:

```
V   view          xterm -l -e less [DSELECTIONS]
```

Some program further require you to provide a shell so they can execute correctly. For instance, running ´ls´ in a command definition requires something like this:

```
L lshome          xterm -l -e bash -c 'ls / | [$PAGER]'
```

In fact, this idiom is so common, you will see variables defined in the example ".twander" file to simplify such definitions (comments removed):

```
SHELL       = bash -c
VSHELL      = [XTERM] [SHELL]
XTERM       = xterm -fn 9x15 -l -e
```

Now you can write the command above like this:

```
L lshome           [VSHELL] 'ls / | [$PAGER]'
```

This causes your command line program to execute in an ´xterm´ context **and** under a shell interpreter.

This is not as much an issue on Windows systems where the first form of the command above works fine. Windows appears to have no problem invoking a new window whether the command is GUI-aware or not.

However, **which** terminal window is used for output can be confusing. If you start `twander` from a terminal session, all terminal output will be sent to **the terminal session you used to invoke the program.** The way to work around this is to start `twander` from a Windows shortcut, using ´pythonw.exe´ rather than ´python.exe´. Now each time you run a command that needs a terminal session for output, Windows will automatically create that session for you.

Using Shell Wildcards In Command Definitions

The {PROMPT:...} Built-In Variable is provided to make it possible to write general-purpose commands which interact with the user. For example, you might want to define a directory listing command for Windows like this:

```
L DirList dir {PROMPT:Directory Of What?} | more
```

When the user presses the "L", they are presented with a dialog box into which they enter their directory name or wildcard pattern such as "*.bat" and everything works as expected.

On Unix-like systems, however, this does not work as expected. Suppose we define the command for these systems to be:

```
L DirList [VSHELL] 'ls -l {PROMPT:Directory Of What?} | [$PAGER]'
```

This works fine **so long as the user does not enter a wildcard pattern** in response to the prompt. Why? Recall that `twander` quotes all Built-In Variable substitutions by default. If the user enters this at the prompt:

```
/kern*
```

The command `twander` tries to execute is:

```
VSHELL Stuff ... 'ls -l "/kern*" | ... pager stuff
```

The argument to ´ls´ is double-quoted. The Unix shells understands this kind of quoting to mean **no expansion of wildcard characters is to be done,** which is the exact opposite of what we want.

You might think that the easy way to solve this problem is to turn off argument quoting with the -t command line flag. However, this is not really practical. Quoting is on or off globally in the program. Turning it off means no Built-In Variable substitutions will be quoted. That's fine so long as no directory/file you select via the user interface has a space in the name. However, you are almost certain to run into such files sooner or later. (Recall that the only way to deal a directory/filename with spaces in it as a single argument is to quote that name.)

So, we need a way to leave quoting on but also properly deal with wildcard string entries from the user. Fortunately, because of the richness of Unix shells, there is a simple way to do this: we'll use a "shell variable" to hold the user's response and the shell's ability to handle multiple commands on one line separated by semi-colons:

```
# Note that the line below is split for printing purposes
# In an actual Configuration File, this needs to all be on one line

L DirList [VSHELL] 'UsrResp={PROMPT:Directory Of What?} ;
                            ls -l $UsrResp | [$PAGER]'
```

Why does this work? Because the shell interprets (and drops) the double-quotes, when the results of the {PROMPT:...} are **assigned to UsrResp .** The later reference to "$UsrResp" returns just the string the user entered without the quotes and the command works as expected.

Interestingly, this problem does not occur when entering text via the RUNCMD dialog. (default: Control-z) Here the text you enter is **not** part of a Built-In Variable substitution, so it is not quoted. (The exception, of course, would be if you entered a {PROMPT:...} reference in the RUNCMD dialog. In this case, the same problem we've just described could occur.)

Modal Operation Of New Windows

Notice our example commands above do not end with "&". These should not be needed on either Unix-like or Windows operating systems. When a command is executed, `twander` starts a process which runs concurrently with `twander` itself. This means you should be able to continue using `twander` while the new command executes.

If you enable the use of threads by setting USETHREADS to True, you may see `twander` locked out while the new command runs - so-called "modal" operation. If this happens, it means your system does not completely or correctly implement threading and you must use conventional "heavy weight" processes (the default) rather than threads.

Windows Don't Disappear On Command Completion

It appears that some X Windows implementations (noted on XFree86 / FreeBSD) do not correctly destroy an ´xterm´ window after a command initiated with -e terminates. This is not a `twander` problem - it is an artifact of thread behavior on such systems and only happens if you set USETHREADS=True. The workaround is to use the default USETHREADS=False setting.

Program Behavior Incorrect When A Window Is Resized

Certain Unix programs such as ´less´ appear to not work correctly when the window in which they are running is resized. The program seems to not be properly informed that the window size has changed. This seems to be an interaction caused by running such programs as threads rather than processes. Once again, the workaround here is to not change the USETHREADS=False default setting.

Really Slow Response Times When Changing To A New Directory

You may occasionally see **really** slow response times when you change to a new directory. This occurs when you enter a huge directory with thousands of file or subdirectory entries. `twander` has to to compute the detail information for each of these entries and this can take a lot of time. On a fast machine with modern hard drives and controllers, `twander` is able to process several thousand entries in just a second or two. However, a number of factors can significantly slow down this process:

- The Autorefresh interval is set too low. Processing the directory takes so long that as soon as one refresh finishes, the next starts right away. The program will appear to hang. There are two possibilities here. Either disable autorefreshing (via the -r command line option or the AUTOREFRESH Configuration File option), or set the REFRESHINT value to some high number so that `twander` has plenty of time to process a directory before the next refresh occurs.

- Slow disk drives. You can really watch `twander` grind if you change to a large directory on a CDROM, for instance. There is no good solution here. These drives are inherently slower than hard drives, and you just have to wait. Make sure you lengthen your refresh interval as described in the previous bullet.

- By far the worst culprit here, though, is when running Windows with ´win32all´ options enabled. It takes a lot more work to get win32all-style information about each directory entry, than the default Unix-style information. Simply turning off ´win32all´ features alone can speed up directory processing by a factor as high as 4X.

When you combine these factors, it is possible to get really long processing times. One test situation we observed was reading a directory with over 4000 entries on a Windows CDROM. With ´win32all´ processing enabled this took over a minute. By disabling these features, the time came down to under 30 seconds.

- For all these reasons, `twander` implements an "adaptive refresh" scheme by default. Whenever a directory is read, the time to do so is tracked. If that time is less than the current value of REFRESHINT - i.e., The directory read took less than REFRESHINT milliseconds to complete - nothing special happens. But, if the actual directory read time takes **longer** than REFRESHINT milliseconds, `twander` adjusts the value of REFRESHINT upwards. That way, you're guaranteed to have time after the read completes to actually do something.

This dynamic adjustment takes place on every directory read. If you go to a slow directory and REFRESHINT gets dynamically adjusted to, say, 25 seconds, when you go back to a faster/smaller directory, REFRESHINT will be reset to its default value. The changing value of REFRESHINT is **not** shown in the program options help menu. The value there is the one set by default or set in the Configuration File. Think of this as the "base" value for REFRESHINT.

If you don't like this adaptive refresh interval business, set the ADAPTREFRESH program option to False. In that case, REFRESHINT will be strictly observed.

Your Configuration File Does Not Produce The Desired Results

It's easy to fall into the trap of treating the `twander` configuration capabilities as a real "programming language". It is not, it is a fairly simple macro language that does very little more than string substitutions. Keep the following rules in mind as you edit your configuration:

- Except for conditional tests, Environment Variables and User-Defined Variables are never resolved **until they appear in a Command Definition.**
- The Right Hand Side of Option Statements, Key Binding Statements, Directory Shortcut Statements, Wildcard Statements, and Condition Test Statements **are treated literally -** No variable substitution is ever done there.
- A Condition Test Statement always involves a variable **reference,** never just a variable name.
- For a Condition Test Statement to be true, the referenced variable **must be defined** and any equality test must be satisfied.
- When testing for the existence of a User-Defined or Environment Variable, `twander` does not care what **value** the variable contains. It is perfectly permissible to have either type of variable set to an empty string. The fact that the variable exists at all is what makes the following construct true:

```
CondVar =
.if [CondVar]
  ....
.endif
```

- You have to be careful when overriding variable or command definitions. User-Defined Variables referenced in a Command Definition are de-referenced **at the time the Command Definition is encountered in the Configuration File.** This means that if you change a User-Defined Variable after it has already been used in a Command Definition, only future references to that variable will reflect the change:

```
FOO = bar

x cmd1 command [FOO]

FOO = baz

y cmd2 command [FOO]
```

In this example, the first command will be defined as "command bar", but the second will be defined as "command baz".

Watch for this, especially, when using the ".include" directive and then overriding a variable defined in that file.

Common mistakes include:

```
#####
# Trying to embed a variable where it will never be resolved
#####

DIRSC03 = [$SystemDrive]\Program Files
MYCOLOR = blue
FCOLOR  = [MYCOLOR]

#####
# Expecting a conditional variable to be resolved before the test
# Suppose $EDITOR is set to "/usr/local/bin/emacs" ...
#
# The following will be False because [EDT] equals
# the string "[$EDITOR]".  It is not replaced
# with "/usr/local/bin/emacs" until [EDT] appears
# in a Command Definition
#####

EDT = [$EDITOR]

.if [EDT] == /usr/local/bin/emacs
          ...
.endif


# Note, however, that *this* would work because
# Environment Variables are permitted in conditionals ...

.if [$EDITOR] == /usr/local/bin/emacs
          ...
.endif

#####
# A badly formed condition is ignored (after a warning)
# which means *all the lines following will be processed*
# (until a valid condition statement which is False is
# encountered).
#####

PROCESS = no
SUBPART = no

# We meant not to process the following but all the
# lines up to the next .if statement *are* processed
# because the bad syntax on the next line means it's ignored

.if PROCESS != no
    ...           # Processed!

    .if [SUBPART] == yes   # *Now* we'll stop
        ...
    .endif
```

```
.endif
```

OTHER

File/Directory name sorting is done without-case sensitivity on Windows systems because the underlying operating system does not observe case.

Because this program has not been tested on anything other than Unix-like and Windows systems, command execution by double-click or pressing Enter is inhibited on all other operating systems by default.

You must have Python 2.2 or later installed as well as Tkinter support installed for that release. In the case of Windows, Tkinter is bundled with the standard Windows Python distribution. In the case of Unix-like systems, you may have to first install Python and then the appropriate release of Tkinter. This is the case, for example, with FreeBSD.

You must install the ´win32all´ extensions if you want to use the advanced Windows features.

You'll find the latest version and, occasionally, Release Candidates of the next version of `twander` at:

```
http://www.tundraware.com/Software/twander
```

You should check this site regularly for updates and bug-fixes. The ´WHATSNEW.txt´ file describes changes since the last public release of the program.

BUGS AND MISFEATURES

As of this release, a number of problems relating to `twander` use have been noted:

- The Configuration File parser does no validation to check the sanity of its various entries for Program Options, Key Bindings, Directory Shortcuts, Variable Definitions, and Command Definitions. It is entirely possible to edit something into this file that makes no sense at all and causes `twander` to misbehave.

- There appears to be a Tkinter/Tk bug on Unix which sometimes inhibits the correct title display when you tear-off a menu. This is a cosmetic defect and may disappear in future releases of Tkinter/Tk/X-Windows.

- Some ´win32all´ features do not work correctly or at all on older Windows OSs. For example, the free/total space available in the Drive List View has been noted to display incorrect values on Win98. Similarly, the owner and group names are displayed as "Unavailable" on pre-NTFS file systems. These are OS limitations which `twander` handles gracefully.

- If you are using ´bash´ as your Unix shell, be aware that, although it sets HOSTNAME automatically, this environment variable appears to not be exported consistently on all systems.

- If you are running Windows and have file or directory names with non-ASCII characters in them, you must configure Python to properly deal with such characters. This is described above in the section entitled, **GOTCHAS.**

- This program has not been tested on MacOS. It has been reported that Python on MacOS X returns ´posix´ as its OS name. If true, `twander` should work as written, though we've not verified this.

Please let us know how/if it works there and any issues you discover.

INSTALLING `twander`

Installation of `twander` is fairly simple and takes only a few moments. The most important thing before installing the program is to make sure you have Python 2.2 (or later) with Tkinter support installed on your system.

One other note: However you install the program, it is probably easiest to get started by editing the example ".twander" file to taste. Be aware that this file is shipped with everything commented out. You have to uncomment/edit the section relevant to your operating system: Unix-like or Windows.

Installing Using The FreeBSD Port

If you've installed `twander` using the FreeBSD port, all you have to do is copy the example Configuration File, ".twander" found in /usr/local/share/doc/twander to your home directory and edit it to taste. (You'll also find documentation for `twander` in various formats in this directory as well.)

Make sure that /usr/local/bin is in your path. To start the program, just type "twander.py" from the shell prompt.

Installing Manually On A Unix-like System

Copy the "twander.py" file to a directory somewhere on your path. (/usr/local/bin is a good candidate). Make sure this file has permissions 755 and owner/group appropriate for your system (root/wheel, root/root, or bin/bin). Copy the ".twander" file to your home directory and edit to taste.

To run the program, just type "twander.py" from a shell prompt.

Red Hat Linux Users Please Note: RH Linux (and possibly other Linux systems) installs two versions of Python. Version 1.52 is called ´python´, and Version 2.2 is called ´python2´. `twander` requires the latter and will not run on the former. As shipped, `twander` invokes Python with the Unix shell "#!" mechanism using the name ´python´ - which in this case is the wrong version. You can work around this problem one of several ways:

- Rename ´python´ to ´python1´ and then rename ´python2´ to ´python´. (Not Recommended - could break other programs.)

- Write an alias or shell script which explicity starts `twander` with the correct version of Python:

```
#!/bin/sh
python2 twander.py $*
```

- Change the first line of the `twander` code to refer to ´python2´ instead of ´python´.

Red Hat users who have upgraded from earlier Linux versions should also note that you may have files in your home directories owned by owners and groups which are no longer defined in the system! `twander` shows the owner and group fields for such files as numbers rather than names. As best as we can determine, this is caused when an RH installation is updated from an older version.

Installing Manually On A Windows System

Copy the "twander.py" file to a directory somewhere on your path, or create a new directory to hold this file and add that directory path to the PATH environment variable.

IMPORTANT NOTE TO WINDOWS USERS: Windows has the old MS-DOS legacy of assuming that a "." begins a file "extension". Although you can create and read files in the form ".something", it is not recommended because many Windows programs get confused when they see this. It is also difficult to remove files named this way with the standard Windows programs and utilities. This is especially the case for older Windows operating systems like Win98. For this reason, it is recommended that you rename the ".twander" default Configuration File provided in the program distribution to something else like "twander.conf" and use the `twander` -c command line option to point to this Configuration File.

On Windows, where to put the Configuration File raises an interesting question. Microsoft operating systems normally do not set the "HOME" environment variable, because they have no notion of a "home" directory - Well, they do, but it is called "USERPROFILE" not "HOME". So, you can either create a new user-specific environment variable called HOME yourself (which points to your desired home directory) or you can invoke `twander` with the -c argument to explictly declare where it can find its Configuration File.

You can run the program several ways on Windows systems:

- Create a Windows shortcut which points to the "twander.py" file using the "pythonw" command to invoke it. Normally, starting a Python program from the Windows GUI creates a parent window which persists as long as the program runs. Using "pythonw" instead of "python" to run your program suppresses the creation of this blank parent window. For example, you might have something like this in the "Target:" field of your shortcut:

  ```
  "C:\Program Files\Python22\pythonw.exe" C:\twander.py \
  ```

  This runs the program starting at the root directory of the current drive (assuming "twander.py" is located in C:\.

- Start a command line window and issue a command like the one above directly from the command line.

- Use Windows Explorer (or better still, an already running instance of `twander`!) to navigate to the directory where "twander.py" is located. Double-click on the file. If Python is properly installed, there should be an association for ".py" file types and `twander` should start automatically.

GETTING HELP: THE `twander` MAILING LIST

TundraWare Inc. maintains an email list for `twander` users to get help and exchange ideas. To subscribe, send mail to:

```
majordomo@tundraware.com
```

In the body (not the subject line) of the email, enter the following text, substituting your own email address as indicated:

```
subscribe twander-users your-email-address
```

DESIGN PHILOSOPHY

Graphical User Interfaces (GUIs) are a blessing and a curse. On the one hand, they make it easy to learn and use a computer system. On the other, they are a real inconvenience to experienced users who are touch typists. Taking hands off the keyboard to use the mouse can really slow down a good typist.

Nowhere is this more apparent than in filesystem browsers. In one corner we have the GUI variants like ´Konqueror´ and ´Microsoft Windows Explorer´. These are very easy to use but you pretty much need the mouse in your hand to do anything useful. In the other corner are the text-based file browsers like ´List´, ´Norton Commander´, and ´Midnight Commander´. These are really efficient to use, but have limited functionality and generally do not operate very well on **groups** of things.

Both of these approaches also suffer from the well-known interface problem of "What You See Is **All** You Get" - Each program has a predefined set of commands and the user cannot easily extend these with their own, new commands.

`twander` is another approach to the filesystem navigation problem which embraces the best of both the GUI-based approach and the text-based approach. It also provides a rich mechanism whereby each user can easily define their own command set and thereby customize the program as they see fit. This is done with a number of key features:

1) The **Navigation** of the filesystem is graphical - you can use the mouse to select files, directories, or to change directories. However, each major filesystem navigational feature is also doubled on the keyboard (using Control keys) so you can move around and select things without ever touching the mouse.

2) `twander` also supports a number of **navigation shortcuts.** It provides single control-key access to changing directories, moving to the previous directory, moving up one directory level, moving to any previously visited directory, (de)selecting any or all files/directories in the current view, and escaping to the operating system to run a command. Some (but not all) of these features are also doubled via GUI/mouse operations.

3) There are **no** built-in file or directory commands. All commands which manipulate the files or directories selected during navigation are user-defined. This Command Definition is done in an external Configuration File using a simple but powerful command macro language. This means that that the command set of the program can easily be changed or expanded without having to release a new version of `twander` every time. Better still, every different user can have their own command set defined in a way that suits their style of working. Best of all, commands can be invoked either graphically (with a mouse click) or via a single keypress to minimize moving your hands off the keyboard.

4) Because `twander` is written in Python using Tkinter, the same program runs essentially identically on many Unix-like and Windows systems. The only thing that may need to be changed across these various platforms are the Command Definitions in the configuration file. You only need to learn one interface (and the commands you've defined) across all the different systems you use.

The consequence of all this is that `twander` is an extremely powerful and highly customizable filesystem navigator. Once learned, both navigation and command execution are lightning-fast (or at least, as fast as your machine can go ;) while minimizing dependency on the mouse.

COPYRIGHT AND LICENSING

AUTHOR

Tim Daneliuk
twander@tundraware.com

DOCUMENT REVISION INFORMATION

$Id: twander.1,v 1.143 2006/12/19 16:52:52 tundra Exp $